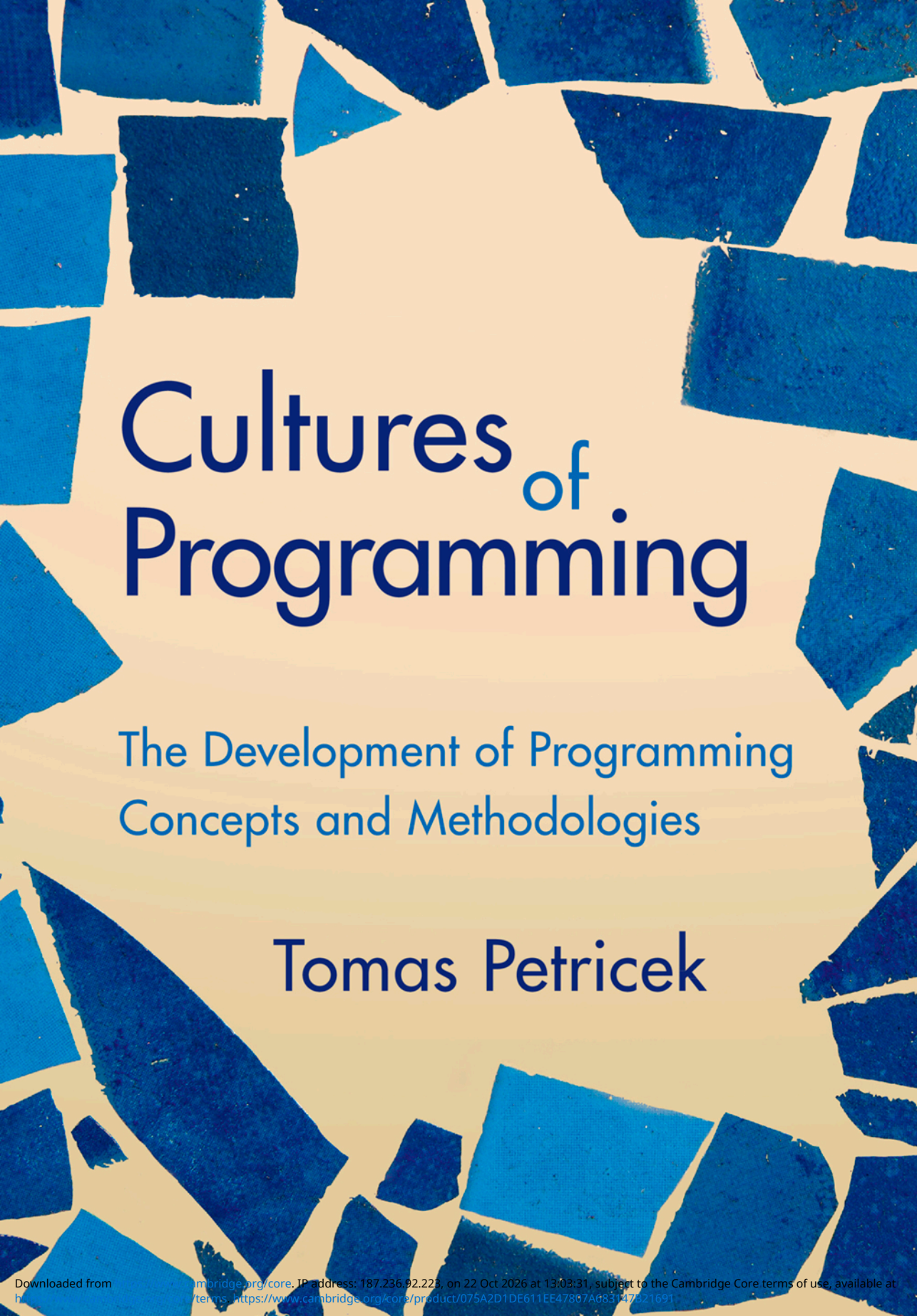


Downloaded from www.cambridge.org/core. IP address: 187.236.92.228, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A6831475521691

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Cultures of Programming

What defines a correct program? What education makes a good programmer? The answers to these questions depend on whether programs are seen as mathematical entities, engineered socio-technical systems or media for assisting human thought. Programmers have developed a wide range of concepts and methodologies to construct programs of increasing complexity. This book shows how those concepts and methodologies emerged and developed from the 1940s to the present. It follows several strands in the history of programming and interprets key historical moments as interactions between five different cultures of programming. Rooted in disciplines such as mathematics, electrical engineering, business management or psychology, the different cultures of programming have exchanged ideas and given rise to novel programming concepts and methodologies. They have also clashed about the nature of programming; those clashes remain at the core of many questions about programming today. This title is also available as Open Access on Cambridge Core.

Tomas Petricek is Assistant Professor in the Faculty of Mathematics and Physics at Charles University, Prague. He uses interdisciplinary research methods to understand programming and to design simple and expressive programming tools. He developed a theory of context-aware computations at the University of Cambridge, worked on the F# programming language at Microsoft Research and built data exploration tools for non-programmers at The Alan Turing Institute and the University of Kent.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

‘Tomas Petricek’s engaging and very readable book argues that programming is a complex human activity that must be understood from multiple perspectives. Written with deep and sympathetic understanding, it provides a unique overview of the history of programming and programming languages and profound insights into current practice that will help demystify the “black art” of coding.’

— Mark Priestley, National Museum of Computing

‘Is programming a practice of hacking, engineering, mathematics, management or humanities? As a computer scientist with the insights and style of a humanist, Petricek leads the reader through the last 70 years of programming to show how these practices have both persisted independently and interacted to produce key technical and methodological innovations through their disputes and synergies.’

— Warren Sack, University of California, Santa Cruz, author of The Software Arts

‘For some, computer programming is an abstract activity like mathematics; for others, an act of craftsmanship; for others, an industrial tool of organisation and control. This book presents a history of programming that preserves all these different points of view; indeed, it shows that these different “cultures” (and others) have been crucial to its development and success.’

— Simone Martini, University of Bologna

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Cultures of Programming

The Development of Programming Concepts and Methodologies

TOMAS PETRICEK
Charles University, Prague

CAMBRIDGE
UNIVERSITY PRESS

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

CAMBRIDGE
UNIVERSITY PRESS

Shaftesbury Road, Cambridge CB2 8EA, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre, New Delhi – 110025, India

103 Penang Road, #05-06/07, Visioncrest Commercial, Singapore 238467

Cambridge University Press is part of Cambridge University Press & Assessment,
a department of the University of Cambridge.

We share the University's mission to contribute to society through the pursuit of
education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781009492348
DOI: 10.1017/9781009492379

© Tomas Petricek 2026

This publication is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, with the exception of the Creative Commons version the link for which is provided below, no reproduction of any part may take place without the written permission of Cambridge University Press & Assessment.

An online version of this work is published at doi.org/10.1017/9781009492379 under a Creative Commons Open Access license CC-BY-NC-ND 4.0 which permits re-use, distribution and reproduction in any medium for non-commercial purposes providing appropriate credit to the original work is given. You may not distribute derivative works without permission. To view a copy of this license, visit https://creativecommons.org/licenses/by-nc-nd/4.0

When citing this work, please include a reference to the DOI 10.1017/9781009492379

First published 2026

Cover image: Simon McGill/Moment via Getty Images

A catalogue record for this publication is available from the British Library

Library of Congress Cataloging-in-Publication Data
Names: Petricek, Tomas author.
Title: Cultures of programming : the development of programming concepts and methodologies / Tomas Petricek.
Description: Cambridge ; New York, NY : Cambridge University Press, [2026] | Includes bibliographical references and index.
Identifiers: LCCN 2025029700 (print) | LCCN 2025029701 (ebook) | ISBN 9781009492348 hardback | ISBN 9781009492386 paperback | ISBN 9781009492379 epub
Subjects: LCSH: Computer programming–History. | Programming languages (Electronic computers)
Classification: LCC QA76.6 .P485 2026 (print) | LCC QA76.6 (ebook)
LC record available at https://lccn.loc.gov/2025029700
LC ebook record available at https://lccn.loc.gov/2025029701

ISBN 978-1-009-49234-8 Hardback

Cambridge University Press & Assessment has no responsibility for the persistence or accuracy of URLs for external or third-party internet websites referred to in this publication and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

For EU product safety concerns, contact us at Calle de José Abascal, 56, 1°, 28003 Madrid, Spain, or email eugpsr@cambridge.org

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Contents

Preface *page* ix
Acknowledgements xii

1 Introduction 1
1.1 The 440-Million-Dollar Bug 5
1.2 A New Perspective on the History of Programming 8
1.3 Program as a Mathematical Entity 10
1.4 The Hands-on Imperative 11
1.5 Software Development Lifecycles 13
1.6 A Proper Engineering Discipline 16
1.7 New Media for Thinking 18
1.8 The Past and the Present of Programming 20
1.9 Developing Software without Errors 21
1.10 Understanding Programs 23
1.11 Programming Education 25
1.12 How Cultures Meet and Clash 27

2 Mathematisation of Programming 33
2.1 Giant Electronic Brains 36
2.2 Sound Body of Knowledge 39
2.3 How Technology Became Language 41
2.4 Mathematical Science of Computing 45
2.5 Many Definitions of Algol 49
2.6 The Minority Report 54
2.7 Go to Statement Considered Harmful 56
2.8 Chief Programmer Teams 58
2.9 Program Proofs and Social Processes 60
2.10 Computing and Cultures of Proving 63
2.11 Fundamental Limits of Program Proofs 66
2.12 Proofs for Machines and Proofs for Humans 70
2.13 Mathematisation of Programming 71

3 Interactive Programming 77
3.1 Spacewar! 81
3.2 Transistorised Experimental Computer Zero 83

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

3.3 Symbol Manipulating Language 86
3.4 Augmenting Human Intellect 87
3.5 A Time Sharing Operator Program 91
3.6 A Step towards Man–Computer Symbiosis 92
3.7 There Should Be No Computer Art 95
3.8 Children, Computers and Powerful Ideas 97
3.9 The Mother of All Demos 100
3.10 Almost Anything Goes 102
3.11 Personal Dynamic Media 106
3.12 Articulate Languages for Communication 109
3.13 Homebrew Computer Club 112
3.14 Beginner's All-purpose Symbolic Instruction Code 115
3.15 The Birth of an Industry 118
3.16 Show Us Your Screens! 122
3.17 Reinventions and Adaptations 125

4 Software Engineering 132
4.1 How Did Software Get So Reliable without Proof? 135
4.2 The Art of Electronic Computer Maintenance 137
4.3 On-line Debugging Techniques 144
4.4 The Debugging Epoch Opens 147
4.5 Orderly and Reliable Exception Handling 151
4.6 System Structure for Fault Tolerance 154
4.7 Professionalisation of Testing 156
4.8 A Slightly Ominous Note for Information Processing Management 159
4.9 Production of Large Computer Programs 161
4.10 Disciplinary Repertoire of Software Engineering 166
4.11 Software Aspects of Strategic Defense Systems 167
4.12 The Mythical Man-Month 171
4.13 Laws of Software Evolution 174
4.14 Paradigm Change in Software Engineering 175
4.15 The New New Approach 178
4.16 A Programming Environment for a Timeshared System 179
4.17 Extreme Programming 181
4.18 The Debugging Paradox 184
4.19 Different Ways of Trusting Software Systems 185

5 Programming with Types 191
5.1 Are Types Invented or Discovered? 195
5.2 Resolving Logical Paradoxes with Types 196
5.3 Evaluating Arithmetic Expressions in Two Modes 197
5.4 Structuring Scientific and Business Data 199
5.5 Towards a Universal Programming Language 202

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

5.6 The Next 700 Programming Languages 206
5.7 Types Are Not Sets 209
5.8 In Search of Types 212
5.9 The Definition of Standard Types 213
5.10 Types as a Lightweight Formal Method 217
5.11 Automating Mathematics Using Types 220
5.12 Programming in Type Theories 222
5.13 The Meaning of Types Is Their Use 226
5.14 Stalking the Elusive Type 229
5.15 Pluralism and Scientific Progress 230

6 Object-Oriented Programming 236
6.1 The Computer Revolution Hasn't Happened Yet 239
6.2 A Language for Describing Discrete Event Systems 241
6.3 Past Ghosts and Present Spectres 246
6.4 A New Medium for Communication 250
6.5 Let's (Not) Burn Our Disk Packs 254
6.6 What Is an Object-Oriented Programming Language? 255
6.7 Structured Programming of the 1980s 259
6.8 The Power of Simplicity 261
6.9 Making Programming Enjoyable for the Serious Programmer 263
6.10 The Enterprise Age of Visual Smalltalk 267
6.11 The Better Way Is Here Now 271
6.12 Managing the Object-Oriented Project 272
6.13 Object-Oriented Programming, Systems, Languages and Applications 277

7 Conclusion: Cultures of Programming 285
7.1 The Most Powerful Tool Available to Human Intellect 288
7.2 Abstracting the History of Programming 288
7.3 Theories of Knowledge 292
7.4 Aesthetics and Cultural Pointers 294
7.5 Exchanging Ideas in a Pluralistic Discipline 298
7.6 Collaborations and Struggles for Control 300
7.7 Why Cultures of Programming Matter 303
7.8 Many Things Go 305

References 309
Index 332

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Preface

The first spark that eventually led to this book was a conversation between two expert programmers that I witnessed at the NDC Oslo conference in 2014. They were advocates of two different programming languages, Erlang and Haskell, talking about how a programming language can help you write good software. The two experts, both knowledgeable and open-minded speakers at the conference, were simply not able to have a reasonable conversation. The Erlang advocate explained how robust error recovery in Erlang makes software reliable, whereas the Haskell advocate was explaining how type systems can be used to eliminate programming errors. One could not understand why bother with types when you will have failures in your software anyway, whereas the other could not understand how restarting a process could make software correct. The two programmers were talking about a similar kind of programming problem, but each of them was looking at the problem from a different perspective and hence saw very different issues.

Despite the contemporary inspiration, this book is about the history of programming. Once you start looking for the kind of disagreements that I witnessed in Oslo, you find them in many places throughout the history of programming, ranging from the 1968 conference that popularised the term 'software engineering' to the debate in the 1980s about whether formal verification of software is logically possible and the rise of Agile development methods in the first decade of the twenty-first century. Fortunately, the interactions between different ways of thinking about programming do not lead just to disagreements. Once you start looking, you also find that many great advances in programming happened when different ways of thinking contributed to a single concept, starting with the very idea of a programming language.

What is surprising about those interactions is that the different ways of thinking about programming that have been shaping the discipline since the 1950s remain surprisingly stable over its 70-year history. In the 1950s, the different ways of thinking, which I refer to as *cultures of programming*, originated from the individual disciplines that contributed to programming including logic, mathematics, electrical engineering, but also business, psychology and military research. In the early days, this diversity may have been a sign of an immature field, but 70 years later, the existence of the same cultures of programming is instead a sign that programming is, and will remain, an inherently pluralistic discipline.

This book retells the history of programming through the perspective of interactions between five cultures of programming: the hacker; the engineering; the mathematical;

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

the managerial; the humanistic. While the five cultures are often linked to specific communities, the ways of thinking they represent also reappear independently in different contexts. As such, my use of the term culture is figurative rather than literal. I use it to highlight that different cultures of programming have different beliefs, assumptions and practices. Many of the cultures I identify in this book are, in some way, a part of computing folklore. This book exposes their more basic nature. To do so, I follow a number of historical strands from the 1950s to the present day, looking at how the different cultures of programming clash over the nature of programming, but also how they contribute to programming concepts ranging from programming languages and software engineering to structured programming, types, objects and tests.

Thinking about the different cultures of programming sheds a new light on the technical history of programming, but it also provides a new way of thinking about contemporary programming issues. The answer to what is a good program and how to create one differs dramatically when we see programs as mathematical entities, engineered socio-technical systems or media for assisting human thought. Similarly, the answer to how to best teach programming differs when you see programming as applied mathematics, a new form of literacy or a craft that can only be mastered through practice. To borrow a programming term, cultures of programming are a useful abstraction.

This book also points to broader socio-technical issues that are often discussed in the context of the history and philosophy of computing. The history of software engineering cannot be told without mentioning the struggle for managerial control and attempts to develop a more masculine notion of a computer professional. Similarly, the history of interactive programming cannot be told without a reference to the 1960s counterculture and the political ideology of free software. Although I focus on how different cultures of programming shape the way we program, the individual cultures can often be associated with particular social and political views. This book makes it possible to draw connections between the technical and the social history. In doing so, it is more technical than most history books and more historical than most computer science books.

My hopes for the readers of this book are twofold. On the one hand, I hope to provide computer scientists and programming practitioners with a useful perspective through which to look at their field, a much-needed context for understanding ‘how we got here’, but also to provide material that can inspire new ways of thinking. On the other hand, I hope to make the rich technical history of programming legible to those who are not programmers themselves, so that it can serve as a source of interesting episodes for further historical and philosophical reflection. Serving this dual purpose requires some careful balancing. In particular, I will sometimes use modern terminology to describe things from the past if it helps to make the text understandable. One specific challenge in writing this book has been the capitalisation of early programming language names where I generally follow the capitalisation used in recent historical reflections written by their creators.

To emphasise the importance of interactions between the five cultures of programming, each chapter starts with a dialogue between a teacher and students that represent

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

positions of the individual cultures. Although I sometimes adapt quotes from actual historical exchanges, the dialogues are a work of fiction.

In reality, individuals are much less explicit about their beliefs and rarely align with a single culture of programming completely. As we will see, the most interesting historical actors often bridge and contribute to multiple cultures over time. The purpose of the dialogues is to illustrate the clashes between the cultures, the ways in which they structure their arguments and how they can contribute to a single programming concept despite having different basic ideas about the nature of programming.

The students in the dialogues are named after ancient Greek philosophers. This distances the dialogues from actual historical actors and debates, but it provides a mnemonic for remembering which character represents which culture. I am well aware of the many inaccuracies and limitations of the name choices, but I hope you can forgive me those and enjoy the dialogues. The five students we will encounter in the openings of each chapter are:

Pythagoras, representing the *mathematical culture*, is named so for his many mathematical discoveries, but also the mystic view that all things were made of numbers. According to Aristotle, he believed that the principles of mathematics were the principles of all things.

Diogenes, speaking for the *hacker culture*, is named so as a critic of a corrupt society who believed that virtue is better revealed in action than in theory. He is also known for his unconventional lifestyle that included sleeping in a ceramic jar.

Xenophon, a philosopher and a military leader, represents the *managerial culture*. He is named so as the leader who managed a military force of 10,000 Greek mercenaries and for his later writings on its organisation.

Archimedes, speaking for the *engineering culture*, is named so as one of the first thinkers applying mathematics to physical phenomena and for his many practical inventions such as a screw pump and a compound pulley.

Socrates represents the *humanistic culture* for his pursuit of truth and knowledge, for his critical Socratic method of inquiry and his influence on the later humanist movement. He famously believed that the unexamined life is not worth living.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Acknowledgements

Looking at the acknowledgements sections of books that inspired me at the start of this project, I was always surprised by how long and how well written they were. Did the authors really talk to that many people, obtain and go through such quantities of feedback and visit that many places? If so, I thought, it must be almost impossible to write a book! Well, my acknowledgements are going to be equally long. When writing a book, you do, indeed, end up talking to many people, receive and incorporate lots of feedback, visit many places, stay at different institutions or even move countries. Alas, my acknowledgements are not going to be equally well written, because this is the last thing I need to finish before handing the final manuscript to my editor.

As a result of the unfortunate compartmentalisation of modern academia, being a computer scientist studying history and philosophy of the discipline raises some eyebrows and leads to the wrong kind of 'research outputs'. I'm grateful to my advisors, group leads and department heads who saw the value in my work and supported me over time, even if it did not fit in the usual boxes. This includes Alan Mycroft at the University of Cambridge, Richard Jones, Simon Thompson, Sally Fincher and Mark Batty at the University of Kent, and Petr Tůma at Charles University. I'm also grateful to David Corfield from the Department of Philosophy at the University of Kent who has been keen to cross the disciplinary boundary from the other side. The fact that the university administration has since decided to close down this thriving and highly ranked department remains a sad postscript to our inspiring collaboration.

I was fortunate to discover that even among computer scientists, many believe in transcending narrow disciplinary boundaries. The informal Cambridge Revolutionary Beers group provided both a sounding board for many of the ideas in this book and moral support over the numerous rejections and revisions that this manuscript went through. Thanks in particular to Antranig Basman, Stephen Kell, Joel Jakubovic, J. Ryan Stinnett and Dominic Orchard. Luke Church, Sam Aaron and Jonathan Edwards may not have often been there in person, but have regularly been there in spirit. The much-needed institutional support for the group has been provided by Fort St George, the Salisbury Arms and The Parcel Yard.

Outside my local circles, the impetus to start writing came in 2017 when Richard Gabriel and Guy Steele encouraged me to submit a paper drawing on my earlier work on programming errors to the Association for Computing Machinery (ACM) History of Programming Languages conference. I'm grateful for the initial nudge, but more importantly for their continued support of the project and feedback on multiple drafts

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

of the manuscript. The work of Dick Gabriel repeatedly reminded me that computer science does not have to be mere paradigmatic puzzle solving and served as inspiration throughout the work on the book. I'm also grateful to James Noble, Gilad Bracha, Rebbeca Wirfs-Brock and Allen Wirfs-Brock who provided interesting additional historical information and feedback on parts of the book.

Other communities that were instrumental for the development of this book centre around a number of industry-oriented programming conferences. These provided the opportunity to encounter first-hand several of the culture clashes that I write about in this book and to talk to the participants in order to figure out what was going on. The long-term organiser of the functional programming track at the NDC Oslo conference, Bryan Hunter, deserves credit for facilitating the meeting that I wrote about in the Preface, but also for many late-night chats that encouraged me to work on things that I truly believe in. Andrea Magnorsky, who brought history and philosophy of programming to the CodeMesh conferences, not only invited me to share my early ideas, but also broadened my perspective in many ways and connected me to another group of like-minded people. I'm grateful to Alvaro Videla and Miles Sabin for their comments. Finally, the organiser of the NewCrafts conference in Paris, Rui Carvalho, managed to create a unique space for discussing the broader picture of programming. I loved coming to learn about intriguing philosophical and technical ideas and for the insightful discussions. The list of NewCrafts community members who gave me interesting ideas and references includes Romeu Moura and Arnaud Bailly, but is much longer. I may have initially come to NDC, CodeMesh and NewCrafts, as well as Lambda Days and other places to share my passion for F# and I appreciate them inviting me again as my interests shifted to history and philosophy.

If the length of the mention in the Acknowledgements was to be proportional to the influence that a person had on my work and thinking, the next couple of pages would have to be dedicated to Liesbeth De Mol. Her thorough knowledge of the early history of programming significantly influenced my writing and her uncompromising personal commitment to honest and rigorous scholarship has been a continuous inspiration. Liesbeth also established the History and Philosophy of Computing (HaPoC) community and the collaborative Agence Nationale de la Recherche (ANR) PROGRAMme project that have provided unique venues for thinking about the history and philosophy of computing. There is so much in this book that has been influenced by the many discussions at a HaPoC conference, History and Philosophy of Programming (HaPoP) symposium or the PROGRAMme retreats. The list of people from those communities with whom I was able to discuss the ideas leading to this book is, again, too long. Those whose work had the most direct effect on this book include Troy Astarte, Pierre Depaz, Simone Martini, David Nofre, Mark Priestley and Julian Rohrhuber. The book *A Science of Operations* by Mark Priestley gave me the first ideas about what research on the history of programming can be. The in-depth debates on programming that took place at the CEUB Bertinoro were equally influential, even if they are not visible in the References. In addition to those mentioned already, the list of those who shaped my thinking and work include Selmer Bringsjord, Maarten Bullynck, Felice Cardone, Martin Carlé, Edgar Daylight,

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Marie-José Durand-Richard, Cliff Jones, Baptiste Mélès, Elisabetta Mori, Pierre Mounier-Kuhn, Alberto Naibo, Maël Pégny, Giuseppe Primiero, David Schmudde, Henri Stephanou, Matte Szabo, Ray Turner and Nick Wiggershaus. The staff at CEUB Bertinoro and the many bars where those discussions took place deserve gratitude for making them possible, often well past the advertised closing time.

In more practical terms, I'm grateful to Karen Merikangas Darling for initial support of the project and to Adrian Johns for feedback on early drafts. The book would never have been completed without the help of my editorial team at Cambridge University Press, Lauren Cowles and Arman Chowdhury, who helped me navigate the tricky details of book production. I'm grateful to the designer of the beautiful book cover, which playfully illustrates how the different cultures of programming fit together and to Fiona Cole, who copyedited the manuscript on an impossibly tight schedule and not only brought consistency to the text but also saved me from countless embarrassing mistakes. I'm indebted to everyone who kindly replied to my email enquiries about image permissions.

I'm also grateful to the many people who work hard to preserve the historical material on which this book draws. This includes Al Kossow, the creator of bitsavers.org, the team behind the Computer History Museum, the Internet Archive, as well as the, now sadly defunct, Living Computers Museum. Petra Hoffmannová from the Library of the Faculty of Mathematics and Physics was able to find a curious article in a long-lost journal. I also greatly benefited from the resources collected by multiple shadow libraries. Their operators and supporters do more than anyone to preserve academic knowledge at great personal cost.

On a more personal note, I would like to thank my parents, who probably caused some of this by surrounding me with mountains of books from my early childhood in the most unexpected locations including the goat barn. I could probably estimate how many banana boxes you need for a bookshelf before I learned how to read. I'm grateful to my family, Evelina, Juliana and Alfred for the joys they bring to the part of life not spent in front of the laptop screen. They made the lonely task of writing a book bearable. I might be tempted to blame the latter two for the late timestamps of most of my git commits, but being a night owl anyway, this would be unfair. I am, however, immensely grateful to the former for making that way of working possible.

The work on this book has been financially supported by the University of Kent and Charles University where I did some of the work on the project. I received invaluable financial support through an ACM History Committee Fellowship, the ANR PROGRAMme project 'What is a computer program?' (ANR-17-CE38-0003-01), from the Charles University-funded research centre 'Language, Image, and Gesture: Forms of Discursivity' (UNCE24/SSH/026) and from the Charles University-funded grant 'Type systems for data-centric programming' (PRIMUS/24 /SCI/021). I am also extremely grateful to the Faculty of Mathematics and Physics and the Department of Distributed and Dependable Systems, for jointly providing financial support that enabled an open access publication of this book.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

1 Introduction

Teacher: The author of this book believes that we all have different ways of thinking about programming. I wonder if that is really the case. Maybe we can first find out whether we agree on what programming is in the first place. . .

Xenophon: I do not see how we could disagree on this. Programming is the process of developing a software system that solves some business problem.

Socrates: This is a painfully limited perspective! Programming is a tool for understanding the world. It can equally inspire and be used to express new creative ideas.

Archimedes: Those are lofty visions, but in reality, most programming is done to solve a business problem. I see a grain of truth in what you say though, because modern development methodologies make understanding of the problem, obtained through programming, a key part of the iterative development lifecycle.

Socrates: You keep treating programming as a boring commercial utility. It is better seen as a kind of literacy. Programming forces you to think about the world in a clear structured way that you otherwise do not experience. It teaches a new way of thinking.

Pythagoras: I'm all for treating programming as a new kind of literacy, but only because it is really a form of applied mathematics. It teaches mathematical thinking. Programming is a process of constructing a mathematical entity in a formal language.

Diogenes: Excuse me! You keep talking about requirements, thinking and mathematics, but continue to completely ignore actual computers. In the end, there are always bits to be shuffled around. Programming is about getting the machine to do what you want and, at its best, exploring the possibilities of what can be done.

Teacher: Even if we disagree about the nature of programming, perhaps we can agree on what counts as paradigmatic achievements in programming. What do you consider as exemplary or the most important development in programming?

Archimedes: I do not want to name a single achievement, but very large computer systems are everywhere around us, ranging from our phones to medical devices, and they generally work reliably. There is definitely something right about the way we are doing programming.

Pythagoras: Your standards are very low! Most software has bugs and you often have to learn tricks to work around those. In 2012, a program bug cost Knight Capital $440 million in just 45 minutes and the bug in the Therac-25 radiation machine actually cost human lives. We are lucky that the high-profile failures are not even more common. . .

Socrates: What I find more worrying is that we often do not understand the effects of programs that we create. Consider the numerous AI chatbots that have learned to imitate the inflammatory and racist language of their users or their training datasets![1]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Teacher: We will get to issues with programming soon enough, but I wanted to start with important achievements and positive examples. Can we please get back to those?

Archimedes: As I said, I do not think there is a single achievement. What matters is that we are gradually getting better at programming and are able to bring value to the society. A good example is the Agile movement, which takes as its central the idea that business people and developers need to work more closely together.

Socrates: Phrases like 'we value individuals and interactions over processes and tools' from the Agile manifesto sound nice. But the Agile movement also subordinates programming into a support role for commercial enterprises. Like earlier software development methodologies, it is a mechanism for control. Not to mention the fact that all 17 authors of the Agile manifesto are men, often in leading consulting roles, so the perspective they can offer is inevitably narrow.[2]

Xenophon: I do not understand the issue you have with control. If you want to build anything interesting, you need a large team of programmers and then you obviously also need some form of team structure or 'control' if you wish.

Diogenes: I agree with *Socrates* that this is a misguided view, but first I'm curious to hear what *Xenophon* finds to be an example of paradigmatic programming achievement.

Xenophon: An example? The development of the Apollo guidance computer (Figure 1.1), which helped to land a man on the moon. The development had its difficulties, but those were resolved thanks to rigorous definition of requirements, followed by careful coding and rigorous testing.



Figure 1.1 Computer scientist Margaret Hamilton poses with the Apollo guidance software she and her team developed at MIT. (Source: MIT, Wikimedia Commons)[3]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Pythagoras: You are playing it safe! Nobody can disagree that landing a man on the moon is an impressive achievement, but even the Apollo guidance software had bugs.

Xenophon: That is correct, but those bugs were well documented and the crew knew how to operate the computer to avoid their effects. Flying with the bugs simply had a lower risk than attempting to fix them at the last minute.[4]

Pythagoras: This is why I find the present state of computer programming unsatisfactory. A program is a formal entity and so you can use formal mathematical methods to show that a piece of software is correct. We should build software that is provably without bugs rather than coming up with post-hoc workarounds!

Diogenes: Has anybody actually done this in practice? What is your example?

Pythagoras: Formal verification is challenging, but there are some good examples such as the formally verified microkernel sel4 that has been used as the basis for systems that are robust against, including one that controlled an unmanned flight of the AH-6 helicopter.[5] But my example of a paradigmatic achievement would be the Algol programming language, which pioneered the idea of treating programs as mathematical entities that can be formally analysed and made all the follow-up work thinkable.

Teacher: Our examples so far include the Agile movement, the Algol language and the Apollo guidance system. *Diogenes* and *Socrates*, would you agree that they are good examples of the great achievements of programming?

Socrates: They may be great, but they are boring. The amazing thing about computers is that they give you an unprecedented creative freedom. What you can do with a computer is more restricted by your imagination than by technical limitations!

Diogenes: This creative freedom of programming inspired the MIT hackers[6] in the 1960s who were instrumental in creating the ARPANET, a predecessor of the Internet, and early computer games like Spacewar! But if I was to choose one example that emerged from those circles and that is relevant to programming today, it would be the UNIX operating system and the C programming language.

Xenophon: I thought that C remains widely used only for historical reasons. Aren't most people trying to find safer and more expressive alternatives these days?

Diogenes: This is a common myth, but the power of C is in that it lets you access and freely communicate with anything on your computer. The C language captured this idea when it was created and it still follows this basic principle.[7]

Socrates: I too think that the early MIT spirit is a source of many good ideas, but I would not choose UNIX and C as the best examples. To me, these two sound like the exact opposite of systems showing the creative potential of computers. I much prefer creative use of programming like the Spacewar! game or the graphical system Sketchpad. But today, the most interesting creative use of programming is live coding of computer music performances at Algorave events.

Pythagoras: What do you mean? Live coding as in people showing how to create small programs live during conference presentations or lectures?

Socrates: I mean using programming in live musical and multimedia performances. In 2019, there were 70 different Algorave events all over the world where you can see live coded audio-visual performances in action in a club! It shows the creative potential of programming in a completely transparent way. Live coding systems like Sonic Pi are also used in classrooms all over the world to teach programming to kids.

Xenophon: When *Diogenes* started talking about hackers, I thought the discussion was becoming a bit odd, but using live musical performances as the most notable example of programming is just crazy! How is that using computers for anything useful?

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Socrates: Do you not find education and exploring creative potential of computers for a new kind of art useful? If you want to question what kind of use of computers is useful, I would worry more about the ways in which they get used by large corporations!

Teacher: Do you have anything specific in mind, *Socrates*?

Socrates: One specific example would be the secret resume screening AI tool built by Amazon that journalists uncovered in 2015.[8] This was supposed to identify good candidates and Amazon trained it on resumes received in the past 10 years. It turned out that the algorithm was heavily biased against women and would penalise resumes including phrases such as 'women's chess club captain'.

Teacher: This might be an interesting case to talk about. Perhaps the plurality of our views on programming will let us better identify what the issues with programming are and find a way of addressing those.

Pythagoras: I agree the Amazon system is unacceptable. But this is not an issue with the algorithm itself. The issue is that it was used poorly. Presumably, the algorithm just learned a bias that was already present in the training dataset.[9]

Socrates: Training data is one issue, but not the only one. This is a perfect example of why you need to think about programming as a human activity. Algorithms are designed by humans, built by humans and used by humans. A human bias can enter the scene at any point during programming and operation of software.

Pythagoras: According to a dictionary definition,[10] an algorithm is 'a set of mathematical instructions or rules that, especially if given to a computer, will help to calculate an answer to a problem'. A set of mathematical instructions is a mathematical entity. An algorithm cannot be any more biased than a multiplication or a monoid!

Diogenes: You are wrong. You can have an instruction `if(gender="M")salary+=1000`, which increases the salary for all men and this is quite clearly biased. . .

Pythagoras: But this is mixing data with your algorithms! What I mean by an algorithm is a fully generic procedure. An artificial neural network on its own does not contain any hard-coded information about a specific problem in this way.

Diogenes: Even if the algorithm is generic, programming still relies on implicit practical human knowledge. In an artificial neural network, you have to set the number of layers, propagation functions, hyperparameters and so on. All of these can be a source of bias. What worries me about algorithms like neural networks is that it is very hard to gain the necessary practical experience to avoid such issues.

Archimedes: You can build systems that guarantee fairness, but not by relying on unreliable practical experience. For systems that make decisions about individuals, you can use counterfactual fairness,[11] which ensures that the result given by the algorithm is the same regardless of the demographic group of the individual.

Xenophon: I have to admit, this discussion is beyond me. I can see that there are more and less problematic algorithms, but which one should I use if I need to conform with the right to explanation required by the EU GDPR regulation? The industry needs to agree on standards that we can adhere to in order to avoid such problems.

Socrates: A regulation might solve your problem in the short-term. In the long-term, programmers needs liberal arts education that includes social sciences, arts and philosophy. Much of the discussion we're just having is already present in Heidegger's work *The Question Concerning Technology* from 1954!

Teacher: We are getting back to arguing about the nature of programs. But I'm still curious if we can use our different perspectives in a more productive way.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Xenophon: I don't see how I could have a productive conversation with anyone who ignores the business context in which programming is typically done.

Diogenes: It will always be counter-productive to just talk. What matters is code!

Teacher: I'm sure the others would disagree with this, but perhaps it points in the right direction. If we shift our attention from the nature of programming to concrete programming concepts, we may be able to find ways through which our different views can contribute to the development of programming.

1.1 The 440-Million-Dollar Bug

The day did not start well for the financial services firm Knight Capital on August 1, 2012. Before trading opened, the firm deployed a new version of its automated routing system for equity orders known as SMARS. The system received parent orders from other components of Knight's trading systems and turned them into one or more smaller child orders sent to stock exchanges. In the 45 minutes after trading opened on August 1, SMARS received a modest 212 parent orders, but generated millions of child orders because of a software issue. These orders resulted in 4 million executed trades of 154 different stocks, greatly exceeding the intended number of trades. Knight Capital lost $440 million as a result of these trades.

This bug reveals the many subtleties of programming and operation of computer systems that we need to understand if we are to make sense of how algorithms and programs affect our society.[12] The subtleties are well documented thanks to an investigation conducted by the Securities and Exchange Commission that eventually made Knight Capital pay a further $12 million for violating a rule that requires firms to adequately control risk associated with direct market access.[13]

The new version of the SMARS system, deployed on August 1, contained new functionality for accessing the Retail Liquidity Program (RLP) provided by the New York Stock Exchange. This was supposed to replace the older 'Power Peg' code that was no longer in use and the developers repurposed a flag that originally enabled Power Peg to instead activate RLP. On August 1, the new version of SMARS was successfully deployed on seven out of eight SMARS servers, but one server kept running the old version of the system. On this one server, the newly enabled flag activated the old Power Peg code. This code was no longer tested and because of other changes in the system, it was not correctly checking that enough child orders had been generated and kept issuing more and more orders. When the technical staff at Knight Capital started investigating the issue, their initial thought was that there was an error in the new functionality and they temporarily rolled-back the previous version of SMARS. This only made matters worse by running the Power Peg code on all eight servers and the whole system was eventually disconnected from the market.

What is interesting about the Knight Capital bug is not just its complex nature, but also the speculations that it provoked concerning how the error could have been

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

prevented. Envisioning future computer revolutions is a popular pastime in the field of computer science[14] and counterfactual speculations about what could have been are often used to argue for a specific vision or agenda. The speculations reveal the different ways of thinking that are specific to the different cultures of programming. I focus on three examples that specifically reference this bug.

The Knight Capital bug makes for a perfect example for a financial startup pitch. Indeed, it was featured in the invited keynote 'Formal Verification of Financial Algorithms, Progress and Prospects'[15] by Grant Passmore at a 2017 workshop dedicated to the ACL2 theorem prover. The speaker, a co-founder of an automated reasoning startup, believes that financial algorithms are 'the killer app for formal methods'. The specific vision outlined in the talk is that matching of financial orders could be mathematically formalised and analysed to formally prove that it follows the various required financial regulations. Knight Capital is mentioned not as a specific example for this goal, but as a general example of 'glitches' that make financial markets 'notoriously unstable'.

What is left implicit in the talk is the idea that formal mathematical methods provide a way of solving all the various issues that plague financial software. This is exactly a type of often unfulfilled promise: 'See what computers are on the verge of doing. It will be revolutionary!' made by computer specialists.[16] In reality, the Knight Capital issue is way more subtle, because the error was caused not just by buggy code for the Power Peg, but also by incorrect manual deployment of the software. To avoid the issue, we would not only need to exactly specify what the desired trading behaviour is, but we would also need to automate and formally verify the deployment. This is, perhaps unsurprisingly, a subject of various ongoing research projects, but it also shows that there will always be potential sources of error, outside of the scope of what has already been formalised.

Treating programs as mathematical entities that can be formally analysed is the cornerstone of the mathematical culture of programming. As we will see in Chapter 2, the idea is commonplace in academic computer science. It often finds its way to industry, but frequently through envisioned revolutions that have not quite happened yet. The fact that the mention of Knight Capital appears in an invited talk at an academic workshop is also not inconsequential. Academic venues are often a mechanism through which knowledge is shared in the mathematical culture of programming.

The Knight Capital bug also prompted many comments from the professional software development community. It happened at a time when the DevOps movement was gaining popularity in industry and Knight Capital served well as a cautionary tale. The movement envisioned a different way of avoiding the issue from the mathematical culture. The term DevOps refers to a range of engineering practices that more closely integrate the 'development' of software with its 'operation'. DevOps aims at a 'rapid IT service delivery' and 'utilise technology – especially automation tools'[17] to make the deployment process more efficient, reliable and free from potential human mistakes. A blog commentary on Knight Capital written by Doug Seven, working at Microsoft at the time, reiterates the belief that 'deployments need to be automated and repeatable' and speculates that this would have prevented the Knight Capital disaster: 'Had Knight

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

[Capital] implemented an automated deployment system [the error] would have been avoided.'[18]

Again, the response illustrates several cornerstones of the programming culture from which it emerged, that is, the engineering culture of programming. Much of the work done in the context of the engineering culture originates from practitioners. Terms like DevOps or Agile, which I will discuss in Chapter 4, often capture sets of practices that have developed organically in the community. The ideas spread through industry conferences, books, blogs, reports and also through commercial training offered by respected engineers. The engineering culture recognises that humans inevitably make mistakes and tries to find tools and processes to make those mistakes less frequent and less severe. In the case of the Knight Capital bug, we have a respected practitioner writing a blog post that recommends the use of tools for deployment automation. Such tools do not offer a formal guarantee, but they would likely be sufficient to eliminate the error. The engineering culture would also advise Knight Capital to remove 'dead code', that is, the Power Peg code which was no longer in use. This would also have avoided the issue.

Last but not least, the Knight Capital bug prompted a direct response by the Securities and Exchange Commission (SEC) that investigated the firm for violating 'Rule 15c3-5',which has been adopted by the commission in 2010 to control the risks associated with automated trading. Among other obligations, the rule requires trading firms with direct market access to implement 'financial risk management controls and supervisory procedures' that 'prevent the entry of orders that exceed appropriate pre-set credit or capital thresholds, or that appear to be erroneous'.[19] The SEC investigation of Knight Capital[20] reports that the 'system of risk management controls and supervisory procedures [adopted by Knight Capital] was not reasonably designed to manage the risk of its market access' as required by Rule 15c3-5. The investigation stops short of saying that having such controls would have avoided the bug, but doing so was the motivation for introducing Rule 15c3-5 in the first place.

The analysis of the Securities Exchange Commission illustrates a way of approaching programming that I refer to as the managerial culture. Rule 15c3-5 is a good example of work done in this culture, because it attempts to address a programming issue through a higher-level regulation of the system. It requires 'risk management controls and supervisory procedures' that may have different forms, but must include several checks before an order is submitted and human oversight of the executed trades. The compliance with the rule is not checked through a formal proof or by a tool, but by a human process. The CEO of the trading firm is required to certify, on an annual basis, that the controls and procedures are implemented. Such format of regulation is typical for the managerial culture. It is a rather lengthy document, written in formal English and produced by a somewhat bureaucratic organisation. We will see that this is often the case for the managerial culture, for example, when we encounter the IEEE standards for testing, documentation and software verification in Chapter 4.

The three responses, mathematical, engineering and managerial do not cover all the cultures that I am writing about in this book, but they are the perfect introduction to the ways in which cultures of programming differ. They show that cultures adopt

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

different ways of thinking about programming, envision different ideals. They also share knowledge in different ways and have very different requirements for trusting software. The remaining two cultures that I write about are the hacker culture and the humanistic culture. The former values direct engagement with the machine and views programming as a highly individual skill. The hackers would likely be surprised that deployment was actioned by a separate person and that the technical staff needed so much time to understand that something was going wrong. Finally, the humanistic culture views programming as the extension of human thought and often envisions broader social implications of technology. They would likely wonder if fully automated trading, without any human involvement, was a socially beneficial idea in the first place.[21]

1.2 A New Perspective on the History of Programming

It is easy to get lost in the various proposals for how the Knight Capital bug could have been prevented. Should the firm have implemented more supervisory procedures, used formal verification, employed more automation or built a system with more immediate feedback? It is likely that any of these would have been sufficient, but how do programmers decide which way to advocate and implement? The concept of *cultures of programming* that I develop in this book provides an effective structure for analysing developments and debates such as this one.

In the case of the Knight Capital bug, the proponents of different cultures of programming offered different views on the source of the problem and, through this, revealed their basic assumptions about programming, but there was no direct interaction between them. In some of the most interesting historical episodes that I will discuss in the rest of the book, this will not be the case. The most interesting cases are the ones where the proponents of different cultures of programming interact. When the basic nature of programming is at stake, they often clash and disagree. Is programming a matter of constructing the source code of a program that is then compiled or executed? should we see it as a process of interacting with a stateful computer system, or should we see it as large-scale engineering effort that needs to be carefully planned and managed?

The proponents of the different cultures of programming can also productively collaborate. This is often the case when multiple cultures contribute to a shared technical concept. A technical concept such as a programming language, a test, a type or an object may mean a different thing to each of the cultures, but this does not prevent other cultures from coming up with new ways of using it and extending it.[22] For example, when the idea of a programming language emerged in the 1950s, it was a formal mathematical language to the proponents of the mathematical culture, a clever trick for making programming easier for the hackers and a way of building software that is independent of a specific machine for the managerially minded users

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

of computers.[23] Despite the different interpretations, each of the cultures was able to contribute something to the shared notion of a programming language, be it a commercial motivation, compiler implementation or a language definition. I will even argue that such collaborations often advanced the state of the art of programming in ways that would unlikely happen within a single consistent culture.

In Chapters 2–6, I revisit five different strands of the history of programming, looking at multiple interesting moments and achievements. I include paradigmatic achievements of specific cultures, but also the achievements that resulted from interactions between the different cultures of programming. We will see that this perspective sheds a new light on interesting events throughout history, ranging from the birth of programming languages in the 1950s to the development of Agile programming methodologies of the 2000s. Although this book is primarily about history, the perspective of cultures of programming is equally useful for making sense of contemporary controversies around programming and possibly even for imagining different directions for the future of programming. I look at how the perspective of cultures of programming sheds a new light on current debates about program errors, understanding of algorithms and the issue of programming education later in this chapter.

The concept of a culture of programming is a post hoc construction that provides an explanatory narrative for the history of programming. I use the term culture to highlight the fact that the different ways of thinking about programming come with their particular beliefs, values, assumptions and practices that shape the views of their proponents in fundamental and often unacknowledged ways and to show that those basic assumptions are often hard to escape. As such, the idea is closer to that of a scientific paradigm than a programming style.[24]

Most of the cultures of programming that I talk about are based on notions that are a part of the computing folklore. For those, this book adds more depth. I will discuss how the proponents of the different cultures think about programming, what methods they use in their work and how they exchange information and build knowledge. Still, the cultures of programming do not exist in some objective epistemological sense. They are derived from the broad range of examples that I discuss throughout this book and they provide a fitting explanation for those. This does not mean that programmers themselves subscribe to a particular culture of programming or that there is some objective way of determining what work originates in which culture. In fact, many of the computing pioneers that we will encounter combined the perspectives of multiple cultures of programming, which may well be the reason why they are remembered.

For these reasons, the characters that appear in the opening dialogues of each of the chapters should not be seen as actual historical actors. They are idealised representations of the different ways of thinking about programming and methods of working. The next five sections provide a gentle introduction to the five cultures of programming. Those are the mathematical culture, hacker culture, managerial culture, engineering culture and humanistic culture, represented in the dialogues by *Pythagoras*, *Diogenes*, *Xenophon*, *Archimedes* and *Socrates*, respectively.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

1.3 Program as a Mathematical Entity

The key characteristic of the mathematical culture of programming is that it treats programs as mathematical entities. Such programs can be studied using formal methods and it becomes possible to prove that they are correct. This may seem obvious to a reader familiar with the basics of computer science, but it is not at all obvious if we look at the history of programming. The first computer programmers came from a variety of backgrounds and included engineers, scientists, mathematicians and human, typically female, computers.[25] In the early days, the mathematicians were involved more in planning of the computation and in the numerical analysis of the equations to be computed than in producing code to instruct the machine. In fact, the Electronic Numerical Integrator and Computer (ENIAC), which was the first programmable general-purpose electronic computer, was initially programmed by physically plugging wires to connect components. A program was the physical setup of the machine, rather than some mathematical entity!

The path to the mathematical way of looking at programming relied on both technical and social developments. On the technical side, programming evolved from plugging of wires to, first, writing sequences of machine instructions and, later, to writing of symbolical formulas that were processed by a computer program such as the FORTRAN translator. On the social side, the mathematisation of programming was a part of a broader attempt to establish computer science as a respectable discipline in the university context.[26] The mathematical perspective provided the, politically much needed, rigorous foundations.

The Algol language, which appeared in the late 1950s, is a paradigmatic example that illustrates both of these developments. It differed from its predecessors in that it has been formally specified and was independent of any particular machine. Algol was designed by a committee set up by the ACM, a professional organisation that was one of the key forces that aimed to turn computer science into a respectable academic discipline. Although Algol was never widely used and was a failure as a practical programming language, it was regarded as an 'object of beauty' by the proponents of the mathematical culture.[27]

The Algol specification included several variants of the concrete syntax, including the so-called publication language to be used in publications involving Algol. This peculiarity has a significance too. Academic publications are the primary way of exchanging knowledge in the mathematical culture of programming and this was directly supported by Algol. For example, variants of the Algol publication language were often used to write algorithms that were published in the regular 'Algorithms' column published regularly by the ACM magazine, *Communications of the ACM*, throughout the 1960s and 1970s. The new way of thinking about programming, established by Algol, inspired a plethora of theoretical and practical developments. I return to the birth of programming languages and Algol in Chapter 2 and discuss one specific development, the notion of types, in Chapter 5.

Perhaps the most basic question that arises in the mathematical way of thinking about programs is whether it is possible to prove that a program is correct.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The proponents of the mathematical culture believe that this is, indeed, the case. For them, a computer program is a mathematical entity and so we can formally describe its properties. The fact that this view was not universally accepted generated a heated debate that I will recount in Chapter 2. A number of objections were raised. First, it is often hard to specify what a correct behaviour is. Second, programs consisting of millions of lines of code cannot be compared to one-line logical propositions, even if just because of their size. Finally, programs are not merely abstract, but have causal effects on the physical world. Today, proving that a program is correct remains an inspiring challenge for those who share their basic conception of programs with the mathematical culture of programming and a deluded illusion for others.

The case of the sel4 microkernel and the unmanned AH-6 helicopter, mentioned in the opening dialogue, shows some of the challenges. The control software is too large to be formally verified as a whole. The authors instead focus on showing the correctness of some of the key components and giving a formal guarantee that untrustworthy parts cannot interact in unexpected ways. Most of the proofs are also too large for a human to check. Instead, they are checked by another computer program, a proof assistant. There also remains a large number of assumptions, both about the hardware and about some of the source code (such as 1,200 lines of startup code in the kernel).[28]

The practical difficulties explain why the Algol language is a better paradigmatic example for the mathematical culture of programming than, for example, the formally verified sel4 microkernel that controls the AH-6 helicopter. Another reason is that it is reasonably possible to see Algol as the product of one particular culture of programming. It would be hard to make such a claim about the sel4 microkernel. Although it partially fulfils a dream of the mathematical culture, its development required a non-trivial number of hacker programming tricks and a huge and well-managed engineering effort.

1.4 The Hands-on Imperative

Whereas the mathematical culture keeps a distance from the technical details of program execution, purportedly to work with a more basic and universal notion, the hacker culture seeks the exact opposite. Direct engagement with the machine and the nitty-gritty details of program execution is the cornerstone of the hacker way of thinking about programming. This way of thinking may appear more natural, historically. The first programmers of the first digital computers were often also their creators and so direct engagement with the computer was a necessity at first. But as we will see in Chapter 4, the machine soon started to disappear from the picture. In the 1950s, most computers were operating in the batch-processing mode where programmers submitted their jobs on punched cards to an operator and then had to wait for several hours to get their results back. The hacker way of thinking about programming had to be reinvented in the early 1960s at MIT.

At MIT, computers have been studied since before the digital era. We may speculate that the early hands-on experience contributed to the birth of the hacker culture in the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

1960s. Some 30 years earlier, Vannevar Bush created his differential analyser, which was a mechanical analog computer for solving differential equations. During World War II, MIT was the home of the Radiation Laboratory that brought together numerous engineers and scientists to work on the radar technologies. Military applications were also behind the Whirlwind computer, built at MIT at the turn of the 1950s, which was one of the first digital computers that operated on real-time input. Finally, in an effort to prototype the follow-up to Whirlwind, MIT engineers built TX-0, which was an experimental machine utilising the new transistor and magnetic core memory technologies, completed in 1955.

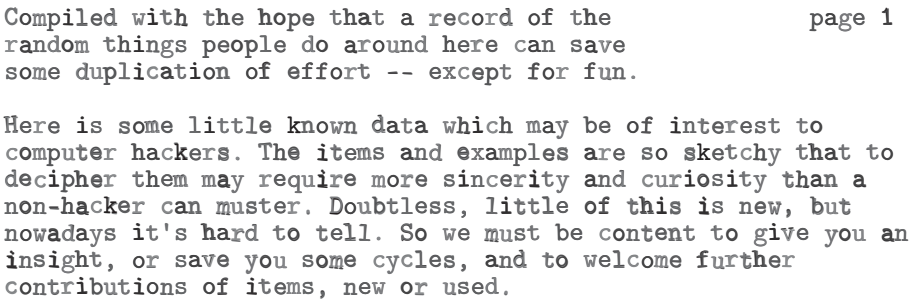
As with the mathematical culture of programming, the reinvention of the hacker culture in the 1960s relied on a combination of social and technological developments. The TX-0, along with its commercial offspring PDP-1, made it possible to directly enter instructions and immediately observe the results, but the two machines also became accessible to the MIT community under a very liberal access policy. Together, these two characteristics made the machines appealing to a community of tinkerers, many of whom came from MIT's Tech Model Railroad Club (TMRC), who started calling themselves 'hackers' and began exploring the possibilities of the machines.

The MIT hacker culture that emerged from this configuration has become a part of computing history folklore and many accounts ignore the broader social counter-cultural context of the hacker culture.[29] In particular, they fail to question the contrast between the masculine reality of the hacker culture and the alleged hacker principle that hackers should be judged merely by their skill and not 'bogus criteria such as degrees, age, race, sex, or position'.[30] Nevertheless, the hacker ethic, documented as part of the folklore, does a good enough job of explaining the way of thinking about programming and the programming practices followed by the hacker culture.

First of all, the hackers believe in direct access to the computer, which gives them the opportunity to understand and improve things. They are keen to use computers for playful and not immediately useful purposes, such as the development of the famous Spacewar! game, which I return to in Chapter 3. Hackers do not care about getting degrees, professional recognition or publications. Instead, they want to be regarded as wizards by their fellow hackers. Hackers at MIT believed that all information should be free and started using the ARPANET (a predecessor of the Internet) for sharing knowledge as soon as it became possible. The hacker approach to knowledge is best illustrated by the introduction to the MIT technical report known as HAKMEM,[31] shown in Figure 1.2, which collects 191 assorted items of knowledge ranging from the variance of a pseudo-Gaussian distributed random variable to an electrical circuit for an amplifier ('submitted without further explanation or cautions').

The MIT hackers shared many of their beliefs about programming with a broader community of similarly minded programmers who did not necessarily call themselves hackers but had the same interest in direct engagement with computers, improving systems and knowledge sharing.[32] Outside of the liberal ninth floor at the MIT Tech Square building where TX-0 and PDP-1 were housed, it was not always possible to follow all the principles of the hacker ethic. The best example of this is the group of programmers at Bell Labs that created the UNIX operating system and the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691


Compiled with the hope that a record of the page 1
random things people do around here can save
some duplication of effort -- except for fun.

Here is some little known data which may be of interest to
computer hackers. The items and examples are so sketchy that to
decipher them may require more sincerity and curiosity than a
non-hacker can muster. Doubtless, little of this is new, but
nowadays it's hard to tell. So we must be content to give you an
insight, or save you some cycles, and to welcome further
contributions of items, new or used.

Figure 1.2 An introduction from HAKMEM, published as AI Memo 239 in 1972. (Source: MIT)[33]

C programming language at the end of the 1960s. UNIX and C were a product of what I call the hacker culture. They were tools created by hackers for other hackers and provided direct access to the machine. Although UNIX was created in a commercial research environment, its authors saw themselves as 'rebels against soulless corporate empires'[34] and it was effectively distributed as free software. The MIT hackers and UNIX hackers fused into a single community, partly thanks to the sharing of software and partly thanks to the ARPANET. When AT&T started distributing UNIX as a commercial product in the early 1980s, many MIT hackers could not accept the new commercial nature of UNIX, leading to the development of the GNU project and the birth of the free software movement in the 1980s.[35]

Perhaps because of its disregard for professional recognition and focus on free sharing of information, the specific contributions and influences of the hacker culture are much harder to follow than those of the mathematical culture with its academic publications and citations. Nevertheless, the hacker way of thinking about programming reappears in multiple forms and in multiple places in the history of programming. Another notable example is the early microcomputer community on the West coast, which was, at least initially, not directly connected to the MIT hackers, and whose work I will discuss in Chapter 3.

1.5 Software Development Lifecycles

Both the mathematical culture and the hacker culture of programming emerged from the university environment, but universities were not the only users of computers. In the 1940s and 1950s, computers were used first for military purposes and later for business applications. Many of those systems were large-scale and required a highly organised and coordinated development effort. The difficulty of programming was a surprise to both individual programmers and to managers.

A prime example that illustrates the challenges of programming is the software for the Semi-Automatic Ground Environment (SAGE), an air defence system consisting of a network of radar stations, connected to a network of computers that was designed

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691



Figure 1.3 Situation display console of the AN/FSQ-7 computer, which was a part of SAGE. (Source: Computer History Museum, Photo by Joi Ito,[36] CC-BY 2.0)

to detect a Soviet bomber formation approaching the USA to perform a nuclear strike (Figure 1.3). The designers of SAGE were focused on the physics and electrical engineering aspects of the project and did not see programming as a major challenge.[37] When SAGE was conceived, the designers imagined the programming task could be specified and then assigned to a contractor, just like the production of conventional electronic components. It soon turned out that the development of the SAGE control system was a task of unprecedented complexity. The initial estimated number of 27 programmers that the project would require first grew to 200 and soon afterwards to 2,000 – at a time when the total estimated number of skilled programmers in the USA was just 200.[38]

Large-scale military and business programming projects, such as SAGE, gave rise to yet another way of thinking about programming that I refer to as the managerial culture of programming. In this culture, programming is seen as a production-like activity. Consequently, the managerial way of thinking places emphasis on specification, planning, organisational structures and the division of labour. The context from which this culture emerged differs from the hacker and mathematical culture both in the environment and its way of working, but also in the kinds of project that it tackled. However, as the case of UNIX (and later Linux) shows, production of large-scale software is by no means limited to the managerial culture.

The SAGE system was eventually built and became operational in 1959. It used the Whirlwind computer built at MIT that I mentioned when discussing the hacker culture and MIT was also initially involved in the development, before establishing the MITRE spin-off for more production-oriented work.[39] Similarly, the RAND Corporation that was initially responsible for the software development established a spin-off company, SDC, which trained over 7,000 programmers and defined one of the first methodologies for managing the development of large-scale computer

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

systems. As discussed in Chapter 4, the development was organised in phases, starting from operational plan and machine specifications; coding was preceded by coding specification that defined the interfaces between components and was followed by several testing and evaluation phases. The careful planning, typical for the managerial approach, aims to minimise the dependence on individuals. Rather than being highly individualistic hackers, programmers came to be seen as replaceable factory workers who are hired, trained and then integrated into a structured development process, where they complete small-scale programming tasks according to the given coding specifications.[40]

The development of the Apollo guidance computer software is another early example of a mission-critical software system that pioneered the idea of careful management of the development process. This was even more necessary because the software development was done at MIT rather than directly at NASA. For example, all modifications to the specification of the on-board software had to be approved by the Software Configuration Control Board and MIT 'could not change a single bit without permission'.[41] The development process itself followed the, later standard, sequence of phases. It started with the requirements gathering phase in which NASA and MIT jointly prepared the Guidance and Navigation System Operations Plan (GSOP), which specified the required functionality to the level of equations to be computed. This was followed by coding, validation that consisted of unit and integration testing and, finally, production. NASA also defined four review points that resulted in the acceptance of outcomes from individual phases.

The development of SAGE and the Apollo guidance computer serve well to illustrate the typical way of thinking within the managerial culture of programming. The projects placed emphasis on the structuring of the development process and controlling of the work done by programmers. This approach is in stark contrast with the hacker culture where code is the only thing that matters. Despite the best managerial efforts, the process was a source of concern and NASA and MIT produced 'quality software, primarily because of the small-group nature of development at MIT and the overall dedication shown by nearly everyone associated with the Apollo program'.[42] In other words, even the Apollo guidance software, which was very much a product of the managerial culture of programming, relied on some degree of hacker qualities.

In the case of SAGE, the managerial culture controlled the development process and the hacker culture played a supporting role. However, the patterns of interactions we will see later in the book are more diverse and contentious. First, the development of 'Software Engineering' that I follow in Chapter 4 can be seen as a revolt against the early hacker-dominated approach to programming. The struggle for control is explicit in the McKinsey 1968 report[43] that urges managers not to leave computer project decisions to programmers. Another typical pattern that we will encounter in the book is the case where the managerial culture adapts a previously mainly technical concept so that it can be used not just for structuring programs, but also for structuring the teams developing them. This is the case with structured programming in Chapter 2, testing in Chapter 4 and also object-oriented programming in Chapter 6.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

1.6 A Proper Engineering Discipline

During the 1950s, it became clear that the task of creating programs was much harder than initially expected. Programming was 'a black art' that relied on 'private techniques and inventions' of individual programmers.[44] The hackers had no problem with this and were happy to continue developing their own private inventions, share them with other hackers and continue exploring what can be done with computers in this way. However, outside the hacker culture, this state of the art was seen as problematic at best. The two cultures of programming that I have introduced already responded to this challenge in their specific ways. Those with a mathematical background started searching for ways to specify programs in more rigorous ways, with the hope that mathematical methods could be used to tackle the complexity of programming. Those with managerial inclinations continued coming up with new ways of organising the work, with the hope that division of labour would lessen the reliance on individual programmer skills.

By the end of the 1960s, many of those involved with the development of technically challenging systems such as programming languages and compilers, or what we would now call operating systems, started to feel that none of the existing approaches were satisfactory.[45] In 1968, a carefully selected group of attendees was invited to participate in the NATO Conference on Software Engineering. The conference showed a resounding agreement that 'the black art of programming [has] to make way for the science of software engineering',[46] but very little agreement on what exactly this means. Despite some disagreement among historians as to the significance of the event,[47] the reframing of programming as an engineering discipline paved the way for yet another means of thinking about programming. The treatment of programming as an engineering discipline is a core principle of what I refer to as the engineering culture of programming.

The engineering culture recognises the difficulty of programming. It does not aim to reduce it, to either a mathematical or an organisational problem. Instead, it aims to tackle it directly, using a combination of good engineering practices such as testing, monitoring and over-engineering. Those practices are often supported by tools that utilise the computer itself to simplify the task of programming. Whereas the proponents of the managerial culture of programming believe that good team organisation is enough to solve the problem of programming, those aligned with the engineering culture place greater responsibility on individual programmers. But unlike in the hacker culture where programmers are free to use whatever black art they can master, programmers in the engineering culture aim to use rigorous and well-documented methods and practices.

Many of the characteristics of the engineering culture of programming link it to traditional engineering. We will often encounter attempts to find a more scientific way of programming, which is in line with the usual definition of engineering as a discipline that employs scientific methods to solve practical problems. As in traditional engineering disciplines, the engineering culture also requires that programmers approach problems with a high degree of professionalism.[48] This includes the willingness to learn from all possible sources, including those produced by the other cultures of programming.[49]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Another way to identify a culture of programming is by the kinds of output it produces. For example, one of the typical outputs for the mathematical culture is a theoretic treatment in an academic publication. For the engineering culture, the three kinds of output we will often encounter in this book are programming styles, programming practices and programming tools. The first of those is rooted in the idea that good structuring of code can make programming easier. A prime example I will talk about in Chapter 2 is structured programming, popularised by Dijkstra in 1968.[50] At the time, many programmers were still writing code as a linear sequence of instructions with jumps that transfer the control to another location in the sequence. Structured programming replaced this with higher-level constructs such as loops, which execute a certain block repeatedly for a given number of times or until a condition is met. This makes programming easier because the structure of the code more directly indicates what happens when the code is executed. We will see the same general principle, structuring code so that it is cognitively easier to work with, when discussing object-oriented programming in Chapter 6.[51]

Programming processes, which is the second kind of output I linked to the engineering culture, were dominated by the managerial perspective for much of the early history of programming. The managerial interpretation reduced the idea to the problem of structuring teams. However, the engineering culture developed an alternative approach to the problem by focusing less on structuring of teams and more on structuring of the process of programming. The first discussions about this idea appeared in the 1970s,[52] but the most prominent example of the engineering approach to programming methodology is the Extreme Programming (XP) methodology that emerged at the end of the 1990s.

Extreme Programming was developed by software engineers and it treats writing code as the central activity. It introduces a range of practices that help programmers produce better code such as pair programming, where code is written by two programmers on a single machine. Another practice, which has the appeal of the scientific method, is Test-Driven Development (TDD). In this method, a program exists alongside a collection of tests that can be run automatically. When programming, programmers first write a test to specify the required functionality. The test should fail, because the functionality itself has not yet been implemented. The programmer then completes the implementation, making the test pass successfully. The scientific appeal of the method is that the collection of automated tests ensures that previously implemented functionality remains correct and so, in principle, the method should gradually converge to a desired result.

The history of testing, which I discuss in Chapter 4, is particularly interesting, because it shows how a single concept can be shared by multiple cultures of programming and used for different purposes over time. Prior to 1979, the notion of testing was seen as a step in a managerial software development process and a way of showing that a program satisfies its requirements. After 1979, the view gradually shifted and testing became an engineering approach to finding errors, which was supported by a range of testing tools. Finally, test-driven development turned testing into a driving force in an engineering-oriented development methodology. This illustrates both of

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

the interactions between cultures that are central to this book. On the one hand, there is a collaboration around a technical concept where different cultures contribute to the notion of a test. On the other hand, there is a struggle for control over the programming methodology. Following the publication of 'The Manifesto for Agile Software Development' in 2001, the light-weight methods of the engineering culture gradually replaced the heavy-weight managerial approaches as the dominant process for software development.

1.7 New Media for Thinking

The mathematical, hacker, managerial and engineering cultures would provide a sufficient enough structure for writing about many of the important practical developments in the history of programming. However, doing so would leave a regrettable gap. There is yet another culture that has played a role in shaping programming. This culture has a less coherent perspective than the others and includes a more loosely connected group of people, works and ideas. Its proponents include educators, artists, as well as computer scientists and programmers influenced by arts and humanities including philosophy and media theory. This is also the culture that has been the hardest to name. I refer to it as the humanistic culture of programming, because the concern for humans and their relationship with computers and programming is often at the core of the work done in the culture, be it work on education, systems that envision the future of computing or artwork involving programming. The humanistic culture of programming is less concerned with how to construct particular programs and cares more about what programs can be created, how humans interact with them, and how they influence the society. The humanistic culture often treats programming not as a mere tool, but as a medium for thinking or as a new kind of literacy.

An influential early paradigmatic example of this way of thinking about computers dates back to before the first digital electronic computer, the ENIAC, became operational. In 1945, *The Atlantic* published an essay 'As We May Think' by Vannevar Bush, who we encountered earlier as the creator of the differential analyser at MIT. In the essay, which was partly based on his pre-war ideas, Bush responded to the problem of information explosion with the idea for a device in which 'an individual stores all his books, records, and communications, and which is mechanised so that it may be consulted with exceeding speed and flexibility.'[53] These ideas have since been regarded as precursors of the Internet, hypertext and online encyclopedias, although Bush assumed the device would be a mechanical computer, like the differential analyser, and would store data on microfilms.

Vannevar Bush himself does not fit squarely into any of the cultures. He has contributed to a wide range of work on computing, but his essay illustrates two typical characteristics that will appear repeatedly in the humanistic culture of programming. First, it presents a vision of the future of interaction between a human and a computer and, second, it imagines a system that will assist humans with thinking and their intellectual endeavours.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Almost 20 years later, the 'As We May Think' essay became one of the inspirations for Sketchpad, a computer system created in 1963 by Ivan Sutherland. Sketchpad used a graphical user interface to let users construct geometric shapes. At the time when most computers were used in batch-processing mode and had no screen, Sketchpad offered a glimpse of a possible distant future. It was also a feat of engineering. It ran on the TX-2 machine, a more powerful successor of TX-0, and utilised the experimental light pen input device. The writing about Sketchpad exemplifies both the focus on humans and a way of thinking about programming. It was motivated by the desire to make computers 'more approachable' by using 'drawing as a novel communication medium'.[54]

The humanistic culture of programming thrives when new ways of interacting with computers become possible. Sutherland's Sketchpad was the first computer program with a graphical user interface, but it relied on using the most powerful computer available at the time with custom hardware modifications. The more broadly accessible way of interacting with computers that became available in the 1960s was through the use of a teletype terminal that enabled users to enter textual commands and receive replies from a remote computer. The new mode of interaction was soon exploited by the Logo programming language, which the authors see as a 'learning environment where children explore mathematical ideas'.[55] Logo is perhaps best known for turtle graphics, a microworld where children interactively control a graphical turtle, but the first microworld developed in Logo was text based. However, even before the graphical screens became commonplace, the authors built a physical robot that could later be controlled using a 'button box' that made Logo accessible to even younger children. As I discuss in detail in Chapter 3, Logo combines a newly developed interactive way of programming with focus on human thinking and critical reflections on education.

The most influential programming language that was significantly influenced by the humanistic culture of programming is Smalltalk. Despite becoming a general-purpose language that contributed to a wide range of programming concepts, the origins of Smalltalk are quite different from those of other early programming languages. A glimpse of the context can be found in the 'Personal Dynamic Media'[56] article written by Alan Kay and Adele Goldberg. The article describes the vision of 'a personal dynamic medium the size of a notebook', which 'could have the power to handle virtually all of its owner's information-related needs'. Smalltalk is described as a communication system and programming is referred to as 'talking to Smalltalk'. In many ways, this article reads much like the next iteration of the vision from the 'As We May Think' essay, except that Smalltalk now provided a realistic prototype. The fact that Smalltalk is now remembered more as the source of object-oriented programming, which has became largely the subject of the engineering culture of programming is another story and I return to it in Chapter 6.

Yet another revolution in how users can interact with computers happened in the 1970s when microcomputers enabled an increasing number of people to buy an inexpensive personal computer. This led to a number of more practical developments, discussed in Chapter 3, but it also enabled one novel and eccentrically creative use of programming. In the 1980s, the music band 'The Hub' brought microcomputers to

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

clubs and used programming as a musical instrument for their performances. The Hub allowed the audifence to see their screens, which became a mantra of the live coding community that would flourish later.[57] The work on education, rooted in Logo, and the work on live coded music that started with The Hub eventually connected in the form of Sonic Pi, mentioned in the opening dialogue, which is a live coding system that has been used both in live performances, but also to teach music and programming in schools.

The work arising from the humanistic culture can take a wide range of forms, but the way of thinking about programming is sufficiently coherent to justify labelling the work as belonging to a single culture. In some cases, there is a direct link that we can follow. For example, a number of individuals and systems were more or less directly influenced by essays like 'As We May Think' and other hallmark publications. However, the way of thinking also repeatedly reappears without a direct connection, for example, in the context of computer education or computer art. This is where my notion of a culture of programming departs from the ordinary understanding of the term in that I associate multiple communities with the same culture of programming even if they are not directly connected, but merely share a closely related way of thinking. The humanistic culture is also, at least to an extent, truthful to its broader humanistic goals. It often aims at greater inclusivity and has been better, at least in contrast to the other cultures, in recognising pioneering contributions of women including Cynthia Solomon and Radia Perlman working on Logo and Adele Goldberg and Diana Merry-Shapiro working on Smalltalk.[58]

1.8 The Past and the Present of Programming

The cultures of programming that I introduced in the preceding five sections capture particular ways of thinking about programming. Those ways can be traced back to the 1950s or 1960s, but the very same ways of thinking can be found in contemporary discussions about programming. The different cultures partly overlap with different communities, such as academic computer scientists or professional software engineers, but this is not the case for all cultures of programming. Moreover, individuals can move between cultures and produce work that combines approaches from multiple cultures.

Vannevar Bush, whom we encountered repeatedly in the sections above, is one example. Bush had an engineering background and his pre-war work on the differential analyser was rooted in traditional engineering. During World War II, Bush worked as an administrator and coordinated much of the US scientific military research. Despite this background, which included engineering and management roles, his later 'As We May Think' essay inspired many working in the humanistic culture of programming.[59]

The computing pioneer John McCarthy is an even more fitting example. In Chapter 2, I will discuss his 1963 paper 'Towards a Mathematical Science of Computation', which is a manifesto of the mathematical culture of programming that envisions much of the later work on formal treatment of computer programs. However, McCarthy was also a supporter of the hacker culture at MIT, affectionately referred to as 'Uncle

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

John', and hired a number of MIT hackers for various projects. He pioneered the idea of time-sharing, which made computers more interactive and enabled not just the first implementation of the Logo programming language, but also a lively hacker exploration of programming that I return to in Chapter 3. Finally, he was also one of the founders of the field of Artificial Intelligence (AI). His work on AI and other creative uses of computers, at a time when computers were mainly scientific instruments, would also connect him with the humanistic culture of programming.

The concept of a culture of programming is an interpretation obtained by looking at the past 70 years of programming. This inevitably simplifies some aspects of the complex history, but it also makes it possible to tell an overarching story about the history of programming.[60] The fact that we keep encountering the same cultures of programming throughout the 70-year history of programming makes the analysis in this book also useful for thinking about contemporary debates and issues involving programming.

One example of this was the discussion about the Knight Capital bug that I discussed in the opening of this chapter. In the next three sections, I sketch how the framing in terms of cultures of programming can shed light on three present-time programming issues, namely the questions of program errors, understanding of algorithms and programming education. There may even be ways in which cultures of programming help us imagine the future of programming. We could, for example, speculate how specific programming concepts used by one culture might evolve if they were adopted by another culture. As interesting as this may be, I refrain from such speculations and focus on the present, in the next three sections, and the past of programming, in the rest of the book.

1.9 Developing Software without Errors

At the end of the 1940s, many believed that programming errors would, along with hardware faults, become less frequent over time.[61] As the many cases of programming errors that we encountered in this chapter show this was an optimistic view and it was soon left behind. Today, many believe that program errors are inescapable. Sometimes, as in the case of Knight Capital, program errors are the source of catastrophic failures. Sometimes, this is not the case. The errors in the Apollo guidance computer software were well understood and carefully mitigated. In some cases, a program error may even be useful as a source of inspiration. Glitch art (Figure 1.4) uses errors for aesthetic purposes, while an error made in a live coding performance can be used as a source of creative ideas.[62]

Different cultures of programming understand program errors differently, but the attempt to eliminate programming errors has been the driving force behind many developments. However, different cultures do not just disagree how to best produce error-free programs. They also disagree about how to define what an error-free program is. For the proponents of the mathematical culture, programming is a mathematical activity and error-free programs are determined by a formal correctness property. This

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

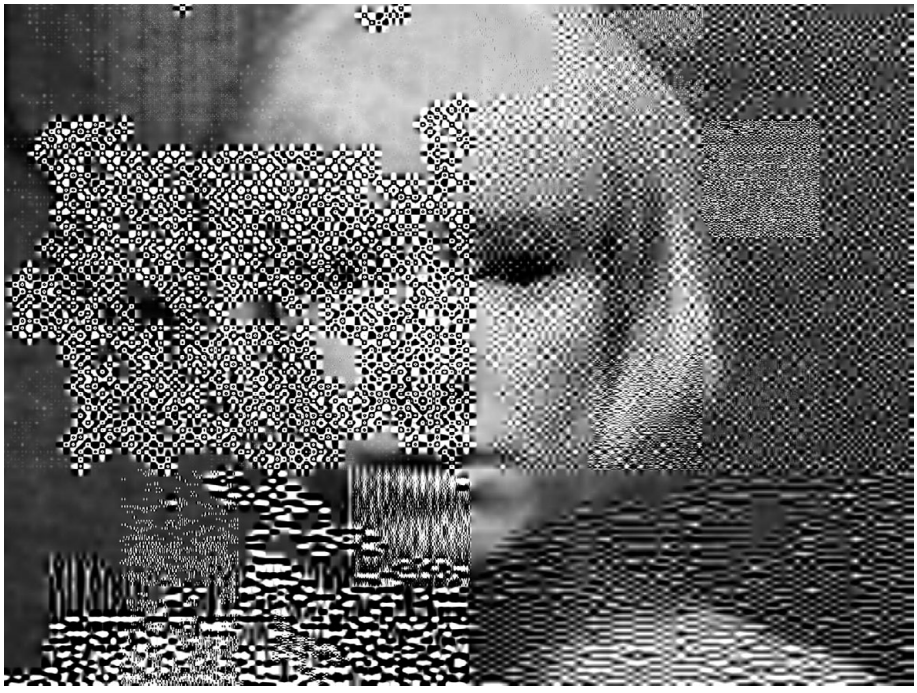


Figure 1.4 Vernacular of File Formats (2009–2010) by Rosa Menkman explore how introducing an error in a compressed file results in different visual effects. The example here is that of the JPEG2000 format. (Source: Rosa Menkman,[63] reproduced with permission)

is typically a correspondence between a program implementation and some abstract mathematical specification. The proponents of the mathematical culture are happy to accept a specification from a potential customer, but do not worry about where exactly it originates from and how. The problem of how to obtain the right specification is not a mathematical concern and is outside the scope of mathematical methods. Dijkstra, for example, refers to the problem of obtaining the right specification using the somewhat derogatory term 'pleasantness problem'[64] and suggests that it is best tackled by psychology and experimentation. Nevertheless, even this carefully constrained perspective does not avoid all problems with using mathematical methods for reasoning about programs and I will return to some of the challenges in Chapter 2.

For both the managerial and the engineering culture of programming, most programs are created for the benefit of a real-world customer. A program is then satisfactory if it meets the requirements of the customer. There is, however, a subtle difference in what exactly this means. Traditionally, meeting the requirements of the customer in the managerial culture meant producing a system according to a given specification. The specification provides the input for the carefuly managed development process that can ensure the requirements are met. A detailed specification, capturing the requirements of the desired system is also the stereotypical artifact of the managerial culture. Unlike in mathematical culture, the specification is not intended to be fully

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

formal. Obtaining the right specification is also seen as an important part of software development and so the culture views 'pleasantness' as inseparable from 'correctness'. For the engineering culture of programming, a satisfactory program should meet their needs, but the proponents of the engineering culture today are well aware that the needs of the customer may not be their initially stated requirements. And the professionalism typical for the culture requires them to work in a way that attempts to uncover and satisfy the actual needs. This difference results in different preferred ways of managing software development and I will discuss the approaches that emerge from the two cultures in more detail in Chapter 4.

Finally, to the proponents of the humanistic culture, the question of correctness is an ill-defined one. Any software has effects on the world and we need to think about those effects. The culture views programming as a social and a political act and so the effects are analysed at such levels. Correct software is one that, for example, empowers desirable social structures as in the vision of 'a truly free society' developed in 'The Telekommunist Manifesto'.[65]

The key lesson here is that the culture determines what methods can be used for studying the question of program errors. Such methods, in turn, delineate what is within the scope of 'programming' and what is outside it. The more systematic the methods, the narrower the scope and the more technical the interpretation of what a good program means. The case of the engineering culture is particularly interesting because, as part of its attempts to capture what software can reliably be built, it had to develop its own new way of reasoning. As we will see in Chapter 4, this relies on logical or even mathematical arguments involving somewhat informal but rigorous notions such as accidental complexity, rate of change in software or forgiveness of the environment.

As the debate about the Knight Capital software error reveals, the different cultures of programming can interpret the same event in multiple ways. The different interpretations are rooted in different worldviews and are, to some extent, incommensurable. Not in the sense that they cannot be comprehended, but because the solutions they lead to solve problems that other cultures do not recognise as relevant. The remarkable fact is that programming has been able to maintain this pluralism throughout most of its history.[66] The different views have coexisted and different cultures also often contributed to a single technical artifact. One example of this, which I return to in Chapter 4, is the notion of testing, which can be seen as a phase in a managerial software development process, a formal mathematical activity, as well as a tool for engineers. Despite occasional clashes and misunderstandings, the overall practice of programming seems to repeatedly benefit from this pluralistic approach.

1.10 Understanding Programs

Most software systems are very complex and opaque. When they work as desired, this is not a reason for concern. Alas, complex software systems often exhibit behaviour that is problematic. A machine translation tool that learns to translate 'Russia' as

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

'Mordor'[67] may be amusing, but a biased job applicant-screening tool,[68] racist predictive policing algorithms,[69] and search engines that discriminate against women of colour[70] highlight a serious problem with the state of software. All of these cases raise questions about our understanding of how the complex and opaque software systems that surround us operate. Although my focus in this book is primarily on technical aspects of programming, the perspective of cultures of programming can help us think about the different ways in which the working of computer programs can be understood and, in turn, shed some light on how different cultures of programming may view, or be blind to, socio-technical issues.

To the proponents of the humanistic culture, the issues with understanding technology are nothing new. They may point to the French philosopher Gilbert Simondon who, already in the 1950s, raised the issue of our relationship with technology and noted that the lack of understanding makes us passive operators of the machines (or software systems) and alienates the user (or the programmer) from the system.[71] This leads to a situation where the user is controlled by the system rather than the other way round. The proponents of the humanistic culture also firmly believe that there are no technological solutions to social problems. They may pursue a range of different ways for raising awareness of the issue, for example through creative art projects, which make the hidden implicit biases in computer systems explicit.[72] They may also pursue projects that aim, inspired by Simondon, to counter the alienation from technology, for example by supporting educational projects that increase diversity among programmers.

The hackers believe that systems should be designed in a way that 'fits a single brain'.[73] If a system can be fully understood, its behaviour can also be fully understood and so a system built with a hacker spirit should be free from hidden surprises and emerging behaviour. Even if undesirable behaviour is found, the users should have 'the freedom to study how the program works, and change it so it does [their] computing as [they] wish'.[74] This view, of course, has many problematic assumptions. Most importantly, it assumes that other people using the system are also hackers, who will want to and will have the resources necessary to understand and modify it. That may have been the case for MIT hackers in the 1960s, a community that I discuss in Chapter 3, but it is not the case for any of the problematic systems I mentioned earlier. Yet, the belief that producing systems that can be understood and modified by their users has enabled numerous developments in areas such as data privacy.[75]

The proponents of the mathematical culture would be ready to admit that the complexity of modern software has outgrown a scale that can fit a single brain. To follow the framework used by MacKenzie in his account of the mechanisation of proofs,[76] the complexity is an inevitable symptom of living in a society of high modernity. In such society, critical infrastructure consists of complex technical systems and the understanding of such systems is delegated to systems of expertise. The proponents of the mathematical culture believe that formal mathematical proofs are the most trustworthy system of expertise. As with the problem of software correctness, the issue becomes correctly defining the mathematical requirements of software systems.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

There are two other approaches to the issues of understanding software and explainability sketched in the opening dialogue. The engineering culture would recognise the earlier examples as engineering failures and would perhaps allude to codes of ethics that govern more traditional engineering disciplines. This does not, however, provide any direct answer as to how an engineer should approach the development of such systems. After a careful ethical consideration they may recognise certain AI systems as unethical and refuse to work on those. For other systems, the engineering approach may consist of being aware of the threats and building tools to mitigate potential issues. An engineer may develop a tool that attempts to detect and eliminate bias from the underlying system. They may also develop a series of tests to ensure that a system does not exhibit algorithmic bias. Finally, the managerial culture would approach the problem from a similar starting position, but would seek a more explicitly defined solution. One approach may be an industry-wide standard or a government regulation that provides guidance for using artificial intelligence algorithms.[77]

1.11 Programming Education

The last topical issue that can be illuminated by framing it in terms of cultures programming, which I want to discuss in this chapter, is the issue of programming education. It is easy to see how contentious the issue is today. A quick search on popular Q&A websites finds hundreds of questions along the lines of 'do I need a computer science degree to get a programmer job?' with hundreds of answers providing contradictory opinions.

Discussions about programming education are revealing, because they are indirectly pointing at the nature of knowledge about programming. Programming education should cover fundamental ideas of the discipline, but as the discussion about education shows, there is little agreement among the five cultures about what those fundamental ideas are. Historically, we can see this conflict in the English naming of the discipline. Those who favoured the mathematical and engineering approach preferred the term *computer science* or *informatics*, business-oriented programmers preferred to talk about *information systems* and *information science*, while many saw computing as a component of a broader field of *cybernetics*.

The first computer science degrees emerged in the 1960s at the intersection of the hacker, mathematical and engineering cultures. They directly reflected the different disciplines involved in building and programming computers at the time. Programming was taught by someone from the university computing centre (typically a service organisation hosting a computer for the use of the whole university), numerical analysis by a mathematician and switching theory and logic design by a person from the electrical engineering department. The focus varied depending on whether the course was based in the department of electrical engineering or mathematics. More business-oriented curricula for information systems appeared in the early 1970s and included development of computer systems, courses on financial and accounting systems, but also on operations theory and the social implications of computing systems.[78]

In later years, the mathematical culture grew in prominence and computer science curricula shifted emphasis to a more abstract notion of programming and algorithms,

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

while the alternative direction, focused on information systems and motivated by more engineering and managerial interests, became secondary. The humanistic culture has, perhaps surprisingly, not been involved in the design of curricula around computing, but has been pursuing its vision as part of broader educational efforts. The Logo programming language was explicitly designed for teaching, but it was not designed to teach programming. It was designed to teach 'powerful ideas' such as mathematical thinking which can be experienced and studied through the use of programming and computers.

Today, opinions on the best programming education in different cultures of programming differ as ever. A formal university education is shaped primarily by the mathematical culture of programming. A typical computer science degree aims to teach fundamental computer science knowledge and, for the mathematical culture, this entails topics such as algorithms, logic and formal languages. Core theoretical topics are typically complemented with material rooted in the engineering culture of programming such as development practices and methodologies. Those evolve more rapidly than mathematical knowledge, making it difficult to keep a curriculum up to date. The hacker culture typically contributes an odd course that involves low-level systems programming. The hacker knowledge is, however, also difficult to convey in a formal academic setting as it often takes the form of tacit knowledge learned through practice.[79] Evidence for this is the marked absence of debugging in computer science education that I revisit in Chapter 4.

The different cultures of programming thus have very different opinions on what would be the ideal model of programmer education. For the mathematical culture, mathematical theories of programming are the most basic form of knowledge and they should be taught at universities. The engineering culture values programming practices and tools which evolve more quickly. This makes non-traditional formats provided by industry, such as coding bootcamps, an appealing alternative to formal education. The hackers view formal education as even less important. If programming is a practical skill learned through practice, aspiring programmers should just start programming, learning from code and guidance of more experienced hackers. Finally, both the managerial and the humanistic culture view computing from a broader perspective and would like to see programming education positioned in a broader context, be it business studies or arts and humanities.[80]

The contributions of the mathematical culture to the discipline of programming may well have played an important role in establishing computer science as a reputable academic discipline,[81] but it also dominated what we consider as fundamental academic computer science knowledge and, in this, alienated the academic mathematical computer science from other cultures of programming. One place where the alienation is apparent is in job interviews. A common kind of interview question is about algorithms, even though this knowledge is rarely needed in a typical programming job. The situation is best understood as a cultural mismatch. Mathematical culture of programming built a solid body of fundamental knowledge. This happens to be easy to use in job interviews, even if most programming jobs require, at best, a combination of knowledge from multiple cultures of programming.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

1.12 How Cultures Meet and Clash

There is more to the history of programming than can possibly fit a single book. There have been thousands of programming languages, an even greater number of programming tools and a wide range of programming methodologies used in innumerable projects. Those are not merely technical entities, but they exist in a broader context that often involves struggles for control, intellectual disputes, biases, stereotypes and discrimination as well as numerous other technical, social, political and economic factors.

The aim of this book is to provide a perspective that sheds new light on important episodes from the history of programming, some of which have not been documented in detail before. As I hope to show in the upcoming chapters, the perspective of cultures of programming helps us make sense of developments involving a wide and diverse set of topics ranging from programming languages, object-oriented programming and types to software engineering and interactive programming. It also helps us make sense of contemporary debates about programming, not limited to those about program errors, understanding of programs and programming education discussed in this chapter.

The five cultures of programming that I identify in this book attempt to capture different basic principles, assumptions and beliefs concerning the nature and practice of programming that emerge recurringly among programmers and computer scientists. This includes different ideas about what constitutes programming knowledge and how to best acquire it, beliefs about what kind of activity programming is and also what it includes in addition to instructing the machine. The five cultures provide meaningful explanation of the history I look at, but they are inevitably simplifications and different accounts of the history of programming may need different or a more fine-grained structure.[82] Yet, I believe that recasting the history of programming as interactions between five different cultures of programming provides as good an overarching narrative for the history as possible. To use a formulation that I will suggest in Chapter 7, cultures of programming can be seen as a useful abstraction for thinking about the history of programming.

Although a reader who is a programmer or a computer scientist might recognise themselves (or their colleagues) in one of the characters in the opening dialogue in this book, an individual does not have to strictly belong to a single culture of programming and accept its assumptions unequivocally. In reality, many of those who contributed to the development of programming combine traits from multiple cultures of programming and change their views over time. Despite their different basic assumptions, proponents of the different cultures of programming do not 'live in different worlds'.[83] They can understand each other and exchange ideas, although they may not agree about the foundations behind such ideas and their importance. Furthermore, the different cultures also share a common ground in the form of program code and concrete software artifacts.[84]

The key idea that I put forward in this book is that the most interesting developments in the history of programming over the last 70 years happened when two or more

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

cultures interacted. The next five chapters provide plenty of evidence. Throughout the book, we will repeatedly encounter two kinds of interaction. On the one hand, different cultures of programming often clash about the basic principles and assumptions. The controversies around program verification (Chapter 2) arise when the cultures clash over what program correctness means. The history of interactive computing (Chapter 3) is a struggle between cultures that is centred around the way in which a human is involved in programming. Finally, in the software engineering debate (Chapter 4), the cultures disagree about the best avenue towards producing software reliably and on budget.

On the other hand, different cultures of programming often contribute to the development of a shared technical artifact. As we will see in Chapter 2, this includes the very idea of a programming language (Chapter 2). The notion of a type (Chapter 5) takes a shape when the hacker, engineering and mathematical cultures meet. The concept of testing (Chapter 4) appears as a hacker method, but is refined by the proponents of engineering and managerial cultures. Similarly, the idea of objects and object-oriented programming (Chapter 6) is first developed in mathematical and humanistic cultures, but it later evolves with an engineering focus in mind and is further adapted by the managerial culture.

In the 1950s, the existence of different cultures in computing and programming could have been explained by the fact that the field was emerging at the intersection of electrical engineering, mathematics and logic, military and business, psychology and many other disciplines. Seventy years later, the existence of the same cultures of programming is a sign that programming is and will remain an inherently pluralistic discipline. The historical episodes discussed in this book show how programming has benefited from this structure. The clashes over basic principles eventually deepen our understanding of the nature of programming. Ideas from other cultures often reinvigorate concepts that have been developed in other cultures and remained stale. The existence of multiple cultures also keeps a greater number of approaches that programmers, as a community, have ready at hand for tackling technical challenges that emerge as the field of computing evolves.[85]

Although the main focus of this book is historical, there is also a forward-looking aspect to this work. To those contemplating the future of programming, the book might point at new, yet unexplored, possibilities that can appear by viewing an existing technical idea from one culture through the perspectives of other cultures.

Notes

1 This applies to chatbots based on large language models (LLMs) such as ChatGPT, as well as earlier examples including Microsoft's Tay chatbot. As pointed out by Neff and Nagy (2016), chatbots ranging from Weizenbaum's 1966 ELIZA to Microsoft's 2016 Tay are often used in ways that their designers did not expect and their position between society and technology raises difficult questions about accountability and agency.

2 The struggle for control in programming is discussed by Ensmenger (2012) and the ways in which early programming in the UK, USA and the later emergence of 'software engineering'

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

led to masculinisation of the discipline are discussed by Hicks (2010), Ensmenger (2010) and Abbate (2012), respectively.

3 https://commons.wikimedia.org/wiki/File:Margaret_Hamilton_-_restoration.jpg.

4 In a case documented by Tomayko (1988), the onboard guidance computer and the ground control computer calculated the time for the de-orbit burn differently and the crew had to manually key in the numbers transmitted from the ground control into the Apollo onboard computer.

5 Klein et al. (2018) provides an overview of some of the verification efforts. I will return to the topic of what exactly formal verification means in these cases in Chapter 2.

6 As I discuss in Chapter 3, the term 'hacker' used here refers to a programming sub-culture documented by Levy (2010) and Tozzi (2017), rather than to security hackers who break into computer systems; a discussion of a broader context, including some of the overlaps between the two can be found in work on piracy by Johns (2010).

7 For a defence of the C programming language against the points raised by *Xenophon*, see the work of Kell (2017).

8 See Dastin (2018) for this particular case. The general issue of algorithmic bias has become a widely recognised issue and is discussed, for example, in a popular account by O'Neil (2016). An important case of how search engines reinforce racism is discussed by Noble (2018).

9 This simplistic position that 'algorithmic bias is a data problem' is indefensible, even if we take a very technical perspective on the nature of AI algorithms as shown by Hooker (2021).

10 https://dictionary.cambridge.org/dictionary/english/algorithm.

11 Kusner et al. (2017).

12 As pointed out by MacKenzie (2014), trading algorithms are not simply 'faithful delegates of human beings' but take the role of more active actors. Computer bugs such as the one discussed here then play the role of 'Heideggerian hammer' in that they force us to examine the role of a system at a more basic level.

13 Murhpy (2013).

14 As pointed out by Mahoney (2005), 'most declarations of the "computer revolution" have rested on future promises rather than on present or past performance'.

15 www.cs.utexas.edu/users/moore/acl2/workshop-2017/slides-accepted/Passmore-AI-ACL2-2017-export.pdf.

16 Mahoney (2005).

17 www.gartner.com/it-glossary/devops.

18 https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale.

19 www.sec.gov/rules/final/2010/34-63241.pdf.

20 www.sec.gov/litigation/admin/2013/34-70694.pdf.

21 Another culture that has been influential in the early days of computing gave birth to cybernetics, a multi-disciplinary study of self-regulatory systems. Despite its prominence in the 1950s and 1960s, it seems that cybernetics has had little influence on the design of programming languages as of yet.

22 The technical concepts play the role of boundary objects, as introduced by Star and Griesemer (1989). They often have concrete technical implementations that have enough shared content, but they are also flexible enough to be used and interpreted differently by different cultures.

23 I return to this insight, based on the excellent work of Nofre et al. (2014), in Chapter 2.

24 I return to the philosophical framing of the notion of cultures of programming in Chapter 7 where I also compare it to related concepts from philosophy of science, including systems of practice introduced by Chang (2012).

25 Ensmenger (2012).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

26 Priestley (2011) documents the technical developments leading from ENIAC to programming languages, whereas Ensmenger (2012) documents the social developments of computer science. Mahoney (1997) follows, more specifically, mathematical theories that contributed to computer science.
27 Perlis (1978).
28 https://sel4.systems/Info/FAQ/proof.pml.
29 Levy (2010); Raymond (1997).
30 The contrast is discussed by Adam (2005); hacking likely follows the pattern of software engineering, documented by Abbate (2012), which emerged as a more masculine redefinition of programming which remained, in its dominant office worker variant, somewhat accessible to women.
31 Beeler et al. (1972).
32 Turner (2010) tells the history of West coast cyberculture, which combines the hacker culture of MIT with the counterculture movement that emerged on the West coast.
33 Beeler et al. (1972).
34 Raymond (1997), quoted by Tozzi (2017).
35 The history of UNIX is briefly discussed in Raymond (2003) and a more detailed historical account of free software, starting from the UNIX hackers, has been written by Tozzi (2017).
36 www.flickr.com/photos/joi/494395374/.
37 As pointed out by Slayton (2013), they saw programming as more flexible than traditional physical electronics and so presumably easier. Programmability was also advertised as an advantage of the system to administrators and policymakers, disincentivising the recognition of the difficulty of programming.
38 Slayton (2013); Ensmenger (2012) talks about the 'labour crisis' in programming.
39 Redmond and Smith (2000).
40 Ensmenger (2012) views this through the perspective of struggle for control between the managers and programmers.
41 Quoted by Tomayko (1988). For a more recent detailed historical account, see Mindell (2011).
42 Tomayko (1988).
43 McKinsey (1968).
44 John Backus, inventor of FORTRAN, quoted by Ensmenger (2012).
45 As documented by Haigh (2010), one example is the perceived failure of the work on Algol 68 in the mathematical culture of programming.
46 Ensmenger (2012).
47 In particular Haigh (2010), argues that the conference is often seen as a crucial turning point by historians of computing, but its actual impact is becoming 'harder to square with the actual historical record'. Nevertheless, the proceedings of the 1968 and the follow-up 1969 conferences (Naur et al., 1969; Buxton et al., 1970) serve as a good account of thinking about programming, in a particular community, at the time.
48 There is, however, a difference between professionalism arising from the needs of software engineers and professionalism imposed from the outside. For example, many of the attempts to develop certification schemes for programmers in the 1960s documented by Ensmenger (2012) seem more aligned with the managerial culture and the same would be the case with many contemporary certifications.
49 The 'Papers We Love' movement (https://paperswelove.org is a recent example exhibiting this characteristic of the engineering culture).
50 In his famous 'Go to statement considered harmful' letter (Dijkstra, 1968), but also in a working paper on 'structured programming' that appeared in the NATO 1969 conference proceedings.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

51 The emphasis on code that programmers can understand in Dijkstra (1968) is remarkably long-lived. For example, recently published book by Seemann (2021) has the idea of 'code that fits your head' in its very title.
52 The obvious place to look for such work would be the IFIP WG 2.3 on Programming Methodology, established in 1969. However, as the report by Gries (1978a) indicates, most work in the group focused either on structuring of code and data or on topics such as program correctness that are more aligned with the mathematical culture. Two exceptions from this are Niklaus Wirth's contribution on 'Program development by stepwise refinement' and Douglas T. Ross' contribution on 'Structured analysis'.
53 Bush (1945).
54 Sutherland (1963).
55 Solomon et al. (2020).
56 Kay and Goldberg (1977).
57 The principle was captured by the slogan 'Obscurantism is dangerous. Show us your screens' formulated by Ward et al. (2004) of TOPLAP, an organisation founded in 2004 to promote live coding.
58 As documented by many, including Misa (2011); Abbate (2012); Hicks (2017), pioneering contributions made by women often remain hidden from history, so the notable fact here is not that there are women contributors, but that their contributions have been recognised. One should not be overly optimistic though as two of the four found a career in programming only after joining the respective labs as secretaries.
59 As the well-researched biography of Vannevar Bush by Zachary (2018) shows, his reservations towards humanities make Bush an unlikely contributor to the humanistic culture of programming.
60 The approach of developing an overarching grand narrative of computing history has recently been followed by Haigh and Ceruzzi (2021). The present book is less ambitious and focuses specifically on programming, but it shares the ambition of providing a comprehensible story that explains many developments throughout the history.
61 This is pointed out by Priestley (2011). At the time, 'programming' was seen as the mathematical design of the program and 'coding' as a translation of the design to machine language; the belief was that coding errors would become infrequent.
62 Blackwell and Collins (2005); for different interpretation of errors in different cultures of programming, see also my earlier paper, Petricek (2017).
63 https://beyondresolution.info/A-Vernacular-of-File-Formats.
64 Dijkstra (1993).
65 Kleiner (2010).
66 This is not unlike the case of competing interpretations of a scientific experiment. As documented by Chang (2012), in the early nineteenth century, some saw electrolysis of water as a process producing phlogisticated and dephlogisticated water, while others viewed it as a process splitting water into Hydrogen and Oxygen. In case of chemistry, however, one view eventually dominated.
67 www.rferl.org/a/27468516.html.
68 Dastin (2018).
69 O'Neil (2016).
70 Noble (2018).
71 Simondon (2016).
72 An example of this approach is the ImageNet Roulette project by Crawford and Paglen (2019), which uses an AI algorithm to assign, often problematic, categories to a person based on their uploaded photo.
73 The Mu project by Agaram (2020) is an extreme example of this approach in that it attempts to build a system that is comprehensible, starting from low-level machine code.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

74 The Free Software Foundation (2022).
75 For example, see the privacy-focused smartphone operating system /e/OS: https://e.foundation.
76 MacKenzie (2004).
77 The work by Cihon (2019) at the Oxford Future of Humanity Institute reviews ongoing work on and argues for such standards.
78 Atchison (1985) provides a review of early computer science education; an early curriculum for 'information systems' degree is proposed by Ashenhurst (1972).
79 Adopting the notion of 'tacit knowledge' as used by Polanyi (1958).
80 A good example of how such rethinking of computing from the perspective of arts and humanities may look is the recent work by Sack (2019).
81 The rise to prominence of the notion of an algorithm and its contribution to the growth of computer science is documented by Ensmenger (2012); Mahoney (1997, 1992) provides a detailed account of the history of computer science and the evolution of the conceptualisations of the discipline.
82 For example, Mahoney (1988) talks about the tripartite nature of computing consisting of electrical engineering, computer science and software engineering, which provides an account of computing as a whole. In later work, Mahoney (2005) discusses more fine-grained communities of computing including data processing, management, mathematical calculation, mathematical logic, human augmentation, artificial intelligence and computational science. Each of those communities would predominantly follow beliefs of one particular culture of programming, in the sense used in this book.
83 In the classical sense of belonging to different research paradigm as introduced by Kuhn (1962). The notion of culture of programming is perhaps closer to that of 'system of practice', developed by Chang (2012) to talk about the history of chemistry.
84 We might see concrete programming languages and tools as trading zones through which cultures can exchange ideas, even if they interpret them differently. A prime example of this is the notion of 'type' discussed in Chapter 5, which is used by many cultures, but in subtly different ways.
85 Much has been written about pluralism in physics. Galison (1997) documented the two subcultures of particle physics and pointed out that the structure makes the discipline more stable, perhaps in a similar way in which the different cultures of programming contribute to its stability over time. Chang (2012) talks about different 'systems of practice' in the early history of chemistry and uses the account as an argument for greater pluralism in science, a call that programming in many ways, but sometimes unconsciously, already follows.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

2 Mathematisation of Programming

Teacher: Although I want to focus on mathematical thinking about programming, it may be useful to start from the broader theme of professionalisation of programming. When did programming start turning into a respectable discipline?

Pythagoras: I have serious doubts whether programming is a respectable discipline even today, given that we are still unable to produce correct programs. But the turning point from which we can even think about making programming respectable is the publication of the preliminary report on the Algol programming language in 1958.

Diogenes: The proliferation of programming languages at the end of the 1950s was a gradual progression. We were slowly learning how to program better. What was so wrong with programming in the 1950s that suddenly changed with Algol?

Pythagoras: In the 1950s, programming lacked the sound body of knowledge that could support it as an intellectually respectable discipline.[1] Treating programs as mathematical entities, which was the intellectual achievement of Algol, made it possible to start building such knowledge in the form of fundamental algorithms...[2]

Archimedes: I'm not sure if algorithms are the right kind of fundamental knowledge, but I would agree that the problem with programming in the 1950s was that it was unsystematic. There were no accepted development methodologies, standard tools or processes and so programming was highly unreliable.

Xenophon: I agree about the unreliability of programming back then, except that it was not due to a lack of algorithms or development methodologies, but because of a personnel issue. The industry was growing too rapidly and there was a shortage of qualified programmers.[3]

Socrates: It is ironic that the attempts to 'professionalise programming' at the time of a labour crisis can also be linked to the exclusion of women from the workforce. Many of the professionalisation efforts tried to turn the perception of programming from low-skilled clerical work, associated with women, to high-skilled masculine work. What programmers actually did has not changed much. The perception did.

Teacher: I do not want to brush aside an issue that continues affecting programming to this day,[4] but let's get back to programming languages for a minute. Would you agree that the birth of programming languages was a notable milestone in the professionalisation of programming?

Xenophon: The interesting shift is that programming languages became entities separate from specific machines. This solved the portability problem that many businesses were facing in the 1950s. At least in principle, you could take your programs in a given programming language, compile them and run them on a new, more powerful, machine without having to start coding from scratch.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Diogenes: You all keep talking as if programming languages were some sudden new invention at the end of the 1950s, but there were many systems that preceded the better-known programming languages. There was the A-0 system by Grace Hopper and Autocode by Alick Glennie, both created in 1952. Already at the end of the 1940s, the Electronic Delay Storage Automatic Calculator (EDSAC) programmers used an 'external form' of instructions and interpretive routines, which are early tricks resembling later programming language ideas.[5]

Pythagoras: You can trace the history of programming languages even further back. In the 1930s, lambda calculus, Turing machines and Post production systems captured the essence of later functional, imperative and concurrent programming languages. The definition of Algol also went hand-in-hand with mathematical work on formal language theory by Chomsky and others.[6]

Socrates: This is a nice story computer scientists like to tell themselves. The actual historical influence of logic is doubtful,[7] but you are right about formal languages.

Archimedes: I think programming languages illustrate a point that we made in Chapter 1. It is a single concept that we all share that brings together ideas coming from different backgrounds. It combines the theory of formal languages that *Pythagoras* talked about, the implementation techniques from earlier systems mentioned by *Diogenes* and the concrete business motivation that *Xenophon* found important.

Teacher: That is a nice observation. But do we also have a shared understanding of what programming languages are?

Socrates: I like to link programming languages to human thinking. In the 1950s, an early perspective was to see programming languages and their compilers as translators from a language that a human understands to a language that a machine understands.[8]

Xenophon: This is not a very practical perspective. The first programming systems in the 1950s were also called 'automatic coding' systems and they tried to make programming easier so that it does not have to be done by skilled programmers. It makes much more sense to see programming languages as one of the approaches to tackling the labour crisis of the 1950s.[9]

Pythagoras: Sure, programming languages involve translation and, sure, they make programming practically easier. But what makes them interesting is that they enable a new mathematical thinking about programming. They make it possible to treat programs as mathematical entities and formally analyse them using the devices of formal logic.

Teacher: Let's focus on this idea of treating programs as mathematical entities for a moment. What exactly does this mean?

Pythagoras: If you treat computer programs as syntactic structures in a formal language defined by the rules of logic, then you can infer properties of programs from their texts by purely deductive methods. In other words, you can write a program and then prove that it is correct.

Diogenes: But how do you make sure that your compiler does what your mathematical model says it does? You make it sound very easy to specify what a program means, but real programming languages and compilers are enormously complicated. After all, one of the criticisms of Algol was that it was very hard to implement properly.

Pythagoras: This is something that people soon started working on. Since the 1960s, there have been several efforts to define Algol formally, so that you could prove that a compiler correctly implements the specification and, consequently, that programs have no errors. It was a visionary idea at the time, but the CompCert project,[10] created in the early 2000s, does exactly this for the C programming language and has been used commercially in mission-critical systems.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Diogenes: So, it took just 60 years to do this and only for a single programming language?

Pythagoras: The programming practice had to evolve in order to make such projects possible. We had to figure out how to write programs in a more compositional way, so that a correct program can be composed from individual correct components.

Teacher: Can we relate the ideas on better ways of organising programs to our more general discussion of professionalisation of programming?

Archimedes: Certainly! The motivation for structured programming in the 1960s was to help programmers write high-quality code by using logical structures with meaning, such as loops for control-flow and records and collections for representing data.

Diogenes: Structured programming is useful, but it is going a bit too far if you remove `GOTO` and other low-level features from a programming language, making it hard to express complex logic efficiently.

Pythagoras: I do not understand why you would want to have `GOTO` in a programming language. The Böhm–Jacopini theorem from 1966[11] clearly proved that any program containing `GOTO` can be transformed to an equivalent structured program...

Xenophon: Structured programming was valuable for a different reason. It made it possible to manage large development efforts in a top-down way. It inspired a new managerial approach to programming where a skilled programmer draws up the overall design and divides it into modules that are then assigned to junior programmers. This led to a successful development methodology adopted by IBM and others in the 1970s.[12]

Archimedes: The whole point of structured programming is to help individual programmers be more confident in the correctness of their programs. Treating structured programming as a managerial practice that turns programmers into unskilled workers goes directly against this idea.

Teacher: If you excuse me, I would like to get back to the issue of mathematical proofs of computer programs. It seems that the basic ideas necessary for this way of thinking have been around at least since the 1960s, yet proving programs correct is something that programmers rarely do today. Do you have any insights into why this is the case?

Pythagoras: This is just a matter of poor economic decisions. Proving programs correct increases the initial costs, so people do not do it. They foolishly ignore the fact that eliminating program errors would save them money in the long run...

Archimedes: I'm not sure if proofs about programs are really the same kind of thing as ordinary mathematical proofs. Unlike in mathematics, formal specifications of programs are often very long and proofs rarely reveal any deeper insights. Maybe people do not use them because they are just tedious...

Socrates: This is the objection raised by De Millo, Lipton and Perlis in 1977.[13] They pointed out that correctness of mathematical proofs is ensured by the social processes that surround them, such as mathematicians excitedly discussing new elegant proofs at the blackboard during a seminar. Proofs about computer programs are too long and boring to support similar social processes and so they cannot be trusted.

Archimedes: This is an interesting problem, but I would not throw proofs of programs out the window yet. Perhaps there is some other way of building trust in program proofs.

Xenophon: You have to realise that you are not trying to produce correct programs for some intellectual enjoyment. If a company decides it is economically worth it, it can ask programmers to write and carefully review the programs. You can set a small target error rate and adjust your development methodology to guarantee that.[14]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Pythagoras: I do not understand why you would accept 'small error rate' when the purpose of mathematical proofs is to give you an absolute certainty.

Archimedes: I thought we agreed that program proofs are not like mathematical proofs and cannot be trusted to the same degree.

Pythagoras: If you have a proof where each step is derived from the assumptions and the previous steps using an inference rule, then it is a proof. It does not matter who is talking about it and how! In fact, you can even automate proof checking and write a computer program that will check that each step of a formal proof is correct. That way, there can be absolutely no doubt about its correctness.

Socrates: Interestingly, De Millo et al. talk about automatic verifiers and they fear what they call the Titanic effect. A failure is still possible, but because of our uncritical trust in the system, it would have catastrophic consequences.

Pythagoras: Even if you have reservations towards full program verification, you should not underestimate the usefulness of small, elegant mathematical proofs about programs. If you study simple, formal models of programming languages, you may not be able to prove that the actual implementation is correct, but you will often find where the tricky corner cases are, so that you know where you need to pay special attention during the implementation.

Teacher: What I find most curious about our discussion is that mathematical knowledge appears as something that does not admit multiple interpretations, yet there are so many different ways of relating to it in the context of programming. It seems that this plurality of perspectives has sometimes been beneficial but it has also led to clashes once we started questioning the foundations of programming.

2.1 Giant Electronic Brains

On February 15, 1946, the recently built electronic general-purpose computer, ENIAC, completed a number of calculations during a public demonstration where it was unveiled in front of some 100 prominent guests. One of the example programs was a calculation of the trajectory of a shell, which it completed in 20 seconds. The same task would take female computers about 40 hours to complete using a mechanical desktop calculator, which is how the military made those calculations before.

As the ENIAC computed the trajectory, the attendees could literally see the numbers being processed on a front panel of the machine. The panel showed the contents of ENIAC accumulators using a grid of faint neon lights. For the demonstration, the inventors of ENIAC, John Mauchly and J. Presper Eckert placed halves of Ping-Pong balls over the lights and painted them with numbers each light represented. When one of the engineers turned off the lights in the room during the actual computation, the result was a captivating show of flashing lights produced by a machine that was soon nicknamed a 'giant brain' by the press.

Although the audience for the demonstration was almost exclusively male, the programmers who set up the machine to undertake the calculation were mostly female. One of them, Betty Jean Jennings (later Jean Bartik), recalled how she and a colleague, Betty Snyder, were busy making the last corrections on the day before the demonstration.[15] Last-minute debugging was apparently as common in the 1940s as it is 80 years later. According to Jennings, the machine was calculating the trajectory cor-

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

rectly, except that it did not stop when the shell hit the ground. When the programmers left around midnight to catch the last train home, the calculation was still buggy. By the morning, Betty Snyder had figured out what was wrong. The program was doing one additional iteration after testing for collision with the ground. She flipped one of the 3,000 ENIAC switches to correct the error and the calculation was ready for the presentation.

The ENIAC was initially designed and built to calculate artillery firing tables for the United States Army. But even before it was publicly unveiled in 1946, it was programmed to perform scientific simulations of neutrons passing through various materials to assist with the design of the hydrogen bomb. This classified work was going on, in parallel with firing trajectory calculations, at the time of the public demonstration of the machine.

The programming of the ENIAC was a complex process that involved novel mathematical analysis, devising computation plans, translation of such plans to ENIAC setups and, of course, all kinds of debugging. Much of the translation work was done by women (Figure 2.1), initially referred to as *operators*.[16] Many of them were recruited from a group of human computers hired earlier to calculate ballistics tables by hand, using mechanical desk calculators and a differential analyser. Women were, at least



Figure 2.1 Four women programmers holding parts of the first four US Army computers, from left to right, Patsy Simmers, Mrs. Gail Taylor, Mrs. Milly Beck, Mrs. Norma Stec. (Source: US Army/ARL Archives, Courtesy of Mike Muuss[17])

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

initially, hired for the work because of the uncertain status of the job and because of the labour shortage resulting from the ongoing war effort. To the surprise of some of the engineers and mathematicians, it soon turned out that the work of computer operators required ingenuity and posed at least as many problems as the engineering and mathematical work.

The ENIAC was, as Betty Jean Jennings eloquently put it, a 'son of a bitch to program'. In its early days, it was programmed by setting switches and plugging wires to connect individual components of the machine. Those included *accumulators*, which were used for addition, subtraction and number storage. Specialised components included *divider and square rooter*, *multiplier* for multiplication and *master programmer* that could be used to control looping and branching. The components operated in parallel and had to be synchronised by connecting cables from pulse outputs (indicating the completion of an operation) to pulse inputs (to start the next operation). This was further complicated by the fact that different components took different lengths of time to complete their work. For some, such as the divider, the time depended on the values with which they computed. Programming the ENIAC was more like constructing new elaborate scientific equipment than programming as we know it today.

The first computers also suffered from frequent mechanical or electronic failures. During the first runs of the ENIAC, the engineers were continually fixing the machine, replacing blown vacuum tubes, resolving short circuits and correcting other failures. But as the story told by Betty Jean Jennings shows, programmers of the first electronic computers also had to cope with some of the same difficulties as programmers do today. To make matters worse, early computers like ENIAC were idiosyncratic and had very limited computing power. This meant that programmers had to invent numerous clever tricks to make computers do even the most basic calculations. This included both technical inventions, to overcome machine limitations, and novel numerical analyses, to cope with the way the electronic computers calculated. At the same time, each machine was different and so required very different kinds of programming tricks. The programming tricks were rarely documented and became a part of each programmer's personal repertoire. This unwritten knowledge later contributed to their reputation for being 'geniuses and mavericks'.[18] In a later recollection, John Backus, creator of FORTRAN, said 'Programming in the 1950s was a black art, a private arcane matter.'[19]

Over time, the mechanical and electronic failures became less frequent. In the case of ENIAC, the engineers realised that vacuum tubes often blew because of power surges created when the machine was turned on. By keeping it always on, they managed to reduce the failure rate to one tube every couple of days. Many believed that programming errors would, over time, become as infrequent as hardware errors.[20] The industry needed almost a decade to fully realise that this was not going to be the case. One of the first computer pioneers who understood that programming was going to be a major concern was Maurice Wilkes, who designed and helped build the EDSAC computer at the University of Cambridge in 1949. Wilkes recalled that, while working on a programming challenge 'the realisation came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

my own programs'.[21] The importance of programming became even more apparent as computers entered the commercial sector, which started happening almost as soon as the ENIAC was built.

Following a dispute with the Moore School of Electrical Engineering about patent rights, the two ENIAC creators J. Presper Eckert and John Mauchly resigned from their university positions and formed the Eckert–Mauchly Computer Corporation. They hired three of the original ENIAC programmers and started building commercial computers designed for business applications. Their first client was the US Census Bureau, but later installations included a number of manufacturing and insurance companies, as well as other government departments.

As computers left the university, companies needed to recruit personnel to program the machines. The early programmers came from a variety of backgrounds including engineering, physics and mathematics, but the growth of the computing industry expanded the range of people who were involved with computers to include statisticians, secretaries, managers, salespeople and many others. All companies that were entering the computing industry faced the challenge of how to recognise who might have a talent for programming. One recommendation was to look for creative people who like intellectual challenge like chess players or mathematical puzzle solvers,[22] but the reality was that nobody knew what a programmer looked like.

Not only was it hard to find programmers, it was also difficult to train them. Much of the knowledge that early programmers needed to have was intuitive and local. It consisted of tricks that worked on specific machines, anecdotes and rules of thumb. Programming at the time had many of the typical characteristics of the hacker culture of programming. Hackers tend to rely on hunches that are derived from experience with a large number of past situations, but are hard to record in an explicit form. More often than not, the knowledge possessed by hackers is tacit knowledge,[23] which is acquired through practice or by working with those who already have the skills. The way hackers learn their craft is not through any formal education, but through practice and apprenticeships.

2.2 Sound Body of Knowledge

Programming as a discipline that relies on tacit knowledge of the hacker culture was good enough in the early days of digital computers when there were only a few distinctly unique machines. As the computer programming sector started growing and diversifying in the 1950s, programming increasingly became the key limiting factor. The growth of the electronic data processing industry in the early 1950s led to an imminent shortage of programmers. Throughout the 1950s, the shortage developed into an industry crisis referred to as the 'programming gap'.[24] The gap was widely reported and discussed in industry journals. It was certainly due to the novelty of the problems and the rapid growth of the industry, but the nature of the knowledge needed by the early programmers may have also played its part. Tacit knowledge that a programmer

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

in the 1940s and 1950s had to master is slow to acquire and it cannot be easily distilled into textbooks or training courses.

The ongoing industry crisis made many people involved with computers ponder how to make programming into something more than a mere practical skill learned through practice. The variety of ideas about what programming should look like served as an early example illustrating the differences between several cultures of programming.

Programmers with academic aspirations found the current state of the programming discipline lacking. One of the assessments of programming in the early 1950s came from Edsger Dijkstra, who was a theoretical physicist by training but started programming in 1952. He later recalled his dissatisfaction when he realised, some three years after he began working as a programmer, that programming lacked 'the sound body of knowledge that could support it as an intellectually respectable discipline'.[25] For those with academic interests, this was not only intellectually dissatisfying, but it also threatened to keep programming a low-status profession in academia.

In academic circles, many of those interested in computers joined the Association for Computing Machinery (ACM) after it was established in 1947. By the end of the 1950s, the members of the ACM became convinced that a more rigorous approach to programming should be modelled on applied mathematics. Although the initial motivation was practical, this focus later led to a disconnect and focus on computer science as an academic discipline as opposed to computer programming done by practitioners.

The focus point of the emerging academic discipline became the notion of an *algorithm*,[26] which was defined as a 'finite set of rules that gives a sequence of operations for solving a specific type of problem'.[27] The concept of an algorithm gave computer scientists a practical agenda with many interesting unsolved puzzles. A prime example of a work done within this newly created mathematical paradigm of computer science became the book series *The Art of Computer Programming* by Donald Knuth,[28] who set out to document the most fundamental algorithms. As of today, the first three and a half volumes of this ever-growing monograph have been published, consisting of over 3,000 pages. Similarly, *The Communications of the ACM*, a journal that was established in 1958, dedicated a large part of its content to the discussion of algorithms.

The practitioners also started calling for the professionalisation of programming, but their interpretation of what this means differed. For managers, the key concern was making programming less reliable on the personal skills of individual programmers. Programmers themselves were concerned with career paths and being recognised as skilled programmers. The topics discussed in the practically oriented computer magazine *Datamation*, launched in 1957, demonstrate all those interests. They include novel equipment for data management, articles on programming education and training, machine compatibility and management techniques. There were some early attempts to define a more practically oriented kind of programming knowledge that would form the middle ground between the managerial and mathematical conception. Professional organisations bringing together practitioners in the field of data processing started developing professional certifications such as the Certificate in Data

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Processing, which originated in 1960,[29] while the development of the SAGE system in the 1950s was supported by early programming training efforts.

The ideas discussed so far are characteristic of three cultures of programming. The programmers belonging to the *hacker culture* keep mastering their craft and inventing new tricks to make programming more manageable. Those with commercial focus that I associate with the *managerial culture* want to make programming more predictable, so that software can be built according to a specification and within a fixed budget. Those with academic interests that I refer to as the *mathematical culture* focus on mathematical analysis of algorithms, which has enabled them to establish computer science as a reputable discipline in the modern university environment that values theory higher than practice.[30]

The developments also show how the different cultures began to diverge and clash.[31] In particular, the focus on abstract algorithms distanced academic computer science from the concerns of commercial programmers, while the managerial attempts to make programming more structured conflicted with the interests of the hackers. The certification and training efforts could be seen as early work of the *engineering culture*, but it would still be some time before programmers would start embracing this perspective.

Given their divergent interests, it seems unlikely that the different cultures could all contribute to a single programming invention. Yet, this is exactly what happened and it revolves around what is quite possibly the greatest invention in programming of the 1950s.

2.3 How Technology Became Language

Programming computers by plugging wires between components, as done in the early days of the ENIAC, did not stay around for long. Most later machines were programmed through numerical codes that were entered by flipping switches and using punched cards. However, each machine had its own encoding with different numerical codes, as well as its own idiosyncratic limitations. As a result, all programs were written for just a single machine.

In the 1950s, 'programming' referred to the process of devising an algorithm, while 'coding' was the more tedious task of encoding the steps of the algorithm as machine instructions. Programmers of the early computers soon realised that they can use the computer itself to make coding easier.[32] The Short Code system, built in 1950 for the UNIVAC I computer, enabled programmers to specify programs in a more human-readable, although still numerical, code that resembled infix mathematical expressions. The computer run a separate program, Short Code interpreter, which read such instructions and performed the specified computations. A more efficient approach implemented by other systems within a few years was to create a program that would read such human-readable code and translate it to the code used by the machine, which would then be executed directly. A computer scientist will recognise these two methods as predecessors of modern interpreters and compilers. The approach

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

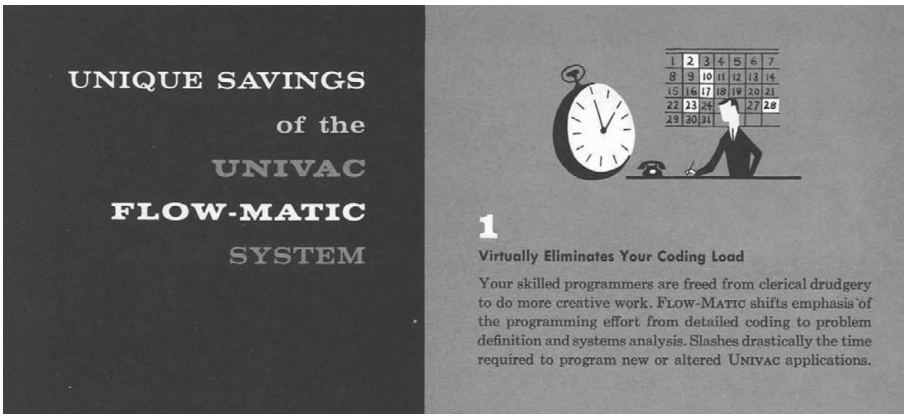


Figure 2.2 A page from a brochure 'Introducing a New Language for Automatic Programming Univac FLOW-MATIC' published by the Sperry Rand Corporation in 1957.[33]
(Source: Computer History Museum,[34] reproduced with permission of Unisys Corporation)

of writing programs using more expressive pseudo-instructions and translating those to the machine language became known as *automatic coding*. As illustrated by the FLOW-MATIC promotional brochure (Figure 2.2) it promised to 'virtually eliminate your coding load'.

A popular description of programming at the time was that the programmer had to come up with a program and then translate it into a language the machine understood. This used the anthropomorphic metaphor of computers as giant electronic brains, which was introduced by the media and was further popularised by ongoing developments in cybernetics. The metaphor made it natural to describe programming as a translation from human language into a machine language. It was not used just by popular media. A computer scientist Grace Hopper illustrated her 1954 programming course at MIT with a drawing (Figure 2.3) that describes the A-2 programming system, a predecessor to FLOW-MATIC, as a robot translating between two languages.

During the second half of the 1950s, the notations used by automatic coding systems became known as *programming languages*. This happened as the language metaphor used in the context of computers 'lost its anthropomorphic connotation and acquired a more abstract meaning, closely related to the formal languages of logic and linguistics'.[35] The FORTRAN programming language illustrates this gradual shift. The 'Specifications for the IBM Mathematical FORmula TRANslating System FORTRAN', published in 1954, did not use the term 'programming language' at that time, but it already consistently referred to the notation as the 'FORTRAN language'.[36]

The most revolutionary aspect of the development was that programming languages became standalone objects, independent of hardware that supported them, and began to be studied on their own. The FORTRAN language is, again, a witness of this development. It was first built for the IBM 704 machine in 1957, but by 1958, IBM was planning to make FORTRAN available for other machines. The early versions of FORTRAN for other machines were not fully compatible with each other, but it

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

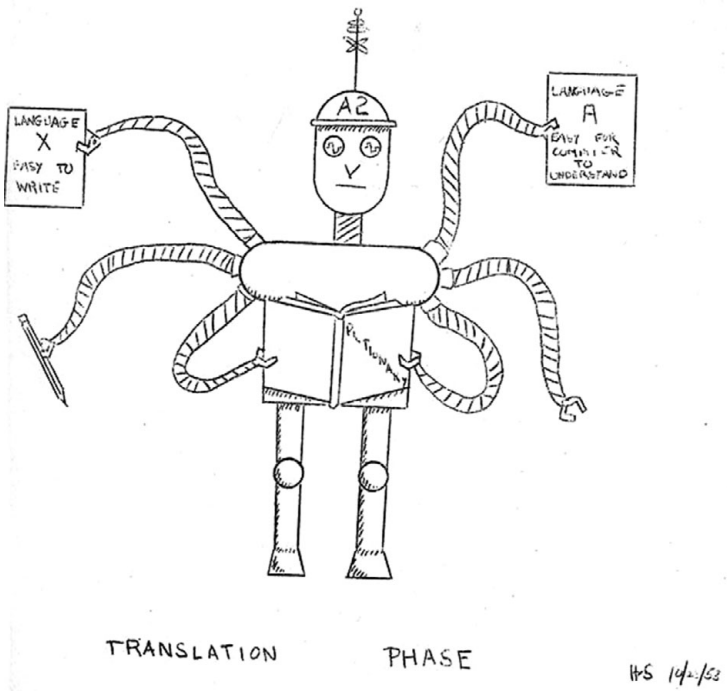


Figure 2.3 An illustration by Grace Hopper that describes the functioning of the A2 compiler for the UNIVAC computer.[37] (Source: MIT, Reproduced with permission)

became possible to think that they should be, and the managers of academic computing centres who wanted to share and reuse programs soon started requiring such machine independence.

At least three different cultures of programming played their role in the birth of the idea of a programming language. The managerial culture provided motivation for detaching programs from individual machines. A programming language makes this, at least in principle, possible and 'the early users of the term *programming language* were principally computer-user groups and computer-installation managers attempting to bring about the cross-machine compatibility of programs' who understood that 'common or universal notations would facilitate the exchange of programs among and within organizations, and would also provide a suitable vehicle for teaching programming in universities'.[38] Technically, programming languages were the next step of the development that started with automatic coding systems such as Short Code, A-2 and FLOW-MATIC. The 'black art' and arcane tricks of the hacker culture of programming were behind the implementations of the interpreters or compilers for the emerging programming languages.

The work on machine-independent programming languages was proceeding on two fronts. The largest computer-user group, SHARE, established a committee to study universal programming languages. The committee failed to make a specific language

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

proposal and some members felt this problem was a 'bucket of worms'.[39] The work was complicated by practical challenges, such as machine differences, but it may also have been hindered by the lack of suitable tools for defining a programming language.

The academically minded members of the Association for Computing Machinery (ACM) similarly established a committee in order to develop a universal language to be used in academic publications and for sharing scientific calculations. The ACM committee eventually developed the International Algorithmic Language (IAL), which was soon renamed to just Algorithmic Language, or Algol. The members of the ACM committee had various academic backgrounds, but unequivocally represented the mathematical culture of programming. This background had a strong influence on the definition of the Algol language. The language was defined using mathematical tools, such as formal grammar, that made it possible to treat programs and programming languages as mathematical objects.[40]

The managerial, hacker and mathematical cultures of programming all contributed to the idea of a programming language. They all recognise the importance of the idea, but they view it from a somewhat different perspective. To mathematicians, programming languages are abstract entities that can be formally studied; to hackers, they are tools to help them with coding and to managers, they are a way to make programs run on multiple machines. Those views are not incommensurable and many individuals span multiple cultures and combine their perspectives. The notion of a programming language plays the role of boundary object[41] in that it provides a common object of interest for multiple cultures. Although the birth of programming languages brought the different cultures closer, it did not permanently bind them together. The idea of programming languages is flexible enough to let the different cultures keep their own distinct perspectives. Three influential programming languages that were designed in 1958 and 1959 illustrate this point.

The Algol language originated with the mathematical culture. It contains a reference to the formal notion of an algorithm in its very name and emphasised the use of mathematical methods for programming. It was recognised as an 'object of stunning beauty'[42] and a remarkable achievement of computer science. Although it was never widely used for building commercial software, it became the de facto language in programming textbooks and academic publications. It was also at the core of many developments in mathematisation of programming, which I will follow in the rest of this chapter.

The language that eventually originated from the commercial efforts to create a portable language for business data processing was COBOL ('Common business-oriented language').[43] The language was born from the managerial culture of programming. Its designers came from commerce and government and did not include anyone with university affiliation. The language aimed to be easy to use, even if that made it less powerful. It aimed to broaden the range of who can use it, for example, by using English-like syntax. Once developed, COBOL was widely adopted by the data processing industry and remains in use in many legacy systems today. At the same

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

time, COBOL was criticised by academics for its poor structure and verbosity and has never been adopted and studied at universities. COBOL's emphasis on data processing meant that it had innovative language features for working with data, a topic that I will return to when discussing types in Chapter 5.

The third language appearing at the end of the 1950s was LISP, which was designed by John McCarthy at MIT.[44] The cultural origins of LISP are harder to untangle than those of Algol and COBOL. The popular story of LISP today is that it was a more or less direct implementation of the lambda calculus, a formal system developed in mathematical logic based on functions. This would place LISP firmly into the realm of the mathematical culture of programming. As pointed out by the programming historian Mark Priestley,[45] the reality is more complex. LISP was motivated by work on symbolic artificial intelligence (AI) that required sophisticated list processing. It was inspired by a range of existing practical AI projects, the FORTRAN language, as well as mathematical theories including recursive function theory and the lambda calculus. LISP certainly has mathematical origins, but it was always a pragmatic language that borrowed ideas from all possible sources in order to solve practical problems. However, LISP soon became the language of choice of the MIT hackers, possibly because it was suitable for the interactive and experimental way of programming favoured by the hacker culture, a history that I revisit in Chapter 3.

2.4 Mathematical Science of Computing

The Algol language may not have been adopted for commercial programming, but it had an enormous influence on academic computer science. The Algol report[46] is written in a combination of formal and informal language. For describing the language syntax, it uses a format that is very similar to that used in formal language theory. The semantics of the language, that is, what the individual constructs mean during execution, is specified using a very structured and precise English language description.

Although the report is formal, it also illustrates that the use of mathematical formalisms in computer science differs from their use in mathematics in that the report presents a definition rather than a proof of a theorem. Such structure would make for a very odd publication in mathematics! Yet, the semi-formal definition was enough to get computer scientists to think about programs in a mathematical way. It was also formal enough to get computer scientists to point out issues with the Algol language definition itself, leading to a revised version published three years later.[47]

The Algol report showed how to think about programs as formal entities and made it conceivable to use mathematical methods for their study. Academic computer scientists could now join the effort to formally study concrete programs, develop mathematical tools for proving their properties and mathematically analyse programming languages themselves. As pointed out by Mark Priestley, the Algol report played a crucial part in

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

transforming the vague feeling that programming should be seen as a mathematical activity into a concrete scientific research paradigm.[48] The paradigm provided the mathematical culture of programming with basic assumptions about what programs are. It identified problems that are worth studying and valid scientific methods for tackling those problems.

One of those who started exploring the possibilities of mathematical reasoning about programs was John McCarthy. In a manifesto of the Algol research program that is relevant to this day,[49] he identifies computer science as a mathematical science in which 'it is possible to deduce from the basic assumptions, the important properties of the entities treated by the science'. He sees programs as such entities and outlines a number of program properties that one can study: Are two procedures equivalent? Does one procedure take fewer steps than another? Does a translation algorithm correctly translate procedures between two programming languages? Many of the questions outlined by McCarthy are still studied by academic computer scientists 60 years later. McCarthy also hopes that the new mathematical science of computing will almost eliminate debugging and claims that, 'instead of debugging a program, one should prove that it meets its specifications'.

To give a sense of the leap from plugging cables on the ENIAC and writing idiosyncratic numerical codes to formal reasoning about Algol programs, let's go through one example discussed by McCarthy. He illustrates the various formal tools he proposes by proving that a program implements the mathematical factorial function. A factorial, written $n!$ is defined as 1 when n is 0 and $(n-1)! \times n$ when n is greater than 0. McCarthy defines a function $g(n, s)$, which computes $n! \times s$ as:

$$g(n, s) = \texttt{if}\ n = 0\ \texttt{then}\ s\ \texttt{else}\ g(n - 1, n \times s).$$

The function calculates the factorial by going down from the highest number to 0. To obtain a factorial of n, you call $g(n, 1)$. In each step, the function multiplies s by the current n and then it continues for $n - 1$ until it counts down to 0 and then it returns the current s. When calling $g(3, 1)$, the value of s at the end will be $1 \times (2 \times (3 \times 1)) = 6$.

The definition is recursive, meaning that g is defined in terms of g and it uses the conditional expression `if` introduced in the Algol report. The conditional expression evaluates to the result of one of the branches, depending on whether the condition $n = 0$ is true or false. This may not seem revolutionary, but it is quite a step from the instructions that machines actually execute. Only some 10 years earlier, EDSAC programmers had to program using memory addresses and two special accumulators instead of variables. To encode conditions, they had to use the `EnS` instruction, which checked if the value in the accumulator is positive and if so, jumped from the current location to the memory location `n` and then continued executing more instructions.

McCarthy proves that the $g(n, s)$ function actually implements mathematical factorial. As we will discuss proofs about computer programs repeatedly in the rest of this chapter, it is useful to see how this is actually done. I will show this in more detail than McCarthy, so that each step involves just one idea. We start with the obvious truth that

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

$n! \times s = n! \times s$. We refine the formula in a number of steps, making sure that we always replace it with one that means the same thing, until we get to our definition of g:

$$
\begin{aligned}
n! \times s &= n! \times s, && (1)\\
n! \times s &= \texttt{if } n = 0 \texttt{ then } n! \times s \texttt{ else } n! \times s, && (2)\\
n! \times s &= \texttt{if } n = 0 \texttt{ then } 1 \times s \texttt{ else } ((n-1)! \times n) \times s, && (3)\\
n! \times s &= \texttt{if } n = 0 \texttt{ then } s \texttt{ else } (n-1)! \times (n \times s), && (4)\\
g(n,s) &= \texttt{if } n = 0 \texttt{ then } s \texttt{ else } g(n-1, n \times s). && (5)
\end{aligned}
$$

The fact that proving the correctness is quite tedious, even for such a simple example, is partly the point of this example, so please bear with me. We start with an obvious truth (1) and then introduce a conditional expression (2). The conditional expression evaluates to either the `then` or the `else` branch, but those are both the same original expression. Next, in (3) we rewrite the expressions using facts about $n!$ In the `then` branch, we know that $n = 0$ and so $n! = 1$. In the `else` branch, we know that $n \neq 0$ and so we can replace $n!$ with $(n-1)! \times n$. We then simplify the expressions (4) and now we have our program, if we just replace $n! \times s$ with $g(n,s)$ on the left-hand side of the formula and $(n-1)! \times (n \times s)$ with $g(n-1, n \times s)$ on the right-hand side of the formula.

The example shows that one can mathematically prove the correctness of a simple program, but if we look at the articles published in the *Communications of the ACM*, a primary journal of the mathematical culture of programming throughout the 1960s, we can see that writing tedious formal proofs had not become a commonplace activity. The journal, however, adopted Algol as the language for publishing algorithms and those were written in a formally precise style. The *Communications* included a column containing a number of new algorithms each month. Between 1961 and 1973 the algorithms were numbered and reached number 472 when the column stopped being regular. This confirms the point, made by Nathan Ensmenger[50] that the notion of algorithm has became central for the mathematical culture of programming. The magazine, however, simply published useful algorithms with a comment on what they do and did not study them formally. Publications with complete formal proofs of program correctness were relatively rare. Even papers that contain proofs, such as the 1969 paper 'Prevention of System Deadlocks',[51] which presents a resource allocation algorithm that never gets into a stuck state known as deadlock, focus more on the general conditions of the mathematical model, rather than on a line-by-line analysis of the code of the algorithm.

Being aware that proving actual programs correct was very tedious, a number of computer scientists focused on finding easier and more practical methods for doing so. Most of those were centred around some way of attaching formal mathematical propositions to specific locations in a program. The idea first independently appeared as *snapshots* in the work of Danish computer scientist Peter Naur[52] and *interpretations* in the work of American Robert W. Floyd.[53] As was typical at the time, both Naur and Floyd got involved with computers through a programming job after their

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

studies of a different scientific discipline, astronomy and physics, respectively. However, the method that gained notoriety in the mathematical culture of programming was described by C. A. R. Hoare in 1969 and became known as 'Hoare triples'. Hoare studied classics and philosophy, followed by statistics, but became a programmer for a small British computer manufacturer in 1960. A year later, he started implementing a compiler for Algol and eventually became chief engineer at the company. In 1968, he moved to academia and began working on an axiomatic description of programs. Building on the work of Naur and Floyd, he proposed a method for formally reasoning about programs in 'An Axiomatic Basis for Computer Programming'.[54] Here, Hoare introduced a notation written as $\{P\}c\{Q\}$. In the formalism, c is a program instruction also called a command and P, Q are formulas of mathematical logic called pre-condition and post-condition, respectively. A Hoare triple $\{P\}c\{Q\}$ asserts that, if a pre-condition P holds before the execution of a command c, then the post-condition Q will hold after the program runs.[55]

There are two interesting aspects of the axiomatic approach introduced by Hoare. The first aspect is that, the pre-conditions and post-conditions P, Q in $\{P\}c\{Q\}$ do not need to talk about everything that is happening in the program. They can only involve aspects that are necessary for the proof and ignore anything that is irrelevant. The second aspect is that the approach is compositional. Let's say that we have two commands c and d with properties $\{P\}c\{Q\}$ and $\{Q\}d\{R\}$ where Q is the post-condition of the first command and, at the same time, the pre-condition of the second command. Now, if we write a program $c; d$ that runs the two commands in a sequence, we know that $\{P\}c; d\{R\}$ because the post-condition of the first command is the same as the pre-condition of the second command. In the first paper, Hoare used his approach to prove the correctness of a simple program that performs division with a remainder, but he later used a similar approach to prove the correctness of a somewhat more complicated program FIND that partially sorts an array of data, which was published in the Communications of the ACM.[56]

To offer a sense of proofs about programs based on Hoare triples, let's look at a different way of calculating the factorial of a number n. This time, the program is written as a loop that iterates from $i = 1$ to $i = n$ and, in each step, multiplies the result s by i:

$s = 1; i = 1;$
`while` $i < n$ `do`
  $i = i + 1;$
  $s = s \times i;$
`end`

The value s is first set to 1. It is then multiplied by $2, 3, \ldots n$ and so the final value will be $s = 1 \times 2 \times 3 \times \cdots \times n$. But how can we formally prove that this actually works? Using Hoare triples, we can focus on the key part of the program, which is the body of the loop. To make the logic easier to follow, I will write i' and s' for the new state of the variables after the body of the loop executes. It turns out that, for the two assignments in the body it holds that $\{s = i!\}i' = i + 1; s' = s \times i\{s' = i'!\}$. This

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

means that, if s is a factorial of i before the body runs then, after the two assignments are executed, s' will again be the factorial of i'. The trick is that i is incremented by 1 and s' has been multiplied accordingly. We also know that, as the program starts, it holds that $\{\}s = 1; i = 1; \{s = i!\}$. This simply tells us that, when we set s and i to 1, it will be true that $s = i!$ because $1 = 1!$ The compositionality of the method means that we can now put all these arguments together. We start with $s = i!$ and this is also the pre-condition and post-condition of the loop body, because the values s', i' will become the new values of s, i of the next iteration. This means that $s = i!$ will be true after the loop finishes looping. But then, $n = i$ and so at the very end $s = n!$ which is exactly what we wanted to show.

It may not seem like that from my examples, but the axiomatic approach did make reasoning about more complex programs easier. This is due to the two aspects that I mentioned earlier. We need to choose our axioms so that they talk only about what matters and the final proof can be composed from proofs about individual parts of programs. Together, these two aspects also enabled later developments of semi-automatic program verification tools, a topic that I will return to at the end of this chapter.

In retrospect, it is difficult to say what the expectations of the authors contributing to the mathematical culture of programming were. Hoare treats programming almost as a branch of mathematics. His hope likely was that programmers would use deductive reasoning themselves to ensure their programs are correct. McCarthy was aware that this is a lot of work and suggests that programs should be involved in checking of the proofs, 'because we can require the computer to do much more work in checking each step than a human is willing to do'.[57] This idea is closer to the way mathematical methods are used in software verification tools today. However, building such tools required both further theoretical innovations and the involvement of the engineering culture of programming in order to build tools that can be used to check non-trivial practical software systems.

2.5 Many Definitions of Algol

Early examples of rigorously applied formal reasoning to computer programs involve very simple programs or algorithms. Even the aforementioned proof of the FIND program by Hoare, which shows 'the construction of the proof of a useful, efficient, and nontrivial program'[58] studies an algorithm that has some 20 lines of code. In practice, even the proponents of the mathematical culture relied on empirical testing. The *Communications of the ACM* magazine confirms this fact. In addition to publishing the algorithms themselves, the magazine also published 'certifications' of previous algorithms, such as the one in Figure 2.4. Those were certainly not proofs. They were personal reports confirming that the reader implemented the algorithm as described previously, tested it, possibly corrected it and obtained the expected result.[59] The practice of publishing certification of algorithms bears a resemblance to the scientific practice of reproducing an experiment and publishing a reproduction study. The

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

CERTIFICATION OF ALGORITHM 58
MATRIX INVERSION (Donald Cohen, *Comm. ACM 4*, May 1961)

RICHARD A. CONGER
Yalem Computer Center, St. Louis University, St. Louis, Mo.

Invert was hand-coded in FORTRAN for the IBM 1620. The following corrections were found necessary:

The statement $a_{k,j} := a_{k,i} - b_j \times c_k$ *should be*

$$a_{k,j} := a_{k,j} - b_j \times c_k$$

The statement **go to** back *should be changed to*

$$i := z_k ; \quad z_k := z_j ; \quad z_j := i; \quad \textbf{go to} \text{ back}$$

After these corrections were made, the program was checked by inverting a 6×6 matrix and then inverting the result. The second result was equal to the original matrix within round-off.

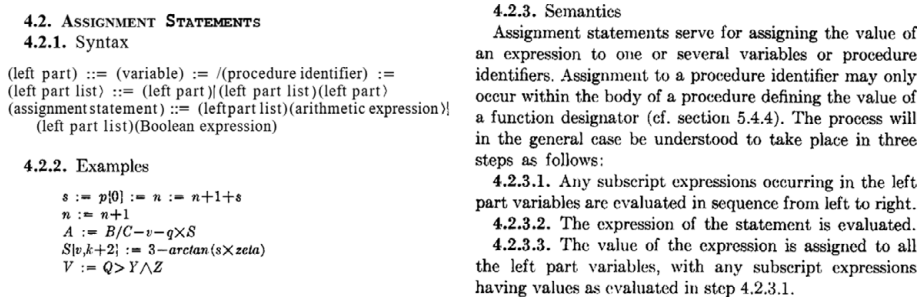
Figure 2.4 An algorithm certification published in the *Communications of the ACM*. (Source: Used with permission of ACM,[61] permission conveyed through copyright Clearance Center, Inc.)

work thus combines aspects typical for the mathematical culture with aspects that will later become typical for the engineering culture.[60]

A prime example that illustrates the struggles with applying rigorous mathematical methods to large software systems is the quest for a formal description of the Algol programming language. Algol emphasised its mathematical structure. As an object of stunning beauty for the theoreticians, it is an obvious target for formal mathematical treatment. Yet, formally defining Algol turned out to be more difficult than expected. The full story, recounted meticulously by Troy Astarte,[62] sheds light on problems that computer scientists still face 60 years later.

Algol was developed by a committee of American and European computer scientists in a meeting in 1958 in Zurich and revised in a subsequent meeting in 1960 in Paris. Figure 2.5 shows an excerpt from a canonical description of Algol, 'Revised report on the algorithmic language Algol 60',[63] which 'gives a complete defining description of the international algorithmic language Algol 60'. To a present-day computer scientist, a language specification would have to define two aspects of the language. The syntax would define the syntactic structures of the language and how they can be composed when writing a program, whereas the semantics would define how programs written using the permitted syntacitc structures evaluate, that is, what happens when a program runs. The Algol report defines the syntax formally using the Backus–Naur form, a notation technique developed for the report and still in use in modern programming language specifications. It provides numerous examples and then proceeds to describe the language semantics. This is done in English which is precise enough for a human reader, but does not define the semantics of the programming language in formal mathematical terms.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691



4.2. Assignment Statements

4.2.1. Syntax

(left part) ::= (variable) := /(procedure identifier) :=
(left part list) ::= (left part)|(left part list)(left part)
(assignment statement) ::= (left part list)(arithmetic expression)|
(left part list)(Boolean expression)

4.2.2. Examples

$s := p[0] := n := n+1+s$
$n := n+1$
$A := B/C-v-q\times S$
$S[v,k+2] := 3-arctan(s\times zeta)$
$V := Q > Y \wedge Z$

4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2. The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

Figure 2.5 An excerpt from the Revised report Algol 60 showing the syntax and semantics of assignment statments. (Source: Used with permission of ACM,[64] permission conveyed through copyright Clearance Center, Inc.)

Contrary to the expectations about mathematical publications, the report does not specify any theorems about Algol. It is merely a 17-page-long definition. This illustrates an important discrepancy between mathematics and programming. Mathematical theorems are shaped by their proofs, counter-examples and the mathematical striving for simplicity and elegance.[65] By contrast, the structure of theorems about programs is mainly shaped by the programs they talk about. The programs are, in turn, shaped by the aim of building software that can be run to achieve some task. Algol may be a carefully designed simple programming language, but as a mathematical object, it remains very complex.

The questions posed by McCarthy, which I discussed earlier, make it clear that the proponents of the mathematical culture were interested in proving properties about the Algol language and programs. To do this formally, they first needed to produce a formal definition not just of the language syntax, but also of the language semantics. An informal English language description of the semantics was not enough. McCarthy made the first step in 1964 by defining a formal semantics of a minimal subset of Algol that he called Microalgol.[66] Microalgol includes the tricky `goto` construct, but omits many other features of Algol. The semantics is defined as a recursive function $\texttt{micro}(\pi, \xi)$, which defines a single step of the program execution. Given a program π and a current state of execution ξ, the result of $\texttt{micro}(\pi, \xi)$ is a new state ξ' that describes what has changed after running a single statement of the program. The states ξ, ξ' consist of vectors of values assigned to variables together with a number representing the next statement to be executed.

McCarthy acknowledges that Algol is considerably more complicated and lists a number of issues, but believes that 'those difficulties can be resolved and that a clear description of the state of an Algol computation will clarify the problem of compiler design'. The case of Microalgol illustrates two important aspects of programming language definition. The first is the meta-language used. This is the language used to talk about the programming language. McCarthy uses normal mathematical notation with concepts like vectors and functions. The second aspect is, how is the meaning of programs defined? In the case of Microalgol, the mathematical definition works as a kind of interpreter. It uses formal mathematical methods to define how programs

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

execute and, step by step, transform a mathematical representation of the state of the computer executing the Algol program.

Hoare, who introduced the $\{P\}c\{Q\}$ notation for reasoning about programs, uses a different mathematical formalism. Rather than specifying how individual language constructs such as the assignment operator transform the state of the computer, he gives axioms about individual language constructs. For example, the Axiom of Assignment[67] states that, if predicate $p(f)$ holds for some f and the program executes an assignment $x := f$ that sets the value of the variable x to f, then a predicate $p(x)$ will hold afterwards. Using Hoare's notation, the axiom states that $\{p(f)\}x := f\{p(x)\}$. The method used by Hoare is quite different from that of McCarthy, but he relies on the same general approach of using a simple subset of the actual Algol language. He omits the tricky `goto` construct and acknowledges that this might be difficult, because the meaning of the command cannot be easily defined using the axiomatic approach. He, however, believes that other aspects of Algol will not cause any great difficulties. Hoare also shares McCarthy's overall optimism and believes that 'the practical advantages of program proving will eventually outweigh the difficulties'.

Scaling from a small subset of Algol to the full language proved to be a significant undertaking and required the development of new techniques for specifying programming language semantics. Peter Mosses, who worked on a formal definition of Algol as part of his PhD at the University of Oxford, followed a novel style developed by his PhD advisor Christopher Strachey. Rather than using mathematics to define an interpreter or provide axioms, his method assigns each statement of the language a meaning, independently of the rest of the program. The meanings are mathematical objects, specifically functions written using the lambda calculus formalism. The individual meanings can be composed to get the meaning of an entire Algol program. The 1974 definition by Mosses[68] covers almost the entire Algol language and uses a highly regular mathematical notation, which a reader with mathematical background (and sufficient dedication!) is expected to understand. The length of the document made Christopher P. Wadsworth, another PhD student from Oxford studying under Strachey's supervision, question the simplicity of Algol. In a letter to his advisor, he wrote about Mosses' definition of Algol:

> I must admit I still feel a little surprised it's as long as it is–I guess Algol 60 is just not nearly as 'well-behaved' as one tends to think it is.[69]

The case of the Algol language definition developed by Mosses reinforces the point that I made earlier about formal methods applied to programs. Despite being mathematical, they are nothing like normal mathematical texts. The report by Mosses consists of a 50-page-long definition, but no theorems. The complexity of the definition is rooted in the complexity of the programming language that it is modelling. Producing the definition is challenging not because one needs to come up with elegant mathematical tricks, but because one needs to produce a structure that can account for all the complexities of the language.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The definition by Mosses, which was published as a technical report by the Oxford University Computing Laboratory in 1974 also illustrates that, by this time, the mathematical culture of programming had established a methodology for doing normal science. The report has only a brief 1-page introduction, which states that it models Algol, as described in the revised report, except that 'own' declarations are omitted. The author does not need to explain what this means and does not need to make any attempt to convince the reader that the model is correct, for example, by comparing the behaviour of the mathematical model with programs produced by a specific Algol compiler. The mathematical culture leaves implicit the fundamental assumption that the model is relevant to actual software. As we will see, both the complexity of the definition and the unclear relationship with actual software will be later identified as problematic by those not strongly committed to the mathematical culture.

Using the ordinary language of mathematics as a meta-language for complex programming language definitions is a limiting factor. Such a mathematical definition has to be written by hand and it has to be checked by other human readers. The computer scientists working on programming languages at the Vienna IBM laboratory hoped that they could develop a compiler for a programming language systematically from its definition through a series of step-wise refinements. For this, the meta-language should not be just the language of human-written mathematics, but instead another formal language that can be processed by a computer. To do this, the Vienna IBM team created a formal language known as Vienna Definition Language (VDL) for specifying the semantics of programs.

The IBM team used the newly developed VDL meta-language to give a series of definitions of Algol. The first was a 67-page definition from 1968[70] that followed the operational style used earlier by McCarthy. Another version developed four years later,[71] aimed to simplify the first one by handling jumps like `goto` in a different way. Finally, the definition that appeared a couple of years later, in 1978,[72] adopted the denotational approach used by Mosses earlier, aiming to produce 'an equally abstract but more readable definition' that is simple enough to fit in a book chapter.

The attempts to formalise Algol illustrate the variety of work done within the mathematical culture of programming, as well as influences of other cultures. Approaches that formalise only a subset of Algol generally aim at explaining interesting aspects of the language to a human. This kind of work is close to traditional mathematics in that it is primarily focused on other human readers. Much of contemporary academic work on programming language design follows this style. A well-chosen language subset makes it easy to illustrate interesting aspects of design choices or programming language features. It also typically limits the scope to an elegant mathematical object. The obvious limitation is that such formal definition cannot be used to provide any guarantees about the full language.

Approaches that aim to formalise the full programming language depart from traditional mathematical practice. They inevitably involve long and complex definitions that are shaped by practical programming concerns rather than mathematical elegance. The methods of defining formal semantics of a full language also often involve the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

development of new software tools such as VDL built in the Vienna IBM laboratory. We might speculate that the influence of engineering culture of programming was stronger in an industrial research laboratory than at the University of Oxford where a similar definition used a more traditional mathematical notation.[73]

2.6 The Minority Report

For the mathematical culture of programming, the 1960s were a period of active development. The first stand-alone university departments of computer science started to appear, the ACM published its first curriculum recommendation in 1968 and universities started to offer undergraduate courses and degrees in computer science. Academic computer scientists were busy developing new algorithms, programming languages and mathematical tools for reasoning about them.

The efforts to formalise Algol were underway, but the language was still also under development. Soon after the publication of Algol 60, the Algol working group started discussing the requirements for the next version of the language. Proposals included more powerful data-structuring mechanisms that I return to in Chapter 5, as well as a plethora of other ideas, such as a more unified syntax, obtained by removing the distinction between statements and expressions, the ability for the language to manipulate its own programs, different mechanisms for handling I/O, as well as new ways of structuring the formalism used in the report that defined the language.[74]

In 1965, the Algol working group made a decision to adopt a proposal for a completely new design of the language, rather than one based on gradual improvements in Algol 60, which was the preferred choice earlier. The new design aimed at an even greater mathematical elegance and uniformity, but many of the working group members saw it as obscure, incomprehensible and overly complex. The design was eventually published as Algol 68, but the publication also included a 'Minority Report' written by working group members who disagreed with the final design. The dissidents felt that the language description methodology used in the project had failed.[75]

The authors of the Minority Report argued that the problems faced by programmers have changed significantly over the last decade and that programmers find themselves 'faced with tasks of completely different and still growing scope and scale'. In an insightful analysis of the Algol dissidents' thinking, computing historian Thomas Haigh pointed out[76] that the group was gradually becoming less interested in the problem of the 'expression [of programs] in some language' and more interested in the problem of 'reliable creation of programs to perform specified tasks'. In other words, they started to see 'languages as tools designed to support or enforce good practice in the design of complex programs'. This new way of thinking presents a shift from the mathematical culture of programming to the engineering culture of programming that was emerging at the time. The Algol dissidents went on to establish a working group on programming methodology (IFIP WG 2.3), which reflected this

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691



Figure 2.6 Douglas McIlroy presenting a lecture on software components at the NATO 1968 conference. (Source: Photograph courtesy of Robert McClure and Brian Randell)[78]

new focus, but was also organised in a much less formal way. A later collection of articles published by the authors[77] shows that the group kept a strong mathematical emphasis, but was gradually interested in broader software engineering issues, architectural metaphors for software development, program structuring and the activity of programming.

Many of the Algol dissidents were also involved in an event that has been widely regarded as a turning point in the history of computing. The event was the NATO Software Engineering conference, which took place in October 1968 in Garmisch, Germany (Figure 2.6). The conference is often linked with the broader industry crisis and the hopes to turn 'the black art of programming' into a 'science of software engineering',[79] although as convincingly argued by Haigh, the purported impact of the conference is hard to 'square with the actual historical record'. In reality, the legacy of the meeting has largely been shaped by the Algol dissidents, many of whom presented their work at the conference and who also served as editors of the final, highly quotable, conference report.[80]

The conference brought together some of the Algol dissidents, but also a broader computing community from both academia and industry, yet there was a heavy emphasis on those working on complex and academically interesting software systems such as compilers and operating systems. The phrase 'software engineering' was novel at the time and was deliberately chosen as being provocative. It implied the need to move, both from the question of programming languages to the question of software production, as well as the need to move from hacker practices to a development process based on theoretical and practical foundations, following the model of established branches of engineering.

Many of the participants left the Garmisch conference with a sense of excitement. Dijkstra recalled that the meeting was a success 'largely because most of the people

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

present were sufficiently high in their local hierarchies that they could afford to be honest, and were'. The common recognition that there was a software crisis was the most exciting aspect for all the participants. To capitalise on the excitement the organisers immediately planned a follow-up conference that was held a year later in Rome. The organisers hoped to move from the recognition of the software crisis to concrete proposals for addressing it. This did not turn out as well as hoped. In the words of the attendees, the conference just 'never clicked' and they 'left with an enormous sensation of disillusionment'.[81]

The two conferences brought together different cultures of programming. This did not prove to be an issue in 1968. The conference was focused on what is wrong with current methods and everyone was able to contribute from their own perspective. There was enough common understanding among the participants to accept points made by others, even if they felt those were not the most fundamental issues. The aim of the follow-up event was to discuss specific issues in detail and, possibly, come to agreements. This proved to be more difficult. As the editors of the second conference report acknowledge 'a lack of communication between different sections of the participants became a dominant feature'[82] of the second conference. Thinking about different cultures of programming offers a useful perspective for understanding this lack of communication.

The organisers wanted to focus on technical rather than managerial problems. By doing so, they already excluded the issues that mattered to the managerial culture of programming, which saw structuring of work and team management as the primary way towards more satisfactory results. Even among the more technical participants, there were widely different perspectives. The discussion about proofs of program correctness was of interest only to the mathematical culture of programming. Time-sharing systems, which I will discuss in Chapter 6, were seen as a powerful new approach by the hackers, but as a tool that teaches sloppy habits by more mathematically inclined participants. The Algol dissidents and like-minded attendees, who were starting to establish the engineering culture of programming, were primarily interested in the development of new ways of structuring programs and in practical tools to simplify programming, but not in management practices or purely mathematical puzzles. The case that perhaps best demonstrates the disagreements between the different cultures is that of structured programming.

2.7 Go to Statement Considered Harmful

The development of new proof techniques, like the axiomatic method based on Hoare triples, came hand in hand with the effort to write programs in a way that makes reasoning about them easier. This included both formal reasoning, but also reasoning in the engineering sense of being able to convince oneself that the program behaves as one intends. The primary construct in Algol that made such reasoning difficult was the `goto` statement. This is a low-level command that is close to how the machine operates and is similar to the `EnS` instruction from EDSAC that I talked about earlier.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

It unconditionally transfers the control, that is, the position in the program that is currently being executed, to another location that is defined by the argument passed to `goto`. To see the construct in action, we can compare our earlier factorial program with a version written using `goto` instead of `while`:

```
s = 1; i = 1;                    s = 1; i = 1;
while i < n do          L1       if i = n then goto L2
  i = i + 1;                     i = i + 1;
  s = s × i;                     s = s × i;
end                              goto L1;
print(s);               L2 :     print(s);
```

The two programs work in exactly the same way. The only difference is that the program on the left uses the `while` statement, whereas the version on the right implements the logic of the looping using `goto`. When $i = n$, the program on the right jumps to the end of the program to print the result. Otherwise it continues to run the sequence of instructions to update the values of i and s and then jumps back to the label $L1$ to again check if it is done.

Let's now consider how the use of `goto` complicates reasoning about programs. In the version on the left, the code has a structure that makes it easier to see that the block nested inside the `while` block will run repeatedly in a loop. If we want to reason about the code formally, we know that we need to understand the pre-conditions and post-conditions of the body of the loop and then compose those with what we know about the code before and after the loop. In the version on the right, the program is just a flat sequence of instructions. This breaks compositionality, because you cannot analyse the behaviour of `goto` $L1$ without looking at the rest of the program and locating the instructions at location $L1$.

A program structured using `while` is easier to study formally, but it is also easier to understand for a human reader. While the mathematicians were primarily concerned with formal reasoning, the readability of programs started to be of importance to the emerging engineering culture of programming that emphasised good practices in program design. Meanwhile, hackers may oppose constructs such as `while`, because the compiler needed to do more work in producing machine instructions and hackers thought it would produce code that is not as efficient as when using hand-written jumps.

Despite some opposition, there was an agreement among the proponents of the mathematical and engineering cultures that writing code in a structured way was preferable. The most vocal advocate of avoiding `goto` became Edsger Dijkstra, who made this position clear in his letter 'Go to statement considered harmful'.[83] Dijkstra explains the issues with reasoning about programs that include `goto` and observes, with his typical eloquence, that 'the quality of programmers is a decreasing function of the density of `goto` statements in the programs they produce'. Following the publication of the letter, Dijkstra also introduced the term *structured programming* for the style of programming that avoids `goto` in a working paper presented at the NATO 1969 Software Engineering conference in Rome.[84]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Dijkstra's letter sparked a debate in the computer science community. Most of those joining the debate accepted that `goto` should not be overused. Some argued that new programming language should not include it at all, while others gave various arguments in favour of it and examples where it is useful. Some also pointed out that any program can be rewritten without the use of `goto`. The canonical reference for this claim, which became known as the *Structured Programming Theorem* became the 1966 paper by Böhm and Jacopini[85] that Dijkstra cited to make this point. The style of the paper is very technical and it uses a somewhat different formulation of the problem and has been 'apparently more often cited than read in detail'.[86] Nevertheless, the theorem was a useful weapon for the proponents of structured programming. As later recalled by one of those who referred to it:

> Us converts waved this interesting bit of news under the noses of the unreconstructed assembly-language programmers who kept trotting forth twisty bits of logic and saying, 'I betcha can't structure this.'[87]

The quote is worth reading carefully, because it gives a number of hints about different cultures of programming. The reference to *assembly-language* programmers suggests that those who opposed structured programming preferred more direct access to the machine, a trait of the hacker culture of programming. Hackers were also concerned about efficiency, which was another argument in favour of `goto` in the debate. It is also interesting to see that a purely theoretical paper, which discusses normalisation procedure for flow diagrams, provided such valuable argument to the proponents of structured programming. The paper has no explicit mention of `goto` or, indeed, programming languages. Even Dijkstra quotes it cautiously, saying that it 'seems to have proved the superfluousness of `goto`'. He also adds that the mechanical translation used in the paper is 'not to be recommended', because it would produce a result that is likely much less readable than the original unstructured code. Yet, theoretical results are deemed as a prime form of knowledge in the mathematical culture of programming and so the result served as a strong argument.[88]

Regardless of the debate, the structured programming approach was soon widely known and adopted, not just by those members of the mathematical culture of programming who were interested in proofs about programs, but also by the broader programming community. As we will see next, the idea also captured the imagination of programming managers, who were less concerned with reasoning about programs and more with control over complexity, budgets and workforce.[89]

2.8 Chief Programmer Teams

Edsger Dijkstra, who introduced the term *structured programming* cannot be pigeonholed into a single culture of programming. He was one of the Algol dissidents and was hoping to find a sound body of knowledge for the discipline of programming. His work was rooted in mathematical methods, yet he was not using mathematics to solve mathematical puzzles, but to improve applied programming. His writing on structured

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

programming illustrates many aspects of what I refer to as the engineering culture of programming. Structured programming is presented as a way that helps the engineer, with their inherent limitations, to do their job better. There is also a strong emphasis on individual responsibility. One can almost imagine that using structured programming when possible would be a requirement in the Code of Ethics that all software engineers should adhere to. This engineering perspective on structured programming remains well aligned with the mathematical view that I discussed above. Structured programming simplifies formal reasoning, which is what mathematicians care about, but it also supports informal human reasoning that an engineer needs to undertake.

However, the idea of structured programming also influenced the management of software projects. Managers saw structured programming as a systematic approach where you divide the overall problem into smaller sub-problems that can be solved independently by less skilled programmers. This was appealing to the managers in the 1970s as the top-down software development approach resembled the hierarchical top-down structure of large corporations at the time.[90]

The success story of the management approach to structured programming was an information retrieval system developed by IBM for the *New York Times*.[91] According to its developers, the system was completed several times faster than comparable systems and ran with almost no errors. The programming followed a top-down methodology inspired by structured programming that the authors called Chief Programmer Team (CPT). In CPT, a single 'chief programmer' was responsible for the overall system design and was supported by a group of assistants who were assigned simpler programming tasks that did not require a full understanding of the system. The chief programmer designs the top-level program structure and directs a team of supporting programmers who solve specific well-defined problems as required by the chief programmer. The hierarchical structure in the program provided by structured programming is thus reflected in the structure of the development team, an idea that would take a new turn in the 2020s.[92]

The originator of the term *structured programming* Edsger Dijkstra never accepted the managerial interpretation of the idea. He was horrified that programming started to turn from an intellectual discipline to an industrial activity with the prevailing American 'management philosophy aiming at making companies as independent as possible of the competence of their employees'.[93] According to Dijkstra, managerial software engineering methods aim for an anti-intellectualism characterised by 'How to program if you cannot'.[94] Despite the disputes, the different takes on the idea often appeared together. We can see this in the December 1973 issue of the *Datamation* magazine,[95] which was dedicated to structured programming. It included the engineering account of `GOTO`-less programming, an article on the Chief Programmer Team methodology, as well as an overview of the structured programming theorem and a light-hearted article proposing to replace the universally hated `GOTO` construct with a 'revolutionary' `COMEFROM` feature.[96]

The concept of structured programming and the disagreements that it provoked illustrate a common pattern of interactions between cultures of programming. Structured programming undeniably benefited multiple different cultures. To the engineering

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

culture, it offered a useful tool for structuring programs so that they are easier to read. It supported compositional reasoning about programs that was appealing to the mathematical culture. The managerial culture used structured programming as a mechanism for structuring the workforce, rather than programs. Yet, as illustrated by Dijkstra's remarks, the proponents of different cultures held conflicting perspectives on the concepts they shared. Those perspectives were not incommensurable, but they were rooted in conflicting assumptions and led to contentious debates.

The cases of programming languages and structured programming are both instructive. Programming languages appeared thanks to a productive meeting of managerial, hacker and mathematical cultures of programming. Programming languages serve as boundary objects that the different cultures could share, but that is interpreted differently by each of them. Programming languages also enable further communication and exchange of ideas between different cultures.[97] This is exactly what happened with structured programming. It appeared as a way of writing code using loops and other structured constructs instead of `goto`. This shaped mathematical formalisations of Algol which sometimes omitted `goto`, seeing it as inelegant and an unnecessary complication. Structured programming then also led to the managerial idea of structuring teams to match the structure of code.

In the case of programming languages and structured programming, the collaboration between the different cultures of programming was enabled by having a shared technical concept, which can be implemented and has an existence of its own.[98] When dealing with a more abstract idea, direct collaboration is harder. The existence of multiple cultures of programming is beneficial in that, when a particular development reaches a dead end, another culture of programming may adopt an idea and find a different way forward. In a way, this is what happened in response to the difficulties with manually writing proofs of program correctness at the end of the 1970s.

2.9 Program Proofs and Social Processes

The opening keynote speaker at the 1968 NATO Software Engineering conference was a prominent computer scientist, Alan Perlis, who had received the Turing Award in 1966 'for his influence in the area of advanced programming techniques and compiler construction'.[99] Perlis was a mathematician by training, although with a clear interest in computers. His PhD thesis from 1950 describes an algorithm for solving integral equations. It does so from a mathematical perspective, but with computing applications in mind.

Despite his mathematical background, Perlis' keynote speech and comments at the NATO conference made it clear that he did not see programs as just another mathematical entity. In the opening keynote, he says that the conference is concerned with objects which 'depend in only very weak ways on the laws of physics' and whose 'structure depends as much on the social laws governing their usage as on their internal constraints'.[100]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Perlis also attended the follow-up 1969 conference and participated in a discussion on software correctness. The conference report contrasts Hoare's conviction that programmers should convince themseves that programs work by inductive mathematical proofs with Perlis' scpeticism about the method and a belief that suitably selected test cases are often sufficient. In response, the report editors counterposed Dijkstra's famous remark that 'testing shows the presence, not the absence of bugs'.[101]

Some five years later, Perlis was sitting in a seminar at Yale University. The speaker was Richard De Millo, who was visiting his collaborator Richard Lipton. De Millo was presenting his research on verifying programs and included an extensive proof of one of the algorithms. Perlis raised his hand and asked 'Why does it take 20 pages to prove a program that is obviously correct?'[102]

The question pointed to a fundamental difference between proofs of mathematical theorems and proofs of computer programs – a difference that De Millo and Lipton, also trained mathematicians working as computer scientists, soon recognised. The question prompted a collaboration that resulted in a 1977 paper 'Social Processes and Proofs of Theorems and Programs',[103] which triggered a controversy in the computer science circles. In the paper, the three authors argue that program verification is bound to fail because of the nature of the proofs that it demands.

What a mathematical proof is may appear simple at first glance. In logic, proofs are finite sequences of propositions, each of which is an axiom, an assumption or follows from earlier propositions by a rule of inference. In practice, proofs are much richer. They use a mix of natural and formal language and they are written to convince another, adequately qualified, colleague. Mathematical proofs are central to mathematical practice. They are taught at universities, shared with colleagues and discussed in whiteboard sessions where mathematicians marvel at the elegance of a proof and clever tricks that make it work. If they trust a proof, they believe that it could, in principle, be written as a sequence of propositions, but they never bother writing it in such formal way.

In programming, proofs are bound to be secondary. Mathematical proofs are a part of an interlinked network of concepts and can be reused to prove other theorems. Mathematical theorems are written to be proved, whereas computer programs are written to be executed. A proof about a program exists merely as a certification that a program matches its specification and it does not play a role as part of a broader network of concepts. Most proofs of program correctness are also quite dull. They typically have a very repetitive structure. Rather than using ingenious tricks, they enumerate a large number of fairly obvious cases.

The point made by De Millo, Lipton and Perlis is that mathematicians gain confidence about mathematical proofs thanks to the rich informal social processes of actual mathematics. Since no comparable social processes can take place among those who work on proving programs correct, formal mathematical approach to programming can never actually eliminate program errors. Although the authors focused merely on the confidence that we can have in the correctness of programs, De Millo was also aware of the idea by philosopher of science, Imre Lakatos,[104] that mathematical knowledge is built through a process of proofs and refutations. The authors knew that if computer

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

scientists treat proofs merely as certifications, they lose an invaluable force that shapes knowledge in mathematics.

Although De Millo, Lipton and Perlis were all mathematicians by training and inclination, they did not uncritically accept the assumption of the mathematical culture of programming that programs are mathematical entities like any other. They believed that programs can be studied using mathematical methods, but accepted that their structure is sufficiently different from ordinary mathematical theorems. Their paper was first presented at the ACM Symposium on Principles of Programming Languages (POPL) conference and a revised version was later published in *Communications of the ACM*.[105] The responses to the two publications show that questioning formal proofs about programs touches on a topic that different cultures treat very differently. Many of the positive letters published in a follow-up issue of the *Communications of the ACM* came from readers working in industry. One reader thanked authors for their wisdom, fairness, style, rigour, and wit while another reader thanked the editors for giving him the first article he enjoyed since joining the ACM.[106]

De Millo later reflected[107] that the three of them likely became spokespeople for computer industry practitioners who felt that academics were pushing solutions that were not going to work for them. To one manager, the article explained why he 'should not look for development of formal proof of the "correctness" of our programs' while another reader argued that he 'cannot recall a single instance in which a proof of a program's correctness would have been useful'. He argued that actual bugs are easy to discover through testing; more serious errors are errors in specification that would not be caught by proofs. As a better approach, the reader points to a review of software engineering techniques, including the Chief Programmer Team approach.

Given their background, De Millo, Lipton and Perlis cannot be clearly associated with a single culture of programming. All three of them have strong mathematical backgrounds, but their view of programs and programming seems more aligned with one that I associate with the engineering culture of programming. They recognise the irreducible complexity of computer programs as well as fundamental human limitations, both of which are typically ignored by the mathematical culture of programming. They also contributed to or advocated for the use of testing in programming: Perlis in his 1968 keynote and De Millo in his subsequent research on mutation testing. Their critical paper was certainly appreciated by those favouring the emerging engineering perspective, but also those in favour of more rigid managerial method.

The proponents of the mathematical culture produced multiple responses that criticised De Millo, Lipton and Perlis. In a letter to the editor, Leslie Lamport pointed out that the recent work of Floyd, discussed earlier in this chapter, 'taught us that a program is a mathematical object' and likens computer scientists before Floyd to 'geometers before Euclid'. Lamport believes that a 'theorem either can or cannot be derived from a set of axioms' and that their correctness is not decided by a social process. In the letter, he also criticises the practice of *Communications of the ACM* of publishing algorithms without correctness proofs in the 'Algorithms' column and notes that the published 'Certifications' provide only a flawed social process. Edsger Dijkstra went even further and labelled the article 'A political pamphlet from the Middle Ages'.[108] Dijkstra does

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

not disagree that communication with colleagues is an essential component of the mathematical culture, but he argues that the authors 'give a complete caricature of program verification' and that how to prove program properties more elegantly is the subject of lively interchange in the field. Dijkstra's response thus does not defend a purely formal view of proofs as that of Lamport. His perspective is more appreciative of the human side, but he believes that human programmers should aspire to mathematical rigour and elegance.

Many historians have written about the debate triggered by the De Millo, Lipton and Perlis paper and tried to assess its significance. Donald MacKenzie[109] suggested that it likely contributed to the decrease of US government funding for formal verification, but computer scientists committed to the mathematical culture who responded to the paper typically disregarded the arguments it made. Yet, looking at the follow-up developments, it seems that the paper captured the issue that the mathematical culture was, perhaps only unconsciously and rather reluctantly, aware of. If the mathematical culture insisted on using conventional mathematical proofs, it seems likely that it would soon reach a dead end. Real-world software systems were simply too complex for this.

I suggest that this is a case where the pluralism of programming with its multiple cultures came to the rescue. Rather than stagnating and eventually falling out of interest, the work on program verification received new impetuses, coming from two other cultures. The resulting two developments were not a direct response to the critique, but they both address the lack of social processes surrounding program proofs. The managerial Cleanroom methodology explicitly creates a new social process, while the engineering work on mechanised proofs replaces the need for human checking with machine checking.

2.10 Computing and Cultures of Proving

The first forays into proofs of program correctness, actioned in the mathematical culture in the 1960s, focused on simple programs and algorithms. This early work understood proofs as largely formal proofs that are actioned and checked by humans. The two examples involving the factorial function, which I discussed earlier, illustrate the approach. They were both done 'by hand' and consisted of a sequence of fairly direct applications of axioms and derivation rules. However, other ways of thinking about proofs about computer programs started to appear in the 1970s and gained prominence in the 1980s.[110]

Some of those working in the industry were aware of the developments arising from the mathematical culture of programming and attempted to adapt them to assist with large scale software development. We already saw one example. The Chief Programmer Team (CPT) methodology adapted the idea of structured programming, shifting focus from structuring programs to structuring teams. The methodology was developed by Harlan Mills, who had a background in mathematics and worked as a research fellow at IBM. With a PhD in game theory, but an industry job, Mills was perhaps the best person to import ideas from the mathematical culture into the world of applied

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

software development. Indeed, Mills soon realised that CPT only addressed one part of the problem. The quality of the resulting software still depended primarily on the skills of the chief programmer. Making sure that a software system or a software component correctly implements its specification thus became an even more important issue. As Mills later put it, clearly showing his appreciation for mathematical methods, 'software engineering without mathematical verification is no more than a buzzword'.[111]

To address the challenge of quality control, Mills proposed an extension of the CPT methodology that he called Cleanroom Software Development.[112] The approach took the mathematical idea of proving programs correct, but adapted it to the managerial context. It also took further an analogy with the surgical team that Mills used when designing the Chief Programmer Team structure. Like the software development team in CPT, a surgical team has a single lead surgeon, supported by a group of assistants. As the 'Cleanroom' name suggests, the new methodology adds a controlled environment, akin to the operating theatre used by surgeons. In the Cleanroom methodology, systems are built by following a detailed specification. A programmer is required to review the conformance to the specification rigorously, first to convince themselves and then to convince other team members. 'It is obvious' is an acceptable proof if everyone agrees. The methodology departs from the earlier understanding of proofs in the mathematical culture in that it admits rigorous but informal proofs, but it shares the property that the process is fundamentally human. The required degree of rigour is, however, a managerial choice and to ensure correctness of the final product, the Cleanroom methodology employs statistical quality control, measuring 'reliability over a probability distribution of usage scenarios'.

The development of the Cleanroom methodology was not a reaction to the De Millo, Lipton and Perlis critique, but it inadvertently addresses the issue that the critique raises, that is, the lack of social processes required for checking the correctness of a proof. Convincing colleagues that a software component matches its specification in the Cleanroom methodology may be as tedious as constructing a formal mathematical proof about non-trivial algorithm. However, the social structure needed to make the proof trustworthy is recreated through business organisation. The enlightened managers who choose to follow the Cleanroom method recognise that the time spent while reviewing tedious conformance to the specification will save time and money later and will require their employees to rigorously do so.

A different answer to the problem of social processes and program proofs was suggested long before the critique by De Millo, Lipton and Perlis. In the 1963 manifesto of the Algol research programme, John McCarthy wrote that 'one should prove that [a program] meets its specifications, and this proof should be checked by a computer program'.[113] McCarthy himself published a paper on computer programs for checking mathematical proofs and such proof checkers later became known as mechanical theorem provers.[114] The word prover might be misleading. A mechanical theorem prover is not expected to derive a proof automatically. This was, and remains to be, an unrealistic expectation. It is merely used to check a proof written by a human in a formal and often exasperatingly detailed way. In other words, 'a mechanical theorem prover does not act as an oracle that certifies bewildering arguments for inscrutable reasons but

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

as an implacable sceptic that insists on all assumptions being stated and all claims justified'.[115]

In the 1950s and 1960s, however, many believed that computers would be able to do more than to just check proofs written by humans. The optimism about Artificial Intelligence fuelled belief that computers would soon be able to prove complex theorems automatically. The earliest work in this area focused on classic mathematical theorems. The Logic Theorist, created by Allen Newell and Herbert Simon in the 1950s,[116] used heuristics to automatically derive mathematical proofs and they were able to prove 38 of the first 52 theorems in Chapter 2 of the *Principia Mathematica*. The work on provers for mathematical theorems prepared the ground for research on theorem provers working with proofs about programs. In 1971, Robert S. Boyer and J. Strother Moore in Edinburgh developed a system later known as the Boyer–Moore theorem prover, which was designed to automatically prove properties of programs in a variant of LISP. This prover was not able to complete non-trivial proofs without a stroke of luck or detailed human guidance, but it was a pioneering system that inspired follow-up research. This includes the ACL2 prover, which was developed in the 1990s as an 'industry strength'[117] version of the Boyer–Moore prover. I mentioned ACL2 in Chapter 1, because a conference on this system featured a talk about verifying financial systems with a reference to the Knight-Capital bug.

Another influential theorem prover designed for checking proofs about programs in the early 1970s was the LCF prover introduced by Robin Milner. The LCF was motivated by the aim to prove the correctness of programs such as compilers.[118] To simplify construction of LCF proofs, Milner's group developed a meta-language (ML), which could be used to write procedures that construct proofs. The idea is that humans should not have to write all steps of the proof. They can, instead, write programs that generate the steps. The programs to generate proofs are, in turn, checked to be correct using types. I will explain how exactly this is done when discussing types in Chapter 5, but the powerful combination gave rise to a family of interactive theorem provers that are still popular today. The ML language soon took a new form as a stand-alone programming language that influenced modern languages including OCaml and F#.

Systems that would automatically prove the correctness of a given program still remain elusive today, but work on theorem provers and other program verification tools remains an active field. On the one hand, semi-automatic mechanical theorem provers are able to do more work automatically. On the other hand, fully automatic tools are able to detect an increasing number of potential bugs. In both cases, the developments rely as much on theoretical innovations as on large-scale engineering efforts. A crucial theoretical development that enabled numerous modern verification tools is separation logic, which appeared in the early 2000s and extended Hoare triples with a mechanism for reasoning about data structures.[119] Practical tools that implement the idea, such as the Infer Static Analyser developed at Facebook, are complex systems. For example, the authors of Infer describe it as a 'general analysis framework which is an interface to the modular analysis engine which can be used by other kinds of program analyses' with instantiations 'for security, concurrency and in other domains'.[120]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The critique of proofs about programs by De Millo, Lipton and Perlis came at an interesting point in the history of the mathematical culture of programming. It received a heated response, perhaps because it questioned the central assumption of the culture that computer programs are mathematical entities like any other. At the same time, many were soon about to realise that formal proofs about programs done by hand were not going to scale up to prove interesting properties of large software systems. Donald MacKenzie uses the term cultures of proving to characterise the different approaches to program proofs.[121] The early mathematical proofs about programs were manual and formal. The Cleanroom methodology utilises proofs that are manual and rigorous, but informal. Finally, theorem provers result in proofs that are formal, but done semi-automatically. As also pointed out by MacKenzie, the different cultures of proving 'mostly coexist peacefully', although they may sometimes clash about 'presuppositions and preferences'.

The two developments discussed in this section are interesting, because they can be also seen as the results of interaction between different cultures of programming. The Cleanroom methodology takes the mathematical idea of rigorous program analysis. It keeps the mathematical rigour, leaves out the full formality, but surrounds it with a managerial team structure. The fact that it was developed by a trained mathematician working in industry is also telling. By contrast, the typical approach of the engineering culture is to develop a new tool or a method to assist programmers. Mechanised theorem provers can be seen as the result of a meeting between the mathematical and engineering culture of programming.

Mechanised proofs also quietly transformed what is considered as a proof. Mechanised proofs are no longer the complex human constructs that mathematicians know as proofs. They are computer representations of sequences of propositions where the prover can derive each proposition from earlier sentences, axioms or assumptions. This notion is close to the notion of proof in logic, with the caveat that we are now talking about computer representations. The mathematical culture of programming gradually adopted mechanised proofs as the hallmark of mathematical computer science research. In doing so, it began diverging from its pure mathematical origins where many mathematicians remain cautious about the use of computers in proofs.[122]

The lack of social process and the complexity of definitions were not the only challenges for proofs of programs that the mathematical culture of programming had to face. Another difference between proofs about programs and proofs in mathematics led to an equally heated debate in the mathematical culture of programming a decade later.

2.11 Fundamental Limits of Program Proofs

By the end of the 1980s, the techniques and tools of the mathematical culture became powerful enough to prove properties of non-trivial systems. This included both computer programs and hardware devices that were increasingly also modelled formally. By this point, the use of mechanical theorem provers became commonplace. What may

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

still have looked like an influence of the engineering culture a decade earlier became fully integrated into the mathematical culture. A subsequent version of the Boyer–Moore theorem prover mentioned above was, for example, used for verifying properties of a small multitasking operating system kernel, a low-level assembly language and aspects of a microcoded CPU.[123]

The adoption of mechanical theorem provers and their use for proving properties of software and even hardware with direct real world effects did not trigger an immediate reflection on the nature of program proofs. Many still saw a program as a mathematical entity and a program proof as a formal mathematical proof like any other. Yet, some practitioners started to realise that there are limits to what one can prove about a program that runs on a real computer and interacts with its environment. A large verification effort that illustrates this was the Software Implemented Fault Tolerance (SIFT) system, which was a flight control system designed for critical functions, such as the 'control of a dynamically unstable aircraft, designed for high energy efficiency'.[124] Boyer and Moore, who worked on the verification of a small but critical part of the system, pointed out that a proof could only be completed if they also had a precise specification of how the underlying microprocessor executes instructions and also the timing of such execution. However, the timing of execution is a property of the real hardware that a typical mathematical model of a microprocessor is likely to ignore. In some cases, such details may not even be deterministic. The timing can, for example, depend on the temperature. The issue is that programs are not just mathematical entities. They are also executed on real hardware.

A similar critical reflection was written by Brian Cantwell Smith, who worked at Xerox PARC. I will discuss his work on programming systems in Chapter 3, but Smith was also a founder of Computer Professionals for Social Responsibility, an organisation opposing the use of computers in warfare. Smith[125] starts with a story of the American Ballistic Missile Early-Warning System (Figure 2.7) that mistook the rising moon for a ballistic missile attack in 1960 and asks whether such a system, if it was formally verified, should be trusted with launching a counter-attack. According to Smith, the answer is no, because 'there are inherent limitations to what can be proved about computers and computer programs'. The core of his argument is that computer programs always work with a model of the real world, which inevitably ignores some aspects of reality. Computer programs 'are not, as some theoreticians seem to suppose, pure mathematical abstractions, living in a pure detached heaven'. They may ignore aspects of the real world, such as timing or the moon, and they trigger actions in the real world that escape the abstractions of the model.

Despite such limitations, Smith still finds formal verification valuable, just for a different reason. A proof of correctness is really a proof of the compatibility between two formal objects: a program and a specification. This is still useful, because a program and a specification are typically expressed in a different way. For example, a specification for a refrigerator will say that it is a device which maintains an internal temperature between 3 and 6 degrees. A program controlling the refrigerator has to specify how exactly this is done. Proving that a program satisfies a specification is

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691



Figure 2.7 Command post for NORAD (North American Aerospace Defense Command) operations, including the Command's surveillance and warning sensors around the globe, taken about 1982. (Source: National Archives, Still Pictures Branch, RG 306-PSE, box 79)

thus a useful added assurance, even if it cannot guarantee anything about an actual refrigerator.

The observation by Boyer and Moore was published as part of a 600-page long interim report submitted to NASA,[126] which was the project funder, while the critical reflection by Smith was initially published as a technical report. Consequently, neither of the two triggered a heated debate in the community. The controversy came in a later paper by a philosopher James Fetzer, which was published by the *Communications of the ACM*.[127] Fetzer became aware of the critique of program proofs by De Millo, Lipton and Perlis. He thought that the lack of social processes was not a major issue, but believed that there is a more fundamental problem, the one that Boyer, Moore and Smith all alluded to earlier. Although both programs and theorems about them are syntactic structures, a program is not a pure mathematical entity. The meaning of a program is that it controls a physical machine. Consequently, as Fetzer argued in more philosophical terms, the very idea of program verification is nonsensical, because it confuses mathematical a priori knowledge with empirical a posteriori knowledge.

According to Fetzer, program verification can mean two things depending on how we treat the nature of the machine that executes the program. If we see the machine as an abstract entity then program verification is a legitimate mathematical activity, but the proof is about a mathematical model, rather than the actual running programs. If we see the machine as an empirical description of a physical system, then we obtain

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

possibly useful, more or less reliable, empirical knowledge about a program, but not a mathematical proof.

Fetzer's critique provoked a fierce response from the verification community. In a letter to the editors of *Communications of the ACM*,[128] 10 prominent computer scientists claimed that his paper 'is not a serious scientific analysis of the nature of verification' and accused the editors of *Communications of the ACM* for 'abrogat[ing] their responsibility, to both the ACM membership and to the public at large' by publishing 'the ill-informed, irresponsible, and dangerous article'. The response does not, in fact, object to the main point raised by Fetzer. It argues that Fetzer attacks a parody of formal verification when he assumes that 'the purpose of program verification is to provide an absolute guarantee of correctness with respect to the execution of a program on computer hardware' and the authors give a number of references 'in which the limits of proofs are carefully drawn'.

One of the references cited in the letter was a current technical report by Avra Cohn, who was very well aware of this issue. Cohn worked at the University of Cambridge and was a part of a team formally verifying the VIPER microprocessor. VIPER was developed at the Royal Signals and Radar Establishment, a research establishment within the UK Ministry of Defence. It was advertised as 'the first commercially available microprocessor with both a formal specification and a proof that the chip conforms to it'.[129] To Cohn, this advertisement was dangerously misleading. In her reflections on the work, she carefully points out that 'a device can be described in a formal way, and the description verified; but …there is no way to assure the accuracy of the description. Indeed, any description is bound to be inaccurate in some respects, since it cannot be hoped to mirror an entire physical situation'.[130] In other words, even if we formally verify a system at all its levels, ranging from the low-level chip to high-level programming language, we are still only working with abstract models. By no means does this make formal verification useless. It just means that it should not have a unique status among other methods for building correct systems.

As with the critique by De Millo, Lipton and Perlis, the paper by Fetzer identified an issue that was, perhaps reluctantly or subconsciously recognised by at least some members of the mathematical culture of programming. According to one commenter who, nevertheless, accuses Fetzer of doing 'a disservice to the cause of the advancement of the science of programming',[131] the idea that no deductive argument can guarantee that a system will work perfectly is a fact obvious to practically everyone.

The likely reason for the raging controversy caused by the critique is that Fetzer questioned the idea that programs are mathematical entities, an assumption that is at the core of the mathematical culture of programming. In the author's response published in a later issue of the *Communications of the ACM*, Fetzer tries to support his argument that the limitations are not obvious to practically everyone in the community. He does this by quoting a number of key papers in the field, such as Hoare, who states that 'programming is an exact science in that all the properties of a program and all the consequences of executing it can, in principle, be found out from the text of the program itself by means of purely deductive reasoning'.[132] As Fetzer points out, such phrasing

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

is clearly in contrast with the idea that limitations of formal verification are obvious to most computer scientists.

I believe that there are two possible explanations of this discrepancy and that both are a part of the truth. First, the ideas about program proofs in the mathematical culture have evolved over the 20 years since Hoare's pioneering work, quoted by Fetzer, appeared. Mechanical theorem provers that draw from the engineering culture did not just provide new tools, but they also imported a more engineering-oriented thinking into the mathematical culture of programming. Second, the idea that programs are mathematical entities was and remains an accepted, but idealised assumption. It provides justification for the formal approach, but does not shape how it is practically done. As such, it is more a rhetorical device that is likely often used without elaborate consideration.

2.12 Proofs for Machines and Proofs for Humans

The two contentious debates that I just discussed share a number of characteristics. In both cases, they point to an issue that is related to core assumptions about what programs are. They also make explicit an issue that those working within the mathematical culture do not publicly acknowledge and, perhaps, are not even consciously aware of. At the same time, both critiques came at a time when the mathematical culture was already transforming how it works. In both cases, the transformation was adopting approaches from another culture of programming. In the case of De Millo, Lipton and Perlis, two ways of replacing the usual social processes of mathematical proofs appeared. The short-lived Cleanroom methodology created an explicit social process through a managerial structure. The now widely accepted engineering-inspired approach based on mechanised theorem proving replaces human checking with machine checking.

In the case of Fetzer, the complications with proving properties of software that controls physical systems were already becoming clear. A few observers started writing about those problems before Fetzer, although in a more reserved way. At the same time, in the 1980s, two subcultures were already appearing within the mathematical culture. Those who wanted to produce as complete proofs as possible treated the problem of program proofs increasingly as an engineering issue. Those who wanted to gain pure mathematical insights treated the problem as a formal one. The latter required working with minimal, formally tractable models and it became the cornerstone of programming language theory, whereas the former became a mainstream approach to software verification.

The first development started with the use of mechanical theorem provers, which are essential for ensuring that all cases in a long and tedious proof about complex systems are covered. A proof constructed using a mechanical theorem prover is really a part of the program. Just like ordinary mathematical proofs are written to convince another human that it would be, in principle, possible to construct a finite sequence of propositions representing the formal proof, a mechanised proof is essentially a program

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

that instructs a machine to construct a computer representation of such a sequence of propositions. Over time, the specification of a correctness property and its proof displaced the program itself as the main object that needs to be created. In modern theorem provers, the programmer only needs to write their proofs and the executable code of a corresponding program is generated based on the source code of the proof. A good recent example using this method is the formally verified CompCert compiler,[133] where a single program represents both the compiler and its correctness proof. In such systems, the proof becomes a technical artifact that is clearly distinct from a formal deductive mathematical proof.

The second development remained more faithful to traditional mathematics. It uses severely simplified mathematical models of programming languages or hardware devices. It is clear that proofs about such models do not provide guarantees about the reality. However, it makes it possible to reason about formal mathematical programs using standard mathematical methods, such as a hand-written proof on a blackboard. This way of working does not guarantee correctness of an actual system that is modelled, but the model may reveal tricky corner-cases that are equally tricky for the real implementation. A failure to prove correctness often points to an issue that also exists in the actual system.

The idea of using severely simplified mathematical models in proofs about programs and programming languages appeared well before the 1980s. It dates back to McCarthy's 1964 Microalgol, which is 'a very small subset of Algol' where 'all the difficult aspects of Algol are eliminated'.[134] Although the work on small mathematical models is rooted in the same mathematical culture as the work resulting in complex mechanised proofs, the language and presentation used in talking about them has gradually became more distinct, possibly highlighting the fact that the object of study is a mathematical object rather than an actual program or a programming language.

The development of two subcultures with two kinds of proof was not a response to Fetzer's critique. It was clearly already happening when the critique was published. Yet, it shows that the issue at the heart of Fetzer's critique is something that has been shaping the mathematical culture. Again, the developments arguably benefited from the pluralism of the field of programming. One of the two directions was influenced by the engineering culture, whereas the other adopted a more purist mathematical approach.

2.13 Mathematisation of Programming

To a contemporary computer scientist, the general idea of using mathematical methods for the analysis of programs and their correctness seems unproblematic. Their only lament is usually that the use of mathematical methods in software industry is not more commonplace. Many academic papers and grant applications that focus on developing new mathematical theories of programs and new tools motivate the work by claiming that only rigorous mathematical approach to programming can guarantee that software will work correctly. History shows that this is a simplistic view. On the one hand, the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

view faces many limitations and the very nature of mathematical methods changes in response. On the other hand, mathematical perspective often reshapes programming in ways that go well beyond checking program correctness.

Despite its many early links to mathematics, programming started as a craft learned through practice. The search for better ways of programming meant different things to different people. Hackers were looking for better tools to control the machine, managers were looking for better ways of organising software production and mathematicians started to think about programming in terms of algorithms. The three cultures came together in the late 1950s and collectively established the idea of a programming language as an independent entity.

The birth of the idea of a programming language did not lead to a single unified way of thinking about programming. Programming was a pluralistic discipline before the 1950s and remains a pluralistic discipline today. The different cultures that I identify in this book each evolve, but they keep a surprisingly stable set of core assumptions and methods. For example, the hacker culture that we encountered in the early days of programming emphasises thorough understanding of and direct engagement with computers. The culture did not disappear once programming languages and methodologies inspired by structured programming gave programming a more rigid structure. As we will see in Chapter 3, the hacker culture quickly came back to prominence once computers became more interactive and it was again possible to work with a machine directly and acquire craft-like skills.

The story of the mathematisation of programming is centred around the mathematical culture of programming and its interactions with other cultures. Those often provide a useful impetus for moving forward in a new direction. The birth of programming languages was motivated by managerial needs and relied on previous hacker knowledge. Yet, it gave the mathematical culture a new way of looking at programs and made it possible to treat them as formal entities written in a formal language. The development of structured programming was motivated by the aim of writing better programs and is perhaps best aligned with the engineering culture, but it was also closely connected with the development of compositional reasoning methods in the mathematical culture. The idea also inspired new ways of managing the software development process.

Another kind of interaction occurs when one culture borrows general ideas or methods from another culture. The two times this happened in the story told here, the borrowing took place at a time of controversial debates, sparked by the questioning of basic assumptions of the mathematical culture. When De Millo, Lipton and Perlis pointed out that proofs of programs were different from mathematical proofs and lacked the necessary social processes, the mathematical culture was already in the process of adopting mechanical theorem provers that ready to replace human checking with machine checking. I suggest that this move resulted from the borrowing of methods from the engineering culture. The fact that the developments happened at the same time as the controversy may be just coincidence, but it is also possible that both were a reaction to problems in the mathematical culture that were slowly emerging on the horizon. Traditional mathematical ways of writing proofs worked for small programs, but it would not work for more complex software. This triggered reflection in the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

community and it also motivated its members to look for ideas outside of their own culture.

The history of the mathematisation of programming and the controversies that we encountered in this chapter has been told in detail in a number of excellent accounts that I have been drawing on in this chapter.[135] What is new in my retelling of the story is the focus on pluralism within the discipline. If we look at how concepts like programming language, structured programming or mechanised proofs came to be, we can see that this is a product of rich interactions between different groups of individuals who have different assumptions and different methods of working.[136] Although different cultures of programming do not share assumptions and use different methods, they do not form incommensurable bodies of knowledge. They can exchange ideas through individuals who sometimes pass between multiple cultures, but they can also exchange ideas through shared technical concepts such as programming languages, programming constructs and programming practices.

Notes

1 Reflecting on programming in the 1950s, Dijkstra (1972) asks '[Where] was the sound body of knowledge that could support it as an intellectually respectable discipline?'
2 The establishment of algorithm as a basic concept of computer science is documented by Ceruzzi (1988) and further discussed by Ensmenger (2012), while Mahoney (1992) discusses the search for the mathematical foundations of computing.
3 Ensmenger (2012) provides a detailed account of the 1950s labour crisis.
4 The roots and effects of gender biases in computing have been documented in various contexts by Hicks (2010, 2017), Ensmenger (2010) and Abbate (2012).
5 The external form allowed programmers to use relative addressing (Wilkes et al., 1951), while interpretive routines made it possible to use pseudo-instructions, e.g., for working with floating point numbers (Campbell-Kelly, 1980).
6 Mahoney (1997).
7 See the work of De Mol and Bullynck (2018) for critical reflection on how the historical story of computer science has been constructed.
8 We return to the media-theoretic perspective in Chapter 3. The perspective seeing compilation as a translation process is documented by Nofre et al. (2014).
9 Ensmenger (2012); Priestley (2011).
10 Leroy (2009).
11 The theorem refers to the result by Böhm and Jacopini (1966). As noted by Harel (1980), the result in the paper is of a more technical nature and has been misinterpreted in the same way as done here by *Pythagoras*.
12 MacKenzie (2004).
13 See DeMillo et al. (1977). A detailed account of the history of program proofs can be found in the work of MacKenzie (2004).
14 We return to this topic when discussing the 'Cleanroom' methodology later in this chapter.
15 The history of ENIAC programming has been documented by historians, especially Haigh et al. (2016), and also by Bartik (2013) herself. There are disputes about some aspects of the history of the early ENIAC programming work. I generally follow the first-hand account of Jean Jennings Bartik, but Haigh et al. (2016) provide a more neutral perspective.
16 The story of six of them has been told by Kleiman (2022).
17 https://ftp.arl.army.mil/ftp/historic-computers/.
18 Ensmenger (2012).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

19 John Backus, quoted by Ensmenger (2012).
20 Priestley (2011).
21 Wilkes (1985).
22 Ensmenger (2012).
23 Polanyi (1958) mentions common law as a system that supports tacit knowledge. This bears some similarity to the HAKMEM report (Beeler et al., 1972) that I mentioned as an example of a document resulting from the hacker culture, which is a large collection of often very specific cases.
24 Documented by Ensmenger (2012).
25 Dijkstra (1972).
26 Ensmenger (2012).
27 Knuth (1968).
28 Knuth (1968).
29 Ensmenger (2012).
30 Here, the field of programming follows the example of physics, whose preference for theory over experimental work has been documented by Hacking (1983).
31 Many of those clashes and struggles are documented by Ensmenger (2012).
32 The early history of programming systems has been documented by Priestley (2011). The birth of programming languages is told by Nofre et al. (2014), in a paper whose title inspired the heading of this section.
33 Univac (1957).
34 www.computerhistory.org/collections/catalog/102646140.
35 Nofre et al. (2014).
36 Backus et al. (1954).
37 Cited in Nofre et al. (2014).
38 Nofre et al. (2014).
39 Quoted by Nofre et al. (2014).
40 The preliminary (International Algebraic Language) report by Perlis and Samelson (1958) and the final report (Algol 60) by Backus et al. (1960) are the primary sources. The development of Algol has been documented in a first-hand account edited by Perlis (1978) and Naur (1978).
41 Star and Griesemer (1989).
42 Perlis (1978).
43 First described by Bosak et al. (1962). First-hand account of the history has been written by Sammet (1978).
44 First-hand account of the history has been written by McCarthy (1978).
45 Priestley (2017).
46 Backus et al. (1960).
47 Backus et al. (1963).
48 Priestley (2011) refers to research paradigms in the sense of Kuhn (1962), although with some caution, because the Algol paradigm exists in parallel with other ways of thinking and participants can exchange ideas with others. This is what I attempt to capture with the notion of 'culture of programming' in this book. Many of the pioneering works of the Algol paradigm can be found in a valuable collection by Colburn et al. (2012).
49 McCarthy (1963).
50 Ensmenger (2012).
51 Habermann (1969).
52 Naur (1966).
53 Floyd (1967).
54 Hoare (1969).
55 A first-hand account of the history can be found in Hoare (1981).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

56 Hoare (1971).
57 McCarthy (1963).
58 Hoare (1971).
59 One cannot avoid an analogy with the seventeenth-century reports of experiments conducted by the members of the Royal Society, which required a confirmation of a trustworthy eyewitness, as discussed for example by Shapin and Schaffer (2011).
60 I am indebted to Mark Priestley for this observation, as well as for another example of engineering influence, which is the notion of 'language maintenance', i.e., the long-term support for the evolution of a language as an entity separate from a particular compiler.
61 Conger (1962).
62 The story can be found primarilly in their PhD thesis (Astarte, 2019), but their earlier publications are also invaluable sources (Jones and Astarte, 2016; Astarte and Jones, 2018).
63 Backus et al. (1963).
64 Backus et al. (1963).
65 Lakatos (1976).
66 McCarthy (1964).
67 Hoare (1969).
68 Mosses (1974).
69 Christopher P. Wadsworth, cited in Astarte and Jones (2018).
70 Lauer (1968).
71 Allen et al. (1972).
72 Henhapl and Jones (1978).
73 As reported by Astarte and Jones (2018), Mosses was working on a tool that would generate compilers from formal language specifications. His Algol description was written in a style used by the tool, but he never actually used the Algol description as input for his system.
74 Lindsey (1996a).
75 Hoare (1981).
76 Haigh (2010).
77 Gries (1978).
78 http://homepages.cs.ncl.ac.uk/brian.randell/NATO/N1968/index.html.
79 Ensmenger (2012).
80 Haigh (2010).
81 Pelaez Valdez (1988).
82 Buxton et al. (1970).
83 Dijkstra (1968).
84 The working paper appeared in Buxton et al. (1970) and in a longer, privately circulated version Dijkstra (1970). Author's later reflections on the history appear in Dijkstra (2002).
85 Böhm and Jacopini (1966).
86 Harel (1980).
87 Plauger (1993).
88 See also historical reflections on the `goto` statement by Leavenworth (1972).
89 Ensmenger (2012).
90 Ensmenger (2012).
91 Reported in the *Datamation* magazine by Baker and Mills (1973).
92 The Team Topologies model by Skelton et al. (2019) can be seen as a new take on the very same idea.
93 Dijkstra (1980).
94 Dijkstra (1988).
95 McCracken (1973).
96 Discussed by Slayton (2013).
97 We can thus perhaps also view programming languages as trading zones, in the sense introduced by Galison (1997).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

98 Concrete technical concepts play a role similar to experiments in physics which, can also 'have a life of their own.' Hacking (1983).
99 https://amturing.acm.org/award_winners/perlis_0132439.cfm.
100 Naur et al. (1969).
101 Buxton et al. (1970).
102 Documented by MacKenzie (2004).
103 DeMillo et al. (1977).
104 Lakatos (1976).
105 DeMillo et al. (1977) and De Millo et al. (1979).
106 Discussed by MacKenzie (2004).
107 Quoted by MacKenzie (2004).
108 Dijkstra (1978).
109 MacKenzie (2004).
110 The different ways of thinking about proofs are documented by MacKenzie (2005), which also inspired the title of this section.
111 Mills et al. (1987).
112 Introduced by Dyer and Mills (1981). The methodology has been extensively discussed by MacKenzie (2004).
113 McCarthy (1963).
114 McCarthy (1962).
115 Rushby and Von Henke (1993), quoted byMacKenzie (2004).
116 Newell and Simon (1956).
117 Moore (2019).
118 The system was first presented by Milner (1972); work on proofs is discussed later in Milner (1979).
119 A first-hand account on the development has been written by Pym et al. (2019).
120 https://fbinfer.com/docs/about-Infer.
121 MacKenzie (2005).
122 For mathematician's reflections on proofs, see for example the collection by Gold and Simons (2008).
123 The applications are discussed by Boyer and Moore (1988).
124 Goldberg et al. (1984), disucssed by MacKenzie (2004).
125 Smith (1985).
126 Boyer and Moore (1983).
127 Fetzer (1988).
128 Ardis et al. (1989).
129 Quoted in MacKenzie (2004).
130 Cohn (1989).
131 Pleasant (1989).
132 Hoare (1969).
133 Leroy (2009).
134 McCarthy (1964).
135 Especially the history of early programming documented by Priestley (2011), computer personnel and power struggles by Ensmenger (2012), history of proofs about programs by MacKenzie (2004), history of software engineering as seen through military software by Slayton (2013) and the history of the programming discipline by Tedre (2014). The book edited by Colburn et al. (2012) collects many of the papers involved in critical reflections on formal program verification and proofs of programs.
136 I use the term *cultures of programming* to tell the story, but the concept is similar to the idea of *systems of practice* introduced by Chang (2012) when analysing the history of chemistry.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

3 Interactive Programming

Teacher: We just saw what becomes possible when we think of computer programs as mathematical entities, but that view leaves out the interaction between a human and a computer through which programs are created. Can we approach programs from the opposite perspective and talk about interactive programming?

Xenophon: Wait, what does interactive programming even mean? Programming languages and algorithms are established topics that everyone understands, but having interactive programming as a topic on par with the mathematisation of programming is odd.

Pythagoras: Programming environments that you use to create programs are interactive, but they support writing of programs in a programming language. Even sophisticated environments for languages like Java are still built around languages.

Socrates: Your assumptions about what programming is determine your answer! We take for granted that program is a static text in some language and we forget that alternative views are possible. Thinking about programming as an interactive process reveals a different perspective and I'm glad we can explore it here!

Archimedes: Are you thinking of something like the famous 1969 demo by Douglas Engelbart, retrospectively known as 'The Mother of All Demos', where he presented his oN-Line System (NLS) that featured mouse, video-conferencing, hypertext and many other aspects of modern interactive computers?

Socrates: This is a great reference, but Engelbart's demo was mainly about using the system, not about programming it.

Diogenes: It is also starting way too late. The first real interactive programming was done at MIT once the hackers[1] from the Tech Model Railroad Club got their hands on the TX-0 computer that was loaned to the MIT Research Laboratory of Electronics in 1958. This was the first time you could program a computer interactively through a terminal.

Xenophon: You can hardly call toying with TX-0 programming! The TX-0 was an experimental machine to test transistor technology for the SAGE defence system. The Whirlwind computer that was used at SAGE before was already interactive, but it was programmed properly, through a carefully controlled process.[2] And anyway, sitting in front of a console in the late 1950s would not scale up as you would be wasting scarce computing resources. You had to do your thinking away from the machine.

Pythagoras: This was soon solved with time-sharing, which made it possible to connect multiple users to a single machine and run their programs concurrently. You could already do this at the start of the 1960s and the Lisp programming language was soon used in the interactive way on time-sharing systems.

Teacher: This raises the question as to whether the emergence of interactive programming enabled new ways of using computers?

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Archimedes: Easier and more efficient? Yes. You could get quick feedback and other help from your programming tools. But conceptually, you still have programmers who create programs, mostly by writing source code, and users who work with those. . .

Socrates: I disagree. Interactivity was the key component of the 'man–computer symbiosis' vision that saw computers as tools for thinking or even as a new kind of literacy. Douglas Engelbart's work on 'augmenting human intellect' places this way of thinking in the broader West coast counterculture movement, whose experiments with LSD used a similar terminology of augmenting the human mind and consciousness.[3] This vision, largely funded by ARPA, inspired a whole new generation of technologists!

Diogenes: Man–computer symbiosis is exactly what you do when you program by directly engaging with the computer. Lisp systems in the 1960s made it possible to construct programs gradually by telling the system how to change the program operation.[4] This is a logical step up from modifying the code at textual level!

Archimedes: Isn't this just about building good engineering tools though? Smalltalk, which came out of the same community in the 1970s introduced many of those. It pioneered object-oriented programming and the use of a graphical programming environment that looked much like the Integrated Development Environments (IDEs) of modern programming languages today.

Socrates: You are looking at Smalltalk from a present-day engineering perspective, but this is not how it was thought about at the time. Smalltalk was a part of the Dynabook project, which was to be a 'personal computer for children of all ages'[5] (Figure 3.1) that would let children experience the excitement of thought and creation!

Xenophon: That is the sort of vision you could expect from the 1960s West coast counter-culture, but how does that help solving real computing problems like modelling the resource requirements of a hospital that a decision theorist may want to program?[6]

Socrates: Smalltalk was envisioned as a general purpose meta-medium that anyone can adapt to their particular needs. A kid may turn it into a painting system, a musician may turn it into an audio synthesis system and a decision theorist can turn it into an agent-based simulation system. . .

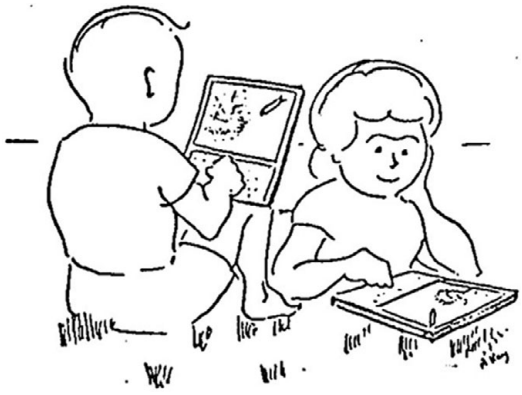


Figure 3.1 A sketch of children using the Dynabook from Alan Kay's 1972 essay.[7] The two 9-year-olds are able to connect their computers, play a game and reprogram its logic. (Source: Courtesy of Alan Kay. Reproduced with permission)

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Teacher: If Smalltalk was so powerful, how come we do not all program in Smalltalk today?

Xenophon: In practice, you do not need a 'meta-medium'. You need a good special purpose software for building simulations, rather than something that you first need to laboriously turn into a system for doing the same thing. I think special purpose software just always ends up being better.

Socrates: Realising the Smalltalk vision is not in the commercial interest of big technology companies. We may have tablets that look like the Dynabook, but you cannot reprogram them from within themselves and share your creation with others! And even Xerox PARC had to label its work as the 'office of the future'.

Xenophon: What is wrong with that? The Xerox Star system that Xerox eventually built in the 1980s (Figure 3.2) failed commercially, because it was too expensive, but it gave you something that was remarkably close to computer systems that office workers used in the 2000s.

Socrates: The problem is that Xerox Star removed all that made Smalltalk interesting! It adopted the desktop metaphor on the surface, but you are now a mere user of something that is given to you. You can no longer shape the system to suit your needs. You cannot learn how it works and make it better.

Diogenes: The vision of a system that you can understand and improve is compelling, but even the earlier systems built at Xerox PARC suffered from a growing complexity at both the hardware and software level.[9] For one thing, it required specialised hardware that existed only inside Xerox PARC!

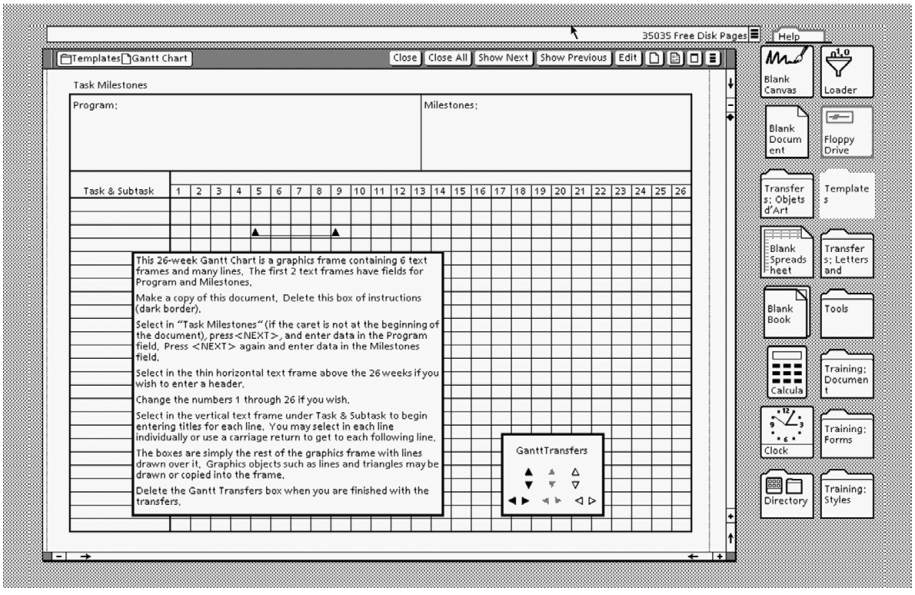


Figure 3.2 Xerox Star desktop showing a template for a Gnatt chart on the left and desktop with further icons on the right. (Source: Screenshot created by the author, using the Darkstar emulator[8])

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Teacher: Was Smalltalk a dead end for interactive programming then? What were the next steps? Did it influence subsequent systems?

Archimedes: Smalltalk had a lot of influence on later object-oriented programming systems, as well as software engineering methodologies. You can also still use Smalltalk today through modern and more efficient implementations. . .

Diogenes: I'm sure these are useful if you think building software is like building bridges! Computers are more flexible and interactive programming is a way to directly control them. In the 1970s, this became clear with the rise of microcomputers, which were small, inexpensive and anyone could build one in their own garage!

Xenophon: You mean toys like the Altair 8800 microcomputer? You had to program that by flipping switches and through a teletype terminal like the TX-0! I do not see how this was the next step for interactive programming. . .

Diogenes: Right, but microcomputers were simple, easy to improve and attracted a huge community of hobbyists. Very soon, the microcomputers of the '1977 trinity' all had screens and an interactive programming environment based on BASIC.[10]

Socrates: This is far from the graphical interfaces of Xerox systems, but I see the connection. Many microcomputers booted directly into the BASIC prompt and so you were doing interactive programming even if you just wanted to load and run a game. Magazines that published source listings for games and other programs further encouraged people to learn how to program.[11]

Pythagoras: It is a shame that microcomputers adopted BASIC and not a sensible programming language like Algol. I agree with Dijkstra,[12] who once wrote that BASIC mentally mutilated a generation of programmers beyond hope of regeneration!

Socrates: Say what you want about BASIC, but the simple kind of interactive programming that became popular on microcomputers would not work with a more structured language. BASIC systems on those machines had a simple line-based interface. If you typed a command, it would run it; if you started with a line number, it would edit the program in memory. This way, you could easily enter programs line-by-line, correct errors and insert more code as needed.

Xenophon: I'm glad people soon figured out that you need a more user-friendly operating system and applications you can buy and use without copying the source code from a magazine. Microcomputers did play an important role in this, because they made it commercially viable to sell and use software like VisiCalc, the first spreadsheet application software. . .

Socrates: Great interactive system, ruined again by commercial interests!

Teacher: Care to elaborate?

Socrates: Interactive programming often emerges with new technologies, like the TX-0, desktop computers at Xerox and microcomputers. They give users more control over the computers and their programming. Businesses find this either inefficient or hard to commercialise, so they repeatedly move to a more closed model where programming software is clearly separated from using software.

Teacher: Has modern software development adopted the closed non-interactive model?

Archimedes: Sorry for repeating myself, but modern software engineering adopted many ideas from interactive programming. This includes interactive shells[13] to prototype ideas and Test-Driven Development (TDD) to get quick feedback on your code edits. Those are all great interactive programming tools!

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Socrates: This is relegating interactive programming to a secondary role. Live coding where you interactively control music synthesis and visual effects does much more. It turns programming into a performance and probes how efficient the interaction can be, because you have to respond to what other musicians are doing.[14]

Pythagoras: This is an odd example, but I have a more sensible one. If you look at how data analysts work in Jupyter notebooks, it is very interactive. You load some data into memory, run various commands to process it, immediately visualise the results and then refine your scripts. Interactivity makes a lot of sense in such a specialised context.

Teacher: There seem to be more examples of interactive programming today than I thought, but they pop up in unexpected contexts.

Socrates: Well, that's what I was saying earlier! Interactive programming often emerges with new technologies, be it microcomputers, data analytics or live coding! Alas, managerial and engineering motivations often call for less flexible but easier to control methods, and so interactive programming is replaced with specialised tools. I just wonder how soon this will happen to data analytics and live coding!

3.1 Spacewar!

At the turn of the 1960s, most computer programming was rather dull. You punched your program on a deck of cards and handed the deck over to a computer operator. The next day, you would get back a listing with 'Program expects a comma. Abort.'[15] because your program had a trivial syntax error. The next day, you would fix your error and try again, without ever getting near the actual computer.

A very different kind of working with computers was taking place in Building 26 at MIT. By 1961, the building became the home of two computers, TX-0 and PDP-1, which were available for interactive use. You could sit in front of those machines, flip the control switches, enter your programs using a Flexowriter terminal, control the attached cathode-ray tube (CRT) screen and even draw on the screen using a light pen, an input device that identifies the screen location at which it is pointing.

The summer before the arrival of the more powerful of the two machines, the PDP-1, to the MIT Electrical Engineering Department in 1961, the staff and students started to think about a way of showcasing the capabilities of the new computer. This inspired a group of current and former MIT students and research assistants to create Spacewar! (Figure 3.3), a game inspired by the science fiction novels of E. E. 'Doc' Smith's *Lensman* series. In Spacewar! two players controlled rockets in space that had a star with gravitational pull in the centre and tried to shoot down the opponent's rocket using torpedoes.

Spacewar! was conceived by Steve 'Slug' Russell, Martin Graetz and Wayne Wiitanen. The three, who mockingly referred to their shared residence at Hingham Street as an Institute, established the Hingham Institute Study Group on Space Warfare and started discussing the idea further. Russell, who was also a member of the Tech Model Railroad Club (TMRC) talked about the idea with other MIT computer hackers. He hoped someone else would write the game and he'd just play so he kept looking for excuses not to do the work himself. He knew he needed sine and cosine routines,

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

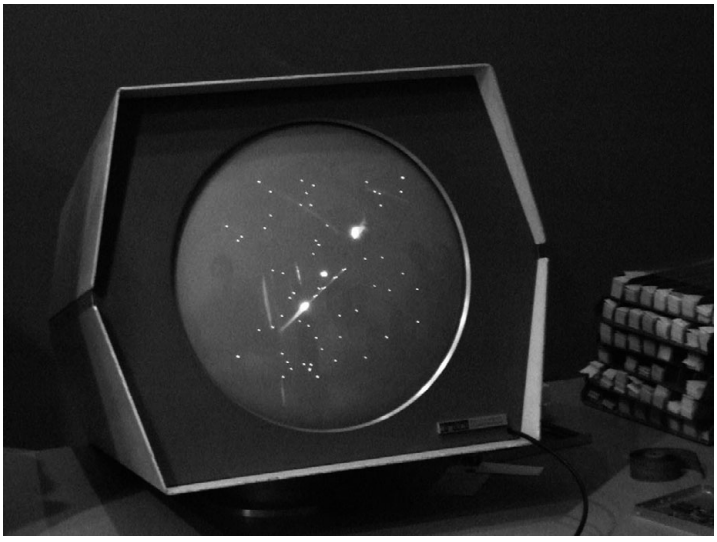


Figure 3.3 Spacewar! game running on PDP-1 at the Computer History Museum in 2007. (Source: Computer History Museum, Photo by Kenneth Lu,[16] CC-BY 2.0)

but claimed he did not know how to write them. In response, Alan Kotok who was a fellow TMRC member and a TX-0 hacker, drove his brand-new Volkswagen Beetle to the headquarters of DEC, the manufacturer of PDP-1. He got a copy of sine and cosine routines that they provided and said 'Okay, Russell, here's the sine and cosine routine. Now what's your excuse?'[17]

After some more hesitation, Steve Russell did write the first version of Spacewar! The game became available to everyone else with access to PDP-1 and, after some debugging and polishing, it became popular enough that the lab had to introduce a policy that playing Spacewar! was the lowest-priority thing the computer could be used for.

Spacewar! was hugely influential and was included in the first 10 computer games to be included in the Game canon, a software preservation effort at the Library of Congress.[18] For this chapter, it is interesting for two other reasons. First, Spacewar! was created interactively. Rather than punching the source code on a deck of punch cards, much of the game was created and tweaked while sitting in front of the computer. Its programmers were using the terminal to modify the code and the screen to test the game. This may not seem surprising today, but it was surprising in the early 1960s when there were just two computers in the whole department, each the size of a room. To use the machine, you would put your name on a signup sheet. Professors and graduate students had some number of allocated hours they could book, but when no one was using the machine, it became available to unsponsored projects and undergraduate students. This encouraged a culture of late-night hacking. Many of the TX-0 and PDP-1 hackers were hanging around the room at night when the machines were in less demand and signed-up users sometimes did not show up.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Second, Spacewar! is an early example that illustrates the hacker ethic which emerged at MIT in the 1960s. Its principles encouraged sharing of source code and improving it.[19] If a program was not deemed good enough by a hacker, the hacker was encouraged to make it better. This happened in a number of ways in the case of Spacewar! In the first version, stars were generated randomly. There was also no gravity, because it was not clear how to calculate it fast enough. Dan Edwards figured out how to make the spaceship rendering faster by generating the instructions to do the drawing for each angle, a kind of just-in-time compilation. This freed enough time to calculate the gravity effect on the two spaceships, but not the torpedoes. Pete Samson, who was an astronomy hobbyist, was dissatisfied with the random stars and decided this had to be fixed. He produced a table of roughly 1,000 stars using an actual star map and wrote code to render them efficiently enough on the screen so that they could be included as the background of Spacewar!

The two aspects are typical of the hacker culture of programming. First, the practical hacker knowledge spreads through personal experience rather than through written materials or formal education. Despite using many innovative techniques, including one of the first just-in-time compilers, the authors did not turn any of those into an academic publication. The hacker ethic encourages sharing of knowledge, but the way it spreads keeps it local and inaccessible to outsiders. Second, hackers value direct engagement with computers. An idea or a beautiful theory is worthless if you cannot actually implement it.

Although Spacewar! is a great example of an interactive program that was also programmed interactively, if we want to trace the origins of the interactive approach to programming, we need to go back a couple more years to the Whirlwind computer, the precursor of TX-0.

3.2 Transistorised Experimental Computer Zero

Towards the end of World War II, the United States Military approached MIT with the problem of designing a flight simulator that could be used to test designs of new airplanes and analyse their aerodynamic stability. In the 1940s, the leading approach was to build an analog electromechanical system. This soon proved too slow and inflexible. The person responsible for the project, Jay Forrester, heard about the ENIAC from a colleague at MIT and decided that a fast digital computer would be up for the task. In 1946, the military agreed to fund the development of Whirlwind, a new digital computer designed to solve the problem.[20]

Whirlwind became operational in 1951. Its application domain posed a number of interesting challenges and as a result, Whirlwind was a revolutionary machine in many ways. Instead of completing individual computational tasks in batches, it had to be able to accept real-time inputs and produce real-time outputs. It also needed a graphical output system and interactive capabilities. Whirlwind had a cathode-ray tube (CRT) screen and was later also equipped with a light pen that was used for screen input. The interactive features of Whirlwind opened the room for new computer applications. The

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

most prominent among them was in the enormous cold war SAGE (Semi-Automatic Ground Environment) air defence system.[21] However, programming the Whirlwind was as tedious as programming other early digital computers. As with the likes of EDSAC, all Whirlwind programming was done away from the machine and debugging mostly involved making sense of lengthy printouts.

Whirlwind used magnetic core memory, a new kind of fast random-access memory, but it was still based on vacuum tubes. To test the use of transistors for a potential future version of the machine, the MIT Lincoln Lab, which housed the Whirlwind, started building TX-0, the 'transistorised experimental computer zero'. After a successful test, the computer was no longer needed and was transferred to the MIT Research Laboratory of Electronics (RLE) on a long-term loan in 1958. Here, it became available for use to qualified users, including the members of the TMRC, without much bureaucracy on a round the clock, seven days a week basis. This arrangement promoted a new, more direct kind of computer programming:

> The idea of working on-line caught on immediately. It was easily demonstrated that a researcher could get a working program off the ground in much less time, and secondly, the output could be monitored so that a useless amount of data was not generated when it was initially apparent that the program was not fully debugged, or that some unforeseen event had not been taken into account.[22]

The new way of working was enabled by the informal access policy, but also by technical characteristics of the machine. The most basic way of manually controlling the TX-0 was through the Toggle Switch Storage (Figure 3.4). This enabled the user to manually set the values of sixteen 18-bit registers in an auxiliary memory storage or set values directly in the magnetic core memory. However, the high-speed large capacity magnetic core memory made possible more than this. In the initial setup, a part of the memory was allocated as a secondary storage medium that could contain program in a symbolic format. This could then be transformed into executable instructions by a primitive compiler. The memory was also used to store a range of utility programs that could co-exist with the user program. The most notable one was a program named Utility Tape 3 (UT-3).[23]

UT-3 enabled interactive programming of TX-0. When loaded, it would wait for commands from the Flexotype terminal. The commands, written in a symbolic language, could be used to examine and modify the contents of registers, search the memory for a particular value or references to a particular address, modify the memory and identify changes between the in-memory program and its version on tape, as well as to run another program in memory using the `go to` command.

The UT-3 was followed by a large number of other utilities, including a more sophisticated symbolic assembler (Macro) and a debugging tool (Flit) that I return to in Chapter 4. A more outlandish utility was developed for debugging a long-running voice recognition software. The utility enabled the user to follow the control path through a program in a flowchart, using a rather elaborate method that involved drawing the flow-chart by hand on a transparent foil and placing it over the computer screen.[24] It also used the toggle switches to slow down or speed up the program.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691



Figure 3.4 TX-0 with the control panel, including the Toggle Switch Storage, 12 1/2" cathode-ray oscilloscope display tube and Flexowriter teletype terminal behind the operator. (Source: Courtesy of the Computer History Museum)

The person in charge of TX-0 after it moved to the Research Laboratory of Electronics in Building 26 was Jack Dennis. As an undergraduate, Dennis worked on Whirlwind and after midnight, he was able to get direct access to it. He knew this was a more efficient way of programming than using printouts and the experience convinced him about the value of direct working with computers. Dennis himself contributed to many of the aforementioned TX-0 tools, including the Macro assembler and the Flit debugger, but he also attracted a new generation of hackers. He gave a number of introductory talks on TX-0 to the TMRC, with which he was formerly affiliated, and established the informal rules that made TX-0 widely accessible.

Many of the TMRC hackers, including Kotok and Graetz who were later involved in the development of Spacewar!, were curious about computers and soon started exploring what could be achieved with the machine. Their way of working included many of the principles of the hacker culture and hacker ethic that I wrote about in the context of Spacewar! To get as much access to the machine as possible, they often used it during the night. Many of their programs were designed for the sake of programming itself, that is, to make programming of the TX-0 easier. Many other programs were curious demonstrations of what is possible, such as a program that controlled an audio speaker to produce computer music. Jack Dennis suggested this problem to Peter Samson who was able to produce a simple melody on TX-0, but was not very satisfied. He eventually installed a hardware extension to TX-0 so that he could get a three-part harmony out of it.[25]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The hacker culture forming around TX-0 encouraged information sharing that had lasting effect on the software industry. It enabled the collaboration around Spacewar! but it also inspired the influential Free Software movement. The limited capabilities of the TX-0 also gave rise to a certain hacker aesthetic.[26] The hackers prided themselves in finding clever tricks to make programs as compact as possible in order to save valuable memory space. On TX-0, writing programs this way was the only option, but low-level assembly was still ubiquitous in the HAKMEM memo,[27] which presented a collection of random MIT hacker tricks a decade later.

3.3 Symbol Manipulating Language

The first experiments with interactive programming were happening at the same time as another development that was making programming easier. This was the emergence of high-level programming languages in the second half of the 1950s. I already wrote about the FORTRAN formula translator, FLOW-MATIC and COBOL that focused on business data processing and ALGOL 58 that made 'programming language' into an independent mathematical entity. All of these were used in the tedious batch-processing mode and did not appeal to the hacker aesthetic. Hackers would eventually find their high-level programming language of choice in Lisp. To follow its origins, we need to get back to a workshop organised at Dartmouth in the summer of 1956 by John McCarthy, before his move to MIT. The workshop brought together a small number of researchers interested in intelligence, learning, abstraction, creativity and computers. McCarthy chose the term Artificial Intelligence for the name of the workshop, coining the name for a new branch of computer science.

Two of the attendees of the workshop were Herbert Simon and Allen Newell. Together with a programmer Clifford Shaw, the two created a program they called Logic Theorist earlier in the year. Logic Theorist was a program for automated reasoning using propositional logic and was eventually able to automatically prove 38 of the first 52 theorems from Whitehead and Russell's *Principia Mathematica*. It worked by selecting and applying appropriate rewrite rules, such as substitution and modus ponens, to a given theorem using heuristics to choose a suitable rule.[28]

Logic Theorist was implemented in a programming language called IPL created by Newell, Simon and Shaw. The language was an assembly language, but it was centred around the idea of list processing. For 1956, this was a revolutionary idea. The language was designed not for numerical computation, but for symbolic list processing and lists proved a powerful tool for representing theorems and implementing their transformations.

After learning about the Logic Theorist and IPL, McCarthy realised that list processing would be a good fit for his own research, but he also wanted a language that would follow the intuitive algebraic notation of FORTRAN. McCarthy explored the idea of implementing support for list processing in FORTRAN in his role as an advisor for an IBM project on reasoning about plane geometry. This led to a language FLPL, which was a set of list processing routines for FORTRAN. Shortly after moving to

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

MIT, McCarthy recruited a group of graduate students and hackers to create Lisp, a more flexible list processing language, for the IBM 704 machine.

Although Lisp was the brainchild of John McCarthy, the language has origins in multiple cultures of programming. This is partly because McCarthy himself belonged and contributed to multiple cultures. His example shows not only that an individual can move between cultures of programming, but also that the views of individual cultures are not incommensurable. His example also supports the hypothesis that the most interesting work on programming arises at the intersection of multiple cultures.

As we have seen in Chapter 2, McCarthy later played a pivotal role in the mathematical culture of programming and Lisp adopted the lambda notation for functions from Church's lambda calculus. However, McCarthy himself also acknowledged that the notation was the only influence as he had not studied lambda calculus before designing Lisp.[29] The Lisp language was also strongly influenced by the engineering issues of list processing and the fact that this was first explored in the context of FORTRAN further shows the influence of the pragmatic engineering culture. Lisp would also never happen without the influence of the early artificial intelligence research. This provided a new kind of programming problem, based on thinking about human thinking, an approach that inspired much work in the humanistic culture of programming that I will discuss later in this chapter.

Finally, some of the interesting innovations in Lisp were also due to the involvement of the MIT hackers. One of those hired to work on Lisp was Steve Russell, who later became famous for his work on Spacewar! In order to explore the computational power of Lisp, the group implemented a function called `eval` that took a Lisp expression represented using lists and evaluated it. Russell then pointed out that `eval` can itself be used as a basis of an interpreter for Lisp, hand coded the function for IBM 704 and Lisp got its first interpreter. Thus an almost accidental realisation of a hacker was at the source of one of the unique Lisp characteristics that is now referred to as 'meta-circularity', that is, the fact that it can largely be defined in itself. Despite the appeal of Lisp, working on it on the IBM 704 computer in batch processing mode was, in the words of Steve Russell, a 'horrible engineering job'.[30]

3.4 Augmenting Human Intellect

Before we continue with the story of Lisp and interactive programming, it will be useful to return to the background of a culture that influenced much work on interactive programming. The proponents of this culture include artists, psychologists, engineers, educators and many others, but they all share a distinct concern for the human and the relationship between the human and the computer. For this reason, I choose to name it the humanistic culture of programming. Unlike, for example, the mathematical culture, humanistic thinking about programming involves a wider range of perspectives. In many ways, the multiplicity of views on programming is one of the defining characteristics of the culture. The work on artificial intelligence, which was concerned with modelling of human thinking, was one of the reasons for the birth of Lisp. At the same

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

time, interest in human creativity inspired programming languages for education and work on computer art.

As the starting point for characterising the humanistic culture of programming, I use two essays that sketched futuristic visions of the relationship between the human and the computer. To use a term adopted later, the two essays envisioned the potential of computers for augmenting the human intellect. The humanistic culture has many different origins and could be with some poetic licence linked to Romanticist ideas through the work of Ada Lovelace,[31] but the two essays directly inspired some of the important follow-up work and serve well to illustrate the aims and ways of thinking of the humanistic culture.[32]

The first essay was written by Vannevar Bush, who was an engineer, inventor and science administrator. Bush joined the Department of Electrical Engineering at MIT in 1919 and worked on a number of electronic devices, successfully commercialising some of his work. He led a team that built the differential analyser, an analog computer for solving differential equations. The machine combined mechanical and electric components and was capable of solving equations with up to 18 independent variables. However, Bush was also a successful research administrator. He became the dean of the MIT School of Engineering and, during the Second World War, managed various scientific research organisations established by the government to support the military efforts.

Before the war, Vannevar Bush started to think about the 'library problem', that is, the fact that investigators in any modern scientific enterprise need to find and plough through hundreds of materials. He thought that some kind of invention, based on microfilm and analog electronic devices, would be able to make such information management much more efficient. After the war, Bush returned to the idea and published an influential article that presented the idea as a direction for peacetime research. In 'As We May Think',[33] Bush introduces the *memex*, a hypothetical device that would store all of an individual's books, records and communications and enable navigation through it via *associative trails*. The vision is strikingly similar to the World Wide Web with its hypertext links, although Bush's thinking was in terms of analog devices, microfilms and he imagined the device as a desk.

The second early proponent for the humanistic way of thinking is J. C. R. Licklider. His background was psychoacoustics, but he became interested in information technology and joined MIT in 1950 where he was involved in the SAGE system for which the interactive Whirlwind computer was built. He later moved to Bolt Beranek and Newman (BBN), a company with close links to MIT which played a key role in the development of many interactive programming tools a decade later. In 1960, Licklider wrote the second landmark essay of our story, 'man–computer symbiosis'.[34] The essay presented a vision of artificial intelligence where the computer is not replacing a human in specific tasks, but instead forms a symbiotic system with the human user. In his own words, the vision is for 'men and computers to cooperate in making decisions and controlling complex situations without inflexible dependence on predetermined programs'. The essay was accessible, widely read and set a well-defined agenda for the humanistic culture of programming.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

In 1962, Licklider got a chance to pursue this agenda in concrete terms when he joined the Department of Defense Advanced Research Projects Agency (ARPA) as a director for its recently established Command and Control Research office, later renamed the Information Processing Techniques Office (IPTO). To Jack P. Ruina, the director of ARPA, who recruited him to lead IPTO, Licklider was known for his work on human factors of SAGE, but his vision of man-computer symbiosis also resonated well with the desire for using computers in military command and control systems.[35]

Licklider never directly collaborated with Vannevar Bush, but he worked on a project studying 'Libraries of the Future'[36] that was inspired by the memex and was started before his time at ARPA, only to be completed after the end of his two-year term as the IPTO director. Both Bush and Licklider outlined a similar humanistic vision of the future where computers are used to augment human mental capabilities, be it by assisting them with information organisation or with making sense of and solving complex problems.

The vision of augmenting human intellect outlined in 'As We May Think' and 'Man-Computer Symbiosis' keeps inspiring programming system designers interested in more creative ways of using computers to this day. One of the earliest examples of this was the Sketchpad system developed by Ivan Sutherland in 1963 and presented in his PhD thesis, 'Sketchpad: A man-machine graphical communication system'. The system enabled users to construct computer drawings using an interactive graphical user interface. This may seem trivial today, but it was a major undertaking in the early 1960s and it required efficient use of cutting-edge computing technology. Sketchpad ran on TX-2, which was a successor of the TX-0, built at the Lincoln Lab two years later. TX-2 was not as easily accessible as TX-0, but it was more powerful and was equipped with the light pen, which Sketchpad utilised as the primary input device.

Sketchpad went beyond just drawing. It supported editing, specification of constraints (e.g., two lines are parallel) and the definition of new reusable symbols. Sketchpad symbols later served as one of the inspirations for object-oriented programming as modifying the definition of a symbol (a class) resulted in change to all its uses (instances), a topic I return to in Chapter 6.

Ivan Sutherland was inspired by both Vannevar Bush and his ideas around memex and by J. C. R. Licklider and his ideas on man-computer symbiosis. The reference to 'communication system' in the subtitle of Sutherland's thesis points to the two organisms, collaborating in a symbiotic relationship. As put by Sutherland himself, Sketchpad provides a more efficient way of communication between the human and the computer:

> The Sketchpad system makes it possible for a man and a computer to converse rapidly through the medium of line drawings. Heretofore, most interaction between men and computers has been slowed down by the need to reduce all communication to written statements that can be typed… For many types of communication, such as describing the shape of a mechanical part or the connections of an electrical circuit, typed statements can prove cumbersome.

Sketchpad envisioned using graphics for drawing and sketching of diagrams, but not for programming the computer itself. It did not take long for this idea to

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

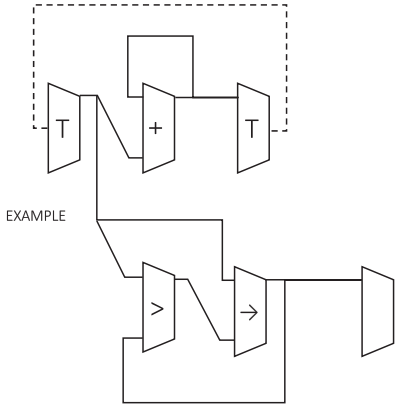


Figure 3.5 A visual program that accepts numbers from terminal and prints their running sum and the maximum, as constructed using Sutherland's system.[37]

ensue. A follow-up project by Bert Sutherland, an older brother of Ivan, added this aspect and described a visual language for 'online graphical specification of computer procedures'[38] (Figure 3.5) in his own PhD thesis in 1966, three years after his younger brother. Not long after, the idea of using flowcharts for visual programming was implemented in the GRAIL system that used a new tablet device.[39]

In the early 1960s, Sketchpad was probing the limits of what it was possible to do. To offer a glimpse of personal computing, it required the full power of the gigantic and enormously expensive TX-2 machine. Only few could imagine that such power would become available for computers that could be owned and used by individuals. One of those who did realise this was Wes Clark, the engineer who designed TX-0 and TX-2 and believed that personal computers were the future of interactive computing. Clark set out to build such a machine in 1961 after completing TX-2. In just one year, he completed LINC, which is sometimes referred to as the first personal computer. The machine supported real-time data processing and was first introduced as a tool for neurophysiological researchers. Following the first public demonstration at National Institutes of Health, the participants could clearly see the potential of the machine:

> It was such a triumph that we danced a jig right there around the equipment. No human being had ever been able to see what we had just witnessed.[40]

Despite the enthusiastic reception, LINC was only moderately successful. The team continued their work and built the first dozen machines for biomedical applications by 1963. Outside of medical systems, the idea was perhaps too far ahead of its time. Only a few could imagine that the cost of computers would drop enough to make the vision a reality. As a result, the team did not receive further funding from MIT, and had to move, which hindered the progress. The work on personal computers continued in various forms throughout the 1960s, but it only gained prominence with the work at Xerox PARC in the 1970s that I return to later in this chapter. Meanwhile at MIT,

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

the work on interactive programming was taking a different, perhaps less visionary but more practical trajectory.

3.5 A Time Sharing Operator Program

The hackers at MIT had access to a single interactive computer at night hours, but this was an exception from the rule. Most other computers at the time, especially those used by businesses, were used through batch-processing. New smaller computers like PDP-1, which was a commercial follow-up to TX-2, were becoming cheaper, but they still cost over $120,000 in 1960 (equivalent to roughly $1.2M in 2024 in terms of purchasing power). At MIT, the cost of PDP-1 inspired a joking series of program names such as 'Expensive Planetarium' (for the code to draw Spacewar! background), 'Expensive Synthesiser' (for the music playing program) and 'Expensive Typewriter' (for an early interactive text editor). At most other installations, the cost of computer time practically prevented interactive use. If interactive programming was to become more commonplace, interactive computing capabilities needed to become accessible to a much wider audience in some way despite the high cost of computers.

One possible answer, which was already in the air at the end of the 1950s, was to let multiple users interact with a single, more powerful, computer at the same time. Humans interacting with a computer through a terminal spent most of the time thinking and typing, so a single machine could, in principle, handle concurrent requests from multiple users. John McCarthy, who had just moved from Dartmouth to MIT, started to call this technique 'time-sharing'.[41] To him, the idea 'seemed quite inevitable' and he thought that 'everybody had in mind to do' it.[42] That was not the case and it took quite some work to get the idea off the ground.

In 1959, McCarthy wrote a memo 'A Time Sharing Operator Program for our Projected IBM 709', in which he argued that time-sharing should be the primary way of using a new machine that MIT was being given by IBM. He saw time-sharing primarily as a way to 'reduce the time required to get a problem solved on a machine'. To get support for his plan, he first had to convince the MIT Computation Centre, which managed access to most computers owned by MIT. This was not hard as the centre was well aware of irate users, complaining about the 3 to 36 hour response time for running programs in the batch processing mode.

McCarthy proposed to do a small-scale experiment using the existing IBM 704 machine. Interestingly, this turned out to require not just writing new code, but also modifying the hardware so that an interrupt is triggered when the machine receives input from a user. McCarthy asked Steve Russell to implement an interactive version of Lisp for the IBM 704 time-sharing system and held the first demonstration in 1960. The idea caught on, attracting both the MIT Computation Centre, which saw time-sharing as a more effective use of their hardware, as well as J. C. R. Licklider, who saw it as a way towards his idea of man-computer symbiosis. In fact, the idea was so closely related that when McCarthy read Licklider's essay on man-computer symbiosis, he

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

thought 'the whole notion was obvious'.[43] Licklider later provided significant funding for time-sharing in his role of the IPTO director at ARPA in the mid 1960s.

The development of a large-scale time-sharing system suffered various technical and political setbacks,[44] but the efforts eventually succeeded. At MIT, the first major implementation was the Compatible Time-Sharing System (CTSS), which was developed as part of the ARPA-funded Project MAC. By the mid 1960s, it ran on a dedicated IBM 7094 machine with over 100 terminals and was able to support 30 concurrent users.

Time-sharing systems were more limited than early interactive computers with display screens, but they were good enough to enable interactive use of high-level languages like Lisp. This gave the MIT hackers a programming language that was *fun* to use. Interactive programming would soon no longer require the use of low-level machine instructions, which was an interesting challenge for hackers, but made it difficult to produce programs of greater complexity.

3.6 A Step towards Man–Computer Symbiosis

The standard way of running a Lisp program at MIT in 1960 was using punched cards. Your program had to be preceded by five cards labelled NYBOL1 and six cards labelled LCON, which you could fetch from the 'Utility Decks' drawer in room 26-265 in the MIT Computation Centre. Lisp came with a tracing functionality that let you debug programs away from the computer. Thanks to the early work on time-sharing, the MIT deployment on IBM 704 also soon made it possible to use Lisp interactively via the Lisp-Flexo system, using the same kind of terminal as when programming the TX-0. The system evaluated entire Lisp expressions entered by the user. The designers of the system recognised that typing a whole expression at once is difficult and offered the so-called TEN-Mode in which an expression could be entered in smaller chunks into 10 buffers (in any sequence with the option to overwrite earlier inputs) before evaluating it. The interactive mode of LISP I was rather basic, but it offered a glimpse of a not-too-distant future.[45]

This started in 1963, when L. Peter Deutsch created an interactive Lisp for PDP-1 at Bolt Beranek and Newman (BBN). Deutsch realised that an interactive way of programming requires 'tools for rapid modification of already existing LISP programs within the LISP system'[46] and started working on an interactive LISP editor. The editor leveraged the fact that Lisp was a list processing language and Lisp programs themselves are represented as lists that contain either symbols or further nested lists. This means that what Deutsch needed to do was to create an editor for nested lists and then use that to edit Lisp programs.

The LISP editor gave programmers a way to navigate through the nested list structure and make changes to it. As an example, Deutsch uses a Lisp function to append two lists with a number of errors and then describes how to correct those. One such error is illustrated in the following snippet:[47]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

```
(LAMBDA (X) Y (COND ((NULL X) Y)
  (T (CONS (CAR X) (APPEND (CDR X) Y)))))
```

The function, defined using the lambda notation, is supposed to take two arguments, `X` and `Y`. If the first list is empty `(NULL X)`, the function returns `Y`; otherwise, it takes the first element of `X` using `CAR X` and appends the remaining ones recursively before adding the removed element. The error here is that the arguments of the function, `X` and `Y` should be written in a single list as `(X Y)`.

After invoking the interactive editor on the function, the programmer can type commands such as `P` to print the currently edited expression. Numbers such as `3` navigate to an n-th element of a list and `0` navigates back to the parent list. The following then prints the expression, focuses on the second sub-list, prints it and then returns back (lines preceded by `*` are typed by the user):

```
*P
 (LAMBDA (X) Y (COND & &)).
*2 P
 (X)
*0
```

When printing, the editor does not print the entire expression, but only the first few levels. More deeply nested lists, such as the arguments of `COND` are replaced with the `&` symbol. To modify the list, programmers could type a number in parentheses to remove or replace the n-th element of a list, so the following corrects the error by first removing the third element and then replacing the second with a new list `(X Y)`:

```
*(3)
*(2 (X Y))
*P
 (LAMBDA (X Y) (COND & &)).
```

The LISP editor was first developed as a stand-alone tool, but it was later revised and improved by others at BBN and incorporated into a Lisp distribution named BBN-LISP in 1971.

Deutsch was a high-school student when he started implementing Lisp for the PDP-1, but he was already an accepted member of the hacker community thanks to his programming skills. The LISP editor, which he developed as a student at the University of California, Berkeley, is also a product of the hacker culture. Despite being one of the first structure editors for programming languages, a technique that would become an active research field in the 1980s and 1990s, Deutsch's LISP editor was only ever documented in manuals and various internal reports.[48] The project also follows the typical pattern where hackers are dissatisfied with their own programming tools and set out to improve them.

Another development that changed how interactive Lisp programming was done was taking place at MIT at the same time as Deutsch created his LISP editor. It was done as a doctoral research project and so it was approaching the problem from a more academic context. In academic research in the 1960s, Lisp was the de facto language for

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

artificial intelligence (AI) research. The practice of AI research provided motivation for Warren Teitelman, who developed an interactive programming system for Lisp named PILOT:

> In artificial intelligence problems …it is important for the programmer to experiment with the working program, making alterations and seeing the effects of the changes. …PILOT is a programming system that is designed specifically for this purpose. It improves and raises the level of interaction between programmer and computer when he is modifying a program.[49]

Although PILOT shares technical characteristics with systems produced by hackers, Teitelman positions his work in the context of 'symbiotic systems', adopting the perspective of 'man–computer symbiosis' developed by J. C. R. Licklider. This is perhaps a necessity as work emerging from the hacker culture was rarely treated as being worthy of an academic publication, but it also suggests that humanistic visions were often easy to combine with the practice of hackers. The thesis further references systems like Sketchpad, further linking PILOT to the humanistic culture of programming.

The central piece of PILOT was a language for specifying list transformations called FLIP. This was then used in various ways to create, edit and modify Lisp programs. PILOT treated programs as collections of procedures and enabled two kinds of modification. *Editing* was the process of changing individual procedures. The `EDIT` function let the user look for patterns in procedure code and transform the matching parts. *Advising* was the process of changing the interface between procedures. The `ADVISE` function could be used to insert other procedures at the entry or exit points of any other procedure. This was used, for example, to insert procedures that would record the history of program execution.

The notion of advising was later further developed by object-oriented systems for Lisp in the 1980s that I mention in Chapter 6, but the concept also inspired Aspect-Oriented Programming that would appear in the early 2000s. However, PILOT is interesting for another reason. In his PhD thesis, Teitelman views Lisp as a *programming system* that the programmer interacts with rather than a *programming language* in which the programmer writes code. This perspective is in stark contrast with the mathematisation of programming and the formalisations of programming languages that followed the publication of the Algol report. This schism remained open for several decades, until, in the 1990s, the 'language' way of looking dominated.[50]

The work on the LISP editor, BBN-LISP and PILOT all came together in Interlisp, which was an implementation of Lisp notable for its many innovative programming tools. It was designed to be an interactive programming system. The programmer using Interlisp was interacting with a running Lisp environment that contained both data and code. The users did not have to worry about files, which were only used when saving the state of the system to an external backup media. Interlisp featured an auto-correction system called DWIM ('Do What I Mean'), also created by Teitelman, that was triggered when running a program resulted in error. It attempted to correct common errors such as typos or incorrect parenthesisation. All such corrections were done in the Interlisp environment, so the corrected version was used next time the code was executed.[51]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The perspective that Lisp is a programming system rather than a language became even more dominant in the late 1970s. The increasingly complex artificial intelligence projects required more memory than the most common research computer could provide and new stock machines were less suitable for running Lisp, primarily because their architecture made common operations with lists hard to implement efficiently. In response, Richard Greenblatt and Thomas Knight at MIT began building a computer specifically designed to run Lisp efficiently. This was the birth of 'Lisp machines',[52] a class of computers that would be the dominant way of running Lisp in the 1980s. Lisp machines were personal computers with displays. Lisp was used both as the systems language to implement everything on the machine and as the way of interacting with the computer. In many ways, the Lisp system fulfilled the role of an operating system.

The case of Lisp is interesting, because unlike Algol and COBOL, it was the product of a mix of cultures. It was used in AI research, it was partly inspired by mathematical formalisms and it was developed by hackers. Interlisp pioneered many of the programming tools that would later become the bedrock of the engineering approach, even though this still required further development. For one, the DWIM auto-correction was reportedly somewhat idiosyncratic to errors made by Warren Teitelman himself. Those hackers making different kinds of errors claimed the acronym DWIM stood for 'Damn Warren's Infernal Machine'.[53]

Different cultures of programming also developed different dialects of Lisp. An interesting example is Scheme, created by Gerald Sussman and Guy Steele in 1975. Their initial motivation was to make sense of the Actor model developed by Carl Hewitt.[54] They decided to implement a toy implementation of the programming model in Lisp and adopted lexical scoping of Algol that Steele was studying at the time. The language initially supported a way of defining functions (lambda) and actors (alpha), but the two researchers soon discovered that the mechanisms are, in fact, the same.

The influence of the mathematical culture on the design of Scheme is easy to retrace, but there is also a more subtle methodological influence of the humanistic culture. In the artificial intelligence research, a common approach of understanding a behaviour was to try to write a program to simulate it.[55] Writing a dialect of Lisp to understand the Actor model follows this approach. Scheme can thus be seen as a product of the methodology of the humanistic culture with ideas of the mathematical culture. It was later used by the authors to explore more mathematical problems of programming languages. In particular, the work resulted in an influential series of papers on programming language semantics ('Lambda the Ultimate') and this was the main force for its success.[56]

Another system that takes a distinct perspective on Lisp was Logo, which emerged from the humanistic focus on education. I will return to Logo after a brief detour that is useful for explaining the typical characteristics of the humanistic culture.

3.7 There Should Be No Computer Art

The humanistic culture of programming is characterised by the concern for the relationship between the human and computer. The futuristic visions of man-computer

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

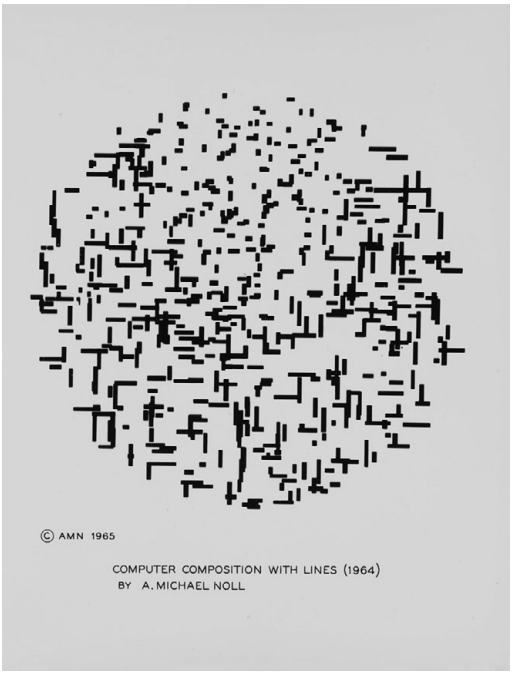


Figure 3.6 Computer Composition with Lines by A. Michael Noll (1964).[57]
(Source: Courtesy of A. Michael Noll. Reproduced with permission)

symbiosis that I discussed above are one instance of this, but the relationship can be probed and explored in a variety of ways. The early use of computers in the context of art approaches the question from a different perspective, but often poses similar questions.

The early work on computer generated art dates back to the 1950s, but the first public exhibitions took place in 1965. Frieder Nake exhibited his work in Stuttgart and A. Michael Noll presented his work in the same year in New York. As Figure 3.6 shows, early computer generated art creatively used the limited capabilities of early computers and single-colour plotters and much of the work involved abstract art, utilising the basic shapes that computers could produce. Despite the limited computer capabilities, the work foretold a number of techniques that would appear in computer art later. In some cases, it also followed the humanistic methodology that I just discussed in the context of Scheme. For example, A. Michael Noll later recruited 100 participants for a study that compared Mondrian's 'Composition with Lines' with a picture he generated using a computer, pointing out that only 28 per cent were able to correctly identify the computer generated picture (and over a half preferred it). Using the program to generate the pattern thus served as the means for probing the nature of art and human perception of it.

Early computer artists also illustrate another important trait of the humanistic culture of programming. It is the self-consciousness and critical reflection of their role in society. Not long after the first exhibitions, Frieder Nake wrote a provocative essay

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

'There should be no computer art'[58] in which he criticises the fact that computer art is being co-opted by the commercial art world, in order to create 'another fashion for the rich and ruling'. Nake does not want computers to become a 'source of pictures for galleries' and argues instead for a use leading to critical reflections on art and the society. Could the kind of art created by Nake and other computer art pioneers be a basis for a more positive transformation in society, rather than hanging on the walls of rich art collectors?

In the vision for children's education pursued by Seymour Papert and his colleagues at MIT, this was exactly the case. The origins of this work are surprisingly technical. In 1961 at a conference in England, Marvin Minsky and Seymour Papert presented similar papers on a technique that became known as reinforcement learning and is the basis of many successful applications of AI to this day. Minsky was a co-founder of the field of artificial intelligence from MIT and the author of numerous PDP-1 hacks, while Papert was working with psychologist Jean Piaget in Geneva studying education. The two were both interested in thinking about thinking; Minsky in thinking in machines and Papert in thinking in children. They soon started a collaboration that lasted several decades and brought Papert to MIT in 1963.[59]

Papert joined Minsky as a co-director of the MIT Artificial Intelligence Lab. He soon became fascinated by the hacker community, which now favoured Lisp as their programming language of choice. In the community, Papert found 'extraordinary examples of learning and problem solving that took place organically'.[60] He also soon started working with the education research group at Bolt Beranek and Newman (BBN) that was led by a mathematician and a musician Wallace (Wally) Feurzeig and included Dan Bobrow and Cynthia Solomon, who had previously taken a job as Marvin Minsky's secretary so that she could learn programming and was then, among other things, involved in the BBN-LISP project.

Papert was inspired by Piaget's theory of active learning, which posits that students should construct their own knowledge by actively interacting with concrete materials. After coming to MIT, he started looking for the right kind of 'concrete material' that would enable children to learn mathematical ideas using the computer. The technical developments such as time-sharing and interactive computing made it possible to think of programming beyond scientific computations and business data processing. The MIT hackers thought of programming as something that every dedicated hacker could master, but it took another leap of thought to see programming as something that every child could learn. This step was taken by the group around Papert, Feurzeig and Solomon who started thinking about a programming language, eventually called Logo, that would make such learning possible.

3.8 Children, Computers and Powerful Ideas

Technically, Logo is a programming language inspired by Lisp, but its aims and culture are very different from anything seen in programming prior to 1967. At the time, professional programmers were learning to program in order to solve business problems;

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

MIT hackers were learning to program in order to do programming, because programming was fun. Papert did not think that children should learn programming for either of those reasons. Instead, he saw programming as a way of communicating with a living native speaker of mathematics. The view is best captured in his later book:

> The computer can be a mathematics-speaking and an alphabetic-speaking entity. We are learning how to make computers with which children love to communicate. When this communication occurs, children learn mathematics as a living language. Moreover, mathematical communication and alphabetic communication are thereby both transformed from the alien and therefore difficult things they are for most children into natural and therefore easy ones.[61]

The group was aware of the early work on computer art and also saw Logo as a vehicle for art. But most importantly, they saw computers as the means that is best used for creative ends. 'As in a good art class, the child is learning technical knowledge as a means to get to a creative and personally defined end.'[62]

Logo was based on Lisp because of Lisp's interactive programming capabilities and because it made list processing easy. The designers explicitly did not want to create a simple language just for children. They wanted a language that would empower students to progress from a simple environment to the most powerful kind of programming available. The key design idea that made this progression possible was later captured by the term *microworlds*.

Microworlds are small environments for thinking about particular problems. The first microworld was built around simple word manipulation and could be used, for example, to create a program that constructs a sentence by generating a random sequence of words of the form 'adjective noun verb adjective noun'. The resulting sentences made children curious about the structure of natural languages and grammar:[63]

```
RED GUINEA PIGS TRIP FUZZY WUZZY DONKEYS
PECULIAR BIRDS HATE JUMPING DOGS
FAT WORMS HATE PECULIAR WORMS
FAT GEESE BITE JUMPING CATS
```

A better known microworld is the later Turtle graphics where the child controls a turtle that draws lines in a two-dimensional space. Turtles first existed in the form of a 'floor turtle', a robot that moved around actually drawing on the ground, but 'screen turtle' provided a similar experience using a display (Figure 3.7). The Logo designers thought of a turtle as an object-to-think-with. Although Papert later said that a turtle should be seen as just one example of such objects-to-think-with, it remains an unparalleled example of a microworld for a number of reasons.

First, the children can easily learn to see the world through the perspective of the turtle. When programming, they can think what they want the turtle to do in relative terms. This makes graphics easier in contrast to using absolute Cartesian coordinates. When Cynthia Solomon later realised that young children had a hard time typing longer commands, she developed a command `TEACH` that lets children instruct the turtle using a simple language of single-alphanumeric commands and is capable of recording

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

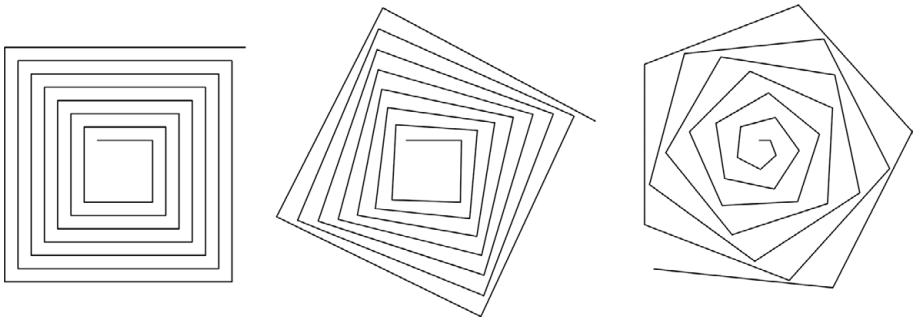


Figure 3.7 Drawing Turtle graphics using Logo. `SPI` is a recursive procedure that moves the turtle forward, rotates it and calls itself with a greater distance. The output shows interesting effects achieved by varying the angle on line 3.
(Source: Recreated by the author using Curly Logo[64])

and replaying such instructions. Second, the turtle metaphor also lets children debug programs by 'becoming the turtle' and literally stepping through the code. Even more interestingly, when the program misbehaves, a child can think 'the turtle has a bug' rather than thinking 'I made a mistake', avoiding negative emotions.

The goal of Logo was to teach powerful ideas to kids. The group used the term 'powerful idea' to refer to concepts that are more general than the examples through which they are taught, such as the idea of anthropomorphisation (becoming a turtle) or metalanguage (language for talking about a language). In many ways, the work on Logo was rooted in the humanistic culture of programming. For one, Logo was more than just a programming language. 'It was also a computer environment made up of people, things, ideas. And it was a computer culture: a way of thinking about computers and about learning and about talking about what you were doing.'[65] Logo designers viewed their work in a broader social and political context. This is well documented in Papert's Mindstorms book where he points out that 'there is a world of difference between what computers can do and what society will choose to do with them'. He acknowledges that his work is political and proposes a revolutionary change, because 'schools as we know them today will have no place in the future'. The work arising from all cultures of programming had a political side, be it struggle for control over software production, support for the countercultural movement, an attempt to establish programmer as a more masculine professional or striving for academic respect.[66] What is unique about the humanistic culture of programming is that it often acknowledges and discusses the political side of programming, even if it often paints an overly rosy picture.

The cultural background of the Logo language had a number of interesting technical implications. In particular, the Logo culture is interesting in how it approaches program errors. Those are seen as sources of serendipity that provide an opportunity for learning. In a hypothetical situation discussed in Papert's *Mindstorms* book, a child wants to draw a flock of six birds 'all the same way up'. When they see four rotated birds instead, they first note that the erroneous result is nice and should be kept for later, before proceeding to debug it by 'becoming the turtle'.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Logo designers were also conscious of the aspects of the system that are important for communication, but are ignored by most other language designers. This includes the syntax, naming but also error messages. A telling story is that of the response of early Logo implementations to an unknown command such as `love`. The initial response was 'love has no meaning', but the designers wanted to avoid telling such a message to children and discussed printing either 'I don't know how to love' or 'You haven't said how to love'. Both of these were criticised. The former overly anthropomorphises the computer while the latter puts too much blame on the child.[67] Finally, the designers were also not talking about just the programming language, but about the entire educational environment. This includes solving problems using Logo on computer, but also group interactions and physical activities. Famously, the teaching done by the Logo group also involved activities like juggling and the Mindstorms book explains different styles of juggling. But those are then encoded as Logo programs, following again the humanistic theme of using programs to understand a problem.

The Logo programming environment is perhaps the earliest and most prominent programming system that was primarily influenced by the humanistic culture of programming, but it had strong influences and contributions from other cultures. The initial motivation for Logo was to teach mathematical ideas and much of the technical developments that it crucially relied on were being built thanks to large-scale engineering efforts, like time-sharing, and playful excitement of hackers who often shared the physical space with the Logo designers.

At the same time as Logo was designed to help children learn to think on the East coast, another project heavily influenced by humanistic visions focusing on improving human capabilities was getting ready for its debut on the West coast.

3.9 The Mother of All Demos

On 8 December, 1968, Douglas Engelbart presented the oN-Line System (NLS) at the Fall Joint Computer Conference in San Francisco. At a time when the most interactive way of using a computer was through a teletype connected to a time-sharing system, the demonstration gave a glimpse of the future. It used graphical user interfaces throughout and featured a way of navigating between documents akin to modern web browsing, interactive display-based text editing not unlike the present-day wikis, a new device for controlling the computer called the 'mouse', as well as a tele-conferencing system bringing some of the speakers virtually from Menlo Park. For all these reasons, the demonstration is often retroactively referred to as 'The Mother of All Demos'.

Engelbart's work was directly inspired by Vannevar Bush. He read the 'As We May Think' essay while serving as radio and radar technician with the United States Navy. It greatly influenced him and motivated him to return to university and enrol in a graduate program at Berkeley in 1951. After completing his studies, Engelbart joined the Stanford Research Institute (SRI) in Menlo Park. He became a valued contributor in the SRI group working on magnetic storage, but his true interest was in pursuing his vision, which he started to refer to as 'augmenting human intellect'.[68]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

By the end of the 1950s, Engelbart got a small grant to work on his ideas and developed a detailed account of his vision, but without actually implementing any of it yet. He outlined the vision in a report 'Augmenting Human Intellect: A Conceptual Framework'.[69] The report is positioned in the humanistic culture of programming through its references to Vannevar Bush and J. C. R. Licklider, as well as its motivation. The objective of augmenting human intellect is:

> [I]ncreasing the capability of a man to approach a complex problem situation, to gain comprehension to suit his particular needs, and to derive solutions to problems. Increased capability in this respect is taken to mean a mixture of the following: more-rapid comprehension, better comprehension, the possibility of gaining a useful degree of comprehension in a situation that previously was too complex, speedier solutions, better solutions, and the possibility of finding solutions to problems that before seemed insoluble.

The report is a combination of a general analysis, specific examples and idiosyncratic details, which are all typical of Engelbart's later work. He analyses general structures involved in working with information, such as different sources of intelligence. He describes the envisioned system that is referred to as H-LAM/T (Human using Language Artifacts Methodology in which he is Trained), next research steps needed to build it, but he also gives concrete examples such as that of an 'augmented' architect designing a building using the system:

> He sits at a working station that has a visual display screen some three feet on a side; this is his working surface and is controlled by a computer (his 'clerk') with which he can communicate by means of a small keyboard and various other devices. …With a 'pointer' he indicates two points of interest, moves his left hand rapidly over the keyboard, and the distance and elevation between the points indicated appear on the right-hand third of the screen.

Engelbart describes what seems like a modern CAD system controlled through a graphical user interface. The vision is closely related to the Sketchpad system that was being developed at MIT by Ivan Sutherland at the same time and was also inspired by ideas on man-computer symbiosis. In the report, Engelbart mentions hearing about the Sketchpad project, but notes that further information was unavailable at the time of writing.

Engelbart's report eventually found its way to Bob Taylor who worked as a research program manager at NASA. Taylor had a background in psychology and did research on psychoacoustics. He was familiar with the pioneering work of the humanistic culture of programming, including the essays of Bush and Licklider, and he soon started supporting Engelbart's work through grants from NASA.

In 1962, Taylor met Licklider who was newly appointed as the director of IPTO. Licklider soon began asking Taylor about his psychoacoustics research. The two became close and Taylor eventually succeeded Licklider as the IPTO director, after it was briefly led by Ivan Sutherland. The NASA and later IPTO provided Engelbart with funding to set-up the Augmentation Research Centre at the SRI and enabled him to recruit a number of early collaborators. The first of those was Bill English, an engineer who, among other things, worked with Engelbart on the development

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

of a computer 'mouse'. English also managed many of the key systems of the 1968 presentation, including the video and audio link between the San Francisco auditorium and the Menlo Park office.

The ideas behind NLS were revolutionary, but the system itself required a lot of expertise in order to be used. As sarcastically remarked by Larry Tesler, who worked on word processing in the 1970s at Xerox PARC, 'they had to justify the fact that it took people weeks to memorise the keyset and months to become proficient, so they came up with this whole mystique about "augmenting intellect"'.[70]

The NLS implementation was equally elaborate. It was built using a number of small languages that a modern-day programmer would call domain-specific. There was, for example, a Machine-Oriented Language for low-level coding and a Content Analysis Language for text manipulation. The languages themselves were implemented in another special language, Tree-Meta, designed for writing compilers. This enabled the team to quickly change the system, but there was still a clear distinction between editing the code of the system and running it. That said, the NLS demonstration was not focused just on video-conferencing and collaborative document editing. It showed a number of features that prefigured tools that appeared in integrated developer environments several decades later. When viewing NLS source code, the programmer was able to collapse structures such as IF statements and each file also contained a list of known bugs and collaboratively edited notes recording, for example, possible future improvements.

Douglas Engelbart established a group of devoted disciples that 'regarded his vision with almost a religious awe'. As one of them remarked, 'he not only made sense; it was like someone turning on a light'.[71] However, it took several more years before anyone tried to implement similar ideas outside of the tight-knit Engelbart group.

3.10 Almost Anything Goes

In the 1960s and 1970s, work on interactive programming was done at a small number of research centres and institutions. They were loosely connected through conferences and meetings, but also through the movement of individuals between the centres and later virtually through the ARPANET. The map of ARPANET from 1972 (Figure 3.8) shows many of the centres. It includes Harvard, Lincoln Lab, MIT and BBN in Boston where the first computer hackers experimented with interactive programming on TX-0, TX-2 and PDP-1 and where Lisp, Logo and time-sharing were developed. It includes the University of Utah where Ivan Sutherland of Sketchpad fame was based and where Alan Kay, whom we will encounter in the next section, did his PhD. It also includes the West coast hub with SRI where Engelbart worked and UCLA from which the first packet-switched message was sent to SRI in 1969.

A key organisation in the development of the community was the ARPA Information Processing Techniques Office (IPTO), initially called Command and Control Research office, of the US Department of Defense. The office was established by ARPA in order to kill two birds. First, they wanted to get into command and control research, but

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

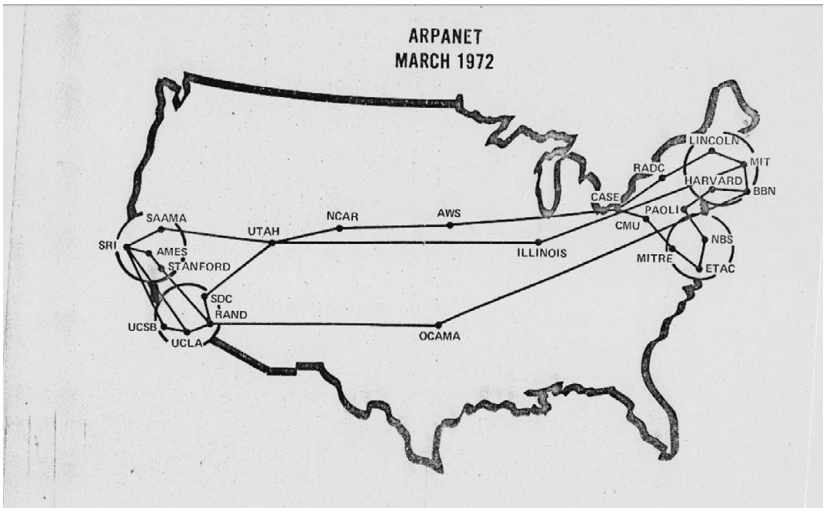


Figure 3.8 A map of the ARPANET from 1972, showing the first 29 connected nodes. (Source: UCLA and BBN, Wikimedia Commons,[72] CC-BY-SA 4.0)

second, they also had a rather expensive computer, built as a backup for the SAGE air defence system, for which they no longer had any use. As one of the earliest projects, the expensive computer was sent to System Development Corporation (SDC) in Santa Monica and used for war game scenario simulations, but IPTO soon started planning further research. The focus on command and control was fortunate as it aligned with much of the thinking around interactive programming. This is clear from the initial purpose of IPTO, which was 'to support research on the conceptual aspects of command and control and to provide a better understanding of organisational, informational, and man-machine relationships'.[73]

After establishing the IPTO, the ARPA managers approached J. C. R. Licklider to become its first director. Licklider was a lucky find as his work on man-computer symbiosis addressed exactly the kind of command and control problems that the IPTO was supposed to focus on. Under Licklider's leadership, IPTO started funding work on time-sharing, graphics, augmenting human intellect and related topics. Crucially, the projects also formed an 'ARPA community' that brought together a diverse range of people from the hacker, engineering and humanistic cultures of programming. The ARPA community continued to develop under the leadership of Licklider, Ivan Sutherland and later Bob Taylor who moved from his earlier position at NASA. The project investigators met regularly at meetings that struck the right balance between being competitive and collaborative, but much of IPTO funding went to students and helped to 'win the hearts and minds of a generation of young scientists'.[74] Under Taylor's leadership, the IPTO also started organising regular meetings specifically for the students it funded, which provided a nourishing environment for future leaders of the field.

The IPTO was pivotal in the development of time-sharing and ARPANET, a direct predecessor of the Internet. By the start of the 1970s, the political context changed and

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

forced IPTO to shift focus from basic research to applied work. Fortunately for the ARPA community, another organisation opened in 1970 and provided a new home for many from the ARPA community. The new organisation was the famed Xerox Palo Alto Research Centre (PARC). Over the next decade, Xerox PARC played a renowned role in the development of modern graphical user interfaces, local networking technologies like Ethernet, object-oriented programming and laser printers. Xerox PARC tapped into the same mix of hacker, engineering and humanistic cultures as the ARPA community. Soon after its founding, its leadership hired Bob Taylor who eventually became the manager of the Computer Science Laboratory at PARC. As in IPTO, he played the role of a visionary and community builder who, somehow, managed to get a large group of computer scientists with diverse backgrounds and interests to work together.

Much has been written about the history of IPTO, Xerox PARC, their key figures and their intertwined histories, as well as the influence of the 1960s counterculture on the developments.[75] The detailed accounts reveal many links and sometimes unexpected connections. For example, Stewart Brand who was the publisher of the counterculture magazine *Whole World Catalog* assisted Doug Engelbart as the video assistant during 'The Mother of All Demos'. Three years later, in 1972, Brand visited Xerox PARC and wrote an article 'Spacewar'[76] for *Rolling Stone* magazine, the title of which was a nod to the computer game that I mentioned at the beginning of this chapter. The article portrayed the Xerox PARC researchers and programmers, including Bob Taylor and Alan Kay, as brilliant, uninterested in conventional goals and plenty of time for messing around, a portrait that shocked the Xerox management and caused much tension between PARC and the higher Xerox management. The countercultural influences were also present in the experiments with LSD, which were often done as part of (more or less) controlled research. Engelbart was interested in those for intellectual reasons as his work on augmenting human intellect was closely paralleled with the idea of enhancing creativity with psychedelic drugs.[77] Engelbart's own experience with LSD did not lead to any new discoveries, but the experiences of many others differed. For example, the HyperCard system that I discuss later in this chapter and which was a groundbreaking hypertext system, was reportedly conceived by Bill Atkinson during an LSD trip in 1985.

The ARPA community and Xerox PARC are interesting as places where multiple different cultures of programming meet. Not just in terms of the broader social environment, but also in more specific terms of how they thought about programming and computing. The early researchers at Xerox PARC came from a range of backgrounds and had past experience with different ways of thinking about computers. PARC hired L. Peter Deutsch, a well-known hacker and a former member of the Tech Model Railroad Club (TMRC) who implemented Lisp for PDP-1 as a high-school student and created the first version of the LISP editor as a student. The group also recruited Bill English, an engineer who was the first member of Douglas Engelbart's group and was instrumental in producing 'The Mother of All Demos'. Further engineers included Chuck Thacker, who previously worked on hardware design for a time-sharing system at Berkeley. His engineering spirit is captured by his lifelong fight against

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

'biggerism'; in his computer designs, one 'never found a logic gate or a ground wire out of place'.[78]

Finally, PARC also had a fair share of members influenced by the humanistic culture. The best known is perhaps Alan Kay. Kay struggled with the rigid education system, but eventually completed a PhD at the University of Utah and joined Xerox PARC soon thereafter. He publicly declared that it is fine to use $3M machines to play games and 'screw around' in a *Rolling Stone* article by Stewart Brand. Kay saw computers as a creative tool. He joined PARC in order to build Dynabook, a personal computer 'which could be owned by everyone and could have the power to handle virtually all of its owner's information-related needs'.[79] Kay's background clearly links him to the humanistic culture of programming. He was influenced by Marshall McLuhan's work on media theory, Jean Piaget's psychology, but also, more directly, by Seymour Papert's work on Logo and Ivan Sutherland's Sketchpad.

Xerox PARC eventually settled on the research program of envisioning the office of the future. This was able to accommodate a wide range of hacking and engineering work, as well as research motivated by humanistic concerns. It included the work of Alan Kay, who was really interested in computers for children, but built systems that were equally interesting for future office systems. The focus still excluded some of the more serendipitous creative work done by artists. For example, Richard Shoup and Alvy Ray Smith, who briefly collaborated at PARC in 1974 and produced the first computer animations, departed too far from the core research program and did not stay around for long. Shoup continued to work on graphics software independently and Alvy Ray Smith went on to co-found Pixar Animation Studios.

The meeting of cultures around IPTO and Xerox PARC provides plenty of evidence illustrating the hypothesis that interesting new developments occur when multiple cultures meet. This was the case for the birth of programming languages in Chapter 2 and it will be the case for the development of types in Chapter 5. The advances on interactive programming are no different. The case of interactive programming is perhaps exceptional in that it brought together people from many different cultures for a relatively long time. In addition to the hacker, engineering and humanistic cultures, the mathematical culture also played its part although less prominently. First, both Papert and Kay were interested in computers as a medium for teaching rigorous mathematical thinking, even though their own work did not lead to the typical outcomes of the mathematical culture. Second, many of the MIT hackers clearly had mathematical interests, as illustrated by the many mathematical puzzles discussed in the HAKMEM memo[80] which is a canonical output of the hacker culture. Finally, Xerox PARC also employed direct contributors to the mathematical culture of programming, such as James H. Morris, whose PhD thesis on lambda-calculus models of programming languages is covered in Chapter 5 when talking about types.

One culture that is conspicuously absent from the list is the managerial culture of programming. In the history of interactive programming, there seems to be little room for well-designed organisational structures, careful advance planning and formalised processes. To paraphrase Paul Feyerabend's mantra of epistemological anarchism, when it comes to interactive programming *almost* anything goes. The community was

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

keen to embrace different methods and principles and often overcame the differences, but this did not apply to managerial methods. There are multiple possible reasons for the absence of managerial culture of programming in the mix. First, interactive programming requires direct and uncontrolled access to computers which is often at odds with managerial methods. The countercultural leaning of the community may be another reason, especially at a time of growing tensions between programming personnel and management[81] illustrated by the 'Unlocking the Computer's Profit Potential' report that I return to in Chapter 4. There are also the personal characters of the individuals. Bob Taylor is widely regarded as an exceptional research manager who built a devoted team of researchers, but his aversion to rigorous management methods resulted in frequent clashes and, ultimately, his departure from Xerox PARC. One may only speculate whether more openness towards the managerial culture of programming would, for example, make the ground-breaking technologies invented at Xerox PARC easier to commercialise.

3.11 Personal Dynamic Media

The meeting of cultures at Xerox PARC created an environment where many thought about programming in a very different way than the creators of COBOL, Algol or even modern programming languages like Python. Alan Kay is now best known as the designer of the Smalltalk programming language and the pioneer of object-oriented programming, but his interest at Xerox PARC was always in building a portable computer that would be easy enough to use for children, while being powerful enough to offer the full flexibility of a programmable computer.

Following this humanistic vision, Kay also saw *using* the computer and *programming* the computer as closely interconnected. When you see a computer for the first time, you may only try running the code that is already there, but as you learn, you will want to be able to customise and modify the pre-existing programs and eventually, develop new programs on your own.[82] In other words, an iPad or other modern tablet may look much like the tablet computer sketched by Kay in Figure 3.1, but the way of working with it is nothing like what Kay was trying to achieve. The fundamental difference is that programming was an inherent part of using the Dynabook. An iPad makes a strict distinction between a user and a programmer and you cannot gradually progress from one to the other without leaving the system. The Dynabook was supposed to make this possible.

Kay imagined that hardware developments will make it possible to build an actual Dynabook within a decade or two. To experiment with ideas, he wanted to build an 'interim Dynabook', which would be powerful enough to show what working with Dynabook would feel like. This coincided with the interests of two PARC engineers, who took it as an opportunity to build a personal computer with an even broader range of uses.[83] The work resulted in the influential machine Xerox Alto shown in Figure 3.9. About 30 machines were built initially. They soon became popular across PARC, as well as outside, and over a thousand Altos were eventually built.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

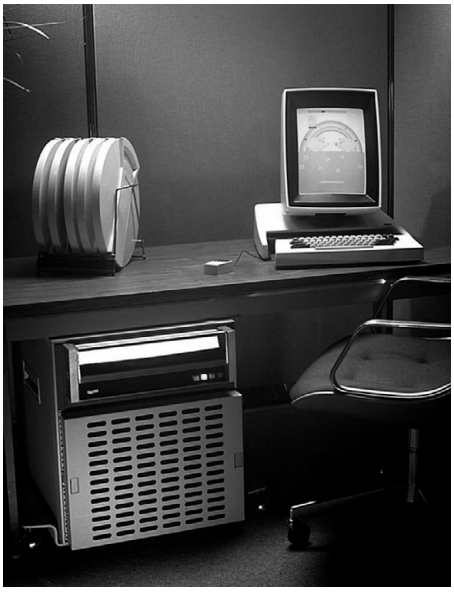


Figure 3.9 Alto with storage disks, computer, vertical display, mouse and a keyboard. (Source: Computer History Museum. Courtesy of PARC. Reproduced with permission of SRI International)

Equipped with a machine, Alan Kay and his team started working on implementing the Dynabook vision. The best account of the concept is the 'Personal Dynamic Media'[84] paper, co-authored with a PARC collaborator and co-developer of Smalltalk, Adele Goldberg. The paper uses a language influenced by media theorist Marshall McLuhan and describes the goal of the group as designing 'dynamic media which can be used by human beings of all ages'. Smalltalk is presented not primarily as a programming language, but as 'a new medium for communication' with the machine.

Kay and Goldberg saw computers as a *meta-medium* that can be used to simulate, or *become*, other media, including a 'wide range of existing and not-yet-invented media'. The example programs described in the paper capture the expected way of using the system. They include 'An Animation System Programmed by Animators', 'A Hospital Simulation Programmed by a Decision-Theorist' and an 'Electronic Circuit Design by a High School Student'. The expectation is that a user will first use the meta-medium of Smalltalk to construct a medium for solving the kind of problems they are interested in, be it animation, modelling or circuit design. The user will then use the resulting medium, or program, to create animations, model decision theory problems or design circuits. Programming is thus a new kind of literacy that everyone will master in order to use the computer as a flexible tool for their work.

The idea that anyone would be able to use a computer in an unrestricted way through Smalltalk was an appealing vision to the proponents of the humanistic culture of programming. The reality turned out to be more problematic. The team taught the first practical version of the language, Smalltalk-72 to local middle school children and,

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

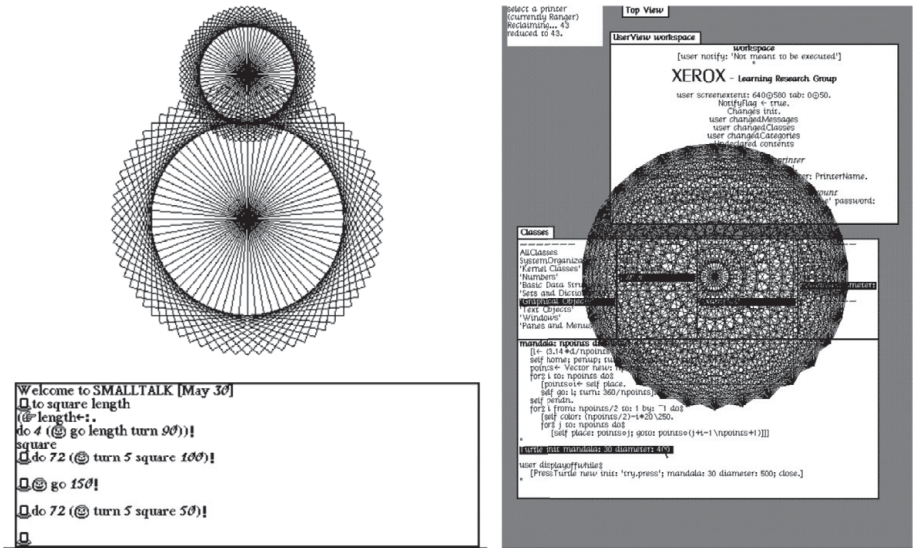


Figure 3.10 Turtle in Smalltalk-72 (left) and Smalltalk-76 (right). In Smalltalk-72, the screenshot shows a dialog window with entered code and the result of running a turtle code. In Smalltalk-76, the screenshot shows the 'mandala' operation in the class browser and the result of running an example included in the method definition.
(Source: Created by the author, using emulators from the Smalltalk Zoo[85])

despite some positive experiences, many kids struggled with rudimentary Smalltalk concepts. Many non-professional adults working at PARC who Kay and Goldberg tried to teach Smalltalk faced similar challenges.[86]

In January 1976, during a team research retreat Kay tried to convince his team to start afresh and make a new attempt at producing a communication medium for anyone to use. However, by this time, Smalltalk was already a sizable project that had a life of its own. Dan Ingalls, who did most of the system implementation, certainly did not want to abandon Smalltalk, but instead wanted to turn it into a full programming system. He went on to create Smalltalk-76, which was a more efficient, fully supported object-oriented concept like inheritance and came with a graphical interface, shown in Figure 3.10 (right). The graphical interface also included the iconic Smalltalk object browser, a window that can be used for browsing the different class definitions that exist in the system and for viewing or editing their methods. At the same time, Alan Kay shifted his focus to a new, more accessible computer and a language called NoteTaker. Here, the Smalltalk language followed the same pattern as the Xerox Alto machine. The PARC community was eager to use its own tools and both Alto and Smalltalk started to be used by others, but typically for their own purposes and experimentation that was somewhat different from the motivations of their creators.

Smalltalk may not have become the universal medium for communication with a computer, but even as a programming language, it retained many of the ideas from which it was born. Most importantly, Smalltalk was not a programming language in

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

which you would write a complete program that would then be run. Instead, you were interacting with an environment that already contained various definitions that you could reuse and modify. In this, it was similar to Lisp systems like Interlisp, but the programming model of Smalltalk was based on objects, a topic we will return to in Chapter 6. Smalltalk-72 also pioneered using graphical interface for programming. You wrote your commands in a dialog window at the bottom of the screen. When editing a definition, the window becomes a structure editor, logically similar to that developed by Deutsch for Lisp, but controlled using a mouse and menu, rather than a terminal console. Smalltalk-72 also followed Logo in the attempt to be more friendly through the use of non-technical language. The manual written by Adele Goldberg and Alan Kay[87] introduces programming as 'talking to Smalltalk' and the language itself uses graphical symbols like a hand for defining a new symbol and a smiley face, as a value representing a turtle, both of which appear in Figure 3.10 (left).

The development of Smalltalk happened in parallel to a major development rooted in other cultures of programming. Alan Kay started working on ideas leading to Smalltalk during his PhD at University of Utah, which he did in 1966–1969. This is only a few years after the Algol language appeared in 1958. Many of the important developments in the Algol research programme that I discussed in Chapter 2 happened around the same time. This includes the rising popularity of structured programming, development of mathematical tools for reasoning about programs and the very idea that a program should be treated as a mathematical entity that can be formally analysed. Smalltalk was influenced by Algol in various technical aspects, but it belongs to a very different culture of programming. In Smalltalk, you use a computer by interacting with it, sometimes through writing code. You work by modifying a rich environment. There is no clear 'program' that you could consider in isolation. Smalltalk is the culmination of ideas leading to interactive programming. It combines the interactive graphical way of using computers, envisioned by humanists augmenting human intellect and implemented in Engelbart's oN-Line System (NLS), with the direct way of interacting with running programs developed by hackers that culminated with Interlisp.

The development of interactive programming no doubt benefited from the mixing of different cultures of programming. Smalltalk appeared as a system motivated primarily by the humanistic vision of programming, but had many other influences and contributors. It did not achieve its original goal, but it was adopted by the hackers and engineers at Xerox PARC for other projects. It is easy to imagine that, if it was not for the people leaning towards other cultures of programming, who took leadership of the project and developed Smalltalk further, it would remain just an abandoned attempt at programming for 'human beings of all ages'.

3.12 Articulate Languages for Communication

Today, Smalltalk is remembered as the programming language that crucially contributed to the development of object-oriented programming. This is true and I follow this strand in Chapter 6. However, Smalltalk is also a flagship interactive

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

programming system that brought together the humanistic vision of programming as communication with graphical user interfaces. In this, it has been followed by a number of programming systems from the late 1970s until today. Although the systems following Smalltalk were not as influential, it is worth looking at two examples as they illustrate the way of thinking about programming in the humanistic culture.

The first is Pygmalion,[88] which appeared in 1975 and is a 'two-dimensional, visual programming system' implemented on top of Smalltalk-72. The second is Boxer[89] which appeared in 1985 and has been described as an 'integrated computational environment suitable for naive and inexperienced users'. Aside from sharing the cultural context, the two systems are both visual programming systems in that they represent the program in a graphical way and let the user construct programs through visual means. In Smalltalk, graphical interface was used for structure editing (in Smalltalk-72) or for browsing through available objects (in Smalltalk-76 and newer). In Boxer and especially Pygmalion, graphical interface is used for constructing the program logic itself. Furthermore, Pygmalion and Boxer both motivate this idea through arguments that are centred around creative thought.

Pygmalion was created by David Canfield Smith during his PhD at Stanford University. Smith joined Stanford in 1967 with a goal to develop a computer that would be able to learn. He eventually changed his direction and approached Alan Kay to be his thesis supervisor. Following Kay's advice, Smith read a number of books on philosophy and art and his work was greatly influenced by ideas on creative thought and schematic thinking from 'Art and Illusion' by Ernst Gombrich.[90]

Pygmalion uses two-dimensional graphical representation because, as Smith argues, visual images provide more descriptive power. Using a term borrowed from Gombrich, the aim of Pygmalion is to design an *articulate* language for communication with a computer, that is, a language that 'corresponds closely to the form used in the mind in thinking about the problem'.[91] Pygmalion, shown in Figure 3.11 (left) is based on two key principles. The first is the principle of *icons*, concrete visual representations of an idea that appear on the screen. In Figure 3.11, the icons are mostly labels in boxes, but the thesis also includes a more graphical example of electronic circuits. The second principle is *programming by demonstration*. A program is constructed not by *telling* the computer what to do but instead by *doing* it. The Pygmalion system implements a 'remembering editor for icons', which stores the sequence of actions done by the user and is able to re-execute them. Incidentally, the same recording mechanism was also created in 1974 to teach Logo programming to young children.

The Pygmalion programming system itself could have been described in a narrow technical way, but Smith's PhD thesis does the exact opposite. Its very subtitle 'A Computer Program to Model and Simulate Creative Thought' makes a reference to creativity and its first two chapters discuss 'a psychological model of creative thought, forming the basis for the Pygmalion design principles'. The thesis also contains numerous references, arguments and examples that connect it with humanistic culture. These include references to the Logo programming language, Turtle graphics, Sketchpad, but also the Spacewar! game developed by the MIT hackers for PDP-1 in 1962.Those

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

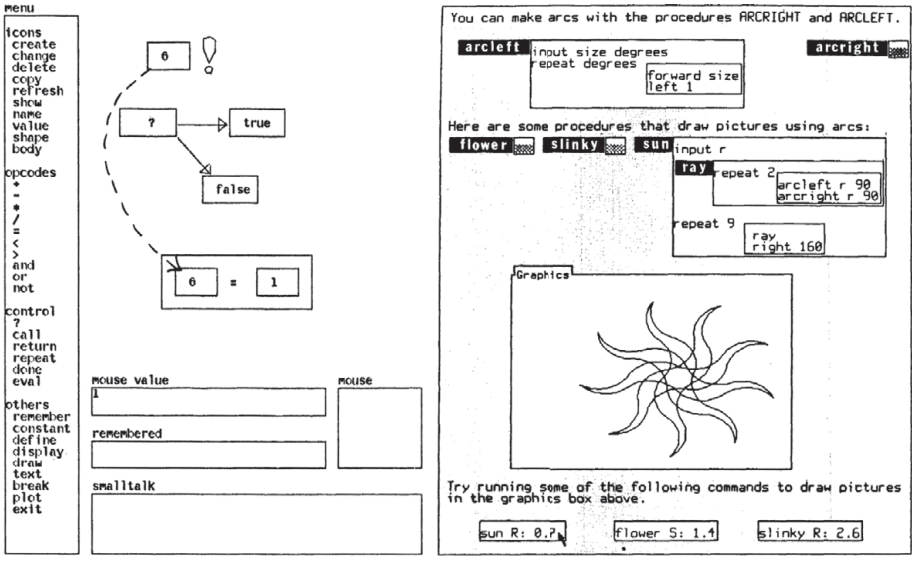


Figure 3.11 Pygmalion (left) and Boxer (right). The Pygmalion example shows a step in the process of computing the factorial function and the Boxer example shows a simple Turtle graphics program.
(Source: Left: Used with permission of Springer Nature BV, from Smith (1977). Right: Used with permission of ACM, from diSessa and Abelson (1986). Permissions conveyed through Copyright Clearance Center, Inc.)

references serve as *cultural pointers*[92] that connect the thesis to the broader humanistic culture of programming.

The motivation for Boxer was more specific. Andrea diSessa, who started the project is a trained physicist, but became interested in education and joined Papert's Logo group at MIT. The goal for Boxer was to build a medium that could be used for creating interactive physics textbooks. A book on optics might include structured text, alongside simulations that perform ray tracing through lens and mirrors. The medium for such a book should be easy to program, but more importantly, it should be *reconstructible*. The user should be able to see the programming, understand it and modify it to fit their needs.

Boxer, shown in Figure 3.11 (right), implements a reconstructible computational medium using two key concepts. The first concept is the use of a *spatial metaphor*. Programs are represented as nested boxes in a two-dimensional space. The boxes can contain text or code, itself represented using nested boxes. For example, in the figure, a box `repeat 2` contains a nested box with the code to be repeated twice. Boxes can also be collapsed to save screen space. The second concept is *naive realism*, which is the idea that what you see on the screen are the computational objects themselves. In Boxer, the program is not hidden. You see it all in the form of nested (possibly collapsed) boxes. A procedure is a box, code to call a procedure is another box and visual output generated by running it is yet another box. An interesting consequence is that the programming language *is* the user interface. There is no need for buttons, because you can run a simulation by clicking on the box that represents the code for the simulation.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

There are many reasons for viewing Boxer as a product of the humanistic culture of programming, including the focus on education: the fact that diSessa was affiliated with the Logo group at MIT, but also the different arguments that support various design choices in the papers about Boxer. Like Pygmalion, Boxer's use of visual representation is motivated by an argument about human cognition. People have a commonsense knowledge about space and the system can leverage that to help users 'interpret the organisation of a computational system in terms of spatial relationships'. The references in papers on Boxer include Pygmalion and a large number of 'integrated computational environments' that target experienced programmers such as Smalltalk and Interlisp.

Ideas from Pygmalion and Boxer all found their way into modern computing and programming tools, but often in ways that leave some of the original motivations behind. David Canfield Smith, who created *icons* in Pygmalion later joined Xerox to design the user interface of Xerox Star, an office system that was an attempt to commercialise some of the technology developed at PARC. The design used icons in a way we would recognise today, but without any of their original computational nature.[93] Boxer led the way to contemporary block-based visual programming environments for kids like Scratch, but those only keep the idea of visual programming, not that of reconstructible media.

Projects such as Pygmalion and Smalltalk aimed to make programming accessible to everyone, but they run on computers like Xerox Alto that were still too expensive to be owned by an individual. The systems started as research projects and their creators correctly expected that the exponential growth of computer power will soon solve the issue of cost. The cost of computers did, in fact, decrease, but this brought very different kinds of machines from those that Alan Kay and his group imagined.

3.13 Homebrew Computer Club

Since its establishment, Xerox PARC was subject to tensions between the researchers who pursued a curiosity-oriented approach and trusted scientific serendipity and the Xerox management that saw PARC largely as a way of preparing for potential future threats to their existing business model.[94] In 1971, PARC researchers reluctantly responded to the pressure from management, embodied by Xerox planning executive Don Pendery, and produced a brief report on the future of computing that was cheekily labelled 'PARC Papers for Pendery and Planning Purpose'.[95] The authors described many of their visions, some relevant for the 'office of the future' and estimated that their technology should be commercialisable by the early 1980s. Xerox eventually tried to do this. In 1977, it established the Systems Development Department, which started to build a system inspired by Xerox Alto that would be usable for office tasks. The resulting Xerox Star system was released in 1981 and was in many ways an impressive technology. It featured a graphical user interface based on icons as we know them today, a modern word processing application, all connected using Ethernet with laser printers. But the system remained too expensive and Xerox was

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

unable to find an effective marketing strategy for selling it. Smalltalk itself was not publicly released until 1980 and even then, it took several more years before it became efficient enough on commodity hardware to be used for business application development.

In the meantime, a new kind of computer appeared. Throughout the 1960s, hardware manufacturers started producing increasingly sophisticated integrated circuits. Finally, in 1971, Intel released a complete 4-bit general-purpose microprocessor, Intel 4004, available as a single chip. Intel was at first unsure about marketing the chip, but they eventually began offering the 4004 as a 'computer on a chip'. This was soon followed by a more powerful family of 8-bit microprocessors, including the Intel 8080 that caught the eye of a small electronics company called MITS. The company was initially producing telemetry modules for rocket models, but it shifted its focus in the early 1970s and used the Intel 8080 microprocessor to create a cheap microcomputer that they called Altair 8800. Altair was a kind of computer that its creators wanted for themselves. Like many of the early adopters, they did not have any specific use for it. They just wanted to play with a computer of their own. This growing community of electronics hobbyists was served by a number of monthly hobbyist magazines. When the editors of one such magazine, *Popular Electronics*, started looking for a computer project based on the Intel 8080 microprocessor, they learned about the ongoing MITS project. Altair 8800 was featured in the January 1975 issue of *Popular Electronics* and the readers could directly order it for $397 (the equivalent of about $2,500 in 2024). Following the publication of the article, the machine became an instant hit.[96]

Altair 8800 was nothing like a modern personal computer. It was also nothing like the Xerox Alto that ran Smalltalk. It was much more like a tiny version of the 1955 TX-0 machine, but even that came with a built-in tape reader and a 12" oscilloscope screen. The Altair 8800 was initially sold as a kit that you had to assemble yourself. The only way to control it was through a sequence of switches on the front and the only way it could respond was by flashing lights. You might even be reminded of the ENIAC computer with its flashing lights covered by halves of Ping-Pong balls in 1946, which I mentioned in the opening of Chapter 2. The dramatic change was that Altair 8800 was affordable enough to reach a new generation of hackers, eager to get their hands on a new electronic gadget and explore what could be done with it.

Using just switches and lights, you couldn't do all that much with the Altair, but the system had a detailed specification that enabled you to modify it. You could build or buy extensions for connecting the Altair to a paper tape reader, teletype or a video terminal. To load a program from a paper tape, you first had to use the front panel switches (Figure 3.12) to manually enter the instructions of a bootloader, which was a very compact program that copied data from a tape into the Altair memory. Then, you would flip a switch to read the program and another to actually run it. If you connected the Altair to a teletype or a video terminal, the program could then interact with the user by printing to and accepting input from the terminal.

To the engineers at PARC, Altair 8800 and its early successors looked desperately primitive and the hackers working with those machines appeared as mere kids with toys. But the community of enthusiasts around those machines grew and started

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691



Figure 3.12 Front panel of Altair 8800, showing the status lights, a row of switches for specifying addresses and entering data and a row of control switches.
(Source: Living Computers Museum, Photo by Michael Dunn,[97] CC-BY 2.0)

creating a wide range of programs. The most iconic part of the community was the Homebrew Computer Club in Menlo Park, California. Many of the founding members of the club built their own microprocessor-based computers, even before the Altair was released.

The early microcomputer community that developed around the Homebrew Computer Club shared some of its characteristics with earlier contributors to interactive programming that I discussed earlier, but it also brought new perspectives. It shared much of its spirit with the 1960s MIT hacker culture in that it emphasised individual technical achievements, information sharing and the Homebrew members believed that software should be free and shared with everyone. The club was also rooted in the same countercultural social context as Engelbart's Augmentation Research Center and Xerox PARC. The members believed in a do-it-yourself approach that was a strong focus of Stewart Brand's countercultural bible, the *Whole Earth Catalog*.[98] In contrast to the early MIT hackers, the Homebrew Computer Club was more open towards commercial pursuits.

The fact that for-profit companies were increasingly involved in the development of hardware and software for microcomputers did not initially have an effect on what cultures of programming were involved. The microcomputer community brought together hackers interested in tinkering with the electronics with humanists who saw programming as a way of advancing the human condition. They were interested in building systems that 'would allow the public to take advantage of the huge and largely untapped reservoir of skills and resources that resides with the people' and believed that computer intelligence should be directed towards 'demystifying and exposing its own nature, and ultimately giving [the user] active control'.[99] The community believed in decentralisation that is at odds with the managerial approaches to programming that were emerging in large-scale commercial computing enterprises. Attempts to transition

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

from the 'black art' of programming to the science of programming, which we looked at in Chapter 2, were also largely ignored by the microcomputer programmers.

Programming of microcomputers was also initially not influenced by for-profit companies because the most notable early businesses focused on building and selling hardware, rather than developing software. The most famous company that originated in the Homebrew Computer Club is undoubtedly Apple, co-founded in 1976 by Steve Wozniak and Steve Jobs, which started with the Apple I do-it-yourself kit. The idea that information should be free led to conflicts with those trying to profit from building software. This was the case of Bill Gates and Paul Allen, who realised the commercial potential of microcomputers, founded Micro-Soft and implemented an Altair 8800 interpreter for the programming language BASIC. The interpreter was licensed to MITS who started selling it on behalf of Micro-Soft. In line with their belief that information and software should be free, the Homebrew community started copying and sharing the BASIC interpreter, to which Gates responded with a contentious 'An Open Letter to Hobbyists' that accused them of stealing.[100] As argued by Joy Lisi Rankin, BASIC was already the 'language of the people'[101] and the dispute did not prevent it from becoming the de facto programming language for microcomputers.

In 1977, three companies released personal computers that sold in millions over the next few years and made computers available to a broader public. The three computers were Apple II, Commodore PET and Radio Shack TRS-80 and they were all influenced by the Altair 8800 or the Homebrew Computer Club in some way. Apple II was a successor of Apple I. It was also designed by Steve Wozniak, but it marks a shift from a machine for hobbyists to a machine for end-users. It was pre-assembled, with all electronic circuitry hidden in a user-friendly box. The Commodore company briefly considered purchasing the Apple design before producing their own machine and TRS-80 was co-designed by another Homebrew member and an Altair kit owner. Perhaps due to the Altair 8800 influence, all three 1977 microcomputers came with a BASIC interpreter that partly served as a primitive operating system. BASIC was not the only way of programming microcomputers and programmers could always choose to write code in low-level assembly. This was favoured by many more sophisticated users and, later, also by companies producing advanced software. But BASIC was the first language that many new computer users would see and it shaped their experience with computers.

3.14 Beginner's All-purpose Symbolic Instruction Code

The history of the BASIC language dates back to 1964 when it was designed by John G. Kemeney and Thomas E. Kurtz at Dartmouth College. BASIC ('Beginner's All-purpose Symbolic Instruction Code') was created as a language to be used for educational purposes for the new Dartmouth time-sharing computer system. It was initially designed for writing numerical programs in a way that would be easier than using FORTRAN. The idea was that every Dartmouth student should have access to

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

$$A_1 X_1 + A_2 X_2 = B_1$$
$$A_3 X_1 + A_4 X_2 = B_2$$

$$X_1 = \frac{B_1 A_4 - B_2 A_2}{A_1 A_4 - A_3 A_2}$$

$$X_2 = \frac{A_1 B_2 - A_3 B_1}{A_1 A_4 - A_3 A_2}$$

```
10  READ A1, A2, A3, A4
15  LET D = A1 * A4 - A3 * A2
20  IF D= 0 THEN 65
30  READ B1, B2
37  LET X1 = (B1 * A4 - B2 * A2) / D
42  LET X2 = (A1 * B2 - A3 * B1) / D
55  PRINT X1, X2
60  GO TO 30
65  PRINT "NO UNIQUE SOLUTION"
70  DATA 1, 2, 4
80  DATA 2, -7, 5
85  DATA 1, 3, 5, -7
90  END
```

Figure 3.13 An introductory example from the 1964 BASIC manual. A program for solving a pair of linear equations (left top) implementing a calculation (left bottom) as a BASIC program (right).
(Source: Dartmouth College Computation Center[102])

it and should learn it during their studies. As such, BASIC was designed to be easy to use, but also to work well on an interactive terminal.

The BASIC language itself was very simple. The introductory example from the 1964 BASIC manual (Figure 3.13) shows many of the available constructs. Programmers could define variables, read data from the `DATA` block at the end of a program, and use conditionals and the `GO TO` statement for transferring the control to another part of the program. In the example, this is used to read multiple inputs and solve multiple pairs of equations. In addition, the 1964 version of BASIC also had a way of implementing subroutines using the `GO SUB` command, `FOR` loop for iteration and a way of defining multi-dimensional variables to represent lists and tables.

Despite the simplicity of the language, there were a few features that made it powerful and interesting enough for hackers. Since the Altair, BASIC had commands `PEEK` and `POKE` for reading data from and writing data to any memory address specified as an argument. This gave programmers access to everything on the machine that was not exposed in another way, such as drawing on arbitrary screen location. Since the BASIC program itself was stored in the memory, you could also write programs that looked at their own source code and modified it. In addition to `PEEK` and `POKE`, there was also the `SYS` command for invoking code at an arbitrary memory location, including various system routines and assembly code written by the programmer. Despite the simplicity of the core language, the system provided enough backdoors for hackers to practise their craft once they mastered all the basic tricks.

Perhaps more interesting than the language was the way of interacting with it. On Commodore PET, BASIC was not just a programming environment, but the operating system of the computer. When the machine started, the BASIC command line appeared (Figure 3.14) and the user could start by typing BASIC programs. Even if you wanted to play a game that you got from a friend on a tape, you had to do this by invoking the BASIC command `LOAD`, which takes a file name and device number as arguments and

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691



Figure 3.14 After turning on, Commodore PET starts and immediately runs Commodore BASIC. Any further interactions were through the BASIC command prompt. (Source: Courtesy of Terry Stewart.[103] Reproduced with permission)

then you had to type `RUN`. This loaded the BASIC source code for the program into the memory of the interpreter and ran it. You could then also view the source code and try to modify it. Although Commodore BASIC is certainly not how Alan Kay and Adele Goldberg imagined 'personal dynamic media', it was in some ways close to the idea. Unlike programming languages born from the mathematical and engineering traditions including FORTRAN and Algol, BASIC did not impose a strict distinction between programming and running a program.

Using BASIC on Apple II, Commodore PET and TRS-80 had an interactive nature not just because you sometimes typed commands in the prompt. The other factor was its use of line numbers, which was a remnant from the era of teletype terminals and was sometimes ridiculed, but it proved remarkably useful. As Figure 3.13 shows, every line of a BASIC program started with a line number. When the program runs, the numbers are used by the `GO TO` statement. For example, line 60 in the sample program contains `GO TO 30`, meaning that when the program gets to this point, it will continue running code on line 30. However, the line numbers are also used for editing. If you load a program like the above and type a line starting with a number, BASIC will put the new line into an appropriate place in the program, possibly replacing existing code. This way, it is possible to use the same simple command prompt for both running commands and editing code. In a way, the prompt serves as an elementary structure editor. It is a very simple one, because the structure of BASIC code is just a list of lines, whereas structure editors in Interlisp or Smalltalk-72 had to let users navigate through a much more complex tree structure.

The authors of BASIC were attempting to make programming that would normally be done in FORTRAN easier. Their design was also influenced by Algol and they

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

referred to BASIC as an 'algebraic language', even though they did not adopt some of the Algol features that were to become influential in the mathematical culture of programming and became recognised as good engineering practices. First, BASIC did not support rich data types which were long established in COBOL and were becoming standard in mathematically minded work on programming. Second, the structure of a program in BASIC is a list of lines, rather than that of nested blocks as in Algol. This makes it harder to use ideas of structured programming and it was one of the reasons that led Edsger Dijkstra, a proponent of the mathematical approach to programming, to condemn BASIC in his 1975 note:

> It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.[104]

The quote is perhaps best interpreted as a clash between the mathematical culture and the hacker cultures of programming. Using the mathematical lens, BASIC programs do not have the elegant structure of Algol programs and the interactive way in which they are created makes them harder to treat as mathematical entities. Yet, to a hacker, BASIC on a microcomputer lets you easily experiment with computers, have fun and (occasionally) also write a program that does something interesting or even useful.

The era of microcomputers came about thanks to a combination of engineering advances that made the microprocessor possible, work of educators who thought that every student should be able to program a computer and hacker efforts to build computers one could cheaply play with. The era of Apple II, Commodore PET and TRS-80 was also perhaps the last time programming was at the centre of using computers.[105] A typical user was a hacker who wanted to explore what can be done with computers and, accordingly, at least one of the operating systems available for each of those machines was the BASIC programming environment. Microcomputers were personal, so nobody needed to introduce a policy that playing Spacewar! is the lowest-priority thing a computer could do. Most users played games and ran other programs, but they were equally interested in fiddling with the machine to see what it could do. This way of looking at computers came to an end when commercially minded programmers realised that useful programs for microcomputers can be built for and sold to non-hackers.

3.15 The Birth of an Industry

At the turn of the 1980s, microcomputers turned from playthings for hackers into commercial tools. This was not just because of the rise of more user-friendly microcomputers and their low cost, but crucially, also because of the development of new ways of using computers. The purchase of a mainframe computer from IBM a decade earlier was a major investment that was typically also associated with the, equally expensive, development of custom software for a specific business task. The low cost

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

of microcomputers would not be so interesting if their purchase also involved the development of expensive custom software. However, new kinds of software, which started to appear at the end of the 1970s, meant that custom software was often not needed. In particular, the growth of the microcomputer industry and its acceptance by business is largely due to the development of end-user software applications for text processing, spreadsheets analysis and database management.[106]

In 1974, Butler Lampson and Charles Simonyi developed the Bravo editor at Xerox PARC for the Alto machine. Bravo was a WYSIWYG (What You See Is What You Get) editor that supported text formatting, fonts and many other features familiar today. Bravo directly influenced Xerox Star, but the work was not known outside Xerox. The first text processor for microcomputers, Electric Pencil, appeared in 1976 without any influence from Bravo. The software was rather rudimentary when compared to Bravo, but it became popular in the microcomputer community and was soon imitated and surpassed by WordStar and later WordPerfect. The capabilities of Bravo were only matched by graphical text processors for Apple Lisa and Apple Macintosh in the mid 1980s.

The case of spreadsheets is more interesting for the history of interactive programming because working with spreadsheets is closer to programming. The first spreadsheet software for microcomputers was VisiCalc, developed by Dan Bricklin and Bob Frankston in 1979. It was initially sold for Apple II and it became the reason why many companies purchased the machine and sometimes even referred to it as the 'VisiCalc machine'. The system was a powerful tool for various financial calculations such as examining alternative budget plans ('what if' problems). VisiCalc owed its debt to the previous generation of interactive computers. Both Bricklin and Frankston worked on the time-sharing system Multics at MIT and Bricklin later worked on word processing software at DEC. As documented by Nooney,[107] 'Bricklin's willingness to develop for a microcomputer rather than a time-sharing system wasn't based in a passion for the hack or a desire to prove anything about the potential of microcomputing at all but in a deliberate set of business decisions largely governed by his most interested potential investor.' VisiCalc was a commercial product of an entrepreneurial spirit. It came with an 'extensive documentation, reference guides, model spreadsheets, and packaging that made users feel like they were dealing with a serious product'.[108]

VisiCalc was not intended as a programming system, but as a specific business tool. The 1983 advertisement for Apple with VisiCalc makes the focus clear: 'Teamed with VisiCalc financial software, an Apple can help anyone make better, smarter, faster business decisions'. Like Bricklin and Frankston, many others who worked on earlier interactive programming projects moved to end-user software development for microcomputers at the turn of the 1980s. David Canfield Smith who created the Pygmalion system joined Xerox to work on the Xerox Star user interface. Here, he developed the idea of icons from Pygmalion into something much more akin to present-day icons in Mac OS or Windows, but in 1983, he moved to VisiCorp, a company selling VisiCalc. Similarly, Charles Simonyi, who created the Bravo document processing system, joined Microsoft in 1981 and started working on a program that would become the first version of Microsoft Word.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The third must-have application for business use of microcomputers was dBase, a database management system first released in 1979. In the first version, dBase allowed users to define a database, manually enter and modify data, but also to write simple data queries. Unlike modern database systems, which operate as services that other applications connect to, dBase was an end-user application. When you defined a database, dBase automatically created a text-based user interface for entering data into the database. Most operations with the dBase database were triggered through commands entered in a simple interactive prompt (much like that of BASIC). Working with data was done through a 'record pointer' that referenced one particular row in the database. You could navigate over the records using commands such as `GOTO`, which jumped to a specified index, or `FIND`, which searched the database for a record containing a certain value.

Both VisiCalc and dBase allowed a limited kind of interactive programming in a narrow domain, but they made the basic kind of programming they supported accessible to a wide range of non-experts. Incidentally, the same remains the case with modern spreadsheet applications like Microsoft Excel. In a way, VisiCalc and dBase are two specific examples of new media that Alan Kay and Adele Goldberg wished for in their 'Personal Dynamic Media' article. Their dream did not come true. VisiCalc and dBase were not created using meta-medium such as Smalltalk that would enable their users modify the systems themselves. They were implemented in assembler and were commercial applications designed for a specific task. This is, finally, where the managerial culture of programming started to shape the programming and use of microcomputers. Not in the sense that VisiCalc and dBase were commercial applications, but in the sense the they were well-defined products that draw a clear distinction between their developers and their users. This distinction is not typical in the hacker and humanistic cultures. For hackers, everyone interacting with a computer should be a hacker. The humanistic visions typically aim to be more inclusive. You start as a user, but can gradually progress and gain access to the capabilities that are available to developers.

A remarkable software application that brought back some of the visions of the humanistic culture is HyperCard, which was released by Apple in 1987 for their Macintosh system. Although HyperCard was an application, rather than a general purpose programming system like Lisp or Smalltalk, it had no fixed application domain. In HyperCard, users created 'stacks of cards' that could contain user interface elements and data. Like modern-day web pages, cards could be connected through links. HyperCard stacks were used and created through the same user interface. The interface displayed the current card and the user could interact with it by clicking on buttons and following links. The interface, however, supported multiple modes (Figure 3.15). In browsing mode, you could not modify the stack; the typing mode allowed you to change text on cards; the painting mode allowed you to change the visual properties of user interface elements; while the authoring mode allowed you to create new controls and specify their functionality through menus, for example, to create a link to another card. Finally, the scripting mode allowed you to specify more complex behaviour using the HyperTalk programming language. The progression from a user to a programmer,

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

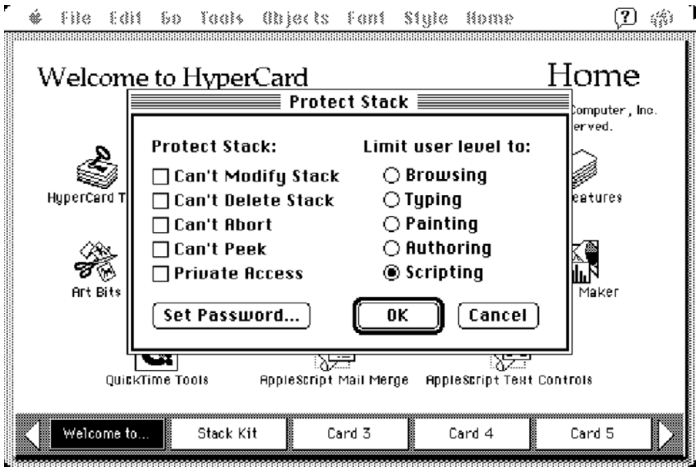


Figure 3.15 Options for protecting the stack and choice of user levels in HyperCard. (Source: Screenshot created by the author, using the Internet Archive HyperCard emulator[109])

which we saw implicit in many systems originating in the humanistic culture of programming, is made explicit in HyperCard. However, HyperCard was also mindful of commercial users in that it allowed them to 'protect the stack' to prevent others from modifying it (Figure 3.15).

Despite being a commercial application like VisiCalc or dBase, many characteristics of HyperCard link it to the humanistic culture of programming, as well as the countercultural origins of interactive computing. The creator of HyperCard, Bill Atkinson, recalled in an interview how the idea for HyperCard came to him while sitting on a park bench at night after taking a dose of LSD:[110]

> Poets, artists, musicians, physicists, chemists, biologists, mathematicians, and economists all have separate pools of knowledge, but are hindered from sharing and finding the deeper connections. …How could I help? By focusing on the weak link. …It occurred to me the weak link for the Blue Marble team is wisdom. Humanity has achieved sufficient technological power to change the course of life and the entire global ecosystem, but we seem to lack the perspective to choose wisely between alternative futures. …I thought if we could encourage sharing of ideas between different areas of knowledge, perhaps more of the bigger picture would emerge, and eventually more wisdom might develop. …This was the underlying inspiration for HyperCard, a multimedia authoring environment that empowered non-programmers to share ideas using new interactive media called HyperCard stacks.

The motivation for HyperCard resembles motivations that we saw earlier in this chapter, be it the man–machine symbiosis of J. C. R. Licklider or augmenting human intellect of Douglas Engelbart. Atkinson wanted to let people passionate about any subject, who are not programmers, use computers to their full potential. HyperCard maybe did not change how the 'Blue Marble team' manages wisdom, but it succeeded in the goal of empowering non-programmers to share ideas. It was surrounded by an enthusiastic community that used it for a wide range of projects, ranging from

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

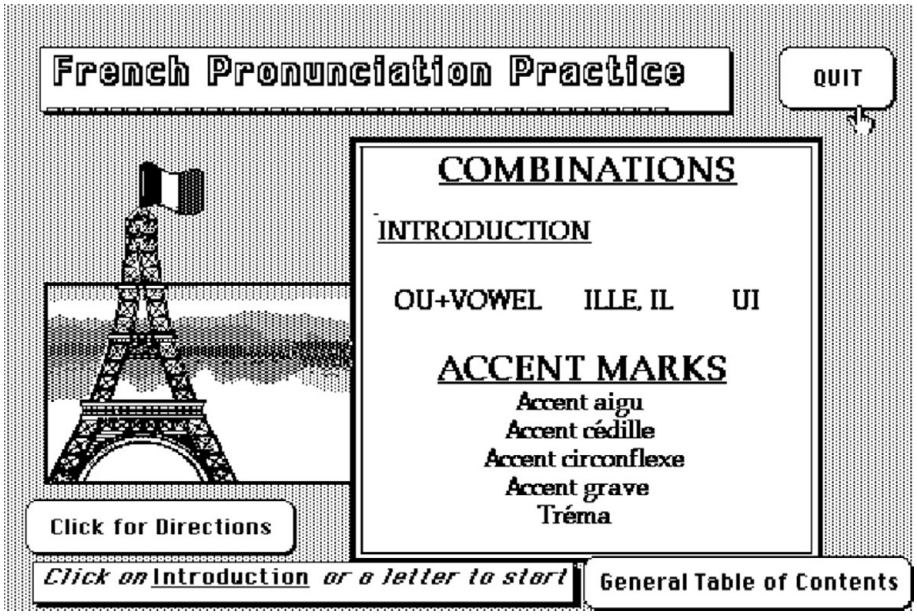


Figure 3.16 French Pronunciation stack created by Charles W. Gidney.
(Source: Screenshot created by the author, using the Internet Archive HyperCard emulator[111])

creative uses like interactive games, educational materials (Figure 3.16) and art pieces to commercial applications like databases and control systems. HyperCard gave users much more flexibility than VisiCalc or dBase and was perhaps closer to systems like Boxer. Yet, the user still had to operate within the constraints of the stacks of cards format.

3.16 Show Us Your Screens!

In the late 1970s, microcomputers became a new platform for work on interactive programming. They provided an impetus for the development of the BASIC programming language and enabled the birth of commercial software. The same technology also supported new creative uses of computers in the art scene. As was the case with the early computer art which influenced the educational Logo language, computer art done using microcomputers also later influenced interactive programming tools for education. This time though, we need to look at computers and music.

In the 1970s, a number of musicians from the experimental music scene in Oakland started to incorporate microcomputers in their work. Many of them were already building their own electronic systems, but microcomputers made it possible for the electronic system to become 'a musical actor, as opposed to merely a tool'.[112] The community started using the KIM-1 microcomputer, which had similar capabilities as the Altair 8800. KIM-1 was a single printed circuit board with a 6-digit display and

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

23 buttons. It was intended as a development board for a new MOS 6502 microprocessor. It became popular with hobbyists, partly because its expansion connectors allowed it to connect to devices such as lights, switches and speakers. Several experimental musicians used KIM-1 to generate early algorithmic music, but a more radical use of the machine was to directly control musical devices. Moreover, you could also connect multiple KIMs in a way where one player could respond to the actions of another. By 1978, an Oakland group started performing using KIMs under the name 'The League of Automatic Music Composers'.

The group played for several years, but they soon realised that they had to make complicated ad hoc connections between the individual computers each time they played (Figure 3.17). As one of the group members recalls, 'this made for a system with rich and varied behaviour, but it was prone to failure, and bringing in other players was difficult'.[113] To 'clean up the mess', the group designed a new architecture where a central microcomputer, The Hub, was used for passing messages between the individual players connected to it. The architecture gave name to a reborn computer

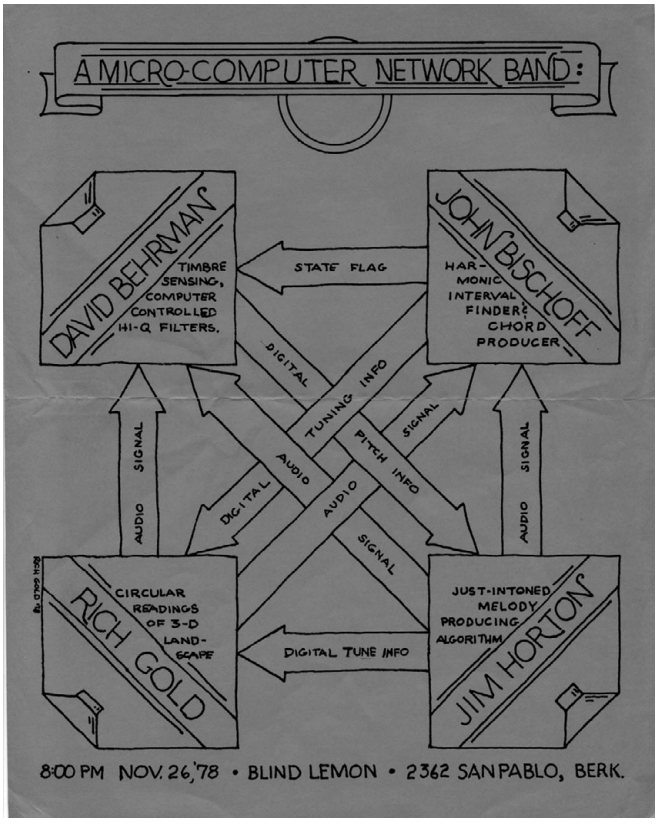


Figure 3.17 Flyer from the League's Blind Lemon concert of Nov. 26, 1978 featuring diagram of League topology. Designed and drawn by Rich Gold. (Source: Courtesy of Rich Gold. Reproduced with permission of Marina deBellagente LaPalma)

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

music group, The Hub.[114] Their performances evolved in many ways over the next decade. They adopted the MIDI communication protocol, when it appeared at the end of the 1980s. They also started using computer screens that the audience could view to see how the performers interact with their instruments, which was a logical next step to showing the bare KIM-1 circuit boards that were used in the early performances.

The history of computer music meets with the history of interactive programming in a number of ways. The Hub was a part of the same counterculture movement as the Homebrew Computer Club. Some of the early programmable music synthesis systems used the Lisp language for its interactivity. At a more fundamental level, using a computer as a music instrument is only possible through an interactive programming system that supports immediate program execution.

At the turn of the millennium, a new generation of computer musicians appeared. As more powerful computers became available, their focus shifted to controlling music composition by interactively writing code in high-level programming languages, an approach that became known as live coding. During a live coding performance, musicians type code on a computer and run it interactively. The performers also typically project their screens so that the audience can see their work and they may also use code to control visual effects on screen. Incidentally, there is an interesting parallel between the development of live coding, which originates in the humanistic culture of programming, and the development of Agile methodologies in the engineering culture at the same time, which I discuss in Chapter 4.[115] In particular, the idea of sharing code on screen with the audience is similar to the idea of pair programming where two programmers work on a single machine, one typing code and one watching, commenting and reviewing code.

In 2004, the live coding community came together to form TOPLAP (Temporary|Transnational|Terrestrial|Transdimensional) Organisation for the (Promotion|Proliferation|Permanence|Purity) of Live (Algorithm|Audio|Art|Artistic) Programming. The TOPLAP manifesto[116] exhibits many of the aspects of the humanistic culture of programming. It puts code at the centre with the slogan 'Obscurantism is dangerous. Show us your screens'. Unlike some of the earlier works originating in the humanistic culture, it takes a more realistic view on readability of code and acknowledges that 'it is not necessary for a lay audience to understand the code to appreciate it'. The organisation also emphasises social aspects of live coding and the statement, 'live coding is inclusive and accessible to all', resonates with Nake's 1971 criticism of computer art. As with the 1950s visual computer art, one of the ways in which live coding benefited society is through work on education.

The system that made the use of live coding in education possible is Sonic Pi created by Sam Aaron.[117] It was released in 2012 and was initially built to teach programming and music in schools using the low-cost Raspberry Pi computer. When using Sonic Pi, performers write code in the Ruby programming language to instruct a synthesiser to play notes and samples. They can select and immediately run portions of the code to play their music. Almost 50 years after the appearance of Logo, the Sonic Pi system shares a number of important principles that characterised Logo. First, Sonic Pi is designed for both schoolchildren and professional musicians, which mirrors the aim

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

of Logo to bring the 'most powerful programming language available', that is, Lisp, to everyone. Second, Sonic Pi and live coding more generally adopt a liberal approach to errors. The performers recognise that in 'musical genres that are not notated so closely …, there are no wrong notes - only notes that are more or less appropriate to the performance'. This means that live coders 'may well prefer to accept the results of an imperfect execution'; they might 'even accept the result as a serendipitous alternative to the original note'.[118] As in Logo, errors are seen as sources of serendipity that provide an opportunity for learning.

Live coding is clearly inspired by earlier work on computer art and so it originates in the humanistic culture of programming. The idea has been, however, influenced by other cultures of programming. The hacker culture influence was prevalent in the early computer music work where performers built their own instruments, following the do-it-yourself mantra of the 1960s counterculture. The extent to which live coding is influenced by mathematical culture is open to debate. On the one hand, the importance of algorithms, a cornerstone of the mathematical culture, is made clear in the TOPLAP manifesto which 'recognises continuums of interaction and profundity, but prefers insight into algorithms'. On the other hand, live coders are performers who use the computer as a musical instrument rather than to construct linguistic entities. A typical performance using a system like Sonic Pi involves programming concepts like loops and processes, but does not typically involve the programming of any algorithms.

In this, programming education using live coding dissents from systems that teach concepts based on the academically dominant mathematical culture. The most prominent example of the latter is the visual programming language for education Scratch, which was developed at MIT Media Lab in the early 1990s. Scratch was inspired by Logo and evolved from the idea of 'visual Logo'. It lets children create graphical programs like games, but it fully adopts the idea of programming as the construction of linguistic entities. Programs in Scratch are constructed visually by arranging blocks, which makes it more accessible to children, but the thinking that it encourages is rooted in the mathematical culture of programming. The difference between Sonic Pi and Scratch illustrates the importance of recognising different cultures of programming. The notion of computational literacy will differ significantly, depending on which culture of programming is advocating for it.

3.17 Reinventions and Adaptations

Looking at the history, we can broadly identify two ways of thinking about programming. The first one is to treat it as a process of constructing textual programs, which are created, modified, formally analysed and then compiled and run. The second is to see programming as interacting with a programmable medium. The two ways of thinking do not result in completely different technologies. A mathematical programming language like Algol also exists as part of an environment comprising editors, tools and compilers that the programmer interacts with, while interactive programming systems like Smalltalk or Interlisp still have a language at their centre.[119] The two ways of

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

thinking, however, determine what the programmers focus on when creating or using the technology.

The two perspectives are noteworthy for my story because each of them is favoured by different cultures of programming. On the one hand, the mathematical culture builds around the idea of treating programs as entities written in a formal language. On the other hand, hacker culture and humanistic culture often rely on treating programming as interaction with a machine or a medium, respectively. To hackers, the interactive way of programming offers direct access to the computer and lets them fully utilise the machine in ingenious, unrestricted ways. To artists, visionaries and educators, the interactive way of programming emphasises the crucial ongoing process, be it exploration, learning or communication with a machine.

When discussing the mathematisation of programming in Chapter 2, it was possible to see direct historical influences and follow the development of the idea and concepts that emerged from it. I did not try to document every single influence and concept, but it was possible to see a continuity, both in the community and in the ideas that it developed. One reason for this is that the hotbed of the mathematical view was in academia, which publishes work as papers and books with references that can be easily followed. Another reason is that the mathematical treatment of programs is a very general idea, not linked to a particular machine or application. This means that it can continually develop even when machines, systems and programming languages change.

The development of interactive programming does not follow such a clear historical line. Work on interactive programming often results in code, hacks, memos and demos rather than publications with references. This complicates the tracing of direct influences. The progression from MIT hackers to the ARPA community and Xerox PARC is an exception from the rule, because there is a well-documented continuity in the community. The same cannot be said about the emergence of the new generation of microcomputer hackers, who share many values with the early MIT hackers, but do not form a part of the same community. Consequently, many ideas of interactive programming reappear, likely independently, in different contexts. Their presentation may differ, but the core ideas have surprisingly much in common. For example, viewing a computer as a programmable dynamic medium, which appeared in the humanistic vision around Smalltalk is not a million miles away from the interactive programming of microcomputers using BASIC, which served for a time as a primary medium for interacting with the machine.

The interactive approach to programming often reappears and evolves when the context in which programming is done changes. Many of the developments that I looked at in this chapter were caused by new hardware or system developments. TX-0 gave the MIT hackers a direct access to the machine, which led to the development of new online debugging tools like UT3 while time-sharing provided a new kind of direct interaction and motivated the development of the LISP editor, which made program modification on a teletype manageable. New contexts that inspire ideas on interactive programming can also be new application domains. Two examples are programming in the educational context and programming as a tool for computer music.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Interestingly, the interactive style of programming found its place even within the mathematical culture. This happened in the early 1990s in the context of interactive theorem proving. Here, the programmer first defines a theorem they want to prove and then enters commands that transform the goal. After each command, the programmer can review the current state of the system such as remaining sub-goals. Interactive theorem provers can be used for proving mathematical theorems, but thanks to an equivalence between types and propositions that I return to in Chapter 5, interactive programming using a theorem prover can also be used to construct programs.

What is striking about the many reappearances of interactive programming in different contexts is that the different incarnations of the interactive style of programming often follow a similar historical pattern. They appear in the creative, humanistic or hacker culture. They are used by hackers to explore new ways of using a computer, visionaries to imagine new ways of thinking using computers and artists to pursue creative goals. Along the way, they produce various ancillary ideas and programming techniques that are interesting on their own and that appeal to more practically minded engineers and managers. To satisfy engineering and commercial interests, the ancillary ideas are often stripped away from their original context and they develop independently. The examples of this pattern are ubiquitous in this chapter. Xerox Star adopted the user interface of Smalltalk and icons from Pygmalion, but removed the underlying programmable medium, while microcomputers stopped booting into an interactive BASIC console and started running spreadsheet and word processing applications.

In other words, each reappearance of the idea of interactive programming enriched the state of the art of programming with new concepts, ranging from debugging tools and structure editors to graphical interfaces and spreadsheets. However, behind those interesting new concepts often lie more fundamental visions and ideas such as those of man–machine symbiosis, augmenting human intellect or programmable meta-media. Those provide inspiration or conceptual framework for the specific technical developments, but those fundamental ideas are often lost as specific new technical concepts become established and find use in everyday commercial programming.

This chapter and Chapter 2 have looked at developments that pursue somewhat idealistic visions for programming. However, both the mathematical perspective on programming and the humanistic and hacker ideals of interactive programming lead to a certain reservation towards practical programming as done in industry. In Chapter 4, I follow developments that were more directly concerned with practical software developments. They made programming more accessible and reliable, but it happened through other means than those advocated by the mathematical, hacker and humanistic cultures.

Notes

1 The term ‘hacker’ here is used to refer to a programmer-hacker, a sub-culture that is a part of the computing folklore documented by Levy (2010) and Tozzi (2017), rather than to security-hackers. The two cannot, however, be fully separated as programmer-hackers were open to violating security for (what they saw) as non-malicious goals.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

2 Programming of Whirlwind was documented by Fedorkow (2021), as part of an effort to recover some of the original Whirlwind software artifacts.
3 For the connections with counterculture, see Markoff (2005) and Turner (2010). The story of the ARPA Information Processing Techniques Office (IPTO) community, which has gained almost mythical status among some programmers, has been told by Waldrop (2001) and Hiltzik et al. (1999).
4 The notion, known as *advising* was developed by Teitelman (1966) in the PILOT system.
5 A phrase used as the title of a speculative essay by Kay (1972b); the motivation has also been described by Kay (1996) in a retrospective article on the history of Smalltalk.
6 The example is based on that given in Kay and Goldberg (1977).
7 Kay (1972b).
8 https://github.com/livingcomputermuseum/Darkstar.
9 Charles 'Chuck' P. Thacker who worked at Xerox at the time used the term 'biggerism' to describe unnecessarily complicated technology (Hiltzik et al., 1999).
10 Montfort et al. (2014) documents some of the early history and illustrates the creative spirit of the community around microcomputers through reflections on a single BASIC program `10 PRINT CHR$(205.5+RND(1)); : GOTO 10`.
11 Halvorson (2020) tells the story of microcomputers, BASIC and how those contributed to the movements to democratise programming.
12 Dijkstra (1982).
13 Also known as REPL (Read-Eval-Print loop), named because the shell repeatedly reads code as input, evaluates it and prints the result.
14 For this view on interactive programming, see Blackwell and Collins (2005).
15 Kotok (2005).
16 www.flickr.com/photos/24226200@N00/364960084/.
17 As recollected in oral histories by Greenblatt (2005); Russell (2017); Samson (2017); Kotok (2005).
18 Campbell-Kelly (2004) discusses the broader context of home and recreational software that followed early games like Spacewar!.
19 The hacker ethic and culture has been described by Levy (2010). This is a canonical, but a naively optimistic reference. Kelty (2008) describes the stories of early hackers as avowedly Edenic and not reflecting inevitable commercial constraints and Tozzi (2017) also provides a more critical perspective. As pointed out by Ensmenger (2015), the idealised view of Levy (2010) and Brand and Crandall (1988) also helped to establish the masculine 'stereotype of the bearded, besandalled computer programmer' at the time when 'in actual practice women were still very much present in most corporate computer departments'.
20 Forrester and Everett (1990); broader history of Whirlwind has been told by Redmond and Smith (2000) and Ornstein (2002) provides a personal account of Whirlwind programming.
21 The SAGE system is at the start of a long history of defence software, followed by Slayton (2013), that shaped the public debate about nuclear defence. The complexity of SAGE is illustrated by Ensmenger (2012) who points out that in 1956, the company developing software for SAGE employed three-fifths of all programmers in the USA and went on to double the number of programmers in the country over the next few years.
22 McKenzie (1974).
23 Gilmore (1958a,b).
24 The 'dynamic flow chart program' is described in 'The Computer Museum Report', Volume 8, 1984 available at www.bitsavers.org/pdf/mit/tx-0/TX-0_history_1984.txt, retrieved May 21, 2022.
25 Samson (2017).
26 The topic of code aesthetic from the perspective of different groups that partly overlap with the cultures of programming is discussed by Depaz (2023).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

27 Beeler et al. (1972).
28 The technical details of Logic Theorist have been described by Gugerty (2006). **?** discusses broader history of proof automation and identifies two strands of work. Logic Theorist aimed to simulate how humans reason, while other systems aimed to use the most efficient computer solution.
29 Lisp is often described as 'the first functional programming language', but as pointed out by Priestley (2017), who documents the origins of Lisp, this is misleading and simplistic. Personal recollections by McCarthy (1978) also refrain from making such claims.
30 Levy (2010).
31 Tan (2020) considers the work of Ada Lovelace through the framework of poetics and uses this as a starting point for arguing for a more profound understanding of code.
32 The various influences of the two essays are documented by Waldrop (2001) who also, in many ways, illustrates the techno-optimism of the humanistic culture of programming.
33 Bush (1945).
34 Licklider (1960).
35 Slayton (2013) discusses the work of Licklider in the context of military command and control. Despite actively contributing to this aspect of military research, Licklider was sceptical of automatic anti-ballistic missile systems that excluded human interaction.
36 Licklider (1965).
37 Sutherland (1966).
38 Sutherland (1966).
39 GRAIL was described by Ellis et al. (1969). Visual programming languages continues to be an active research field today and is frequently aligned with the humanistic culture, such as its focus on education. An example is the block-based visual language Scratch developed by Resnick et al. (2009).
40 November (2004).
41 McCarthy (1992).
42 Quoted in Waldrop (2001).
43 Quoted in Waldrop (2001).
44 Documented in Waldrop (2001).
45 Fox (1960); McCarthy (1978).
46 Deutsch (1967).
47 The code is based on an example given by Deutsch (1967), but is reformatted, corrected and adapted for modern Lisp systems.
48 The report (Deutsch, 1967) does not seem to be cited by any of the later work on structure editors. However, Deutsch published a paper 'An online editor' (Deutsch and Lampson, 1967) about a general-purpose text editor that resembles the LISP editor and is more widely known.
49 Teitelman (1966).
50 Gabriel (2012), who reflects on the history of the schism, suggests to view it as two scientific paradigms. I would rather associate the 'systems' and 'languages' views with the hacker and the mathematical cultures of programming, respectively.
51 Documented in the Interlisp manual (Teitelman, 1974) and in a paper on the history of Lisp (Steele and Gabriel, 1996b).
52 Bawden et al. (1974).
53 Quoted in the humorous hacker's 'Jargon file' that was maintained by the hacker community in various formats at MIT and Stanford and was later published (Raymond, 1996).
54 For the first-hand account of the history, see the work by Steele and Gabriel (1996b).
55 A book by Schank and Riesbeck (1981) illustrates this approach by collecting 'miniature' versions of AI programs in order to explain them. For a reflection on this way of thinking

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

about computer programs, see also the work by Sack (2022). I am grateful to Richard P. Gabriel for suggesting this interpretation of the history of Scheme.

56 According to Steele and Gabriel (1996b).
57 Noll (2016).
58 Nake (1971).
59 This interpretation of the interests of the two researchers is due to Solomon et al. (2020).
60 Solomon et al. (2020).
61 Papert (1980).
62 Papert (1980).
63 This example is given by Solomon et al. (2020).
64 https://github.com/drj11/curlylogo.
65 Solomon et al. (2020).
66 Many of those are documented, for example, by Ensmenger (2012); Turner (2010); Abbate (2012); Hicks (2017).
67 Solomon et al. (2020).
68 Parts of the story of Doug Engelbart have been told by many, including Rheingold (2000); Hiltzik et al. (1999); Waldrop (2001) and perhaps most comprehensively by Bardini (2000).
69 Engelbart (1962).
70 Hiltzik et al. (1999).
71 Quoted in Hiltzik et al. (1999).
72 https://commons.wikimedia.org/wiki/File:Arpanet_1972_Map.png.
73 Barber (1975).
74 Waldrop (2001).
75 For the former, see Waldrop (2001); Hiltzik et al. (1999); for the latter, see Markoff (2005); Turner (2010).
76 Brand (1972).
77 As documented by Markoff (2005).
78 Hiltzik et al. (1999).
79 Kay and Goldberg (1977).
80 Beeler et al. (1972).
81 Documented by Ensmenger (2012).
82 This envisioned progression was made more explicit later (Reenskaug, 1981).
83 Some of the motivations are discussed in the internal 'Why Alto' note (Lampson, 1972).
84 Kay and Goldberg (1977).
85 https://smalltalkzoo.thechm.org/.
86 Hiltzik et al. (1999); Kay (1996) himself also acknowledges that the children selected from Palo Alto schools were of 'hardly an average background'.
87 Goldberg and Kay (1976a).
88 Smith (1977).
89 Di Sessa (1985); diSessa and Abelson (1986).
90 Gombrich (1961).
91 Smith (1977).
92 A term introduced by Lennon (2019).
93 The design has been documented by Johnson et al. (1989).
94 According to an interpretation by Hiltzik et al. (1999).
95 Damouth et al. (1971).
96 This history is a part of broader context of the development of a personal computer (Ceruzzi, 2003).
97 www.flickr.com/photos/acidhelm/39802981903.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

98 Interactive programming is a technological link between Engelbart's NLS, Kay's Smalltalk and microcomputers. Their role in the countercultural movement provides another link (Turner, 2010).
99 From 1975 issue of the People's Computer Company, quoted by Turner (2010).
100 The development of BASIC and the now-legendary letter are discussed by Freiberger and Swaine (1984); Ceruzzi (2003); Montfort et al. (2014).
101 Rankin (2018).
102 www.bitsavers.org/pdf/dartmouth/BASIC_Oct64.pdf.
103 www.classic-computers.org.nz/blog/2011-07-04-pet-rom-upgrades.htm.
104 Known as EWD498 (Dijkstra, 1982).
105 Possibly with the exception of the era of the 1990s web programming when many web users were also experimenting with creating their own websites and they were able to learn from others through the 'view source' functionality of their web browsers.
106 For a more detailed account of the birth of the personal computing industry, see the excellent account by Nooney (2023) and also by Ceruzzi (2003).
107 Nooney (2023); the history of VisiCalc has also been told by Grad (2007).
108 Nooney (2023).
109 https://archive.org/details/hypercardstacks.
110 Atkinson (2016).
111 https://archive.org/details/hypercard_french_pronunciation.
112 Chris Brown and John Bischoff (2002).
113 Chris Brown and John Bischoff (2002).
114 The history of The Hub has been documented by Gresham-Lancaster (1998) and Brümmer (2021).
115 The parallel has been pointed out by Zmölnig and Eckel (2007).
116 Ward et al. (2004).
117 Aaron (2016).
118 Blackwell and Collins (2005).
119 For a more detailed analysis of the difference between the programming language perspective and the programming systems perspective, see also Jakubovic et al. (2023); Gabriel (2012).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

4 Software Engineering

Teacher: We talked about mathematical methods and interactive programming and both times we concluded that they are not used as often as they should be. So, how do programmers make sure all the software that is around us actually works?

Pythagoras: Looking at the industry practices, the answer is a mix that includes debugging, testing and quality assurance, rigorous management processes and over-engineering.[1]

Teacher: I propose we start by going through this list in a historical order. What were the first industry practices that helped us produce more reliable systems?

Diogenes: Well, debugging got a lot easier thanks to the birth of time-sharing and interactive computing at the end of the 1960s, because you could explore the state of a program while it was running, but primitive forms of debugging go back to the first digital computers. On EDSAC, you could debug programs by running them instruction-by-instruction and inspecting the state of a part of a memory on a CRT screen...

Xenophon: Such a waste of valuable computer time! It's not surprising that this use of digital computers was soon banned on most installations.

Archimedes: Well, one interesting consequence of the limitation was that it inspired programmers to come up with new approaches. It resulted in the first debugging tools that we use to this day. In the case of EDSAC, those were postmortem dumps and interpreters that printed diagnostic information while running the program.

Pythagoras: I was wondering about the early days of debugging, but it is practically impossible to find any academic literature on debugging techniques!

Diogenes: The debugging tools built at MIT for TX-0 and PDP-1, called FLIT and DDT, are documented well enough in memos published by the MIT Lincoln Laboratory. An academic paper about them would not be very useful. They were clever, but simple tools and hackers learned how to use them through practice...

Xenophon: I see. The black art of programming strikes again! When did debugging turn into a more rigorous discipline?

Archimedes: People started taking debugging more seriously in the 1960s. Once hardware got reliable and high-level programming languages made it possible to write more complex programs, people realised that the number of bugs in programs became the main limiting factor for the use of computers.[2]

Pythagoras: Very well, but what novel debugging capabilities did this new focus give us?

Diogenes: Good question. The focus may have changed, but the debugging tools I was talking about were quite powerful already. You could step through instructions, set breakpoints and even patch your program on the fly. This worked for assembly at first, but in the mid 1960s, similar tools started to appear for Lisp too.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Pythagoras: This sounds like the debugging tools we have today. See? This lack of progress is surely the consequence of hiding all the information in internal memos and not advancing knowledge gradually through academic publications!

Archimedes: Well, modern debugging tools are more sophisticated. They use efficient incremental compilation rather than interpretation and reverse or time travel debugging makes it possible to step back in time. But the primary improvement is a greater conceptual clarity. In the 1960s, debugging became clearly distinct from testing. As a result, debugging and testing tools started to evolve independently.

Xenophon: I do not see how anyone could confuse the black art of debugging with testing as understood in rigorous software engineering. Around the time of the 1968 NATO conference, testing became a part of the software development process. It was used to certify that a system has been implemented according to requirements of the customer. How comprehensive testing is required was also agreed with the customer.

Pythagoras: Right, but at the NATO conference, Dijkstra also correctly pointed out that testing shows the presence of bugs, never their absence![3]

Archimedes: Full correctness was not the aim. It was a step before handing the system off to a customer and it led to the development of the first automated testing tools in the 1970s. You cannot deny those give you some assurance about software correctness!

Pythagoras: Well, the automation inspired interesting mathematical analyses. One looked how to generate good input data for tests, while another developed a probabilistic theory based on how much of the code was covered by tests.[4] Mind you, this still gives you weaker guarantees than formal verification, but at least it is a rigorous analysis.

Teacher: Let me observe that we seem to have much more to say about testing than about debugging. I wonder why that is the case...

Socrates: It seems to me that a 'test' is an entity that can be shared by multiple people, even if it means different things to them. The consequence is that different cultures of programming develop the idea in different directions. You can find a meaning for 'test' even in my favourite domain of live coding. There, it is different still and is more akin to practice before an art performance!

Archimedes: And we did not even get to the most interesting development which followed when testing turned from a phase in a managerial process into an engineering tool in Test-Driven Development (TDD).

Pythagoras: Isn't TDD using tests just as a poor substitute for formal verification?

Archimedes: No, no, no! In modern use, tests capture the requirements of the customer and serve as a lightweight specification. As a methodology, TDD is a kind of scientific method for building software. You repeatedly add a test and run it (i.e., run an experiment) and then you revise your theory (i.e., your program), until the test passes.[5]

Pythagoras: But using tests as your specification is the wrong way round! You should write a test based on your specification instead.

Archimedes: Except that, in practice, you never have a detailed specification upfront. Writing tests forces you to think about the right behaviour and discuss it with your colleagues and business domain experts. Tests capture the resulting knowledge.

Socrates: Interesting! If tests are something that you discuss with your colleagues and customers, then they provide a focal point for valuable social processes. Exactly the kind of social processes that proofs of program correctness were lacking!

Xenophon: When you put it in this way, I can see how TDD gives you an appealing light-weight software development process. That said, if you have enough time and resources, you should still produce a detailed specification up-front to avoid surprises!

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Diogenes: Do you suggest we should be using the Waterfall development process? I do not understand how anyone was able to build any software this way!

Xenophon: You are thinking of a caricature of the process. Structured methodologies that divided the process into phases like requirements-gathering, analysis, development and testing were used by the military already in the 1950s. People were also continually improving the process. For example, the oft-cited 1970 paper by Winston Royce[6] is not about Waterfall itself, but about making the process more adaptive.

Socrates: I'm curious how we got from these methodologies to something like the Test-Driven Development. Did Waterfall just get out of fashion?

Xenophon: It worked well enough in the 1970s and 1980s, but in the 1990s, the PC market and the rise of the Internet changed what kind of applications companies needed. It was crucial to build software faster and adapt to changing market requirements.

Diogenes: Is this where the Agile methodologies come from?

Archimedes: It was a more gradual process. Extreme Programming and SCRUM were two methodologies that tried to address the situation, but you are right that the term Agile is the most famous one. The Agile manifesto, written by 17 thought-leaders in 2001 gave those methodologies a common name and popularised them.

Socrates: It is sad that it takes 17 men to re-invent something that has been described more clearly by a woman over a decade earlier!

Archimedes: What do you mean? Please tell us more.

Socrates: In 1987, Christiane Floyd described software engineering methodologies inspired by the Scandinavian idea of 'participatory design' where you work closely with users during a product design.[7] She contrasted the dominant product-oriented approach with an emerging process-oriented approach. The latter is based on a close collaboration with the customer, learning and building from a shared understanding.

Archimedes: That does, indeed, resemble the key ideas of the Agile movement!

Teacher: It seems we have covered testing, debugging and development processes already, so what else was on our list of real-world engineering practices?

Pythagoras: We could talk about exception handling, but that would be missing the forest for the trees. Isn't there some more fundamental theory of software engineering?

Archimedes: Software development is inevitably a human task. You cannot hide the inherent complexity and make simplifying claims. That said, you can identify common patterns, characterise different classes of programs and their general properties and also make rigorous analytical arguments grounded in the understanding of the relationship between the human world and software.[8]

Xenophon: Do you mean observations like Brooks' law, which states that 'adding manpower to a later software project only makes it later'? In my experience, that is the case but I do not have enough data to prove this with certainty![9]

Archimedes: Right, but Brooks' argument does not rely on experience. He explains the 'law' in terms of communication overhead. Look at the two charts in Figure 4.1. If you have a task that can be perfectly divided among team members, the amount of time needed decreases proportionally with the number of people. But programming complex systems requires communication between team members and, in general, you need each member to talk to every other member. So you have to add an overhead of $n*(n-1)$. This is quadratic and so, even for a small amount of communication, it eventually dominates the gains you get by dividing the work.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

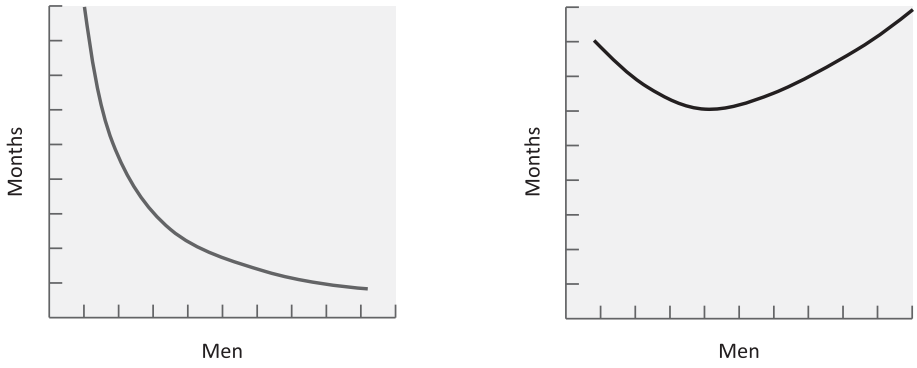


Figure 4.1 Time versus the number of workers for a 'perfectly partitionable task' (left) and a 'task with complex inter-relationships' (right).[10] (Source: Drawing by the author, labels as in the original)

Xenophon: Do you suggest to measure the number of minutes needed for programming and communicating and then compare those based on the team size? That sounds interesting, but I doubt the numbers would generalise well across teams and projects...

Socrates: You do not need concrete numbers. This is a very nice argument, because the logic is convincing without concrete numbers. I wonder what other problems have been studied using this kind of reasoning?

Archimedes: Another example is from the debate about anti-ballistic missile systems (ABMs).[11] In 1969, Joseph Weizenbaum argued that ABM cannot reliably be built because the rate of the change in the environment is greater than the rate of the change we can maintain in the software system. This also relies on a sort of mathematical argument that is convincing enough, even without specific quantities.

Pythagoras: As far as I'm concerned, this is exactly the kind of software that should have a formal specification and a proof that the implementation conforms to it!

Xenophon: That is unrealistic and cannot work in practice. You need a solid process to ensure that system will have the characteristics it needs.

Teacher: We have drifted to the problem of trusting software. What do others think?

Diogenes: If you want to hear my position, then I would like to remind you that software is ultimately built by people. I only trust programs created by people that I can trust.

Socrates: Do not forget, though, that software is written by people for people. I only trust software that is trying to do the right thing, because you can never separate the technical from the social. Getting the social-side of software right has become the limiting factor these days!

4.1 How Did Software Get So Reliable without Proof?

In March 1996, the International Symposium of Formal Methods Europe took place in Oxford. The symposium evolved from a series of events focused on the Vienna Development Method (VDM), a formal method that started with the work on semantics of programming languages in the IBM Vienna lab at the end of the 1960s. The organisers struck a positive note about the industrial applicability of formal methods and a quarter of the presentations at the symposium were reports on industrial use of formal methods.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The opening keynote was presented by a veteran of the field, C. A. R. Hoare, who developed foundational theories for reasoning about programs that we encountered in Chapter 2. In his talk, 'How Did Software Get So Reliable Without Proof?' Hoare asks a question that may have rightfully puzzled many proponents of the mathematical culture.

The talk started with a reference to an analysis by Donald MacKenzie who attempted to determine how many people had died in computer-related accidents prior to 1992. Only about 30 deaths out of the collected dataset of 1,100 were caused by a computer bug. Of those, 28 were the consequence of a faulty software in the Patriot missile system. The majority of the deaths were instead caused by 'faulty human–computer interaction' such as 'poorly designed interfaces' or 'organisational failings'.[12] Hoare acknowledged that 'formal methods and proof play a small direct role in large-scale programming' and reflected on the situation:

> Twenty years ago it was reasonable to predict that the size and ambition of software products would be severely limited by the unreliability of their component programs. … Fortunately, the problem of program correctness has turned out to be far less serious than predicted. … Similarly predictions of collapse of software due to size have been falsified by continuous operation of real-time [telecommunications] systems. So [a question arises]: why have twenty years of pessimistic predictions been falsified?[13]

Hoare's answer is that software gets reliable through a mix of practical engineering practices, including good project management, rigorous testing, debugging, over-engineering and innovations in programming methodology. As one might expect from a proponent of the mathematical culture, Hoare does not see this as a failure of formal methods. Instead, he argues that many such practices owe much to the theoretical ideas developed 20 years prior to his paper. Formal methods and proof may play a small direct role in large-scale programming, but 'they do provide a conceptual framework and basic understanding to promote the best of current practice, and point directions for future improvement'.

Despite the title, Hoare's keynote was an optimistic one. He concluded that the 20-year gap between theory and practice that he identified 'is actually an extremely good sign of the maturity and good health of our discipline' where basic research conducted by theorists provides fundamental basic understanding and where industry adapts and adopts useful theoretical innovations such as structured programming, informal mathematical reasoning, data types and information hiding.

Hoare's keynote paints an idealised image of a gap between theory, which explores new directions for advancing the state of the art, and practice, which adopts proven research ideas. Using the perspective of cultures of programming that I follow in this book, Hoare would see the mathematical culture of programming as the primary source of new scientific ideas and the engineering and managerial cultures as those who apply scientific ideas in the real world. As we saw when looking at the origins of programming languages in Chapter 2, and as we will repeatedly see in the chapters to come, the interactions between cultures are not so simple. An idea may emerge in any culture of programming, it may be influenced and reshaped by ideas from other cultures and even return back in a different form. Much like the history of programming languages

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

and interactive programming, the history of testing, debugging, error handling and development methodologies that I follow in this chapter is full of such interesting and unexpected twists.

4.2 The Art of Electronic Computer Maintenance

The idea that programming a computer will become a major challenge was completely unexpected when the first digital electronic computers were created. To a contemporary programmer, this may seem naive, but it is less surprising if we recognise that digital electronic computers did not appear out of the blue in the 1940s. In many ways, they were the next evolutionary step from analog computers which used mechanical, hydraulic or electrical quantities to model numerical problems. Analog computers did not require sophisticated programming and their operation was often fully transparent. A wonderful late example of this idea is the hydraulic computer that models the economy, which was created in 1949 and was called MONIAC, 'to suggest money, the ENIAC and something maniacal'.[14] In MONIAC, the flow of money in the economy is modelled as a flow of coloured water. Various mechanisms in the economy are modelled using valves controlled by floats, pulleys and other hydraulic devices as shown in Figure 4.2. The machine had a limited accuracy, but it was unprecedented in aiding the understanding of complex ideas of economics.[15] The user of the MONIAC can always see the full state of the system and noticing that the machine (mis)behaves is just a matter of watching how the water flows through the system.

Electronic analog computers removed this kind of transparency, but kept the simple fixed structure where a small change in the input leads only to a small change in the output. The last traces of this simplicity were finally lost with the advent of digital computers. The operation of digital computers is not just hidden from sight, but it also involves complex 'evolution of a meaning'.[16]

In spite of this complexity, the job of an operator responsible for setting up the computations on the early digital computers was thought of as relatively mundane. Many of the original computer operators of the ENIAC computer were women, recruited from a group of human computers hired to calculate ballistics tables by hand. The setting up of the ENIAC was, at first, done using switches and by connecting individual components of the machine using cables. The operators of the ENIAC had to figure out how to translate a more high-level mathematical plan of a computation into actual wiring of the machine. At least initially, the program on ENIAC was the physical setup of cables and switches.

Why start a discussion about the history of good software engineering practices with something as rudimentary as the ENIAC? As soon as the challenges of programming and debugging appeared, programmers started to look for ways of dealing with those. Many such methods are not unlike the methods that programmers use to program and debug computers today. First of all, different individual programmers approached the problem of programming in different ways. In her memoir, Betty Jean Jennings (Figure 4.3), whom we encountered as one of the first ENIAC programmers

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

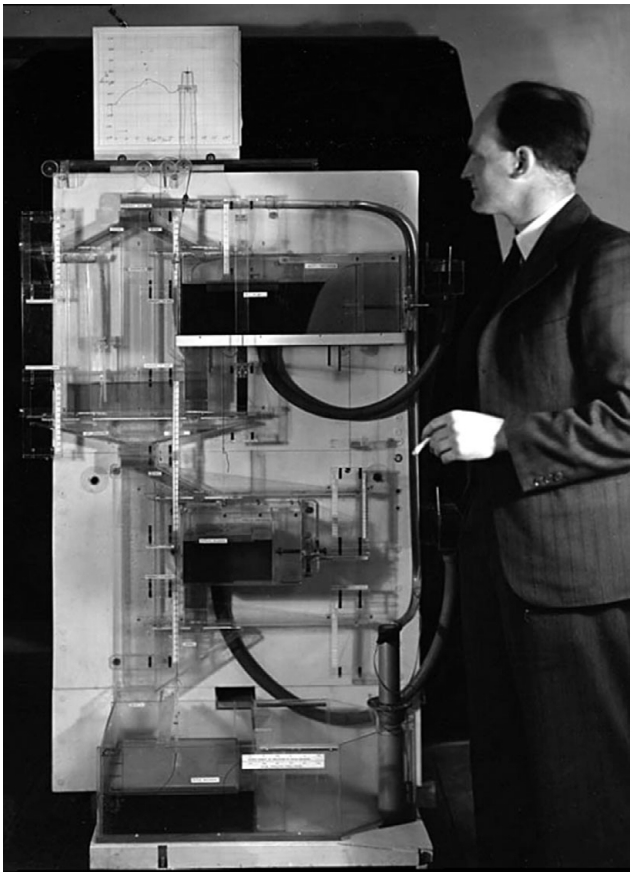


Figure 4.2 Professor A. W. H. Phillips with the MONIAC computer also known as the Phillips Machine. (Source: LSE Library)[17]

in Chapter 2, recalled how her approach to programming differed from that of her colleague Adele Goldstine:

> Adele was an active type of programmer, trying things very quickly. I was more laid back and given to attempting to figure out things logically before doing anything.[18]

Computer programmers never resolved this schism and never settled on one right way of programming. We find the same difference in approaches between the hacker and mathematical culture of programming today. On the one hand, mathematically minded programmers still favour logical reasoning over experimentation. This approach is often taught at universities, in part because it can be more easily written down. On the other hand, hackers often solve problems by quickly iterating on a solution. Being an active type of programmer like Adele Goldstine today may require less formal preparation, but more intuition and experience. A skilled hacker does not try just any

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

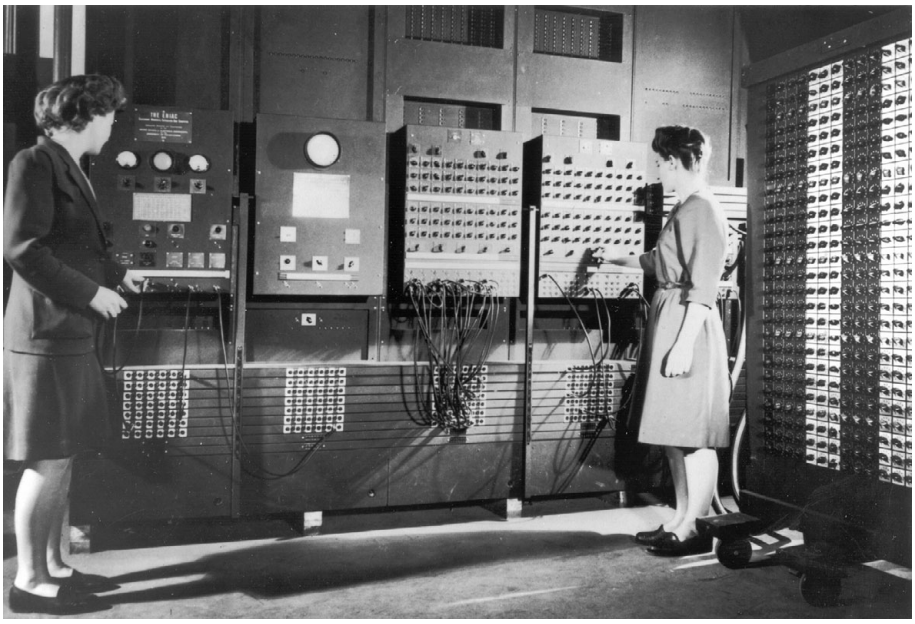


Figure 4.3 Betty Jean Jennings (left) and Frances Bilas (right) operating the ENIAC's main control panel. (Source: US Army/ARL Archives, Courtesy of Mike Muuss)[19]

random solutions. They approach the solution gradually, in a well-chosen sequence of steps.

The programming of the ENIAC also involved two ideas that are characteristic of the engineering culture of programming, which often strives to find a better or a more reliable method to solve a problem. The first engineering method was used when debugging the program for calculating trajectories. When developing the program, two of the ENIAC programmers first manually calculated a single trajectory in a way in which the ENIAC was supposed to calculate it. When the machine was set up to do the work, it was first run on this test trajectory. If the results differed, the programmers knew something was wrong. They could stop the execution of the ENIAC and run it manually pulse-by-pulse to find the source of the error. This was useful not just for finding errors in the program wiring, but also to identify blown vacuum tubes, which was a common hardware failure of the ENIAC. The idea of using a hand-computed test run as a baseline for testing the program is the kind of robust method that arises from the engineering culture. In fact, the approach is not unlike that of a much later Test-Driven Development (TDD) methodology that we will discuss later in this chapter.

The second engineering technique that is illustrated by the history of ENIAC is the idea to use the computer itself to simplify its programming. In 1948, a reprogramming of the ENIAC changed how the machine operated. The new approach was based on the experience with the ENIAC and also ideas for EDVAC, a subsequent machine that was in the works. The reprogramming wired the ENIAC such that it would execute instructions encoded on punched cards using a '60 order code'.[20] The programming of the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

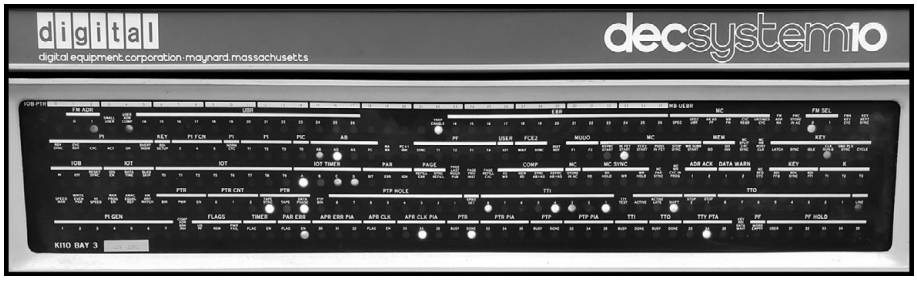


Figure 4.4 Indicator lights at the top of the 'Bay 3' (which also included the main control panel) of the KI10 variant of the PDP-10 computer, released in 1973. (Source: Living Computers Museum, Photo by David Bridgham.[21] Reproduced with permission.)

machine thus no longer required tedious fiddling with cables. The new programmable wiring was kept until the end of ENIAC operation in 1955. Again, using a computer to build a tool that makes the programming easier is an engineering method that we will encounter many times in this chapter.

Despite being electronic and digital, the ENIAC had a certain degree of transparency. The state of the memory was made visible using a grid of light indicators on the control panel, which the inventors used to put on a show during the 1946 public demonstration. Running the machine pulse-by-pulse also made it possible to follow what is going on. A similar degree of transparency was offered by many of the early electronic digital computers (Figure 4.4). An even more curious hacker trick was used to make transparent the later BINAC machine, built by the Eckert–Mauchly Computer Corporation in 1949. One of the engineers discovered that they could connect the output of one of the BINAC circuits to a radio and listen to its sound, much like the hacker-minded narrator in *Zen and the Art of Motorcycle Maintenance*[22] who notices an issue from the different noise of the motorcycle engine:

> Mind you, this weird music wasn't like anything being broadcast on the radio–it was just a sequence or pattern of sounds emitted from the radio as the BINAC was running. Al Auerbach said he'd quickly discovered that he could tell whether the BINAC was running correctly by the sounds coming out of the radio when he played the test program.[23]

In the case of ENIAC, hardware failures were still very common and so debugging using, for example, the manually computed trajectory, often involved looking for both hardware failures and programming errors. However, hardware soon started becoming more reliable, partly thanks to the fact that computers were made of a large number of repeated components. No such repetitive structure exists in computer programs and so programming issues grew in prominence. As mentioned in Chapter 2, one of the computer pioneers who understood the complexity of programming early on was Maurice Wilkes, who designed the EDSAC computer at Cambridge.

The EDSAC computer shared an architecture with many computers built after ENIAC. It was controlled through a program, represented as a sequence of *order codes* that was loaded from a tape or punched cards. The orders, or instructions,

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

were then stored in computer memory and executed. EDSAC recognised 17 different instructions for performing calculations (`AnS` to add the number in storage location `n` into the accumulator), memory access (`TnS` to transfer the contents of the accumulator to storage location `n`), but also for conditional jump (`EnS` to continue executing orders in storage location `n` if the value in the accumulator is greater than or equal to zero).

The practical matter of programming the early computers required 'black art' skills that are characteristic of the hacker culture of programming. Programmers had to invent ingenious tricks, often using the fact that a program itself was stored in memory, to implement operations that seem obvious today, but were not included in the instruction sets of early machines. For example, EDSAC did not have an instruction for indirect addressing that programmers would use today when accessing an element in an array at a specified index. If you wanted to access a storage location based on an index computed by your program, you had to use the transfer instruction `TnS` to first overwrite the part of your program (which was loaded into a known location in the memory) that specified the argument of the next `TnS` transfer instruction. The ACE computer designed by Alan Turing did not even have a conditional jump and programmers had to use the very same trick of overwriting the target location of an unconditional jump.[24]

The first non-trivial program that has likely been subject to debugging is a program to calculate the Airy integral, created by Maurice Wilkes for EDSAC. A detailed account of the debugging process has been given by the historian Martin Campbell-Kelly,[25] who analysed an early tape with the program, finding 'approximately twenty errors in the 126 lines of the program'. Many of those were simple punching errors, but some likely required a notable debugging effort. The unexpected fact that producing correct programs is difficult meant that EDSAC did not initially have any dedicated program debugging tools. As also pointed out by Campbell-Kelly, 'the way to debug a program at Cambridge, and at most other places, was to sit at the console and execute the program manually, instruction by instruction, while observing the registers and memory on monitor tubes'.[26] This tedious process was referred to as 'peeping'. On EDSAC, it was done using the 'Single E.P' button, which executed a single instruction of a program and a CRT monitor that displayed the contents of a portion of the computer memory. Some early computers went even further and provided switches for manually modifying the instructions in memory.

As was the case with the history of ENIAC programming, the history of EDSAC programming also includes a primordial form of engineering programming techniques that later become standard. On EDSAC, the most interesting methods were possible thanks to the fact that the program was stored in memory where it could be programmatically modified. This made it possible to use the computer itself to build tools that make programming and debugging the computer easier.

The first trick simplified the construction of 'large' programs like the Airy integral calculation. Maurice Wilkes and David Wheeler adopted the idea of a subroutine that was proposed in the 'First Draft of a Report on EDVAC' written by the ENIAC team at Moore School. Subroutines were sequences of instructions, eventually stored on tapes in a library, which implemented common tasks and could be added to the program after the main routine. Doing this manually would be laborious, because EDSAC

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

instructions used absolute addressing in instructions that accessed memory and so all addresses in the subroutine would have to be adjusted. To avoid this issue, Wilkes and Wheeler introduced 'initial orders', which was a rudimentary program loader, formed by 40 or so instructions, that loaded subroutines from a tape and replaced relative addresses in its instructions with absolute ones.[27]

A number of other tricks were devised to simplify the debugging of programs. Debugging programs through peeping, by running them instruction-by-instruction on a real machine was not only tedious but it was also taking valuable computer time. As the demands on the machine grew, peeping was recognised as 'extravagant use of computer time'[28] and was outlawed by the designers of EDSAC. Fortunately, the engineering method of using the computer to simplify programming led to two new approaches to the problem of debugging in the new setting.[29]

The first attempt was the development of a 'postmortem dump routine' that was triggered when the program terminated abnormally. The routine printed out a specified region of the computer memory and the programmers could then inspect the state and look for errors offline. The routine is the precursor of the idea of a core dump or memory dump that is produced, for example, when a contemporary operating system crashes.

The postmortem routine did not help when the program worked, but computed an incorrect result. Debugging such issues required a more sophisticated tool. This was implemented by Stanley Gill in 1951. His 'interpretive checking routine' was able to print diagnostic information while the main program was running. To do this, it iterated over the instructions of the main program and executed them one-by-one, simulating the behaviour of the EDSAC. This was about 50 times slower than running the instructions directly, but it provided an invaluable debugging tool. In modern terms, the routine implemented an EDSAC order code interpreter with detailed logging.

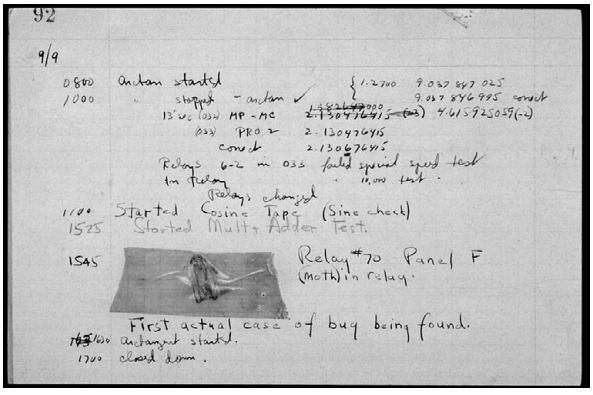


Figure 4.5 'First actual case of a bug being found' – a dead moth removed from the electromechanical Harvard Mark II computer by Grace Murray Hopper. (Source: Division of Medicine and Science, National Museum of American History, Smithsonian Institution)[30]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Before we continue, it is useful to review the terminology used to talk about programming issues in the 1950s. This has since evolved and so the historical meaning of the terms differs from the contemporary one. So far, I have been using the term 'debugging' to refer to the work that programmers and computer operators had to do in order to get their programs to run correctly. In the 1950s, this was one of the frequently used terms, although its meaning was broader and included activities that present-day programmers might call testing. The term debugging has its origins in the engineering slang used at the time. Despite a popular tale, the term 'bug' was not introduced into the vocabulary of programmers when Grace Murray Hopper removed a dead moth from one of the circuits of the Harvard Mark II computer and taped it into her logbook (Figure 4.5). The logbook with the dead moth is probably not hers,[31] but Hopper promoted the adoption of the term bug, which has been used by engineers to refer to malfunctions from the late nineteenth century. The use of the term for talking about programming issues was a notable rhetorical move. As pointed out by Kidwell,[32] by using the term 'bug' rather than a 'failure', programmers suggest that the issues are small faults that can be corrected. This proved an ironic naming, given that the estimate cost of work done on fixing the 'Y2K bug' at the end of the 1990s was over $300bn.

Another term that was often used alongside 'debugging' was 'checking' a program or 'program checkout'. An early example that uses this terminology is a paper 'Checking a Large Routine'[33] by Alan Turing from 1949. The paper illustrates that the early terminology around getting programs to run correctly was not clearly differentiated. Turing opens the paper by asking 'How can one check a routine in the sense of making sure that it is right?' He suggests that programmers can help the program checker by 'making assertions about various states the machine can reach'. Program checking in Turing's paper may be read more as an early attempt at proving a program correct than as debugging or testing. In other early systems, 'program checkout' was done, for example, by inspecting log traces produced by a tool such as the EDSAC interpretive checking routine. Throughout the 1950s, the terminology differentiated and the typical view was that you wrote a program, debugged it to make sure it ran and then 'checked it out' to make sure it produced the correct results. The task of 'program checkout' was more akin to what present-day programmers would call testing. Turing's early work that understood program checking more as proving programs correct was not immediately followed. The formal approach to program checking was, likely independently from Turing, reinvented in the mathematical culture of programming in the 1960s.

The shift from direct interaction with the machine to debugging tools was influenced by the business context. The machine time was expensive and postmortem and checking routines enabled programmers to debug their programs while the computer was available for running other programs. From the early 1950s, the operation of EDSAC was so streamlined that a full-time operator to run programs on behalf of the programmers was hired. The combination of managerial requirements and work of the engineering culture thus sidelined the direct approach of the early hackers. However the hacker culture did not have to wait long for another chance at transforming how we program and debug.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

4.3 On-line Debugging Techniques

Running a computer in a batch-processing mode was the norm in the 1950s and so debugging programs using memory dumps and diagnostic logs was the only option for most programmers. The exception from the norm were the MIT hackers who got their hands on the TX-0 computer, which we encountered in Chapter 3, after it was moved to the MIT Research Laboratory of Electronics (RLE) in 1958. The interactive input/output capabilities that were added soon after the move to MIT RLE made it possible to debug programs interactively, although the tools for doing so were initially very limited. The 1958 memo 'TX-0 Direct Input Utility System'[34] first goes to great lengths to justify the very idea of 'debugging at the console' and then acknowledges the current limitations:

> Debugging at the console is not new. In fact, it is the oldest form of debugging. For the past few years it has been regarded as a great professional sin, but like most sins, it still exists, is rarely talked about, and hardly ever admitted. …
> At the present time most utility programs do not provide enough immediate information to enable a programmer to find his error while he is still at the console. And if he can find it, few systems are capable of allowing him to make an immediate change without the preparation of a card, paper tape, or some other secondary input medium, especially if he wants to retain the symbolic language of the conversion program.

The Utility Tape 3 (UT-3) tool, which I mentioned in Chapter 3, made it possible to control the computer interactively, by typing commands on the console, but it was very rudimentary. Among other things, programs had to be typed in octal numerical system. The major advance in interactive debugging was the FLIT utility, written by Thomas Stockham and Jack Dennis. The latter was, in principle, in charge of the TX-0 operation at RLE, but was, in fact, more interested in playing with the machine himself. In a pun typical for the hacker culture of programming, the FLIT utility was named after an insecticide, but was presented as an acronym for Flexowriter Interrogation Tape.[35]

The FLIT utility made it possible to enter programs interactively using a symbolic notation that was very close to the assembly language used when writing the program in the first place. Rather than entering instructions as octal numbers, programmers could work with symbols as in their source programs. FLIT also introduced a debugging technique that remains the bread and butter of debugging to this day. The `break` command made it possible to set a breakpoint at a given location in the user program. When the program was run and reached the location, the control was transferred to FLIT, which printed the contents of registers and gave the user the options to interactively modify the state and program instructions or resume program execution.

Only one TX-0 computer was ever constructed and so the FLIT utility never left the world of MIT hackers. A successor tool was created by another hacker, Alan Kotok, for the DEC PDP-1 computer as soon as the computer was delivered to the MIT RLE in 1961. Following the insecticide-inspired naming tradition, this became known as the DEC Debugging Tape (DDT). The DDT utility was similar to FLIT and let the users modify programs using a symbolic assembler, referring to parts of a program using

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

the symbolic names used in the source code as well as set breakpoints. An interesting characteristic of both FLIT and DDT was their concise language of commands. In DDT, most operations were triggered using a single character and so the tools were very efficient in the hands of experienced hackers.

The appearance of the first time-sharing systems in the early 1960s renewed interest in on-line program development and debugging tools, although much of the work was still happening close to MIT and Cambridge. One such collaboration was between Thomas Evans at Air Force Cambridge Research Laboratories and D. Lucille Darley who worked at Bolt Beranek and Newman (BBN), a Cambridge-based company with close links to MIT that we encountered in Chapter 3. The two wrote an extensive review of existing on-line debugging techniques and created a new tool named DEBUG.[36] The DEBUG tool addressed two issues with FLIT and DDT. First, it made it easier to modify a program by automatically relocating parts of a program to make space for the new code. Previously, this had to be done by inserting the new code in an unused part of memory and inserting error-prone jumps to the right places in the code. Second, DEBUG also made it possible to easily export a clean corrected version of the debugged program. It kept a table of edits performed manually during the interactive session and was able to print out on an 'alter tape' after the end of the debugging session.

In their review of on-line debugging techniques,[37] Evans and Darley remark that time-sharing systems make on-line debugging economically feasible and that facilities for program debugging is an 'area of critical importance for effective utilisation of such time-sharing systems'. The motivation for their review was to document the systems that existed at the time, either in early time-sharing systems or for smaller computers such as TX-0 or PDP-1, which were used interactively at some installations. In doing so, the review also highlights one point that clearly positions work on debugging utilities in the realm of the hacker culture of programming. Evans and Darley noted that 'much of the work in this field has been described only in unpublished reports or passed on through the oral tradition'. Moreover, some of the tools they review 'are far from completely described even in internal memoranda'. In other words, debugging relies on personal knowledge that is difficult to write down and is gained through practice and mentorship. The reliance on oral tradition may also have been a reason why much of the early work on debugging tools was confined to MIT and affiliated organisations.

Tools like FLIT and DDT were a step up from UT-3 and flipping of switches on early machines in that they made it possible to use a symbolic language during their debugging sessions. Their users no longer had to modify the numeric codes representing the instructions, but could instead type assembly language instructions. In 1966 when Evans and Darley published their survey, on-line debugging techniques existed for both assembly languages and for high-level programming languages including Lisp and FORTRAN, although the ones for high-level languages were 'less well-developed and less widely used'.

Programming systems with on-line debugging facilities for high-level programming languages started to develop in the mid 1960s on early time-sharing systems. The time-shared IBM 7044 computer came with a FORTRAN implementation QUICKTRAN

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

and a multi-user FORTRAN also existed for the Berkeley time-sharing system. Those systems generally allowed users to set breakpoints (by inserting appropriate instruction into a program) and inspect the values of variables (using the symbolic name of the variable). QUICKTRAN was based on an interpreter and so it also enabled users to insert and delete statements in the program source code, which was not possible in systems that executed compiled code. The idea of an incremental compilation, which is used by some debugging tools today, already existed in literature, but was not implemented in a real-world system at the time.

The higher-level language with the most advanced interactive programming and online debugging features in the mid 1960s was Lisp. Lisp has been around since the end of the 1950s, but the first implementations were written for batch-processing systems. By 1964, Lisp was available on a number of interactively used machines such as PDP-1 at Stanford, as well as on time-sharing systems including the IBM 7094 computer running the Compatible Time Sharing System (CTSS) at MIT. The first time-sharing implementations of Lisp featured fairly basic interactivity. The time-sharing Lisp running at MIT let users set breakpoints, evaluate Lisp expressions interactively and print the backtrace (recording what functions were being called) when an error occurred, but this was soon about to change.[38]

Thanks to its nature, Lisp has been the perfect language for the development of advanced debugging tools and Lisp hackers soon started to exploit this characteristic. In Lisp both data and code are represented as lists, a feature known as *homoiconicity*. This means that code of a running program can be easily represented and modified during the program execution. As we saw in Chapter 3, a number of Lisp implementations also included editors that let users edit Lisp code not as text, but by directly modifying the list structure representing it. As in the case of FORTRAN, running the code modified interactively required having an interpreter, in addition to a compiler that turned the initial program into efficient instructions. The work on Lisp editing and online debugging came together in MacLisp, which ran on the Incompatible Timesharing System (ITS) at MIT. In the aforementioned survey of debugging tools, Evans and Darley describe a scenario illustrating the possibilities of the system:

> With some care, it has been possible, for example, to find a bug while at a breakpoint in running a test case, call the [structure] editor to make a correction, run the program on a simpler test case to verify the correctness of the change, then resume execution of the original test case from the breakpoint.[39]

One interesting aspect of this scenario is how it integrates the different activities involved in getting a program to run correctly. It starts with running a test case, which is an activity we would today refer to as testing; it follows with making a correction in a tool that we would now refer to as the debugger and then verifies this correction, which is something that a present-day programmer would perhaps do interactively using a REPL (Read-Eval-Print Loop) shell. In other words, the on-line debugging techniques of the 1960s do not yet make a clear distinction between development, debugging and testing. More interestingly, error handling is also closely interconnected with the other activities.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

A system that illustrates this well is the BBN-LISP and the corresponding project manual published in 1967.[40] The list of authors reflects the capabilities of the system. It includes Daniel Bobrow, D. Lucille Darley (a co-author of the DEBUG tool and the aforementioned survey), L. Peter Deutsch (who created the structure-based editor for Lisp), Daniel L. Murphy and Warren Teitelman (whom we encountered in Chapter 3 as the author of the PILOT system).

The connection between debugging and error handling is immediately clear in the report, because the two topics are discussed in a single section, 'Error Handling and Debugging Functions'. The section opens by explaining that there are two classes of errors, '`H` errors for which the user can provide Help on the spot; and $\overline{\texttt{H}}$ errors for which no help is possible'. By default, the $\overline{\texttt{H}}$ errors stop all computation, printing functions that have been entered by the failing computation. The error can be handled in code using the `errorset` function, which behaves as an exception handler in modern programming languages. It runs a computation specified as an argument and, if no error occurs, returns its result. If an error occurs, the special value `NIL` is returned.

`H` errors allow the user to fix the mistake and let the program continue. This includes cases where a function or a symbol is undefined (or misspelt). The user can fix such errors interactively by typing the correct name or defining the function during the debugging session. The user can also induce (trigger) an `H` error interactively to break the execution as soon as the next function is entered. In addition to this, BBN-LISP supported more conventional breakpoints, set by invoking the `BREAK` function. This modified a specified function to stop the execution when the function was called, optionally also checking that a certain break condition was satisfied.[41]

By the end of the 1960s, many different tools for on-line debugging, testing and error handling were developed. However, there was still no clear distinction between debugging, testing and error handling and the different tools were often used together. This required practical hacker knowledge held by the practitioners that was more often shared through personal interactions, than through written documentation. However, the ongoing shift from the black art of programming to software manufacturing based on rigorous scientific methods was about to transform the practices of debugging, testing and error handling.

4.4 The Debugging Epoch Opens

While the on-line debugging techniques for the early interactive computers and time-sharing systems were appearing in the 1960s, most commercial programmers were still debugging their programs using the two methods developed over a decade before for EDSAC: the postmortem memory dump and various forms of tracing that printed diagnostic information during the program execution. This was increasingly recognised as a major obstacle for software development and, in November 1965, Mark Halpern made the problem clear in his article for the *Computers and Automation* magazine titled 'Computer Programming: The Debugging Epoch Opens'.[42] In the article, Halpern commends the recent developments in hardware and programming languages

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

and discusses the next major challenge: 'We have machines big and fast enough to execute most useful programs, and software varied and reliable enough to get those programs written; what now? Now: debugging'.

Halpern, who had experience working at IBM and was now based in a research laboratory at Lockheed, saw 'the debugging problem' as the new limiting factor for the growth of the computer industry. He used the term debugging in the broader sense of identifying and correcting programming errors and proposed a number of new debugging techniques. One such technique that remains commonplace to this day was distinguishing program operation in 'Debugging Mode' and in 'Production Mode'. Other techniques he proposed were not unlike some of those that were already becoming possible in Lisp environments developed at the time by the hacker culture. Halpern identified a problem that was becoming increasingly visible at the time and envisioned some tools that would become available later, but he did not anticipate a different kind of response to the problem that was starting to take shape.

At the end of the 1960s, many stopped seeing the difficulty with software development as the debugging problem of eliminating small program flaws and started seeing it as an industry crisis instead.[43] This was a major cultural shift. As discussed in Chapter 2, practical programming in the 1950s was dominated by the hacker culture of programming. It was a black art that relied on personal knowledge and the results were not reproducible or reusable. This state of the art was not acceptable to either the mathematical culture, which sought formal foundations for programming, nor the managerial culture, which wanted to be able to produce software of a predictable quality at a predictable cost.

Tackling the problematic state of computing became the subject of the NATO Software Engineering conference held in 1968 in Garmisch. The idea for the conference came from Fritz Bauer who was a professor at the Technical University of Munich. Bauer was actively involved in the development of Algol 68 and although he considered the language 'too fat',[44] he was not a signatory of the Algol 68 Minority Report that I discussed in Chapter 2.

The conference was organised by Bauer and the NATO Study Group on Computer Science, which was established in the autumn of 1967 by the NATO Science Committee.[45] The organisers decided to structure the conference program along the three themes of design of software, production of software and service of software. They identified group leaders for each of the topics, many of whom were previously involved in the Algol efforts, and invited a limited number of 'carefully chosen leading figures' from different backgrounds and different countries.[46] The attendees included a diverse group of academic and industry programmers, but the attendee list notably lacked representatives of the hacker cultures of programming. The MIT hackers that we encountered in Chapter 3 would likely disagree with many of the assumptions of the conference, but they also did not fit the traditional profile of an academic or industry leader in the field.

Consequently, the conference attendees quickly agreed on the problem with the 'black art' approach of the hacker culture of programming. The agreement on the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

problem at the 1968 Garmisch conference, however, contrasted with the disagreement on the solution at the follow up conference in 1969 in Rome. The different cultures of programming interpreted the term 'software engineering' differently and were more interested in pursuing their own agenda than agreeing a common approach. According to the mathematical culture, the answer was the development of methods for formal reasoning about programs. The managerial culture saw the answer to the software crisis in scientific management of programming teams. However, the conference also marked the rise of the engineering culture of programming that hoped to address the software crisis by the development of new and better tools and programming practices to assist the programmer.

The development of software engineering brought a more structured approach to thinking about software and its development.[47] A concrete example of this was structured programming that I discussed in Chapter 2 and that soon acquired both engineering and managerial interpretation. However, the structured nature was apparent in the very organisation of the Garmisch conference. The conference program was divided, following the steps of the emerging development process, into sessions on design, production and service. The NATO conference report also shows more differentiation in the terminology used for talking about debugging and testing. It only mentions debugging in a general sense, but includes a number of more thorough perspectives on software testing.

First of all, the authors of the working paper 'The Testing of Computer Software'[48] relate testing to the aim of the conference to provide a scientific basis for software development. The authors note that 'testing is one of the foundations of all scientific enterprise' and argue that software should be 'rigorously tested for conformity with the specification'. They go on to distinguish two different aspects of such testing:

> This testing has two aspects: that carried out by the producer of the software to satisfy himself that it is fit for release to his customers and that carried out on behalf of the customer in order to give him sufficient confidence in the software to justify his paying for it.

The working paper notes that the unfortunate current strategy is that 'most customers accept the software without any form of test' and suggests a testing strategy that includes checking the documentation, system availability, verification of functionality and performance assessment. In other words, they describe the kind of testing that we might today refer to as acceptance testing.

A more fine-grained perspective distinguishing different kinds of tests is from John Nash at IBM whose working paper includes an illustration (Figure 4.6) that distinguishes unit tests, component tests and system tests. As we will see, this kind of division was becoming more commonplace as one aspect of the structured development process that later became known as Waterfall. Finally, another interesting idea from the NATO 1968 report is from a working paper 'Aids in the Production of Maintainable Software'. The author argues that testing should be automated and that 'a collection of executable programs should be produced and maintained to exercise all parts of the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

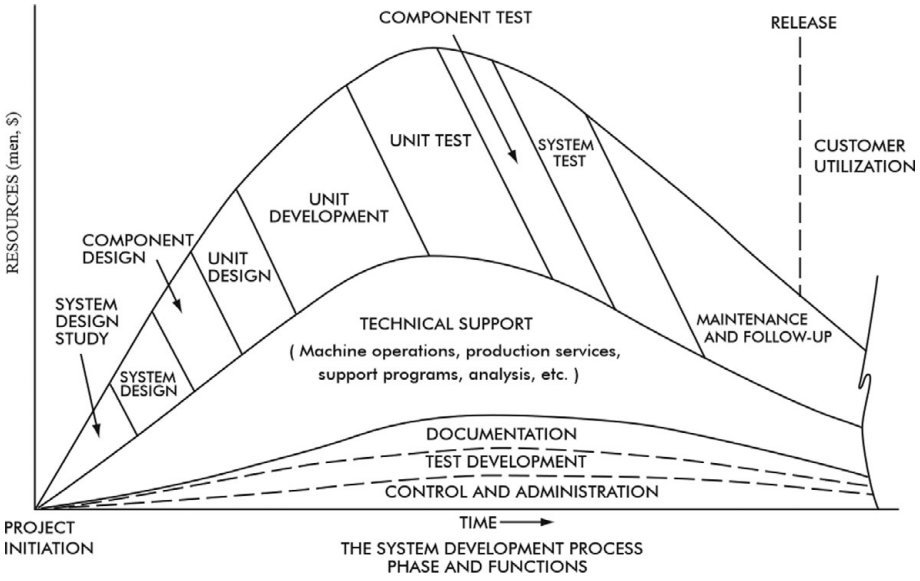


Figure 4.6 Figure 'Some problems in the production of large-scale software systems' from the NATO 1968 Software Engineering conference report. (Source: Drawing by Robert M. McClure.[49] Reproduced with permission.)

system'. This suggests the idea of test automation that will be implemented in a number of systems in the decades to follow.

The NATO conference proceedings show that the task of getting a program to run as required, which was interchangeably called checkout, debugging or testing in the 1950s, started to differentiate into a number of distinct concepts. Most importantly, the notion of testing was becoming a multi-faceted concept that covers the more managerial notion of acceptance testing done by a customer, as well as the engineering idea of automated testing as practised for example by modern unit testing tools.

Testing and debugging also appears in the report from the 1969 Rome conference. Interestingly, most of the discussion appears in the 'Software Quality' section under the 'Correctness' subsection, which covers formal correctness ('how to prevent bugs from ever arising') and debugging ('how to catch them when you have got them'). It is the section on formal correctness that contains an argument in favour of careful testing by Perlis alongside a famous remark by Dijkstra that '[t]esting shows the presence, not the absence of bugs'.

The discussion on debugging in the report is also enlightening. The list of participants of the 1969 Rome conference includes Warren Teitelman, who worked on the on-line debugging tools for Lisp and is possibly the sole clear representative of the hacker culture in the list of attendees. His working paper, 'Toward a programming laboratory' presented much of his work on Lisp, including the PILOT system and interactive error correction. The working paper sparked a number of discussions at the conference, in particular on the relative merits of off-line debugging and on-line

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

debugging. Niklaus Wirth, who worked on the Pascal programming language at the time, expressed his worry about the new developments:

> There has been, since the advance of timesharing and on-line consoles, a very hectic trend towards development of systems which allow the interactive development of programs. Now this is certainly nice in a way, but it has its dangers, and I am particularly wary of them because this conversational usage has not only gained acceptance among software engineers but also in universities where students are trained in programming. My worry is that the facility of quick response leads to sloppy working habits and, since the students are going to be our future software engineers, this might be rather detrimental to the field as a whole.[50]

Wirth's comment comes from the perspective of the mathematical culture, arguing that programmers should first thoroughly, and perhaps even formally, think through the problem before writing any code. The engineering perspective defended by Alan Perlis in response is that 'people don't think like that; conversational systems permit programmers to work from the top down and to compute even with incomplete specifications'. Teitelman unsurprisingly shares this view and argued that conversational systems promote 'a more efficient, systematic approach to debugging and testing'. In the report, Teitelman does not elaborate on what he means by 'systematic', but making that explicit would likely pose a challenge. The ambition of the NATO conferences was to transform programming into a proper engineering discipline. This had almost no effect on debugging, which remained a hacker practice that is learned through experience and relies on the skills of individual programmers. The systematicity of debugging remained rooted in the personal unwritten knowledge of hackers. The shift also did not visibly affect the work on interactive programming, which we followed in Chapter 3. The conferences did however, mark the start of a transformation of exception handling and software testing.

4.5 Orderly and Reliable Exception Handling

When a program running on a 1960s batch-processing system encountered an unexpected situation, it would terminate and optionally produce a postmortem dump. In interactive programming systems, an unexpected situation would break the execution so that a programmer could analyse the situation, correct it and resume the execution. In both cases, the assumption was that the unexpected situation would eventually be handled by a programmer. However, one of the premises of software engineering was to produce reliable programs that can gracefully recover from unexpected situations.

In Lisp, the burden of handling unexpected situations could be moved from the programmer to the program itself. LISP 1.5, the first widely available version, included a function `errorset`, which I mentioned when discussing on-line debugging. When an error occurred inside code wrapped in `errorset`, the function returns `NIL` and the programmer can write code that recovers from the error. The LISP 1.5 manual is careful to say that continuing computation in a different direction when one path results in an error is only possible if the failing code has not modified any variables or

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

caused other damage. In other words, doing the recovery correctly is still a task for a competent hacker.

Another early programming language that contributed to the development of exception handling is PL/I. The language predates the birth of software engineering and was developed at IBM as part of a broader effort to bridge the gap between scientific programming (typically done in FORTRAN) and business data processing (typically done in COBOL). In the 1960s, IBM set out to build a single system to cover all possible applications of computers. It designed the IBM System/360 computer range as well as the operating system OS/360 and also started work on a new multi-purpose programming language. The designers of PL/I believed that the strict separation of scientific and business computing was a false stereotype and that programmers needed a single language to complete all their programming tasks. Some also hoped that having a single language may relieve IBM from the burden of maintaining multiple compilers.[51] Although PL/I was modelled after FORTRAN, it included rich data structuring capabilities and built-in functions for the handling of I/O operations. Those operations for reading data were a common source of program failures, which influenced the design of the language:

> [Languages before PL/I] had few general facilities to define actions which were to be taken when conditions such as reading an end of file, overflow, bad data, new line, occurred. Often the conditions could not even be sensed in the language, and assembly language patches were required. … The [PL/I] designers were determined that entire real applications should be programmable in the language and these programs be properly reliable and safe.[52]

To make programming of reliable and safe applications in PL/I possible, the language designers defined 23 types of error conditions and the `ON` construct which could be used to specify an action to execute in case an error condition occurred. The construct was cumbersome to use, partly because passing information to the exception handler often required using global variables and because the action of registering a handler had to be cautiously explicitly reverted.

Although LISP 1.5 and PL/I introduced much of what is needed for modern exception handling, it took another decade until more sophisticated exception handling started to became commonplace. The developments in LISP 1.5 and PL/I are notable, because they contributed to the differentiation of the various tasks involved in getting programs to behave correctly. Whereas error handling in the 1950s was closely interconnected with testing and debugging and was a matter for hackers, the constructs like `ON` and `errorset` make it a part of the programming language itself. The terminology shows that this was a gradual process. When talking about the `errorset` function, the LISP 1.5 manual[53] talks about 'errors', while the PL/I reference[54] talks about 'exceptional conditions' and introduces the `ON` statement in a section on exception control statements.

Internalising error conditions into the programming language moves the problem of error handling into the realm of the engineering and even mathematical culture. PL/I is a good example. The language originated from engineering efforts at IBM. It was partly inspired by Algol, but the PL/I design lacks the focus that would be typical for the mathematical culture. It is not done with reference to a theoretical basis and does not

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

emphasise the ease of proving correctness. However, PL/I was also extensively studied at the IBM Vienna lab, which produced multiple formal definitions of the language. For some of the formalisation techniques, Algol served as a testbed and this resulted in the Algol specifications that I covered in Chapter 2.[55]

Programming language work on exception handling has been inspired by a more general focus on reliability in the software engineering community that started to emerge in the wake of the NATO Software Engineering conferences.[56] In 1975, John B. Goodenough published two papers that made major contributions to exception handling and testing. We will return to the latter in the next section. Goodenough was working at a software consultancy developing software engineering tools and solutions, so he came from a markedly engineering background. Yet, his proposal for an exception handling mechanism was presented at the ACM Symposium on Principles of Programming Languages (POPL) where it appeared among talks mostly rooted in the mathematical culture of programming. It was then revised and published in the widely read *Communications of the ACM*.[57]

The exception handling design proposed by Goodenough shares many aspects with the modern exception handling mechanism as known, for example, from Java. Exception handlers are associated with blocks of code and can specify code to run for various exceptions that may be triggered during the execution of the enclosed code block. There are a number of built-in exceptions, but users can also define their own. Furthermore, procedures are annotated with exceptions that may be raised and Goodenough argues that the compiler should check that all possible exceptions are handled. In a feature that goes beyond what is supported by most programming languages today, the handler can also resume the execution of the code that raised the exception. There are three different constructs for raising exceptions: `ESCAPE` requires termination, `NOTIFY` forbids termination and `SIGNAL` where the termination behaviour is decided by the exception handler.

Goodenough's paper was a detailed proposal for an exception handling mechanism. It was a theoretical publication and did not come with a specific implementation, but it influenced the design of two programming languages from the 1970s, Clu and Ada, that embody the ideals of software engineering in the form of a programming language. Clu was created by Barbara Liskov at MIT and was motivated by the work on programming methodology and information hiding. Its key innovation was the design of abstract data types, which I cover in Chapter 5, but Liskov also wanted to support 'robust or fault-tolerant programs … that are prepared to cope with the presence of errors'.[58] For this reason, Clu adopted an exception handling mechanism that was inspired by both Goodenough's paper and the mechanism in PL/I. The designers of the Clu language made two design decisions related to exception handling that are common in programming languages today. First, they referred to the mechanism as 'exception handling' and, second, they only implemented the termination model, that is, there was no way to resume the program from the original location after an exception.

The development of the Ada language was led by the US Department of Defense (DoD) and was the result of a managerial 'Software Initiative' to reduce the 'High Costs of Software'. The designers believed that 'a common language for use through-

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

out the DoD would be a first step to building an engineering discipline of software, to supersede the then current "black art"'.[59] The design of Ada was a product of thorough and detailed analysis and was described in a 250-page long 'Rationale for the design of the Ada programming language'.[60] John B. Goodenough was directly involved in the Ada design process and the Rationale makes a reference to his work, as well as other ongoing research on exceptions. Just like Clu, it adopts the terminology of exception handling and also the termination-only mode of exceptions.

Whereas the development of exception handling in PL/I was focused on solving the specific problem of handling of specific error conditions related to input and output issues, much of the later work influenced by the rise of software engineering had a broader realm. Already in Lisp, the exception handling mechanism based on `errorset` was used 'not to trap and signal errors but for more general control purposes (dynamic non-local exits)'[61] and MacLisp later added a pair of primitives `catch` and `throw` so that the original exception handling primitives could be reserved just for error handling. More interestingly, Goodenough also saw the proposed exception handling mechanism 'in general, as a means of conveniently interleaving actions belonging to different levels of abstraction'.[62] This was the case especially with exceptions that are intended as notifications for the caller who then resumes the execution of the code triggering the notification. Similarly, in Clu, exceptions were seen as 'a general way of conveying information from a called procedure to its caller'.[63]

In the case of Lisp, the LISP 1.5 manual indicates that `errorset` can be used to 'continue computation in a different direction when one path results in an error',[64] but does not indicate that the mechanism can be used for purposes not related to errors. The more general use of `errorset` for control flow was likely the result of the ingenuity of the hacker culture, which was able to exploit the mechanism for originally unintended clever programming tricks. Goodenough's paper presented a cultural shift and exceptions were seen as a language mechanism that enabled multiple good software engineering practices, including both error handling for reliability, but also good structuring of code into multiple levels of abstraction. However, Clu, Ada and later Java again constrained the use of the mechanism to just error handling. In a perhaps surprising plot twist, the trend has again been reversed in the 1990s in research on algebraic effect handlers that are, yet again, an exception-inspired programming language construct for interleaving actions across multiple levels of abstraction.[65]

4.6 System Structure for Fault Tolerance

The work on exception handling in programming languages provided a way of recovering from errors and suggested a more general program structuring mechanism. It also served as the basis for over-engineering, another practical engineering practice that Hoare later saw as contributing to software reliability. In the analysis of exceptions that Goodenough presented in his paper, he classified exceptions into two categories. A *range failure* 'occurs when an operation … finds it is unable to satisfy its output assertion' and *domain failure* 'occurs when an operation's inputs fail to pass certain

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

tests of acceptability'. In other words, if something goes wrong when performing the operation, it is a range failure, but when the operation is invoked incorrectly, it is a domain failure.

The former category of program errors is inevitable. A program cannot prevent real-world situations such as a corrupted file. However, in a perfect program imagined by the mathematical culture of programming, a domain failure would never happen. A program would be written so that the acceptability of inputs is ensured by the callers of the operation and, in an ideal world, even guaranteed by a formal proof. The idea that an operation should test acceptability of inputs is thus a basic form of over-engineering, or, pre-empting errors that should not happen in theory, but does happen in practice. In modern programming terminology, the practice of checking for invalid input is widely adopted as part of the *defensive programming* methodology that also dates back to the mid 1970s.[66]

The software engineering research on software reliability inspired a number of other programming language designs that would simplify over-engineering. One example that illustrates possibilities beyond those that are supported in main-stream programming languages today came from Brian Randell. Randell was a signatory of the Algol 68 Minority Report, attended both of the NATO conferences, was the editor of the proceedings for the first one, and dedicated much of his career to work that contributes to the engineering culture of programming. In 1969, he moved from IBM to Newcastle University and started working on software reliability. In 1975, he published an influential paper 'System Structure for Software Fault Tolerance'[67] that introduced *recovery blocks*.

Recovery blocks are a programming language feature designed for building of fault-tolerant systems that can recover from design inadequacies. A recovery block is similar to an exception handler, but the body is guarded by an acceptance condition (a check for a range failure). If the condition is violated by the primary implementation, the block proceeds to execute one of the alternative, possibly simplified, implementations, until it obtains a result that satisfies the acceptance condition. Another mechanism implementing the idea of over-engineering that emerged from the software engineering community in the mid 1970s was to build systems with a self-checking mechanism.[68] Such systems would actively monitor themselves to detect range and domain failures. While none of the mechanisms going beyond exception handling appeared in widely used programming languages at the time, they normalised over-engineering as a way of tackling programming issues and introduced a mechanism that was, in various forms, implemented in later programming systems such as Erlang.[69]

Following the birth of software engineering, exception handling moved from a manual task for a programmer into a condition to be handled automatically, by the program itself. Consequently, it also turned from a feature of an interactive programming system into a programming language construct. Those working on exception handling started to come less from the hacker culture and more from the engineering and mathematical culture of programming. Testing, another aspect of reliable software development, was also soon to be transformed by the birth of software engineering.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

4.7 Professionalisation of Testing

New efforts to organise the community around software testing emerged soon after testing became recognised as a separate activity. In 1972, Bill Hetzel organised the Computer Program Test Methods Symposium at the University of North Carolina that brought together many of those interested in testing. Mirroring the narrative about the state of software development as a whole, Hetzel complained that testing 'is all art' and lacks an 'established discipline to act as a foundation'.[70] The symposium and the book 'Program Test Methods',[71] which was edited and published by Hetzel following the symposium, were an attempt to put testing on a rigorous basis.

Many saw Hetzel's efforts as a promising step. According to a report written for the ACM by one of the symposium attendees, David J. McGonagle, 'the programming industry took another step toward maturity'[72] with the symposium. This step was more in the ambition of the symposium than in its contents and the presentations at the event illustrate that there was not yet a coherent testing community. The presented papers include managerial reports on testing done for large software systems, engineering reports on new testing tools, proposals for standardising software testing practices, but also a number of mathematical talks on proving program correctness. As reported by McGonagle, the event 'served to bridge the gap between the proof of correctness people on one hand and the large systems users on the other'. Although McGonagle doubted that the event 'convinced any of the non-believers of the critical nature of this area of computer systems work',[73] the symposium was the first step. It was followed by a series of events on testing that, over the next two decades, established software testing as a recognised software engineering discipline.

The organiser of the 1972 symposium later co-founded a consultancy focused on software testing together with David Gelperin. The two played an instrumental role in promoting software testing and co-authored a paper 'The Growth of Software Testing'[74] , which also further helped to establish testing as a discipline by providing its origin story. In the paper, Gelperin and Hetzel recognise the early 1970s as the era when the level of professionalism in software testing significantly increased. In this period, the first software testing conferences took place and testing became the topic of an increasing number of academic papers. Companies started to recognise software test engineering as a speciality and started to hire specifically for positions such as 'test manager' or 'test technician'.

In the 1970s, testing became clearly distinguished from debugging. The goal of the latter was to 'make sure the program runs' while the goal of the former was to 'make sure the program solves the problem'. As the positive phrasing suggests, testing was seen as a way of showing that the software satisfies its requirements. As such, it had the same ultimate objective as the work on proving programs correct, but it utilised engineering rather than mathematical methods. This led to a debate between the proponents of the engineering culture and the mathematical culture, which questioned the extent to which testing can guarantee program correctness.

At the 1969 NATO Software Engineering conference, C. A. R. Hoare argued that one can easily show the 'ultimate futility of exhaustive testing'. In the report, this is

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

contrasted with a point made by Alan Perlis that '[m]uch of program complexity is spurious and a number of [well-chosen] tests will exhaust the testing problem'.[75] The two quotes illustrate a debate about the limits of testing between the mathematical culture and the engineering culture. The proponents of the former argued that testing is inferior to proofs, while the proponents of the latter saw testing as a promising practical tool. Interestingly, the most convincing arguments trying to show that testing can guarantee the absence of bugs came from engineers who utilised mathematical methods to counter the arguments made by the proponents of the mathematical culture by using their own means.

Three prominent examples of this approach appeared in the mid 1970s.[76] The first was from J. C. Huang whose PhD was in switching and automata theory, but who also had practical experience as an engineer.[77] Huang acknowledged that doing exhaustive testing, for a very simple program that takes two 32-bit integers as inputs and completes in 1 millisecond would take 50 billion years. His answer was to not treat the program as a black box, but analyse its structure and find a set of tests such that each branch in the program would run at least once (Figure 4.7). John B. Goodenough, who we encountered in the section on exception handling, developed another method for choosing test inputs.[78] He also indirectly responded to Hoare's comment claiming that 'properly structured tests are capable of demonstrating the absence of errors in programs'. Finally, at the same time, Thomas J. McCabe, who was employed by the National Security Agency and later established his own software consultancy, published the 'cyclomatic complexity' measure.[79] The measure is defined in terms of the number of branching points, paths and loops in a program flowchart. Although this was later used without reference to testing, McCabe suggested using the metric as a basis for a testing strategy. If the number of tests was lower than that required for a program of a certain complexity, either more tests would need to be added or the program structure would need to be simplified.

The next period of software testing that Gelperin and Hetzel refer to as 'destruction-oriented' started in 1979 with the publication of the textbook *The Art of Software Testing* by Glenford Myers.[80] It is ironic to see 'art' appear in the very title of the book, but the author does not advocate the earlier 'black art'. Instead, the title is intended to suggest that the book takes a practical rather than theoretical perspective. The author cautions the readers that 'many subjects related to testing such as the idea of mathematically proving the correctness of a program were purposefully excluded'. The book adopts an engineering perspective that draws on methods for test selection developed throughout the preceding decade. Yet, it still recognises the hacker heritage of software testing. In a section on 'Error Guessing', Myers admits that some people seem to have a knack for smelling out errors and that their error guessing technique is largely an intuitive and ad hoc process.

The book is a significant milestone not just because it is the first book solely dedicated to software testing, but because it shifts the perspective on the purpose of testing. Meyers sees testing as 'the process of executing a program with the intent of finding errors'. This way of thinking about testing presents a shift from the earlier view which saw testing as a mechanism for showing program correctness. The new perspective is

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

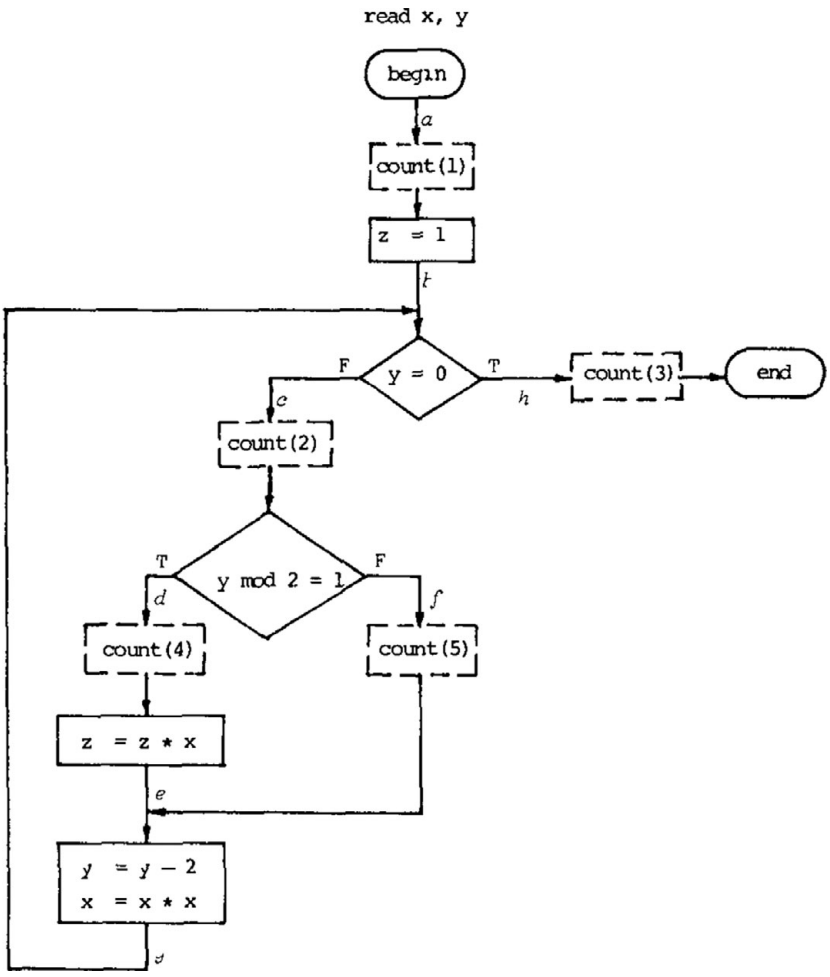


Figure 4.7 A flowchart for a sample program. Each branch (arrow) is annotated with a counter `count(n)` with different number `n` for different counters. A good set of tests will make sure that each counter is incremented at least once. (Source: Used with permission of ACM,[81] permission conveyed through Copyright Clearance Center, Inc.)

a shift from a managerial thinking of testing (acceptance testing done by a customer as part of the development process) to an engineering thinking (testing done by engineers to help them with their job). The perspective happens to be more practically effective as it encourages test selection that is more likely to find faults. It prompts testers to look for edge cases that might not be considered if the aim of testing is to make sure that a program works as required. Finally, the new perspective on testing aligns it with other approaches to fault detection. For example, Myers' book talks about code inspections and code walkthroughs and labels those as non-computer-based (or human) testing.

The hallmark method of the engineering culture is to create new tools that assist with solving the problem at hand. Such tools tend to be less well documented in the aca-

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

demic literature, but there are a number of good examples for the destruction-oriented period of software testing. The idea for one such tool occurred to Richard Lipton in 1971 when he was a student, long before he became known as a co-author of the paper 'Social Processes and Proofs of Theorems and Programs', which I talked about in Chapter 2. The idea was termed 'mutation testing' and it was published both as a paper and as a tool presentation in 1978.[82] The idea of mutation testing is to automatically change some aspect of the program logic and see if this change is detected by the tests. For example, a program that sorts a list of numbers will contain a comparison `X<=Y` to check if one element is less than or equal to another element. A mutation testing tool might change this check to, for example, `X!=Y` testing for an inequality instead. A well-designed test suite can detect such changes in logic. If the tests fail to detect that the modified program is incorrect, it is a hint that more tests need to be added.

In parallel to the engineering developments that I followed in this section, testing also existed in the context of the managerial culture, where it was one of the phases in the application development lifecycle. To document this, we first need to return to the end of the 1960s and follow the managerial response to the industry crisis.

4.8 A Slightly Ominous Note for Information Processing Management

For the managers working in the commercial computing industry, the 1960s started on a high note. The industry was growing at an impressive annual rate of 27 per cent and the number of computers installed in the USA grew from 5,400 to 75,000 during the decade.[83] Yet, many were also increasingly aware of the limitations and issues with the use of computers. There was a growing body of anecdotal evidence of problems with complex computer systems that experts in the field were familiar with.[84] The Mariner 1 spacecraft was lost due to a programming error in its guidance system in 1962. The IBM OS/360, delivered in 1967, was delayed by nine months and cost the company half a billion dollars, four times the original budget. Fred Brooks Jr., the project manager of the delayed IBM OS/360 system later likened the development of large software systems at the time to a tar pit struggle:

> No scene from prehistory is quite so vivid as that of the mortal struggles of great beasts in the tar pits. … Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems – few have met goals, schedules, and budgets. Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty – any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion.[85]

The reasons for the issues were both technical and managerial. The rapid growth of the computing industry meant that computers were used to solve increasingly large and complex problems. It was no longer possible for a small group of highly skilled hackers to keep the structure of an entire system in their head while building it, and software development started to require large teams which, in turn, meant that much more coordination and communication was needed. The programming languages at

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

the time provided only basic mechanisms for structuring and organising large code. The fact that a 'subroutine library' referred to a filing cabinet with punched cards is merely testimonial to that.

For the engineering culture, the NATO Software Engineering conferences are often seen as the pivotal moment. The conferences brought together participants from universities, industrial research labs, computer manufacturers, computer software and service firms as well as governments. But as pointed out by Thomas Haigh[86] a more careful look reveals that most attendees were involved in the development of technically interesting software such as operating systems, programming languages and compilers. Notably absent were discussions of application software, which is the bread and butter of programming such as travel booking or payroll systems.

The managerial perspective on the growing problems with the rise of computers was best captured in a report 'Unlocking the Computer's Profit Potential'[87] (Figure 4.8) published by McKinsey & Company in 1968. The report was widely discussed in computer magazines after its publication. The 'Editor's Readout' of the *Datamotion* magazine from August 1968 labels the report as 'a slightly ominous note for information processing management', while the republication of the report in the *Computers and Automation* magazine in April 1969 ensured it reached an even broader audience. The authors of the report studied the use of computers at 36 companies. Despite the technological advances, from a profit standpoint, 'computer

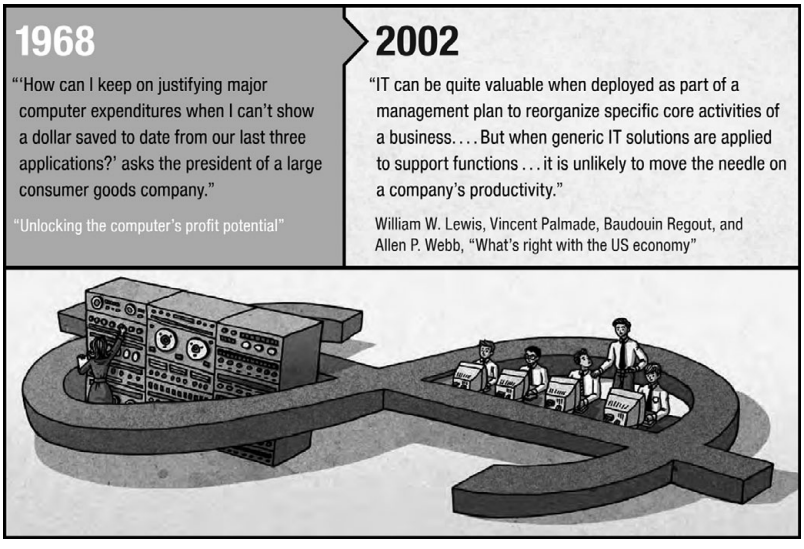


Figure 4.8 A quote from the 1968 report 'Unlocking the Computer's Profit Potential', reprinted alongside with a 2002 quote under a title 'Some things haven't changed much...'. (Source: Exhibit from 'Management, the next 50 years', *McKinsey Quarterly*, 2014 3, www.mckinsey.com. Copyright (c) 2024 McKinsey & Company. All rights reserved. Reprinted by permission.)

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

efforts in all but a few exceptional companies are in real, if often unacknowledged, trouble'.

The basic problem, according to the report, is that most companies completed 'conversions of routine administrative and accounting operations to computer systems', but are now finding it difficult to effectively apply computers in other areas. The computer staff who completed the conversion projects are now leading computerisation efforts, but are 'seldom strategically placed (or managerially trained) to assess the economics of operations fully or to judge operational feasibility'. The McKinsey recommendation was for the top management team to take a more active role in the management of computer projects:

> Many otherwise effective top managements, however, are in trouble with their computer efforts because they have abdicated control to staff specialists … Only managers can manage the computer in the best interests of the business. The companies that take this lesson to heart today will be the computer profit leaders of tomorrow.

The report coincided with a shift in thinking about the nature of programming jobs.[88] Before the end of the 1960s, programming was seen as a creative activity that required unique skills and often also unique treatment. Towards the end of the decade, the literature aimed at top management stopped talking about programmers as weird but ingenious hackers and started seeing them as any other kind of worker. The managerial approach to turning programming into a 'proper engineering discipline' was thus modelled after industrial factories and industrial production lines. The managers hoped that such a model could lead to a reliable and predictable software production, without relying on particular skills of individual programmers. They started looking for a methodology that could control the development process, not so much by appropriately controlling and structuring the code being written, but by appropriately controlling and structuring the team writing it.

Many different attempts at managing programmer teams soon appeared. In Chapter 2, I discussed the Chief Programmer Team methodology and the Cleanroom software engineering approach, which were modelled on the structure of a surgical team and were developed by Harlan Mills at IBM throughout the 1970s. Other approaches include egoless programming,[89] which was a democratic management structure focused on collaboration with a rotating leadership. However, the dominant approaches were top-down development methodologies inspired by the engineering idea of structured programming. The system had to be specified, decomposed into parts and then developed according to the specification by individual teams of programmers. The most prominent structured approach to software engineering became known as the Waterfall development model.

4.9 Production of Large Computer Programs

The software development model that became known as Waterfall in the 1970s was, in fact, already in use in the 1950s during the development of the Semi-Automatic

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Ground Environment (SAGE) system that I briefly mentioned in Chapter 1. SAGE was a system of computers to process live data from multiple radar sites across the USA that was responsible for identifying a Soviet nuclear strike and coordinating the response. It was an effort of unique scale. The core development team 'grew from 125 personnel in 1956 to 1,400 in 1962' and the organisation behind SAGE 'trained more than 700 programmers between 1955 and 1957'.[90]

The development methodology that was eventually used for the development of SAGE has been presented in a 1956 paper 'Production of Large Computer Programs' at the Symposium on Advanced Programming Methods for Digital Computers in Washington DC, but without any specific reference to SAGE, which was not yet publicly announced. The paper came perhaps too early and was largely forgotten, until it was reprinted a quarter of a century later.[92] According to the paper, the development of the main SAGE control program proceeded in nine phases, illustrated in Figure 4.9. The operational plan defines the design requirements and is used to produce operational specifications. This should be sufficiently detailed so that programmers can prepare the program using only machine and operational specifications. Coding specifications then describe the structure of the program with enough detail that makes it 'possible to predict precisely the output of [each component] subprogram for any configuration of input items'. The coding is followed by two-phase testing. In parameter testing, each component is tested according to its specification in isolation, in 'an environment that simulates pertinent portions of the systems program'. This is then

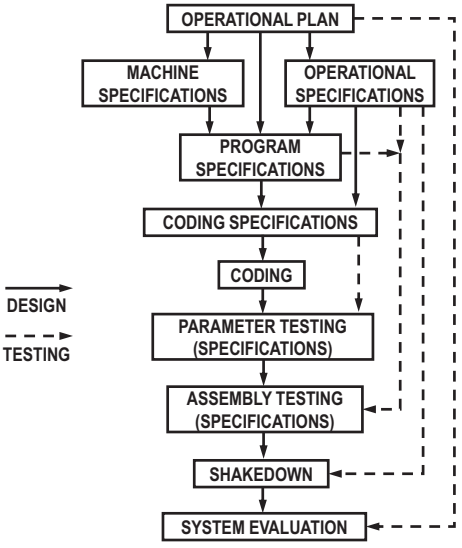


Figure 4.9 'Production of a large program system proceeds from a general operational plan' which defines broad design requirements for the complete control system and is jointly prepared with the user of the system. The dashed line indicates that testing is guided by a specification, not by the coded program.[91]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

followed by a testing of the assembled program, first using simulated and then using live data.

The development methodology of SAGE may not have directly influenced later managerial development methodologies, but it has many aspects that are characteristic of methods that became widely used in the coming decades. Most importantly, it splits the process into a number of phases that are undertaken in sequence. Testing is treated as a separate phase and tests are based on the developed specifications, rather than being based more directly on system implementation. The goal of testing in the SAGE methodology is to show that the system satisfies the specification, that is, it falls into the 'demonstration-oriented' period of testing as classified by Gelperin and Hetzel. The author of the write-up is, however, also anticipating the later debate about the viability of this approach:

> It is debatable whether a program of 100,000 instructions can ever be thoroughly tested – that is, whether the program can be shown to satisfy its specifications under all operating conditions.

The individual phases of the SAGE development methodology are remarkably similar to those of the Waterfall development model that became widely known and used from the start of the 1970s. The Waterfall model is sometimes attributed to Winston Royce, who describes it in a paper 'Managing the Development of Large Software Systems' published in 1970,[93] although without using the term 'Waterfall'. The model, illustrated in Figure 4.10 starts with requirements gathering and analysis, follows with program design, actual coding, and testing, and concludes with program operation. However, Royce describes the basic model only as a starting point for further discussion. He 'believes in this concept', but points out that the basic method is risky and invites failure.

Royce starts with a naïve version of the process that proceeds only in the forward direction, but soon notes that the methodology needs to be iterative. The chief difficulty, according to Royce, is that we may sometimes need to go back by multiple steps. For example, the testing phase, which only occurs after coding is done, may reveal timing, storage and input/output issues. Those are not precisely analysable upfront and may require design changes, which then, in turn, require new implementation. The result is 'up to a 100-percent overrun in schedule and/or costs'. Royce goes on to suggest five ways in which the process can be improved, including two that will become major themes in later developments that I will revisit later in this chapter. The first is to arrange matters so that the version finally delivered is actually a second version. This is an approach that Fred Brooks recommends in his later 'Plan to Throw One Away' essay.[94] The second is to involve the customer prior to the final delivery. This was a method used in participatory design, which emerged in Scandinavia in the 1970s, but it is also the essence of a shift in software development that started in the late 1980s and eventually led to the Agile development methods.

Yet another way in which the process can be improved focuses on testing and is documented with a diagram shown in Figure 4.11. Here, Royce recommends involving a dedicated team of 'specialists in software product assurance' who should test the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

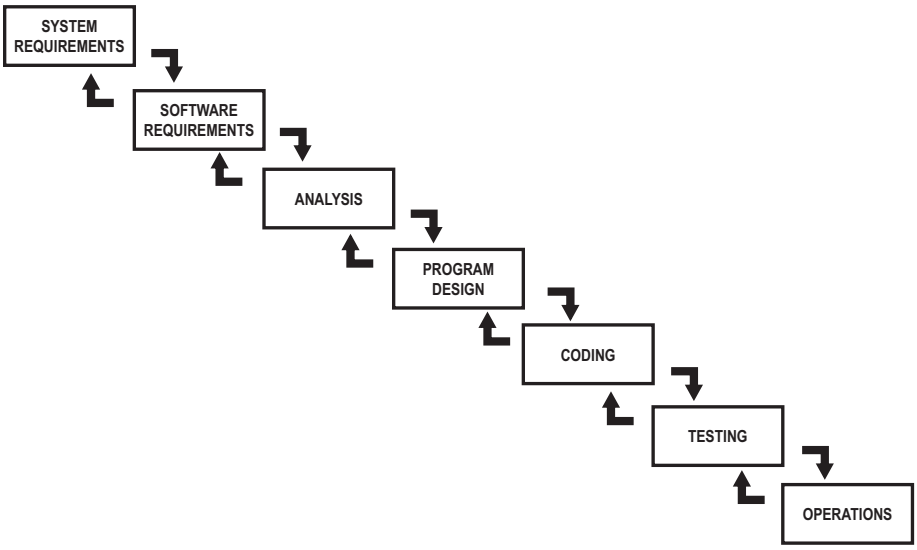


Figure 4.10 'Implementation steps to develop a large computer program for delivery to a customer' as proposed by Royce with iterative interactions, hopefully confined to successive steps.[95]

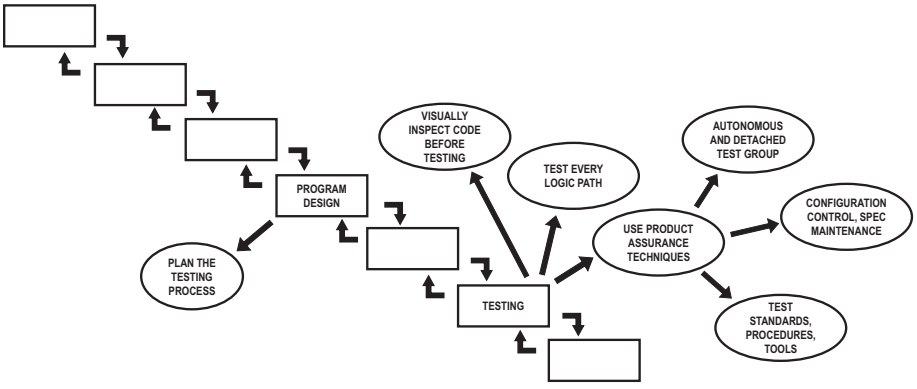


Figure 4.11 Additional aspects for testing outlined by Royce in a section 'Plan, control and monitor computer program testing'.[96]

system based on a detailed documentation, performing manual inspection of code to catch simple errors and manual checkout for identifying more complex bugs. An interesting proposal is to 'test every logic path in the computer program at least once with some kind of numerical check'. This is an informal sketch of the testing approach that was to be developed more formally over the next five years in work on test data selection, which I discussed in the previous section.

Software development methodologies like Waterfall impose a structure on the development process. They organise the work into phases, each of which results

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

in outputs such as documentation, specification or code. This makes it possible to define milestones and monitor project progress. In other words, such methodologies make the opaque hacker process of building software legible to a managerial outsider. The early work on development methodologies illustrates how the managerial culture approaches the problem of programming. In order to professionalise software development, the proponents of the managerial culture also saw the need to standardise the development methodologies and capture the best practice as an industry standard.

There are three notable organisations that contributed to the standardisation of software development processes and the role of testing in those. The Institute of Electrical and Electronics Engineers (IEEE) established a task group on software engineering in 1979. The task group did not, at first, attempt to standardise the process or the best practices for testing. It started by developing a standard for test documentation,[97] which should be used for documenting test activities that occur as part of the software development process. This was followed by two more specific standards, specifying the process for ensuring that products meet their requirements (verification) and that they satisfy user needs (validation),[98] as well as a detailed process for testing individual software components (unit testing).[99] At the same time, the National Bureau of Standards (NBS) published its 'Guideline for Lifecycle Validation, Verification, and Testing of Computer Software',[100] which was intended primarily as a guideline for federal agencies developing new software.

Gelperin and Hetzel chose the 1983 publication of the NBS guideline as the beginning of the 'evaluation-oriented' period of software testing. In this era, tests are used not only to find bugs, but also to assess the project's progress. The NBS guideline recommends a number of different kinds of testing throughout the software development lifecycle. Unit testing is to be done at the end of the programming phase, integration testing and system testing is done at the end of the testing phase, while acceptance testing is done manually, by the customer, at the end of the software installation phase. The guideline also includes regression testing, in order to prevent the reintroduction of errors during the operation and maintenance phase. The presence of multiple separate test phases in the process further increase the legibility of the process, because managers can use it to easily track the project's progress.

The standards published by IEEE and NBS were merely recommendations, but several organisations soon started requiring that software providers follow a specific process. Most notably, the US Department of Defense made its own standard of the software development process a requirement. The standard was published under the code DOD-STD-2167A in 1985. A contractor for the Department of Defense was required to follow a development process consisting of a number of activities such as system and software requirement analysis, preliminary and detailed design, coding, as well as unit testing, integration testing and system testing.

Despite the standardisation efforts, the question of how exactly testing should interact with other activities in the development process remained open. An interesting perspective came from Gelperin and Hetzel, who were the authors of the 1988 paper 'The Growth of Software Testing'.[101] The two were software engineering consultants,

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

who established a joint consulting business in 1986, started organising a conference series on software testing and published a well-regarded book on the topic in 1988.[102] Their retrospective on testing was not written just to document the history of testing, but also to define the new trends. After discussing the 'evaluation-oriented' period, the authors turn to an advocacy of their own approach which they label as 'prevention-oriented' and which purportedly started a year before the publication of their paper and the book.

In the envisioned 'prevention-oriented' period, test planning, test design and development and testing is a ubiquitous activity that happens in parallel with all other steps of the software development lifecycle. Testing is no longer done at certain stages of the process, but throughout the development. Gelperin and Hetzel captured this idea in their consulting company motto, 'Test, then code', which they wore proudly on a blue pin at the Fourth International Conference on Software Testing in 1987.[103] Gelperin and Hetzel discussed their ideas using the vocabulary of the managerial culture. They talked about development phases, testing lifecycle and outputs such as the 'Master Test Plan'. Yet, some of their ideas are closely related to Agile development methods that we will encounter later in this chapter. But first, I want to discuss a more conceptual change that followed the birth of software engineering.

4.10 Disciplinary Repertoire of Software Engineering

The engineering culture of programming that grew in prominence in the 1970s shaped new development methodologies and was behind new approaches to testing and error handling. More importantly, though, it also established a new way of thinking and reasoning about software systems. The new way was distinct from the formal reasoning of the mathematical culture and organisational reasoning of the engineering culture. In her account of the history of missile defence, Rebecca Slayton coined the term 'disciplinary repertoire' of software engineering to refer to this emerging kind of argumentation.[104] The disciplinary repertoire made it possible to look for fundamental laws and principles of software engineering and use those to shape public debates about software systems, such as the missile defence documented by Slayton.

As we saw in Chapter 1, the central notion of debates in the mathematical culture of programming became that of an *algorithm*. The disciplinary repertoire for reasoning about algorithms include primarily formal mathematical methods. For example, the undecidability of the halting problem is a mathematical theorem with a proof. It states that it is impossible to construct an algorithm, which would determine whether an arbitrary computer program will finish running or continue running forever. For the mathematical culture, this has wide-ranging theoretical implications about what kinds of programs can never be constructed. But to the engineering culture, it seems like a theoretical curiosity that tells little about practical limits of building large-scale software applications. The rise of software engineering gradually led to the recognition that a software *system* has a very different nature than a computer *algorithm* and that a different language for talking about each is needed.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The difference between an algorithm and a software system was also crucial for the critique of program verification by DeMillo, Lipton and Perlis that I discussed earlier. When questioning whether program verification can scale up to complex systems, they argue:

> [T]here is no continuity between the world of FIND or GCD and the world of production software, billing systems that write real bills, scheduling systems that schedule real events, ticketing systems that issue real tickets.[105]

The fundamental difference is that 'the specifications for algorithms are concise and tidy, while the specifications for real-world systems are immense, frequently of the same order of magnitude as the systems themselves. The specifications for algorithms are highly stable, stable over decades or even centuries; the specifications for real systems vary daily or hourly. These are not differences in degree. They are differences in kind.' In other words, the disciplinary repertoire for talking about algorithms may work well for algorithmic programs, but it is not applicable to large software systems.

The large software systems that software engineering is concerned with are not 'composed of nothing more than algorithms and small programs'. According to De Millo, Lipton and Perlis, 'the colorful jargon of practicing programmers seems to be saying something about the nature of the structures' that programmers work with. The implementation of any such large systems is made of 'patches, ad hoc constructions, bandaids and tourniquets, bells and whistles, glue, spit and polish, signature code, blood-sweat-and-tears, and, of course, the kitchen sink'. From the late 1960s, software engineers started to figure out what kind of fundamental laws and principles may apply to software systems constructed in such a way.

4.11 Software Aspects of Strategic Defense Systems

The key debates that shaped the disciplinary repertoire of software engineering were reflections on concrete software systems. The focus on concrete problems and systems is characteristic of the engineering culture of programming. We saw that the work on software development methodologies started in the 1950s with the reflections on the development of the SAGE system and that exception handling mechanisms were a response to the concrete problem of dealing with input/output errors. The laws and principles discussed in this section will similarly arise from concrete problems and systems.

The arguments about the anti-ballistic missile (ABM) defence, documented by Slayton,[106] were forming the disciplinary repertoire of software engineering from the the end of the 1960s until the late 1980s. The disciplinary repertoire started taking shape at the time of the NATO Software Engineering conferences, although the most active participants in the discussion about ABM were absent from the two conferences. The USA has been developing various forms of the ABM defence since the 1950s, but the announcement of the new Safeguard system in 1969 raised public profile of the program and triggered a new debate. The Safeguard anti-ballistic missile system

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

was intended to protect US nuclear launch sites from limited forms of enemy attack. A number of policymakers as well as computer scientists opposed it. This was often for political reasons, but opposers understood that a more effective argument needs to be based on technical reasons why such a system cannot reliably work.

The first contribution to the debate came from J. C. R. Licklider, who was instrumental in the development of time-sharing systems and the Internet during his time as a research administrator at ARPA and was then a professor at MIT. Licklider was invited to write a chapter on reliability for a study on the feasibility of the ABM. He soon concluded that the difficulties with the development of similarly complex systems are a common wisdom among the practitioners, but that there is very little well-documented evidence of past failures and their reasons. In his report 'Underestimates and Overexpectations',[107] Licklider resorts to talking about 'common wisdom', which is that 'for a certain class of systems, costs and times tend to be grossly underestimated and performance tends to be mercifully unmeasured'. The tricky class of systems has a number of characteristics. They are complex, work in operating conditions that are not fully under the designer's control, their task changes over time and they are difficult to test in a realistic scenario. Licklider discusses a number of examples, including the SAGE and the American Airlines travel booking system, which both suffered from underestimates and overexpectations. He concludes that:

> [C]omplex software subsystems can be 'mastered' … and made to provide useful and effective service if they can be developed progressively, with the aid of extensive testing … and if they can be operated more or less continually in a somewhat lenient and forgiving environment.[108]

Licklider's analysis raises serious concerns about the possibility of building the software component of the ABM system, which is a complex system that cannot be dependably tested and which operates in an evolving hostile environment. At the same time, Daniel McCracken, who was a consultant and the author of multiple popular programming books, started a more plainspoken effort, the organisation Computer Professionals against ABM (CPAABM). In a press release, the signatories simply stated that the 'Safeguard antiballistic missile system should be abandoned because its control computer will not be able to perform the complex job assigned to it'.[109] This is because the 'pattern of development that must of necessity be followed with Safeguard, is highly unlikely to lead to a successful computer system'. The group argued that there are three main issues with the system. First, its precise task cannot be defined, because the countermeasures developed by the enemy cannot be expected; second, realistic testing of the system is impossible; and third, the Safeguard system cannot be developed through evolution.

The arguments used by the CPAABM are similar to those of Licklider. They focus partly on the inherent complexity of such a system and partly on the properties of the environment in which the system needs to be built and operated. However, the arguments are not based on formal proof or impeccable empirical evidence. They are generalisations based on the somewhat anecdotal evidence about past systems.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The question of necessary evidence for making such claims sparked a debate among the computing community. Herbert Bright, an ACM council member, was one of those who believed that past experience is not sufficient for claims about future systems. He expressed 'a profound distaste for saying we believe it won't work when the truth is merely that we suspect it won't work'.[110] The system would not violate any known physical or mathematical law and so 'it's unfeasible to develop a sound basis for a proof that the proposed system cannot work'. Joseph Weizenbaum did not feel obliged to provide a mathematical proof: 'My suspicion is strong to the point of being belief. I don't think that my statement as a professional that I hold this belief obligates me to a mathematical proof.'[111] However, Weizenbaum did not base his suspicion merely on personal hunches or the analysis of past failures. His belief, interrogated and summarised by Slayton, was based on an informal, but logical argument:

> Computer systems could only become reliable if 'the environment that they are to control or with which they are intended to cooperate must have a change rate smaller than that of the [computer] system itself'. But since the missile defence 'environment is actively hostile and destabilising by intent', the software could not keep up with its changing requirements: 'convergence is logically impossible'.[112]

Weizenbaum's argument is an example of the developing disciplinary repertoire of the engineering culture of programming. It uses logical reasoning to talk about things such as the 'environment' and a 'change rate', which are crucial for large complex systems, but can hardly be defined in a fully formal way. We will see this approach to reasoning about software systems repeatedly in works such as 'The Mythical Man-Month' by Fred Brooks, but also appeared in subsequent debate on the anti-ballistic missile defence.

The Safeguard system was eventually built at one site and was briefly operated, before being shut down, because the politicians accepted that its utility 'will be essentially nullified in the future'.[113] This was not the end of the US ABM efforts, however, and in 1983, Ronald Reagan's administration proposed a new system, Strategic Defense Initiative (SDI), nicknamed the 'Star Wars program'. The Star Wars program was intended to protect the USA from ballistic missiles by destroying them before they hit their target. The program explored various ways of doing this, ranging from laser beams to projectiles dropped from a satellite (Figure 4.12). Again, the system required the development of a very complex control software that would detect the attack, calculate missile trajectories, coordinate interception and discern decoys from actual warheads.

In spring 1985, the SDI established a 'Panel on Computing in Support of Battle Management' consisting of a small number of computer experts. One of those invited to join was David L. Parnas, a computer scientist known for his software engineering work on information hiding. However, on June 28, 1985, Parnas resigned from the panel and published a series of notes that he wrote while being a panel member. In the notes, republished as 'Software aspects of strategic defence systems'[114] by the Communications of the ACM, Parnas concludes that, 'we will never be able to believe, with any confidence, that we have succeeded' in building a trustworthy software for the SDI

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691



Figure 4.12 An artist's concept of a ground/space-based hybrid laser weapon. (Source: US Army, Wikimedia Commons)[115]

and that 'nuclear weapons will remain a potent threat'. Parnas is clear that he opposes the initiative on technical rather than political grounds. His objections are not against any kind of military software, but rather 'on characteristics peculiar to this particular effort'. What makes the SDI software unattainable is its inherent complexity, the fact that it cannot be tested in realistic conditions prior to its actual use, as well as the fact that there won't be sufficient time for human intervention when the system responds to an attack.

The reason that testing and gradual development is the only way to build reliable software is the inherent complexity of large software systems. In his analysis of 'the fundamental technological differences between software engineering and other areas of engineering', Parnas, like Weizenbaum some 15 years earlier, employs logical reasoning for talking about informal concepts. He explains the reasons for software complexity with reference to analog and digital engineering products.

Analog systems have an infinite number of states within a broad operating range and can be modelled using continuous functions. 'Small changes in inputs will always cause small changes in outputs' and so analog systems 'contain no hidden surprises'. It suffices to ensure, through well-understood mathematical analysis, that 'the system components are always operating within their normal operational range'. By contrast, digital systems have a finite number of states that cannot be modelled as continuous functions. Before the advent of computers, the number of states of such systems was typically small enough that exhaustive testing was possible. This is not the case for digital computers, which have a very large number of states.

To obtain correct and reliable hardware, engineers were able to leverage the fact that hardware is built using many copies of smaller digital subsystems, which can be analysed and exhaustively tested independently. But this cannot be applied to soft-

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

ware systems, which have a much larger number of states and no repetitive structure. According to Parnas, 'this difference clearly contributes to the relative unreliability of software systems and the apparent lack of competence of software engineers'. Most importantly, the logical reasoning used by Parnas in his analysis lets him conclude that this 'is a fundamental difference that will not disappear with improved technology'.

4.12 The Mythical Man-Month

The only commercial company that was developing software systems of the same scale as the US military was IBM. It dominated the computer market and was rapidly growing. By the end of the 1950s, IBM had a complex family of products, including multiple ranges of incompatible low-end and high-end computers. In 1961, the company embarked on a new direction and started building a single family of computers that would share the same software and have interchangeable peripherals. The System/360 (Figure 4.13), as the new family was called, was announced in April 1964 at the company's auditorium and was paralleled with press conferences in 164 US cities and 14 other countries. The announcement was followed by a flood of orders from eager customers, but 'it would take all the resources of the company working 60-hour weeks, and in some cases round-the-clock shifts, for nearly two years before the System/360 would be called a success'.[116]

The development of the operating system for the new family of machines, called OS/360, was led by Fred Brooks Jr., who joined IBM several years earlier, after com-



Figure 4.13 The IBM 360/75 Computer, first released in 1965, at the Johnson Space Center in 1975. (Source: Space Frontiers/Stringer/Archive Photos via Getty Images)

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

pleting a PhD in applied mathematics at Harvard. The OS/360 system included two major features that made the development complicated. It supported running multiple programs at once and the ability to handle multiple users through time-sharing interactive use. It also came with 44 new peripherals that all required software support. As the system became delayed, IBM reallocated some of its programmers working elsewhere to the OS/360 effort. The team working on the system eventually consisted of over 1,000 people and the cost of the development, grew from $40 million to $500 million. When Brooks left the company in 1964, the CEO asked him why managing software projects was harder than managing hardware projects. This planted a seed for a series of essays, published as a now classic book *The Mythical Man-Month*[117] a couple of years later.

Brooks' reflection on the development of the OS/360 includes a number of principles that have became well-known slogans in the software engineering community. The most famous one is known as Brooks' law and is based on his experience with adding programmers to the OS/360 development effort in order to make up for the slips in schedule. The law states that 'adding manpower to a late software project makes it later'. The law that was certainly counter-intuitive to Brooks and others at IBM during the development of OS/360 is not just an aphorism based on personal experience. As with the principles I talked about in the previous section, it is supported by a logical argument involving the informal notion of communication.

The reason for the law is that developing a software system in a team requires communication. Most programming tasks cannot be divided into a number of completely independent smaller parts and so the developers will spend some of their time coordinating their work with others. Brooks expresses this point formally when he explains:

> If each part of the task must be separately coordinated with each other part, the effort increases as $n(n-1)/2$. Three workers require three times as much pairwise intercommunication as two; four require six times as much as two. If, moreover, there need to be conferences among three, four, etc., workers to resolve things jointly, matters get worse yet. The added effort of communicating may fully counteract the division of the original task …[118]

In other words, as the team gets bigger, the communication overhead grows faster than the added programming capabilities and the overhead eventually outweighs the gains. The mathematical argument is clear enough even if we do not know the specific numbers, such as the percentage of time spent on communication.

Fred Brooks' essays are not merely about misconceptions to be avoided. He also documents a number of developments in software engineering that he believes can have a positive effect. Brooks argues for a rigorous practice in writing documentation, but he also welcomes new technological developments [sic] such as high-level programming languages, on-line debugging tools and electronic word processing tools (for maintaining a shared up-to-date documentation). He also endorses the surgical team structure developed by an IBM colleague Harlan Mills.

Brooks' analysis of software engineering did not stop with OS/360 and Brooks' law. In 1986, he wrote a new essay that grew out of his later experience reviewing military

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

software. The essay, titled 'No Silver Bullet: Essence and Accidents of Software Engineering', was reprinted a year later in the IEEE Computer magazine. Brooks makes a bold prediction:

> There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.[119]

Again, this is not a baseless claim and Brooks supports it with a logical argument. He argues that software construction involves *essential tasks* and *accidental tasks*. The former deal with the 'complex conceptual structures that compose the abstract software entity', while the latter are concerned with 'the representation of these abstract entities in programming languages and the mapping of these onto machine languages within space and speed constraints'. Essential complexity is caused by the problem that the software solves. A software for filling out tax returns in 10 different countries has to do 10 different things. By contrast, accidental complexity is caused by the imperfection of our programming tools.

According to Brooks, the hard part of producing a software system is dealing with the essential tasks, that is, the specification, design and testing of the abstract software entity. The accidental complexity has been largely simplified by the development of high-level programming languages, although it is still something that programmers need to deal with. The crux of Brooks' argument is that, unless dealing with the accidental complexity 'is more than 9/10 of all effort, shrinking all the accidental activities to zero time will not give an order of magnitude improvement'. Again, the exact number does not matter, as long as we believe that it is less than 9/10. Even if the ratio between essential and accidental complexity was 1:1, reducing the accidental complexity to 0 would only lead to a two-fold improvement.

In 'No Silver Bullet', Brooks also elaborates on the two sources of the essential complexity that we already encountered in the discussion about anti-ballistic missile defence, that is, the inherent complexity of large digital systems and the complexity of an evolving environment. In the discussion of the former, Brooks shares an argument with Dave Parnas who was, for a time, his colleague at the University of North Carolina at Chapel Hill. Software entities are, according to Brooks, 'more complex for their size than perhaps any other human construct'. They are composed of a large number of distinct, discrete elements and, as software gets larger, the situation only gets worse:

> [S]caling-up of a software entity is not merely a repetition of the same elements in larger size, it is necessarily an increase in the number of different elements. In most cases, the elements interact with each other in some non-linear fashion, and the complexity of the whole increases much more than linearly.[120]

The second source of complexity of software systems is that they are 'embedded in a cultural matrix of applications, users, laws, and [machines]. These all change continually, and their changes inexorably force change upon the software product.' These characteristics of the environment in which software systems operate have been the primary concern of another influential contribution to software engineering from the 1980s.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

4.13 Laws of Software Evolution

Like many of those developing the engineering perspective on programming, Meir 'Manny' Lehman had both academic and industrial experience. After studying at Imperial College London, he worked at Ferranti, Israel's Ministry of Defense and IBM's research division before returning to Imperial College. During his term at IBM, the Director of Research at IBM, Arthur Andersen, came across a paper from Bell Labs claiming that a move to interactive programming improved programming productivity three-fold. Andersen came to Lehman exclaiming that 'Anything that Bell Labs can do, IBM can do better!'[121] and asked him to investigate what projects could improve productivity of IBM programmers. The question changed Lehman's research focus for the next 25 years.

The immediate result of this task was a 1969 report 'The Programming Process',[122] which reviewed individual technical advances like interactive programming and high-level programming language, but concluded that the 'solution of the programming problem is seen in the development of a total system-oriented methodology and the associated support technologies and tools'. The focus on methodology assisted by tools positions Lehman's approach firmly in the engineering culture of programming. So do the references in his report, which include works of Dave Parnas and the report from the 1968 NATO Software Engineering conference.

Shortly thereafter, Lehman started looking at the evolution of the OS/360 operating system, which provoked his interest in the dynamics of software growth and the laws that govern it. Together with a colleague, Les Belady, he started using the term *software evolution*. After moving back to Imperial College, Lehman continued studying software evolution by analysing large real-world software solutions and eventually formulated eight laws of software evolution that were published in an influential paper 'Programs, life cycles, and laws of software evolution'.[123]

Lehman opens his analysis by pointing out that program evolution has been always associated with the concept of large programs, but that characterising 'large' and 'non-large' programs has been challenging. To address this, Lehman introduces an alternative and more insightful classification of programs, distinguishing between S, P and E-programs. The types of programs are distinguished not based on their size, but based on their relationship with the environment and the degree to which they can be fully specified. An S-program (*specification*) is an implementation of an algorithm. The problem it solves does not change and the solution can be fully described by a specification. A P-program (*problem*) solves a problem that can be precisely defined, but the quality of the solution is determined by the effect it has in the real world. Lehman uses a chess-playing program as an example. Chess has a clear set of rules, but the solution will involve complex hard-to-specify algorithm and the program quality will depend on the subjective judgement of the user of the program. Finally, E-programs (*environment*) do not even have a fixed problem specification. They 'mechanize a human or societal activity', but in doing so, they are embedded in the environment and change it, often also changing the problem statement. Examples given by Lehman include operating systems, air-traffic control and stock control.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The first consequence of the classification is, according to Lehman, for the notion of program correctness. Whereas an S-program can be fully formally specified and its implementation checked against the specification, the validity of an E-program 'depends on human assessment of its effectiveness in the intended application'. Although Lehman somewhat optimistically suggests 'that as programming methodology evolves still further, all large programs (software systems) will be constructed as structures of S-programs', his laws talk primarily about the most challenging kind of programs, the E-programs.

The focus on E-programs, which is typical for many of the discussions in this chapter, is in a stark contrast with the typical focus of the mathematical culture of programming, which often assumes that programs can be fully specified and, by doing so, restricts itself to talking about what Lehman classifies as S-programs. An eloquent advocate of the mathematical culture, Edsger Dijkstra, describes his focus clearly in a position paper on software reliability[124] by contrasting the 'scientific questions of correctness' with 'the unformalized question whether a tool meeting [its] specifications is in such-and-such unformalized and ill-understood environment a pleasant tool to use'. The narrow focus of mathematical methods is a frequent source of clashes between the mathematical and the engineering culture. For example, Peter Naur responded to Dijkstra by pointing out that pleasantness is 'a somewhat euphemistic word to use about such calamities as airplanes colliding in mid air or atomic reactors exploding'.[125]

Lehman's realisation that programs need to evolve in parallel with the continually evolving environment in which they are embedded poses a major challenge for top-down software development methodologies that analyse the requirements at the beginning of the process, but ignore the ongoing evolution. In other words, the advances in understanding of software development in the engineering culture of programming outpaced the understanding in the managerial culture, which was busy standardising the steps of a top-down development process.[126] By the end of the 1980s, engineers started using their understanding to rethink software development processes.

4.14 Paradigm Change in Software Engineering

Already in 1970, when Royce wrote his critique of the simple i software development model, he was aware of the drawbacks of a process that proceeds forward in phases, from analysis to specification, implementation, testing and deployment. Royce pointed out that more iteration and feedback is needed in the process. He suggested a 'do it twice' method in which a pilot project is completed to understand the problem before building the real system. The same approach was advocated later by Fred Brooks in the 'Plan to Throw One Away' essay, included in his book *The Mythical Man-Month*.[127]

Another suggestion by Royce was to involve the customer not just in the initial requirements gathering stages, but repeatedly throughout the development process. The chief motivation for developing such processes is that fully understanding the problem upfront is almost never possible. This point was also common wisdom in

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

the community. It was pointed out by Dave Parnas when discussing the limits of software engineering methods in the context of anti-ballistic missile software. He argued that 'it is hard to make the decisions that must be made to write [an upfront project specification]. We often do not know how to make those decisions until we can play with the system.'[128]

Throughout the 1980s, a variety of innovations in software development were proposed to address the issue. Newly created development models included *prototyping*, where some part of the cycle is done multiple times to support learning, as well as *incremental* models, where the system is broken down into smaller parts that are developed gradually. A development model that brought many of the preceding ideas together was developed by Barry Boehm. As is typical for the key actors in this chapter, Boehm had an academic background, but spent most of his career in industry. In 1988, he developed the *spiral development model* that is illustrated in Figure 4.14. Rather than proposing a single methodology, the model proposes a way of determining the most appropriate process based on the analysis of risks at each stage. Boehm later described the model

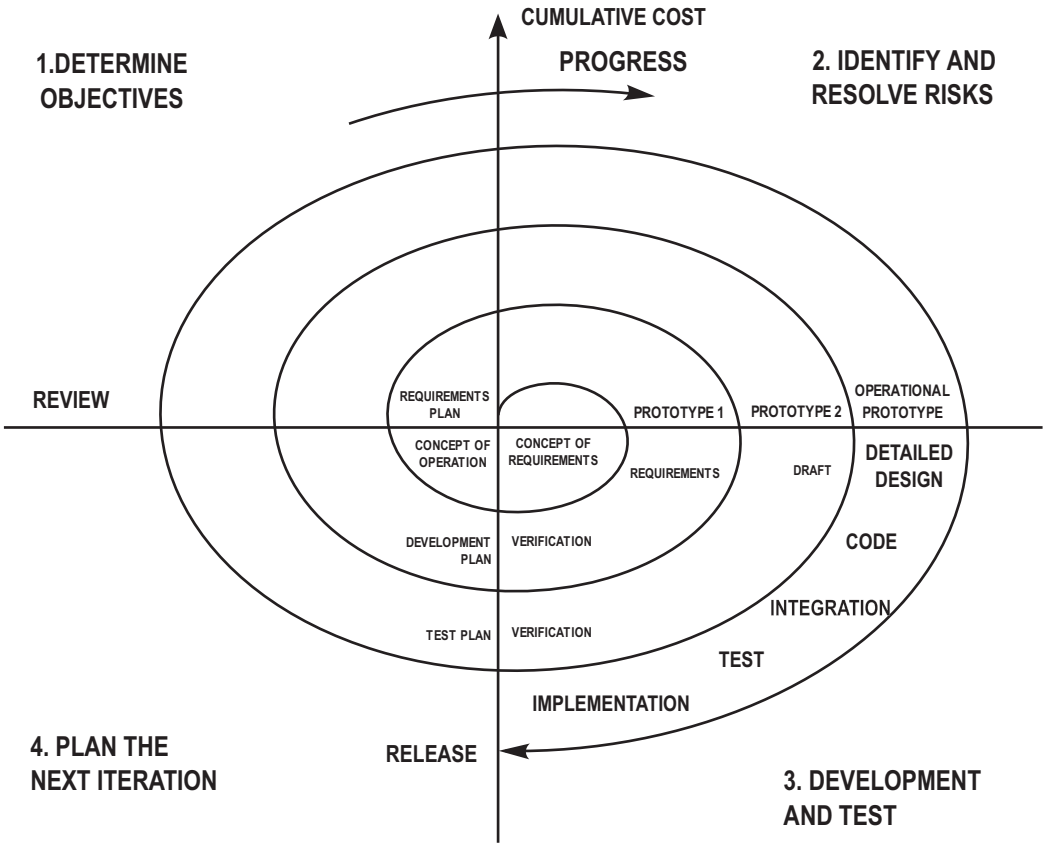


Figure 4.14 A diagram illustrating the Spiral development model. (Source: Adapted by the author from Wikimedia Commons)[129]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

as a 'process model generator', highlighting the fact that the resulting process can dynamically take a number of different shapes.[130]

Despite the numerous innovations, the work on software development models never fully separated itself from the managerial idea of control over the the development process and the programmers involved. In other words, even the more flexible and adaptive development models still aimed to transform the 'black art of programming' into a rigorous discipline that is reproducible, predictable and does not rely on the skills of individual programmers. The processes do this by wrapping the inherent uncertainty in the programming process with enough additional layers to make software development more deterministic.

A more radical vision of how software should be built came from Christiane Floyd. After completing a PhD at the University of Vienna, she worked in several academic and industry jobs, before becoming the first female professor of computer science in Germany in 1978. Since joining the Technical University of Berlin in 1978, Floyd and her group have been working on a new approach to software development 'which is primarily concerned with the development of software as an adequate tool for users'.[131] The approach transcends 'the conceptions of software engineering prevalent in [Germany]. These see it as being based on fixed requirements of the software products to be developed, ignoring the social context in which they are embedded.'[132]

In pursuing this goal, Floyd closely collaborated with Scandinavian colleagues. Following two field trips to Denmark, Norway and Sweden in 1986, she published a trip report that documents the 'Scandinavian Approach'[133] to software development, which makes the user a more integrated part of the process. In Scandinavia, such an approach emerged in the 1970s under the term 'participatory design'. While in Scandinavia, work on participatory design led to the birth of an active research field on human–computer interaction, Floyd used the participatory design approach as the basis for a software development methodology that she named STEPS (Software Technology for Evolutionary Participative System Development). Perhaps the most visionary write-up arising from Floyd's work appears in a chapter 'Outline of a Paradigm Change in Software Engineering',[134] published in the proceedings of a conference held in Aarhus titled 'Computers and Democracy'. In the chapter, Floyd describes the dominant paradigm that emerged following the 1968 NATO Software Engineering conference as *product-oriented* and summarises the emerging *process-oriented* alternative.

According to Floyd, 'the product-oriented perspective regards software as a product'. It 'considers the usage context of the product to be fixed and well understood, thus allowing software requirements to be determined in advance'. By contrast, the process-oriented perspective 'views software in connection with human learning, work and communication, taking place in an evolving world with changing needs'. This approach reflects the changing environment in which software development happens and the difficulties with producing upfront specifications that were gradually being recognised by many researchers and practitioners of software engineering. In the process-oriented approach, the final product 'is perceived as emerging from the totality of interleaved processes of analysis, design, implementation, evaluation and feedback, carried out by different groups of people involved in system development

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

in various roles'. The process-oriented approach admits that requirements change and understanding is built gradually.

In the product-oriented perspective, process aspects such as requirements definition, quality assurance, user acceptance and software modifiability are considered 'as additional aspects outside the realm of systematic treatment'. Floyd holds that 'this situation is inherently unsatisfactory and can only be remedied if we adopt … the richer process-oriented perspective as our primary point of view'.

The perspective from which Floyd approaches software engineering cannot be easily classified as belonging to any of the individual cultures that appear repeatedly in this book. Like other development methodologies, STEPS can be seen as emerging from a managerial attempt to structure the development process. However, the process-oriented methodology also makes individual engineers more autonomous as they are involved in learning about the problem and shaping the final product. The recognition that software cannot be precisely specified upfront resonates with the insights developed within the engineering culture of programming, including the understanding of E-programs developed by Lehman. However, Floyd's work also emphasises human learning and the Scandinavian participatory design, which connects it to the humanistic culture that we followed in Chapter 3. Her work thus illustrates my claim that interesting ideas often emerge at the intersection of multiple cultures of programming.

When Christiane Floyd wrote about the paradigm change in software engineering, she talked about an 'ongoing controversy between rivalling ideas and attitudes'. She clearly saw a glimpse of an emerging new approach to software engineering, but it took 15 more years before the new paradigm took a form that enabled it to dominate the field of software engineering.

4.15 The New New Approach

At the same time Christiane Floyd was thinking about the Scandinavian approach to software development and about the ongoing paradigm change, a similar idea emerged in the USA in the more general context of product development. Hirotaka Takeuchi and Ikujiro Nonaka had management and operational theory backgrounds and gained experience in both industry and academia in the USA and Japan. In 1986, they published an article 'The New New Product Development Game'[135] in the *Harvard Business Review*, in which they analysed a 'new approach to managing the product development process'.

Takeuchi and Nonaka looked at six specific products including a Honda city car and two Canon cameras and reported that companies are 'increasingly realising that the old, sequential approach to developing new products simply won't get the job done' in a fast-paced, competitive world where speed and flexibility are essential. They liken the new approach of the pioneering companies to rugby 'where a team tries to go the distance as a unit, passing the ball back and forth'. Using the new approach:

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

> The product development process emerges from the constant interaction of a hand-picked, multidisciplinary team whose members work together from start to finish. Rather than moving in defined, highly structured stages, the process is born out of the team members' interplay.

In contrast to the humanistic Scandinavian approach focused on shared knowledge building, the wording here is more managerial and emphasises teams and initiative. Yet, the ultimate vision of a collaborative and a more dynamic approach is closely related. The authors talk about a new product development organisation, but the organisation is nothing like that of the managerial structures of the 1970s. In the new approach, teams have a high degree of autonomy, knowledge is shared and exchanged between team members and the process is controlled by more subtle indirect mechanisms.

Takeuchi and Nonaka talked about product development more generally, but in the early 1990s, the ideas about the 'new approach to product development' served as a motivation for the next step in the evolution of software development methodologies. The SCRUM methodology, developed by Ken Schwaber and Jeff Sutherland, was first presented at a workshop on Business Object Design and Implementation,[136] associated with the OOPSLA conference. By now, the authors took for granted that software development occurs under rapidly changing circumstances and that the development requires maximum flexibility. They contrast SCRUM to other development methodologies, notably Waterfall, iterative and spiral models. In SCRUM, the development is done in short sprints which are always followed by review, which makes it possible to adapt more rapidly to the lessons learned during the development.

SCRUM remains rooted in the managerial approach to programming. It describes how teams and the development process should be organised, rather than, for example, recommending particular tools. It does, however, reflect a shift in the managerial culture. Whereas the managerial approach to software development in the 1970s attempted to eliminate dependence on individual programmers and their skills, the new methods that emerged in the 1990s see programmers as creative and responsible professionals that should be trusted with making technological decisions and empowered to learn and contribute to product development.[137]

4.16 A Programming Environment for a Timeshared System

Before we continue our discussion of software development methodologies, we need to look to another source of ideas for how programming should be done. In the 1970s, the hacker culture that advocated for interactive programming and developed on-line debugging techniques in the 1960s continued mastering their craft. The main vehicle for the continued development was Lisp, which was now seen as one of the most prominent 'AI languages' and was used for research and development of expert systems and systems for natural language processing and knowledge representation. Lisp was used either on time-sharing systems or on personal 'minicomputers' and in particular on Lisp machines (mentioned in Chapter 3), which was a class of minicomputers designed to efficiently execute Lisp.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The hacker culture continued to develop at MIT on the East coast, but also at Stanford where John McCarthy moved in 1962 to establish the Stanford AI Laboratory (SAIL) and also at Xerox PARC following its founding in the year 1970. There continued to be an active exchange of ideas between the centres, especially between SAIL and PARC which were a mere two miles apart. As is typical for the hacker culture of programming, how exactly programming was to be done was primarily the unwritten knowledge of hackers, but we can get a glimpse of some of the practices and tools from a rare report on the programming environment that was in place at SAIL from the mid 1970s and from personal recollections of some of the participants.[138]

The programming environment at SAIL was built around a time-sharing system and while personal computers were becoming increasingly commonplace at the time, the authors thought that 'it is not necessary to move to personal computers for a very usable programming environment'. Users interacted with the system through a text-based video display and custom-made keyboards. The primary interface for working with the system was an interactive, 'display-oriented text editor' called E, which later influenced the well-known EMACS editor. Unlike the LISP editor created by L. Peter Deutsch in the 1960s, E was text-based and was used to interface with Lisp as well as other subsystems of the SAIL time-sharing system.

Following the hacker way of thinking about programming, the SAIL programming environment was interactive. The E editor was not used merely for editing source code, but for controlling the system. It was possible to use it to edit files in the storage, but it was equally possible to use it in a special EVAL mode 'which allows the user to type text directly to Lisp'.[139] In this mode, output from running the Lisp code was returned to a selected destination in the editor. Notably, this was also the case when the Lisp program encountered the `break` command. The debugging was, again, done through the interface of the E editor. This way of working is reminiscent of modern Read-Eval-Print Loop (REPL) shells, but the SAIL system went even further. The designers believed that 'if a person can do something while sitting in front of a computer, it should be possible to write Lisp code to do the same thing'.[140] The E editor was built around the notion of commands and it could accept commands from the user, as well as from any subsystem. It was thus possible to write and run Lisp code that would control the editor. Despite having a programming language underneath, the programming environment at SAIL was interesting as a stateful, interactive programming system. This is the perspective that is easily recognised by the proponents of the hacker and humanistic cultures, but that is difficult to grasp for those working in the mathematical and perhaps even the engineering cultures.

The way to master the development of complex systems in the hacker culture of programming was thus not by developing a strict methodology, but by having an adaptable programming environment, a community of skilled hackers and a local 'Lisp wizard down the hall'.[141] Learning how to work in such an environment required learning from the local wizards. At the SAIL system, this was made easier by the fact that the system was equipped with a video switch. The switch made it possible for multiple people to interact with the same (virtual) terminal through their own display and keyboard, a practice called 'mapping'.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The system gave rise to two practices.[142] First, it was possible to share the terminal when tackling a difficult programming problem. A programmer could send a message to the local wizard (through a facility implemented in E) to ask for help. The wizard could then connect to the other person's terminal and control it through his own keyboard, while using an internal phone system to talk. Second, a newcomer could learn how to program and debug in the environment by connecting to the virtual terminal used by an experienced hacker (the local wizard down the hall) and watching them program. This degree of openness was expected by hackers, but it was unexpected for outsiders. As recalled by Gabriel:

> [V]isitors and new professors complained about people being able to 'spy' on them while they were working. In response, the systems people implemented a way to hide a screen (prevent mapping), but the wizards demanded a way to override that, which was also implemented. The privacy freaks demanded that they be notified when such an override was invoked, and the systems people implemented that as well as a way to override the override notification. Information is free.

The way of working that was commonplace in the hacker culture of programming included many aspects that resembled or prefigured new ways of thinking about programming methodologies in the engineering culture. Partly because of personal beliefs, but partly also as a consequence of working in a more experimental problem domain, the hacker way of working rarely needed an upfront specification. Programming was seen as learning and problem exploration. This may have looked like 'black art' in the 1960s, but was becoming an increasingly widely adopted perspective in the 1980s. The programming practices that hackers developed are also precursors of later engineering practices such as pair programming.

To what extent and how exactly hacker practices influenced later developments in engineering development methodologies is difficult to trace precisely. However, SAIL was only two miles away from Xerox PARC where Smalltalk was developed at the same time and, as we will see in Chapter 6, the community that emerged around commercial Smalltalk in the 1980s, included the two pioneers of Extreme Programming. It is likely that at least some of the practices developed by hackers contributed to the subsequent work on programming methodologies.

4.17 Extreme Programming

The need for a new development methodology was in the air at the end of the 1980s and new ideas have emerged from a variety of programming cultures. The Scandinavian process-oriented approach adopted many humanistic ideals, the SCRUM methodology followed a managerial approach, while the hacker practices influenced how practical programming was done in the engineering culture of programming. The mix resulted in a new development method that emerged in the 1980s and gained prominence in the 1990s when it started to be called Extreme Programming (XP). As with the work of Christiane Floyd and the SCRUM methodology, XP tackled the typical issues of its time, that is, the development of software in an environment where requirements are

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

vague and rapidly changing. What makes XP interesting for my story is not the various ways in which it is similar to or differs from SCRUM or the Scandinavian approach, but the fact that it has quite different origins.

Extreme Programming was not developed by academics or management consultants, but by software engineers. The methodology was developed and later publicised by Kent Beck,[143] who nevertheless credits many ideas to 'common knowledge' and also his collaborator Ward Cunningham. Both Beck and Cunningham were consultants, working on commercial object-oriented systems developed using the Smalltalk programming language. They were looking for better ways of building systems and explored a range of ideas, such as building on the concept of pattern languages conceived by the architect Christopher Alexander, which I return to in Chapter 6. They were also a part of the Smalltalk community, which had a long tradition of empowering individuals and allowing them to shape their tools. The Smalltalk community, in turn, had close connections to the hacker culture of programming that was working in their own interactive programming environments such as that developed at the Stanford AI lab.

All of these aspects are mirrored in the XP methodology, which is based on decentralised decision making, rapid feedback and adaptation as well as focus on tools and best programming practices. The grounding in the engineering culture of programming is perhaps best summarised in Erich Gamma's foreword of Beck's book, which opens by saying that 'Extreme Programming nominates coding as the key activity throughout a software project. This can't possibly work!'[144]

The engineering origins of the Extreme Programming methodology shift the attention from managerial issues to engineering, but the methodology still talks about a number of management aspects. In particular, XP identifies a number of different roles in the team. The two most prominent ones are a programmer and a customer, who is now a direct part of the team, in accordance with what Christiane Floyd envisioned. The role of the customer is a challenging one. The customer decides what the software should do, specifies the tests for the required behaviour, while doing their regular job. In the first widely discussed project completed using the XP methodology, the Chrysler Comprehensive Compensation System (C3) payroll system, the customer was able to do all this work, but eventually found the work too stressful and moved on to a different position. The team soon realised that finding another 'XP customer' was more challenging than expected, which resulted in an ominous presentation 'Will Extreme Programming kill your customer?' presented at the OOPSLA conference in 2001.

Extreme Programming shows its engineering origins in that it describes a number of specific programming practices. Many of those are not new. As explained by Kent Beck, 'To some folks, XP seems like just good common sense. So why the "extreme" in the name? XP takes commonsense principles and practices to extreme levels.'[145] Beck lists 12 practices covering areas such as project planning, customer involvement and testing. For example, 'pair programming' requires that all code is written by two programmers at one machine. The practice should not be used, for example, only when solving a particularly tricky problem, but all the time. 'Testing' requires continually written unit tests created by programmers and customer-created tests demonstrating

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

that individual features are completed. In contrast to earlier methodologies, testing in XP becomes the driving force behind development. Another principle is that of 'refactoring', which asks programmers to continually restructure code in order to improve its structure, simplify it and ensure it communicates the intent better.

Extreme Programming brings back to prominence the practice of software testing that keeps appearing, disappearing and evolving throughout this chapter. When discussing the professionalisation of testing, I pointed out that testing became clearly distinguished from debugging in the 1970s. Companies started hiring 'test technicians' to work on automated software testing tools and, with the start of the destruction-oriented period, testing started to focus specifically on finding errors in programs. In most of the software development processes that I have talked about so far, testing exists as a separate phase to be done at a specific point during the development. Sometimes, the process also draws an explicit distinction between unit testing, which focuses on individual units of functionality, integration testing that focuses on the overall system and acceptance testing to show that the product matches requirements specified by the customer.

In XP, testing took the extreme form of Test-Driven Development (TDD). In this practice, an automated test for a required functionality needs to be created even before the programmer writes any code to actually implement the functionality. The idea was not new. As discussed earlier, Gelperin and Hetzel's consulting company Software Quality Engineering promoted the motto 'Test, then code' as early as 1986, and even put it on their lapel pins at a software testing conference in 1987. Yet, TDD treated the idea in a new way in that it took it as the basis for controlling the development process.

Test-Driven Development, which eventually evolved into a software development methodology of its own,[146] fully completes the shift from the managerial culture of programming to the engineering culture. It turns tests from something that can be done manually to a software artifact that has to be automated. Tests are not used just for ensuring that a certain functionality is implemented correctly, but they serve as a description of the system. In TDD, programmers start by writing tests for a minimal system that could be possibly run and then gradually add tests that describe further functionality, ensure that the tests pass by implementing the necessary logic and then proceed to make the code simpler and more reusable before moving on to the next feature. The tests can also be seen as capturing the understanding of the problem, jointly developed by programmers and the customer. Although proponents of the mathematical culture would find the idea absolutely inadequate, tests can even be treated as a partial executable specification.

TDD shows its connection to the engineering culture in that it is structured around tools, automated test runners. At the same time, it makes another attempt at fulfilling the vision outlined at the 1968 NATO Software Engineering conference, which was to turn the black art of programming into a science of software engineering. Already in 1968, one of the conference participants, A. I. Llewelyn, working at the short-lived UK Ministry of Technology, pointed out that 'testing is one of the foundations of all scientific enterprise'.[147] Test-Driven Development develops the parallel between programming and science in that it outlines a systematic software development process

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

based on testing. But just as actual scientific work is more complicated than its naive image, so is software development even when following the TDD methodology.

4.18 The Debugging Paradox

In 1965, Mark Halpern concluded his article 'The Debugging Epoch Opens' by claiming that what 'is certain is that the debugging problem must be confronted squarely and soon if the computer is to take on some of the critical roles it is presently being cast for'. He warned that 'the danger is real and imminent' and argued that the 'remedy will require the commitment of the programming community's full resources'.

In many ways, Halpern was correct. Eliminating program bugs and, more generally, making sure that programs do what they are intended to do has become a major effort in the proceeding decades and remains as important an issue as ever. In one way, however, his prediction was mistaken. The interesting developments that we followed in this chapter focused on testing and software development processes, but very little on *debugging*. This may be a historical accident, because the meaning of phrases like program checkout, testing and debugging kept shifting and evolving throughout the 1960s and 1970s. Yet, when Halpern wrote his article, debugging was already increasingly associated with 'making sure that programs run' whereas testing referred to 'making sure that it solves the problem'.[148]

So, why didn't I have more to say about debugging? The reason is that the basic practices of debugging today are not unlike that discussed in a review of on-line debugging techniques by Evans and Darley in 1966.[149] Just like when debugging programs using a Lisp system in 1966, programmers today set breakpoints to pause program execution, explore the values of variables, modify the program and then test the result. Unlike in 1966, this works reliably for high-level programming languages, which is a notable technical achievement, but there have been few conceptual innovations.[150] It is difficult to say what the reasons for this are with any certainty, but thinking about the interactions between the different cultures of programming in this chapter offers one possible answer.

In the 1950s, programmers talked about debugging, testing, program checkout and error handling somewhat interchangeably. This was the era of the black art of programming and dealing with errors was a 'private arcane matter'.[151] As testing became distinguished from debugging, the term 'test' slowly started to be used not just as a verb, but also as a noun. A 'test' became a description of a check to be performed, first manually and later automatically. This, however, also made a test into an artifact that could be shared with programmers across cultures. In other words, a test became a boundary object.[152] This may have made it easier for other cultures of programming to contribute to work on testing. Managers could make the running of tests part of their processes, mathematicians could study tests formally and engineers could build new testing tools. Eventually, Test-Driven Development made an executable test into a primary mechanism for controlling the software development process.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The case of error handling shares similar characteristics with that of testing. In the early days of programming, the handling of program errors was fully within the realm of the hacker culture. In the 1960s, Lisp and PL/I added a programming language construct for working with errors; in the 1970s, the exception handling mechanism appeared and was adopted by Ada and many programming languages that followed. Here again, the vague programming practice results in an artifact, or a boundary object, that could be shared across cultures. Engineers started implementing programming languages with exceptions, mathematicians were able to formally analyse programs using them, as well as to propose new ways of working with exceptions.

Similarly to tests, which gave rise to a new development methodology, exceptions gave rise to a new programming style in the 1980s. While working on the development of fault-tolerant systems for a telephone exchange, Joe Armstrong at Ericsson came up with a new way of handling exceptions for their programming language Erlang. In telecommunications, systems are designed to 'run forever' and must be resilient to hardware failures. As such, other computers must respond to errors. To support this, Erlang programs are structured as sets of lightweight processes, which may execute on separate machines. When an exception occurs, the affected process is killed. The crash will then be handled by a dedicated supervisor process that may restart the process or activate another system. In Erlang, exceptions became a key building block for building reliable fault-tolerant systems.[153]

In contrast to testing and exception handling, debugging remains a hacker practice with no simple boundary objects that could be exchanged with other cultures. As is characteristic of the hacker culture of programming, the knowledge is often unwritten and it remains the case that 'much of the work [on debugging] has been described only in unpublished reports or passed on through the oral tradition'.[154] Even the internal memoranda documenting debugging tools often merely provide a reference manual for the tools, rather than documenting how to actually use them.[155] Building sophisticated modern debugging tools is an engineering challenge and mathematical work on program analysis contributed valuable ideas for debuggers, but the process of debugging remains a private arcane matter.

4.19 Different Ways of Trusting Software Systems

Some of the most interesting developments discussed in this chapter happened when different cultures interact, often by exchanging and contributing to a specific technical concept. Each culture brings its own methods and ways of thinking about programming. While the concept of debugging remained within the realm of the hacker culture, concepts such as testing were broadly adopted and developed. Different cultures of programming are also apparent in the way programmers think about the limits of what software can be built and what they see as sufficient criteria for trusting that a system is correct.

As we saw in Chapter 2, for the mathematical culture of programming, a formal correctness proof is the ultimate goal. A fully trustworthy software system should come

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

with a proof showing that the implementation corresponds to a formal specification and the trust in proofs may be established through a social process or through mechanical checking. In reality, proving an entire system correct is often unachievable. In practice, the proponents of the mathematical culture will also, even if uneasily, trust systems where the formal proof concerns a simplified model or only a core part of the system.

The mathematical culture also recognises mathematical laws as the only limitation of what software can be built. To put it simply, it is impossible to build a system that solves the halting problem, but anything else is doable. The proponents of the mathematical culture believe that there is no fundamental conceptual difference between a compiler and a software to detect and target ballistic missiles. The question whether a sufficiently precise specification of the system can exist is out of the scope for computer science. To make this point explicit, Edsger Dijkstra distinguishes between the problem of correctness (whether a system meets a specification) and of pleasantness (whether the system is one we wanted to have).[156]

The managerial and engineering cultures are more concerned with practical constraints of building complex software systems that lack clear specifications and are built in an evolving environment. The disciplinary repertoire of software engineering made it possible to claim that certain systems are impossible to build, for example, when the change rate of the environment is higher than the change rate at which the system can be built or when progressive development with continuous testing is impossible.

What exactly is required by the managerial and engineering cultures in order to trust a system has been evolving over time. In the managerial culture of the 1970s, a system can be trusted if it was built by correctly completing all the required phases of the selected software development methodology including, most notably, acceptance testing by the customer. The trust is thus mediated through a process agreed ahead of time. In the case of the engineering culture, the trust in software system relies on the responsibility of individual programmers, but it can be supported by tools. For example, in Test-Driven Development, a tool can measure code coverage, which is a metric of what proportion of code has been executed during testing. While some engineers will emphasise that this is a simplistic metric, many are likely to trust systems with a higher code coverage.

Finally, in the hacker culture of programming, the trust is mainly placed in individuals. In the 1950s, those were the masters of the 'black art' of computer programming, but a similar level of trust still exists in highly technical, especially open-source, programming projects such as the Linux kernel or in the development of programming languages. For example, some open-source communities award the Benevolent Dictator for Life (BDFL) title to the project leader who is trusted with making the ultimate decisions about the project.

Donald MacKenzie discusses the issue of trust with reference to historical and sociological research on risk and trust in the societies of high modernity. His summary is that 'in traditional communities face-to-face interactions typically obviated the need for formal structures of objectivity'. In a society of a high modernity which, among other things, relies on increasingly complex computer systems, 'face-to-face solutions

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

are no longer viable. Modernity's "trust in numbers" is a substitute for absent trust in people'.[157]

In the case of software development, the question of trust does not follow a simple progression from traditional communities to a society of high modernity. The hacker culture still relies on the equivalent of face-to-face interactions, whereas the mathematical culture is fully committed to the modernist trust in numbers. The managerial and engineering cultures are somewhere in the middle of the spectrum. Although individual responsibility is important in both engineering and managerial culture, the two cultures rely on processes and tools, respectively, as crucial components for trust.

Notes

1 Based on the answer suggested by Hoare (1996).
2 Inspired by Halpern (1965), who predicted that the 1960s and 1970s would be the epoch of debugging.
3 Buxton et al. (1970).
4 Goodenough and Gerhart (1975), Musa (1975).
5 This recent way of thinking about testing has been documented, for example, by Mugridge (2003).
6 Royce (1970).
7 Floyd (1987).
8 The list is based primarily on works by Lehman (1980), Brooks Jr (1975) and Parnas (1985) that will be discussed later in the chapter.
9 As pointed out by Licklider (1969), lessons learned in the 1950s and 1960s have resulted in 'common wisdom'. His work was one of the first attempts to make such knowledge more explicit.
10 Brooks Jr (1975).
11 Documented by Slayton (2013) and discussed below.
12 MacKenzie (2004).
13 Hoare (1996).
14 Gschwind (1952), quoted in Bissell (2007).
15 Bissell (2007).
16 Using a term introduced by von Neumann and Goldstine (1947). The historical significance of the problem of controlling the evolution of a meaning is discussed in detail in a chapter on notations by PROGRAMme (2022).
17 www.flickr.com/photos/lselibrary/3833724890/.
18 Bartik (2013).
19 https://ftp.arl.army.mil/ftp/historic-computers/.
20 Haigh et al. (2016).
21 http://pdp10.froghouse.org/KI10/.
22 Pirsig (1999).
23 Bartik (2013).
24 Documented by Priestley (2011).
25 Campbell-Kelly (1992).
26 Campbell-Kelly (2011).
27 Initial orders and subroutines have been described in a textbook by Wilkes et al. (1951). A useful historical summary can be found in Campbell-Kelly (2011).
28 Campbell-Kelly (1992).
29 Both presented in Wilkes et al. (1951) and documented by Campbell-Kelly (2011).
30 https://americanhistory.si.edu/collections/nmah_334663.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

31 National Museum of American History (2024).
32 Kidwell (1998).
33 Turing (1949).
34 Gilmore (1958b).
35 As is typical for the hacker culture of programming, the system was only ever described in an internal memo (Stockham and Dennis, 1960), written for the benefit of the MIT hacker community.
36 See Evans and Darley (1966) for the review and Evans and Darley (1965) for the DEBUG tool.
37 Evans and Darley (1966).
38 As usual, the systems have been described in various internal memos including ones by Martin and Hart (1964), Teitelman (1965) at MIT and one by Russell (1963) from Stanford.
39 Evans and Darley (1966).
40 Bobrow et al. (1967).
41 The `EDIT` and `BREAK` functions were first described by Teitelman (1965).
42 Halpern (1965).
43 This interpretation is offered by Ensmenger (2012).
44 Lindsey (1996b).
45 Pelaez Valdez (1988).
46 The history of the conference has been documented by Pelaez Valdez (1988).
47 As discussed in Chapter 2, the extent to which the NATO conferences contributed to the rise of the engineering culture is open to discussion. Ensmenger (2012) talks about 'major cultural shift', while Haigh (2010) argues that the impact is hard to 'square with the actual historical record'.
48 Llewelyn and Wickens (1968).
49 http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.
50 Buxton et al. (1970).
51 The context in which the PL/I language was developed has been documented by Astarte (2019).
52 Radin (1978).
53 McCarthy et al. (1962).
54 IBM (1967).
55 For a detailed account of the history of PL/I and its formalisations, see Astarte (2019).
56 Ryder et al. (2005).
57 Goodenough (1975a,b).
58 Liskov (1993).
59 As recalled in the history of the Ada project written by Whitaker (1993).
60 Ichbiah et al. (1979).
61 Steele and Gabriel (1996b).
62 Goodenough (1975a).
63 Liskov (1993).
64 McCarthy et al. (1962).
65 The idea has been introduced by Plotkin and Pretnar (2009) and a more accessible review of the feature can be found in a report by Chandrasekaran et al. (2018).
66 The modern term for the specific practice is *offensive programming* and seems to have been introduced in the late 1990s, but the term defensive programming dates back to a 1975 book by Yourdon (1975).
67 Randell (1975).
68 A notable example of this is the system designed by Yau and Cheung (1975).
69 Armstrong (2007).
70 Quoted in MacKenzie (2004).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

71 Hetzel (1973).
72 McGonagle (1972).
73 McGonagle (1972).
74 Gelperin and Hetzel (1988).
75 Buxton et al. (1970).
76 The three examples are referenced by MacKenzie (2004).
77 Huang (1975).
78 Goodenough and Gerhart (1975).
79 McCabe (1976).
80 Myers et al. (1979).
81 Huang (1975).
82 A demo, presented at the AFIPS conference by Budd et al. (1978) and a paper by DeMillo et al. (1978).
83 Ensmenger (2012).
84 Reported by Licklider (1969) and discussed by Slayton (2013).
85 Brooks Jr (1975).
86 Haigh (2010).
87 McKinsey (1968).
88 The shift has been documented in detail in the work of Ensmenger (2012).
89 Introduced in 'The Psychology of Computer Programming' by Winberg (1971).
90 See Slayton (2013), who documents not just the development of the SAGE, but the entire socio-technical context of missile defence software.
91 Benington (1983).
92 Benington (1983).
93 Royce (1970).
94 Brooks Jr (1975).
95 Royce (1970).
96 Royce (1970).
97 IEEE Standard for Software Test Documentation, ANSI/IEEE (1983).
98 IEEE Standard for Software Verification and Validation Plans, ANSI/IEEE (1986).
99 IEEE Standard for Software Unit Testing, ANSI/IEEE (1987).
100 National Bureau of Standards. (1984).
101 Gelperin and Hetzel (1988).
102 Hetzel (1988).
103 This and many other events in the history of testing are documented in the excellent online reference by Meerts and Graham (2010).
104 Slayton (2013).
105 De Millo et al. (1979).
106 Slayton (2013).
107 Licklider (1969).
108 Licklider (1969).
109 Available at https://digitalcollections.library.cmu.edu/node/16631.
110 Quoted in Slayton (2013).
111 Quoted in Slayton (2013).
112 Slayton (2013).
113 Finney (1975).
114 Parnas (1985).
115 https://commons.wikimedia.org/wiki/File:Ground-Space_based_hybrid_laser_weapon_concept_art.jpg.
116 As reported by Boyer (2004).
117 Brooks Jr (1975).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

118 Brooks Jr (1975).
119 Brooks (1987).
120 Brooks (1987).
121 As recalled by Lehman in an oral interview conducted by William Aspray (Lehman, 1993).
122 Later republished as part of a joint book (Lehman and Belady, 1985).
123 Lehman (1980).
124 Dijkstra (1977).
125 Naur (1993).
126 The different pace of development in the two cultures of programming is reminiscent of the intercalated periodisation in the history of microphysics by Galison (1997), where instrument, theory and experiment sub-cultures also advance at a different pace.
127 Brooks Jr (1975).
128 Parnas (1985).
129 https://commons.wikimedia.org/wiki/File:Spiral_model_(Boehm,_1988).svg.
130 Boehm (1988).
131 Floyd et al. (1989).
132 Floyd et al. (1989).
133 Floyd et al. (1989).
134 Floyd (1987).
135 Takeuchi and Nonaka (1986).
136 Schwaber (1997).
137 As noted by Fowler (2001), the shift did not happen solely in software development, but is a more general move in product design and manufacturing.
138 Gabriel and Frost (1984) and Richard P. Gabriel (personal communication, 17 February 2024).
139 Gabriel and Frost (1984).
140 Richard P. Gabriel (personal communication, 17 February 2024).
141 Steele and Gabriel (1996b).
142 As recalled by Richard P. Gabriel (personal communication, 17 February 2024).
143 The publication that popularised the idea is a book by Beck (2000).
144 Beck (2000).
145 Beck (2000).
146 Outlined by Beck (2003).
147 Naur et al. (1969).
148 Gelperin and Hetzel (1988).
149 Evans and Darley (1966).
150 Modern Smalltalk-based programming environments are perhaps an exception from this rule, but the innovations developed in those systems rarely leave the particular technical context from which they originate.
151 Backus (1980).
152 Using the terminology introduced by Star and Griesemer (1989).
153 Armstrong (2007).
154 Evans and Darley (1966).
155 Using the terminology of Polanyi (1958), debugging is a kind of personal knowledge.
156 Dijkstra (1993).
157 MacKenzie (2004).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

5 Programming with Types

Teacher: Types are a good follow-up to a chapter on software engineering. They are another widely used concept that makes programming safer and more reliable. Are they related to concepts like tests and exceptions?

Pythagoras: Types are more fundamental! They originated in work on formal logic before there were any computers. Bertrand Russell invented types in 1903 to avoid paradoxes of the kind 'sets of all sets that are not members of themselves'. Alonzo Church then adapted the idea for his lambda calculus in 1940. Programming languages with a reasonable notion of type directly follow the schemes devised by Church.

Archimedes: This is a fascinating history, but your retelling is misleading. The notion of types that is similar to the one introduced by Church, which you like to call 'reasonable', only appears in typed functional languages in the 1970s!

Teacher: Were there any types in earlier programming languages, say in FORTRAN?

Archimedes: If you search for the term 'type' in a 1956 manual,[1] you'll find that FORTRAN has 32 types of statements, two types of constants and two types of variables, three basic types of decimal-to-binary and binary-to-decimal conversions... None of these sound like the kind of types we are looking for.

Diogenes: The predecessor of types in FORTRAN was called *modes*. They determined how bits that represent numbers in memory should be interpreted. In FORTRAN, function arguments and result could be in either fixed or floating point mode. You indicate that a function is in the fixed point mode by starting its name with the letter 'X'.

Archimedes: I can see how this is similar to different primitive types that we have in modern programming languages, but I don't think you can talk about types if you can't represent richer data types or even just textual data.

Teacher: Where should we look for the origins of richer data types, then?

Pythagoras: Both FORTRAN and Algol 60 only had numerical types. Record types were added first in Algol W in the mid 1960s and then in the final Algol 68 design.

Xenophon: Outside of your scientific programming bubble, the business data processing language FLOW-MATIC supported the concept of records already in 1955. It was inspired by the existing use of paper records, so you could say that the true predecessor of types is the filing cabinet (Figure 5.1) rather than the lambda calculus!

Pythagoras: Well, yes, records in FLOW-MATIC and COBOL were an inspiration for the later work on richer data types in theoretical computer science, but much work was needed for getting the field of business data processing on a firm theoretical footing.[2]

Archimedes: Records in COBOL look very similar to records in some of the contemporary programming languages. What kind of theoretical footing were they missing?

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

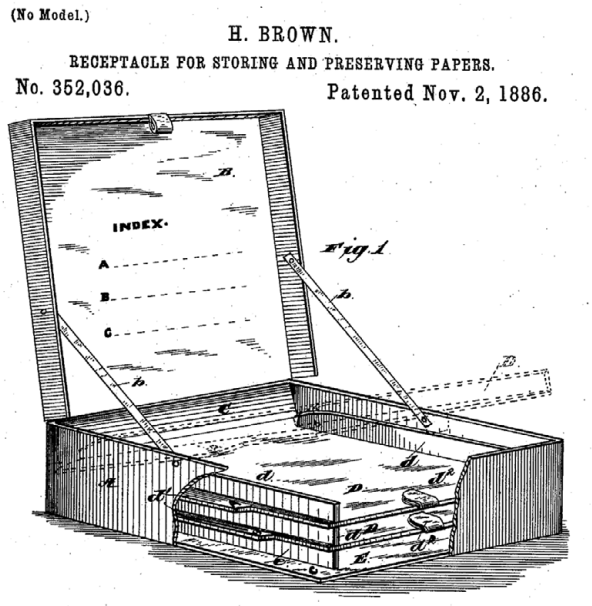


Figure 5.1 Sketch of a 'receptacle for storing and preserving papers', the predecessor of the filing cabinet for storing (paper) records, patented in 1886 by Henry Brown. (Source: U.S. Patent 352036A[3])

Pythagoras: Programming is a mathematical activity, so you need to think about types as mathematical objects, rather than bits in computer memory or paper records in a metal case! This was done by John McCarthy in 1961, when he proposed to model basic numerical data spaces as sets and then construct derived data spaces using set operations such as a product to model records.[4]

Xenophon: I'm sorry, but I'm getting a bit lost. I hoped to learn what's the deal with types, but then *Diogenes* talked about 'modes' and now *Pythagoras* is talking about 'data spaces'. This is very confusing! Are we even talking about the same thing?

Teacher: It seems to me that a type is yet another concept that has brought together work from many different cultures. It unifies modes, which were mainly used to correctly interpret bits in memory, records used for business data processing and mathematical formalisms that model types as sets.

Diogenes: I'm happy to use the term type, but I object to the idea that we should think of types as sets. This may be useful when talking about numbers and records, but how do you model other useful types of values like pointers?[5]

Pythagoras: You really should stop thinking about programming as fiddling with bits. What do you need addresses for? To represent data structures such as lists or trees. Those can be mathematically modelled as recursive structures, which gives you a higher level of abstraction, more suitable for reasoning about programs.

Archimedes: I agree that we should not think of types as sets, but there is a more fundamental reason. In the 1970s, types started to be seen as a protection mechanism, for ensuring that programmers only work with values using the provided procedures. They prevent you from relying on the underlying representation, which you may want to change. This is not something that the 'types as sets' model can capture.[6]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Xenophon: Now we are getting back to software engineering! This view is related to *information hiding*, which was introduced in the 1970s as a better way of structuring programs. It helps you decompose a large system into components that can be developed independently by smaller teams.[7]

Teacher: Has the idea of types became a managerial concept like structured programming?

Archimedes: Types retained their identity as programming language concepts, but ideas on information hiding shaped how they were used. In Clu, an *abstract data type* was a custom type that could be used only through the specified operations. Its representation was hidden and only available within the *cluster* that implemented it.

Pythagoras: Clu illustrates why getting back to the logical origins of types is useful! In early versions, it did not do all type checking at compile-time and had to include some run-time type checks. It turns out that you check everything at compile-time, but this only became clear when John Reynolds created a formal model of types based on a simply typed lambda calculus with user-defined types in 1974.[8]

Teacher: Types seem to be used in a number of different ways. I think this raises an important question of whether it is helpful to use the same term for all of those uses?[9]

Socrates: Well, it does make the discussion somewhat confusing, but I think there is value in having a single notion of a type. It allowed multiple people, following different perspectives, to contribute to a single concept. For example, as we just saw, the logical tradition provided a checking mechanism for the engineering notion of an abstract data type. This may not have happened if we used different terms!

Diogenes: I can see the utility in joining some of those ideas. Invoking an invalid operation may well be the programming equivalent of a logical paradox. But information hiding is a social control mechanism, rather than a useful programming tool.

Pythagoras: You wouldn't expect me to defend any idea advocated by *Xenophon*, but information hiding became crucial for the use of types in the automated theorem prover LCF. The system used abstract data types to represent provable theorems. You could only construct proofs from pre-defined axioms, by using inference rules modelled as built-in functions. The types guaranteed that you constructed only valid proofs.

Archimedes: Now we are getting to an interesting topic. The metalanguage used for composing theorems in LCF later became a stand-alone practically useful programming language ML. This only happened once ML was extended with features like records and module systems, but the basic idea of the language originated in the work on LCF.

Pythagoras: Most importantly, all those extensions to the ML type system were defined in a formal mathematical specification. This made it possible to prove that well-typed program cannot go wrong.[10] The same theoretical method was later used for checking how programs communicate over a network, what resources they access, how they work with memory, as well as to control aliasing.[11]

Socrates: But now you are talking about program behaviour, not about values used in programs. I wonder if this is stretching the concept of a type?[12]

Pythagoras: That depends on how you think about types. You are right that these examples do not make sense if you think of types as sets. Today, a more common interpretation is to see types as a lightweight formal method.[13]

Archimedes: Why do we need to be so sophisticated about this? Types are a programming assistance tool, just like test runners, linters and many other tools.

Pythagoras: But that is really downplaying what types can do. If you use a language with dependent types, you can, for example, define a type `Vect` n α to represent a vector of length n containing values of type α. An append function that takes `Vect` n α and `Vect` m α then returns `Vect` $(n + m)$ α. The type is a partial specification of the function behaviour that is automatically checked for you!

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Xenophon: Wait a minute! Specification should be separate from the program and written by someone who is a domain expert, not a programmer. A specification should facilitate a conversation about the problem, but your fancy types only make it harder.

Archimedes: Actually, types facilitate these conversations.[14] In the process of writing a type definition, you have to precisely understand the domain in order to devise the correct representation. This involves talking to a customer and other team members, or even showing the type definition to the customer.

Socrates: Interesting. We are again getting back to program proofs and social processes. It seems that types enable some of the healthy social processes that other kinds of formal proofs about programs failed to support.

Teacher: Should we then really view types as lightweight specifications?

Socrates: If you see types this way, they become just obstacles that get in the way of your creative thinking. But I actually think there is a useful alternative. In systems with dependent types, you can write a type of a function and, when you start implementing it, the system can automatically suggest parts of the implementation based on the type information. With this way of looking at them, types can contribute to the vision of man–computer symbiosis!

Pythagoras: You need to distinguish between the language and the editor. The editor may be interactive, but the types in the finished program are still specifications. This is how they give you correctness guarantees about your programs!

Archimedes: But they do not have to do that in order to be useful tools.

Pythagoras: What a crazy idea! Can you please explain what you mean?

Archimedes: Happy to. Look at the TypeScript programming language. Its type system does not give you absolute guarantees. The language extends JavaScript and adds type checking that has a number of carefully considered aspects where the designers choose simplicity and compatibility with JavaScript over safety.[15]

Pythagoras: But this is ridiculous! How is that supposed to be useful?

Archimedes: It works very well in practice. Types in TypeScript support the structuring of large programs, they serve as a useful documentation, they help you write code via editor auto-complete and they help you avoid many common programming errors.

Teacher: After we got over the diverse origins of types, it almost looks like we'll find an agreement, but now we see multiple diverging perspectives again. Has this recurring disagreement been a curse or a blessing?

Pythagoras: The lack of a clear definition is an obstacle to the progress in programming languages, because people make confused claims. It is unfortunate that the engineering notion of data type, modelled later as sets, got mixed up with the logical notion of a type, modelled after the lambda calculus.[16]

Diogenes: You might not be able to directly compare and synthesise different theoretical contributions concerning types, but that is not what programming is about! In the end, I care about what programs I can write using types. Many different kinds of types have been used to build developer tooling, so just judge by looking at the end results!

Socrates: I think the fact that those overlapping conceptions of a type share the same name has been useful. It helps different communities exchange ideas. We saw how Clu and TypeScript both adapted logical notions for more engineering-oriented purposes and I'm sure the theory has equally benefited from engineering innovations. I doubt the ML language would have ever been born if we kept all those notions separate!

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

5.1 Are Types Invented or Discovered?

In February 2016, some 350 functional programmers came to Kraków to join the programming conference Lambda Days. The opening keynote titled 'Propositions as Types'[17] was presented by a computer scientist, Philip Wadler, who is well known for his many contributions to the theory of programming languages.

The topic of Wadler's keynote can be accurately related in two ways. A dull technical summary is that the talk described a correspondence between two formal systems, a natural deduction system for intuitionistic logic and the simply typed lambda calculus. A more captivating account is that the talk showed a remarkable correspondence between logic and functional programming, suggesting that functional languages are not mere human inventions, but are instead rooted in deeper truths about the universe.

Seen from the perspective of the initiated, the talk documents how the idea of types, which originated in Bertrand Russell's work in formal logic, made it into the first functional programming language, the lambda calculus, before ending in programming languages that functional programmers use today, including Haskell, OCaml and F#.

A historian or a philosopher of mathematics attending the talk would raise many objections.[18] First of all, the two systems related in the talk were both developed in response to the same problems in work on the foundations of mathematics and they are carefully constructed to match, so their correspondence is not as surprising as it may seem. A listener who is not acquainted with the assumptions of the mathematical culture may be surprised by another aspect of the talk. Wadler talked about the relationship between two formal systems that originated in the field of logic before the first computers existed. Yet, he clearly suggests that one side of the relationship is about programming languages, for example, by saying that the 'lambda calculus is the world's oldest programming language, being defined in the 1930s, and it is the coolest because it was defined decades ago before the first stored program computers were built'. Such a claim only makes sense if we see programming languages as abstract mathematical ideas that exist independently of computers, but not if we see programming languages as the result of human engineering efforts.

Wadler's keynote makes for a good opening of a chapter on types. It illustrates many of the positions typically accepted by the mathematical culture of programming, which had a strong influence on the idea of types in programming languages. The early proponents of the mathematical culture, which we encountered in Chapter 2, included mainly academic computer scientists. With the increasing popularity of functional programming, many of those views became common among practitioners, such as those attending the Lambda Days conference. As we will see, this combined the mathematical notions with more engineering-oriented perspectives and methods.

The origin story of types in programming languages, as told by the mathematical culture, starts with the history of formal logic in the early twentieth century. The idea of a 'type' that appeared in the early work on logic has, indeed, influenced types in programming languages. However, this influence is more indirect and takes place some

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

15 years after types first appeared in programming languages, largely independently of work on formal logic. With this caveat, I will start by reviewing the history of types in early twentieth-century logic, before returning to programming in the late 1950s.

5.2 Resolving Logical Paradoxes with Types

In 1902, Bertrand Russell wrote a letter to Gottlob Frege to let him know that he had discovered a paradox in his formal system published in 1879. Using a modern notation, the system made it possible to define predicates such as $p(x)$ to denote that a property p holds for an argument supplied in place of the variable x. Negation of a predicate, written as $\neg p(x)$, would denote that the property does not hold.

Russell realised that Frege's system made it possible to define a predicate $p(x)$ such that it is true if and only if $\neg x(x)$. This is a clever trick, because it uses the variable x for both the predicate and the argument that it should be talking about. The definition is not just clever, but it is also paradoxical. In Frege's formal system, it should be the case that either $p(p)$ or $\neg p(p)$ holds. Either the predicate or its negation should be true when applied to itself. To determine which is the case, let's assume that $p(p)$. The definition of $p(x)$ is $\neg x(x)$ and so if we substitute p for x in this definition, the result is $\neg p(p)$. Conversely, if we assume that $\neg p(p)$ and substitute p for x in the definition of p, we get $\neg\neg p(p)$. The double negation can be eliminated and this is equivalent to $p(p)$! Russell was thus able to show that both $p(p)$ and $\neg p(p)$ hold in Frege's system, which is a contradiction.

Russell's solution evolved gradually over several years.[19] 'Appendix B' of his 1903 book *The Principles of Mathematics*[20] (not to be confused with the later and better known *Principia Mathematica*) defines the so-called doctrine of types. Here, Russell introduces a hierarchy of types for classifying objects and classes (these can be thought of as sets containing individuals or other sets). In the doctrine of types, the lowest type represents *individuals*, the next type *classes of individuals*, the next *classes of classes of individuals* and so on. Russell does not go as far as imposing types on *propositions*, which he considers 'harsh and highly artificial'. This means that types do not apply to predicates like p, which was the source of inconsistency in Frege's system. Russell's doctrine of types from 'Appendix B' thus does not eliminate all contradictions from the system.

The flaw was corrected in a 1908 paper 'Mathematical Logic as based on the Theory of Types'.[21] Here, Russell describes the 'vicious-circle principle' and requires that 'whatever contains an apparent variable must not be a possible value of that variable'. The type of proposition that contains a variable must be higher than the type of values assigned to the variable. This rules out the contradiction caused by $p(p)$ discussed earlier. In the definition $p(x)$, the proposition p contains a variable x. If x is a variable of a certain type, the type of p must be higher and so p cannot be used as a value of x. Writing $p(p)$ is illegal.

The next major step in bringing types to programming, at least in the eyes of mathematically minded programmers, was made by Alonzo Church in 1940. Like

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Russell, Church was interested in the foundations of logic. Around 1928, he began working on a formal system that was based on the concept of functions, rather than sets (or classes). The system was published in 1932 and later became known as the lambda calculus. In 1940, still before any modern digital computers existed, Church published a paper 'A formulation of the simple theory of types'.[22] In the paper, Church describes a variant of Russell's theory of types which 'incorporates certain features of the calculus of λ-conversion' or the lambda calculus as we now know it. Church's hierarchy of types for the lambda calculus defines ι as the type of individuals, o as the type of propositions and $\alpha\beta$ as the type of functions with a parameter (independent variable) of type α and result (dependent variable) of type β. In a modern notation, function would be written as $\alpha \to \beta$ and the system makes it possible to define types for functions that take other functions as arguments or return them as results such as $(\iota \to \iota) \to (\iota \to \iota)$.

The purpose of types, as introduced by Russell, was to eliminate paradoxes from a formal logical system. To serve this role, we do not need to know much about what types are. Russell defines a type as the 'range of significance of a propositional function, that is, as the collection of arguments for which the said function has values'.[23] However, he also later notes that it is unnecessary to know what objects belong to the lowest type and that, in practice, only the relative types of variables are relevant. Church makes a similar remark when he 'purposely refrains from making more definite the nature of the types [of propositions and individuals]'.[24] This is in stark contrast with types as they first appear in programming languages, where there are no relative types and the nature of the primitive types is all that matters.[25]

5.3 Evaluating Arithmetic Expressions in Two Modes

As in formal logic, types in contemporary programming languages are often used for specifying what the range of possible values of a variable is. If we have a variable `COUNT` of type integer, this indicates that its values may be numbers such as $1, 2, 3, \ldots$, but not a floating point number like 3.14 or a string '`Bertrand`'. As in logic, this may be used for restricting what programs are valid. In statically typed programming languages, types are checked before a program is run. If you write a program that attempts to multiply '`Bertrand`' by 3.14, the program will be ruled out as invalid. The compiler will know that multiplication can only be applied to arguments of numerical types. The value '`Bertrand`' is not of a numerical type and so the checking procedure will report an error.

Unlike in logic, the nature of primitive types is central in programming. A compiler typically uses types to check that primitive operations are called with values of correct types as arguments. It may also use the fact that a variable is of a certain type to allocate an appropriate number of bytes of memory for storing it. However, different modern programming languages use types very differently. None of the above is universally true and it should serve merely to give a good basic intuition.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The idea of types is so commonplace in contemporary programming that it is perhaps surprising how long it took before programmers started consistently using the term 'type'. The IBM 704 FORTRAN manual,[26] published in 1956 uses the term 'type' in its everyday English sense, often interchangeably with words such as 'class'. The report talks about '32 types of statements', 'three basic types of decimal-to-binary or binary-to-decimal conversions', but also 'two types of variables':

> Two types of variable are also permissible: fixed point (restricted to integral values) and floating point. Fixed point variables are distinguished by the fact that their first character is I, J, K, L, M, or N.

In a more formal description of functions, expressions and arithmetic formulas (chapter 3 of the manual), the authors use the term 'modes' and distinguish the fixed and floating point modes. For functions, each of the arguments as well as the result can be in either of the two modes. This yields a number of different mode configurations and a separate function name must be given for each of the supported configurations. The IBM 704 version of FORTRAN also required a particular naming of functions. All function names must end with `F` and, if the mode of the function is fixed, the name must also start with `X`. For example, a function `XINTF` truncates an argument in the floating point mode and produces a fixed point number.

Arithmetic expressions are also in one of the two modes and this is used to define what expressions are valid. For example, if `E` and `F` are expressions of the same mode, then `E+F`, `E-F`, `E*F` and `E/F` are also expressions of the given mode. However, applying numerical operators to expressions in mismatching modes is not allowed. In retrospect, the similarity with types in formal logic is not too hard to see. Just like Russell used types to make writing of the self-referential proposition $p(p)$ invalid, FORTRAN uses modes to make writing `E+F` invalid for expressions representing different kinds of numerical values. Yet, it took almost a decade before someone made this connection. This should not be a surprise. As we saw in Chapter 2, at the time when FORTRAN was being created, the very idea of programming languages as independent entities, more so mathematical ones, was only just appearing.

It seems possible that FORTRAN adopted the term 'mode' because the mode of an expression was used to determine what kind of machine code the compiler produced for the expression. The machine code for `X+Y` was different in the fixed and the floating point mode. In other words, the mode of an expression determines one of two possible modes of compilation. If this is the case, the terminology used in FORTRAN was shaped by the hacker culture of programming in that the name of the concept was derived from a technical implementation trick that was used to compile FORTRAN code. The creator of FORTRAN, John Backus, also made the link with the hacker culture explicit when he later recalled that 'programming in the 1950s was a black art'. Interestingly, the term 'mode' outlived its hacker origins. We will see that the term was still in use in the late 1960s and the 1970s when it was adopted by the mathematical culture, possibly before it fully embraced the term 'type' with its links to the early twentieth-century formal logic.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The first modern usage of the term 'type' is in the 1958 preliminary report on the Algol programming language,[27] which was then still known as International Algorithmic Language (IAL). The term was introduced during a meeting between the American ACM group and the European group working on IAL in Zurich at the end of May 1958. Tracing the history of types, Simone Martini suggested that 'the technical term appears to be just a semantical shift from the generic one; in particular, there is no clue that in this process the technical term "type" from mathematical logic had any role'.[28] Many of those present at the meeting were notable proponents of the mathematical culture of programming, so it is possible that they were familiar with the use of types in logic, but there is no acknowledgement of an influence. Even if the term was, in fact, imported from logic, the authors of the report did not find this significant enough to discuss the relationship explicitly.

Types or modes in FORTRAN and Algol 60 were rather basic compared to what programmers know as types today. Algol 60 had three primitive types. In addition to integers and real numbers, it also used a Boolean type to represent logical values. Both FORTRAN and Algol supported arrays, but those were not treated as types. An array was represented as a 'subscripted variable' where the subscript determined a position (index) of a value in the array. The designers of Algol considered other types including complex numbers, vectors and matrices but those did 'not seem to have stirred much attention during the discussions'.[29] One feature that was notably lacking from Algol 60 was a mechanism for representing structured data such as records consisting of multiple fields. It is telling that the mathematically minded designers of Algol were thinking about vectors and matrices, but not about a mechanism for representing structured data that was becoming essential in business data processing at the very same time.

5.4 Structuring Scientific and Business Data

The late 1950s was when the idea of a programming language as a stand-alone object, not linked to a specific computer emerged. This brought together the managerial need for portability, mathematical tools for thinking about languages and hacker expertise in implementing programs that translated human-readable code to machine code. A number of programming languages came out of this effort. FORTRAN was a successor of various automatic programming systems developed previously. It was a product of the 1950s 'black art' of programming, but it was also shaped by the engineering focus on supporting efficient scientific applications. Algol 60 kept the same focus on scientific applications as FORTRAN, but approached the issue from a mathematical perspective.

At the same time, computers were also becoming increasingly used for commercial applications. The Eckert–Mauchly Computer Corporation, created by the ENIAC inventors, ran into financial difficulties and was sold to Remington Rand in 1950, where it continued as a division and started selling the UNIVAC I computer in 1951. IBM soon joined the electronic computer market with the IBM 701, which became available in 1952. Programmers at both companies faced the problem of creating business data processing systems. The implementation of those relied on the same amount

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

of hacker ingenuity as the implementation of systems for scientific applications, but the structure of the problems they posed was shaped by commercial rather than scientific needs. In particular, processing business records required ways of working with structured data that consisted of both textual and numerical information stored in files. Two early programming systems that tackled this challenge were FLOW-MATIC and Commercial Translator (COMTRAN). Both systems brought together ingenuity of the hacker culture with the problem focus of the managerial culture. FLOW-MATIC was developed by Grace Hopper at Remington Rand and it was released for UNIVAC to customers in 1957. COMTRAN was developed at IBM, under the leadership of Bob Bemer and was publicly presented in 1957, although the implementation only became available later.

FLOW-MATIC and COMTRAN may seem like obscure historical artifacts, but they became sources of ideas for a programming language that appeared in 1959 and is still in use today, the Common Business-Oriented Language (COBOL).[30] The language was produced by the Committee on Data Systems Languages (CODASYL) that was established with the goal of establishing a common language for data processing. When CODASYL started looking into the idea, it first established a Short-Range Committee that would conduct a study of existing business compilers and provide materials for a task group that would design the language. In a perhaps unsurprising turn of events, the Short-Range Committee was later asked to define an interim language that would be used before the final design was agreed. The interim language was published as COBOL 60. The final design never materialised, but COBOL received a widespread adoption in industry.

FLOW-MATIC and COMTRAN influenced COBOL and future commercial programming languages in a number of ways. One notable design influence of FLOW-MATIC was the use of readable English words rather than symbolic code. However, the most interesting feature for our discussion about types is the fact that both FLOW-MATIC and COMTRAN separated 'data description' from 'program description'. The data description specified the way data is stored in terms of files, records and fields. Figure 5.2 shows a data description from the COMTRAN manual for a sample payroll system. Records are hierarchical and nesting is indicated by the 'level' column on the data description card. (Nesting was supported in COMTRAN, but not in FLOW-MATIC.) Type `RECORD` defines a new record whereas the `COPY` type indicates that the structure of the value is the same as that of another record. For individual fields, the `DESCRIPTION` column specifies the format of the data. For example, `9(5)V99` specifies a numerical value formed by up to five numerical characters, followed by a decimal point and two decimal digits. The COBOL language, which was developed in 1959, adopted both the general concept of records, as well as the formatting specifier, also known as the 'PICTURE clause'.

A contemporary programmer can easily see COMTRAN records as precursors of the record data type that exists in many recent programming languages, but the way of thinking about data descriptions in business-oriented programming languages was very different from thinking about types (or modes) in scientific programming. In COBOL, data description was one of three 'divisions' that constituted a COBOL pro-

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

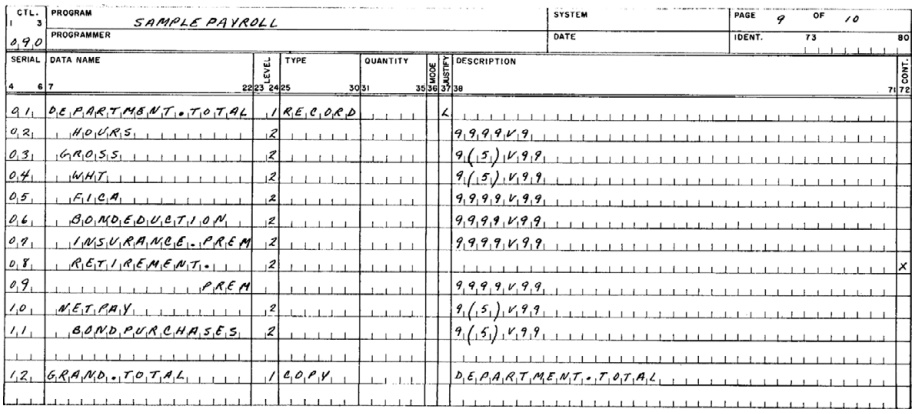


Figure 5.2 A sample data description card from the COMTRAN system, which defines a structured record representing total payroll spending and spending per department. (Source: IBM Commercial Translator manual.[31] Reprint Courtesy of IBM Corporation ©.)

gram, alongside 'procedure division' used for specifying program code and 'environment division' which specifies machine-specific information such as input and output devices or machine configuration. As the following extract from a 'Detailed description of COBOL'[32] shows, the COBOL designers thought of data division as the part of the program that specifies the physical structure of data on a tape, even though in a somewhat higher-level way:

> The `DATA DIVISION` is concerned with information from files, data which are developed during the program and placed into working storage, and constants defined by the user. … The basic concept used in the data organisation is that of 'logical record' which can be defined to be any consecutive set of information. … It is important to note that several logical records may occupy a block on a tape (i.e. physical record), or a logical record may extend across several physical records.

Making the connection between data definitions in business programming languages and types in Algol required bridging the difference in thinking about data and types. Moreover, the exchange of ideas was hindered by the social circumstances and prejudices. The discussion about types and Algol happened among the mathematical and engineering cultures, often in conferences and publications supported by the Association for Computing Machinery (ACM). This did not involve programmers working on business data processing who read the *Datamation* magazine, met through the volunteer-run SHARE user group and collaborated through CODASYL. The division between the mathematical and the business culture was not complete though. Since 1951, the Western and Eastern annual Joint Computer Conferences (WJCC and EJCC) provided a venue where communities mixed and in 1960, the conference was expected to attract some 3,000 attendees (Figure 5.3).

The mathematically minded members of the ACM were aware of the work done in the data processing industry, but felt that their publications 'failed to report fundamental research in the data processing field'.[33] The ACM members were not interested in

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

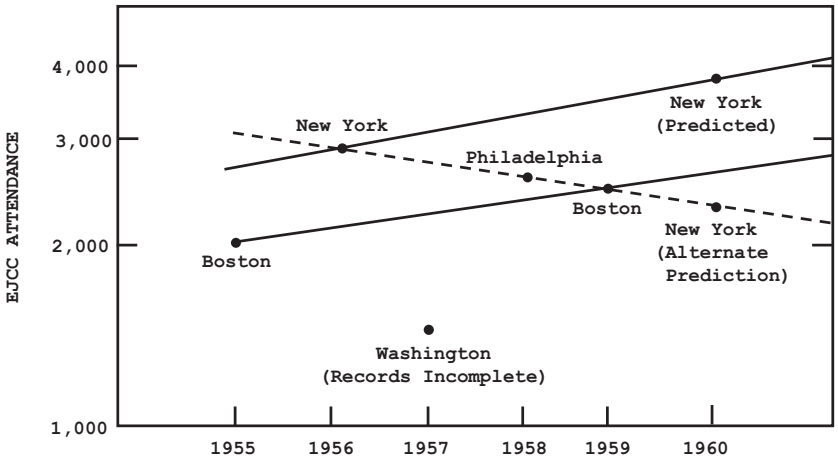


Figure 5.3 Predicted number of attendees at the EJCC based on growth in attendance in Boston (full line) or based on overall eastern trend (dashed line). The General Chairman noted that, if the optimistic prediction is correct, there is no hotel in New York that can hold the EJCC.[34] (Source: Drawing by the author, labels as in the original)

'how someone else solved his payroll problem' because they believed that 'the solution is almost sure to be too specifically oriented to the company for which it was solved and [lacks] general applicability'.[35] Yet, learning how COBOL programmers solved their payroll problem is exactly what was needed for the next step in the evolution of types.

5.5 Towards a Universal Programming Language

The limitations of scientific programming languages for working with other kinds of data structures were soon widely recognised and many mathematically minded computer scientists attempted to design a way of representing richer data structures modelled after established mathematical structures.[36] One of those mathematically minded computer scientists was John McCarthy whom we encountered in the preceding chapters as the author of Lisp and a proponent of the mathematical culture. At the culture-bridging Western Joint Computer Conference, McCarthy presented a paper 'A Basis for a Mathematical Theory of Computation',[37] which was the first step of his effort to develop a science of computation. According to McCarthy, his paper contained contributions to a number of goals, the first of which was the development of a 'universal programming language', which would support both scientific and business-oriented programming.

McCarthy believed that 'Algol is on the right track but mainly lacks the ability to describe different kinds of data', which has become a key feature of business data processing languages. In McCarthy's view, 'languages such as FLOW-MATIC and

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

COBOL have made a start in this direction, even though this start is hampered by concessions to what the authors presume are the prejudices of business men'. Interestingly, the concessions do not have much to do with the way data descriptions are structured. McCarthy mainly criticises the use of English-like notation in COBOL, which he finds unsuitable for formal treatment. It is worth noting that McCarthy was exercising his own prejudices here. The syntax of COBOL was defined in a formal meta-language that was not unlike the Backus–Naur form used in the definition of Algol 60.[38] The semantics of COBOL was described, just like the semantics of Algol, using plain English, albeit with perhaps less technical rigour.

To McCarthy and other proponents of the mathematical culture, introducing the ability to describe different kinds of data first required the development of a mathematical theory of data types. The specific details of a concrete programming language would only come later. In an acknowledgement that remains typical for work of the mathematical culture, McCarthy acknowledges that 'the formalism for describing computations in this paper is not presented as a candidate for a universal programming language because it lacks a number of features, mainly syntactic, which are necessary for convenient use'.

Much of McCarthy's paper is dedicated to various ways of defining functions in terms of base functions, conditional expressions and recursive definitions. Inspired by Church's lambda calculus, he also discusses functions taking functions as arguments and adopts the λ notation used by Church. In a later part of the paper dedicated to data, McCarthy models the values that functions can accept as arguments and return as results using mathematical sets. To make the terminology in this chapter even more confusing, McCarthy refers to such sets of values as 'data spaces'. He does not adopt the term 'type' even though he cites lambda calculus and the term 'type' was already in use by the definition of Algol. This is unexpected only in hindsight. Types in lambda calculus were used to avoid logical paradoxes, whereas types in Algol described the kind of data accepted by a subroutine. In both cases, types could rule out certain invalid expressions, but this link was not necessarily obvious at the time.

The specific ideas presented by McCarthy include a number of ways of defining new data spaces in terms of existing data spaces. The constructions use the usual mathematical operations for working with sets. A Cartesian product of data spaces A, B creates a data space $A \times B$ consisting of pairs (a, b) such that $a \in A, b \in B$. Although McCarthy does not say so explicitly, the construction roughly corresponds to records in COBOL, which also combine multiple values of other primitive types. McCarthy also defines a union of non-intersecting sets $A \oplus B$ and a power set A^B which models function values and remarks on the importance of recursive definitions. The only example of a recursive data space given by McCarthy is the data space S defined as $S = A \oplus (S \times S)$. This is the model of S-expressions, a notation used in his new programming language Lisp. The non-intersecting union $\oplus$ operator represents a choice, so an S-expression is either an atom $a \in A$ or a pair (x, y) composed of two S-expressions $x, y \in S$.

The notion of a 'data space' as used by McCarthy, type as used by Russell and type as used in Algol come closer together in a collective monograph *Structured Programming*.[39] The book was published in 1972 and includes three chapters. In the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

```
type local car = (make: manufacturer; regnumber: carnumber;
                                owner: person; first registration: date);

type visitor car = (make: manufacturer; regnumber: carnumber;
                                              origin: country);

type car =  (local: localcar,
                foreign: foreign car).

type car = (make: manufacturer;
            regnumber: carnumber;
            (local: (owner: person;
            first registration: date),
            foreign: (origin: country))
            ).
```

Figure 5.4 Two ways of representing data about cars using records (written using ';') and discriminated unions (written using ','). Above, a car is a either local (including its owner and a first registration date) or foreign (including country of origin). Below, the type of a car is a record containing the shared attributes and a nested discriminated union containing additional information about either a local or a foreign car. (Source: Dahl et al. (1972))

first chapter, Edsger Dijkstra writes about his notion of structured programming. In the second chapter, C. A. R. Hoare refines and extends the line of work started by McCarthy. He adopts the term 'type' and explicitly discusses the similarities between the notion of a type used by programmers and logicians. He cites the notion of a type in Algol, Russell's use of types to avoid paradoxes, as well as the use of the notion in informal mathematical writing. Finally, the third chapter by Ole-Johan Dahl and C. A. R. Hoare discuss program structuring in SIMULA, an object-oriented programming language discussed in Chapter 6. This is presented as a synthesis of 'the design of data and the design of programs', which are the topics of the first two chapters of the monograph.

Hoare presents his ideas on data structuring using a hypothetical unimplemented programming language with a concrete syntax for type definitions. The syntax includes records, fixed-size arrays, potentially infinite sequences, but also discriminated unions that represent a choice of alternatives, powersets to model function values and recursive type definitions. For each of those, Hoare provides a set-theoretical model and notes on a possible implementation. The set-theoretical models of records and discriminated unions are like the $\times$ and $\oplus$ operations introduced by McCarthy. Hoare makes it clear that he is talking not just about mathematical models, but about real computer programs. Consequently, his examples that include a range of realistic problems, for example, use a data structure to store information about cars shown in Figure 5.4.

Meanwhile, the designers of Algol 60, united in the international IFIP Working Group 2.1, embarked on the task of designing a successor version of Algol. Improving the limited support for data structuring in Algol 60 and including richer data types was one of the goals for a successor language. As I mentioned already, the design process was far from smooth and the process involved 'discord, resignations, unreadable doc-

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

uments, a minority report, and all manner of politicking'.[40] We do not need to get into the particulars of this contentious history, but it is worth noting that the dissenters who felt that the proposed language fails to offer a suitable tool for structuring programs included C. A. R. Hoare and Edsger Dijkstra, two of the authors of the later *Structured Programming* monograph.

The definition for a successor language was eventually produced and the language became known as Algol 68. A common criticism of Algol 68 was that the definition was unreadable to an uninitiated reader. This also plagued the terminology of data structuring. In a section of the Algol 68 report[41] that compares Algol 68 with Algol 60, the authors note that:

> Whereas ALGOL 60 has values of the types integer, real and Boolean, ALGOL 68 features an infinity of 'modes', i.e., generalisations of the concept 'type'.

The authors of the Algol 68 report decided to keep the term 'type' for primitive types, but they re-introduced the term 'mode' when referring to structures that are constructed by 'mode declarers' which compose other modes or primitive types. Rather than adopting the business term 'record', the report talks about structured values. The notion of a mode in Algol 68 still keeps some of the original implementation-centric hacker aspects. For example, a mode can specify whether a value will be stored on the global heap or in a local stack. In most other respects, however, Algol 68 is a product of the mathematical culture. This is apparent in its attempt to achieve orthogonal design that minimises the number of primitive concepts, the formal writing style of the report as well as idiosyncratic typography and new terminology.

The re-introduction of the term mode for talking about composed types in the Algol 68 report was only a brief digression in the history of types. Another of the dissenters from the design committee, Niklaus Wirth, went on to develop the programming language Pascal,[42] which was arguably the most successful attempt to resurrect the spirit of Algol 60 and develop it into a universal programming language. Pascal not only switched the terminology back and referred to custom data structures as types, but it also adopted forms of structuring data that were familiar to COBOL programmers including records (representing fixed number of components of possibly different types) and files (representing a sequence of components). Records in Pascal can also have multiple variants, which is arguably a more business-friendly approach for implementing McCarthy's union.

It is worth noting that the idea of introducing new user-defined types in a program was not always seen as an inherent aspect of programming. Some authors used the term 'extensible languages' to refer to languages that can be extended through custom type (or mode) declarations in order to create new languages, more suitable for solving problems in a certain domain. An example can be found in the review 'Definition mechanism in extensible programming languages'.[43] It recognises 'mode constructors' as one of the mechanisms of an extensible language. Interestingly, the review puts this extension mechanism alongside other extension mechanisms such as the ability to define new language syntax using macros.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Despite the objective to produce a language that would be suitable for both scientific computing and business data processing, most of the work on structured data types was done at the intersection of engineering and mathematical cultures. Throughout the process, there was an interesting borderline between more mathematical and more engineering designs. For example, Niklaus Wirth was commissioned by the Algol 68 design committee to produce a report representing the 'views of the pragmatists'. The pragmatic engineering design included a type for pointers, which are much closer to how a machine operates, but break the otherwise elegant mathematical theory based on sets. By contrast, the mathematically minded idealists favoured recursive definitions, because they have an elegant set theoretical model.

The mathematics used for talking about types in the late 1960s was mostly set theory. This provided a model for records, unions and recursive data types. References to types as known from mathematical logic started to appear, but those were not treated as types in the sense used in programming language. In particular, Hoare[44] only referenced Russell's notion of types for avoiding logical paradoxes, while McCarthy[45] referenced Church's untyped lambda calculus. The confusing and ever-changing terminology involving types, modes and data spaces only confirms that the notion of a type was still in flux. The link between programming languages and lambda calculus was slowly gaining importance throughout the 1960s, but the crucial connection was established in work on programming language semantics.

5.6 The Next 700 Programming Languages

The mathematical development in programming languages, through which types in programming and types in logic became one and the same thing, features one of the most bizarre characters of the history of logic. Mervyn Pragnell was not an academic and never published any papers, yet he appears in the stories told by many British pioneers of theoretical programming language research. Pragnell was the organiser of an underground logic reading group where many of them met. His group was literally underground. It met in the basement of Birkbeck College, without the knowledge of the college authorities. Pragnell knew a lab assistant at the College who would let the attendees in. Pragnell recruited members of the reading group in various ways, such as by lurking around the Foyles bookshop, looking for people buying books on mathematical logic and inviting them to join the group. Several members of the group 'recall a distinctly theological atmosphere, taking turns to read aloud pages of hefty books on logic and mime the formulae!'[46]

Although there are no written records of the group meetings, it is likely that the lambda calculus was one of the topics discussed in the reading group. The topic may have come to the group through Christopher Strachey. As documented by Troy Astarte,[47] Christopher Strachey had been introduced to the lambda calculus by Roger Penrose around 1958. Strachey ran a consulting business and, between 1960 and 1964, he employed Peter Landin who was another member of the reading group interested in the lambda calculus. Yet another member of the group and early proponent of

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

functional programming, Rod Burstall recalled that 'Landin taught [him] about lambda calculus and functional programming in the pub round the corner from the College, The Duke of Marlborough'.[48]

At Strachey's consulting business, Landin worked on a compiler for the Feranti Orion computer. The machine architecture made it awkward to produce optimal code directly. Landin suggested using a language inspired by the lambda calculus as an intermediate program representation.[49] Based on this idea, Landin produced a series of papers that describe the approach in a more basic academic manner. In the first paper of the series, 'The Mechanical Evaluation of Expressions',[50] Landin introduces a lambda calculus-inspired formalism of 'applicative expressions'. He describes how 'some forms of expressions used in current programming languages', including lists, conditional expressions and recursive definitions, can be modelled using applicative expressions. He then describes a mechanism for evaluating such applicative expressions. The paper reflects Landin's mathematical background and the evaluation mechanism is described using a formal mathematical language. In modern terms, Landin presents an abstract machine with a state modelled using mathematical structures shown in Figure 5.5. The evaluation of the machine is modelled using rules that turn a state, written as (S, E, C, D), into a new state (S', E', C', D').

The next step in Landin's work came when he was invited to a conference on Formal Language Description Languages (FLDL) in Vienna in September 1964. In his presentation,[51] Landin extended his method and used it to define the semantics of Algol 60. The details of this work were later described in a two-part paper in the *Communications of the ACM*.[52] The papers follow the same approach as the earlier work. Landin describes how to translate Algol 60 programs to his 'applicative expressions', extended with features necessary to support imperative programming like assignments and jumps; he then describes a model of evaluation for those expressions that he now

A *state* consists of a *stack*, which is a list, each of whose items is an intermediate result of evaluation, awaiting subsequent use;

and an *environment*, which is a list-strucutre made up of name/value pairs;

and a *control*, which is a list, each of whose items is either an AE awaiting evaluation, or a special object designated by '*ap*,' distinct from all AEs;

and a *dump*, which is a complete state, i.e. comprising four components as listed here.

We denote a state thus:

$$(S,\ E,\ C,\ D).$$

Figure 5.5 The state of an abstract machine for evaluating applicative expressions (AEs). The abstract machine models the evaluation of a lambda calculus-inspired intermediate representation of programs. (Source: Landin (1964))

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

calls the 'SECD machine', after the symbols that appear in the definition of the machine state.

Landin's use of the lambda calculus in the series of papers published while working with Strachey is certainly a step towards bridging the gap between formal logic and programming. Algol 60 programs are translated to lambda calculus-like language, but the two are still to some extent separate kinds of entities. The papers have also not yet elaborated the connection between the logical types and types in programming. After attending the FLDL conference, Landin moved from Strachey's consulting business to work on experimental programming languages at the UNIVAC division of the Sperry Rand corporation. The move from work on programming language implementation to work on programming language design provided an incentive for developing a different way of using the lambda calculus. In the aforementioned papers, lambda calculus serves as an intermediate language to define the semantics of another programming language. After moving to UNIVAC, Landin published a paper called 'The next 700 programming languages'[53] which uses lambda calculus not just for the semantics, but also as a source of language design ideas.

The ISWIM ('If you See What I Mean') language presented by Landin in the paper attempts to provide a general system for compositional programming, an approach that became the cornerstone of the functional family of programming languages. Landin explains his approach in the introduction:

> ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of 'primitives'. So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives.

The framing also explains the number 700 in the paper title. The paper starts with a quotation reporting that there are over 1,700 programming languages used in over 700 application areas. By supplying appropriate sets of primitives, ISWIM can be used for the next 700 desired application areas. To use contemporary terminology, ISWIM consists of a purely functional sub-system with impure extensions including assignment and a form of jump (using so-called program points). Programs are written as expressions with auxiliary definitions given using `where` expressions. In a number of places, the paper makes explicit references to mathematical logic. Landin is not just borrowing the λ notation as McCarthy did for Lisp, but he is marrying programming language theory with work on lambda calculus. An example is his discussion of the β reduction rule from the lambda calculus, which ensures that an expression L `where` $x = M$ is equivalent to a program obtained by substituting the expression M for a variable x in the expression L. The rule makes it possible to eliminate `where` from programs, which Landin connects to the Church–Rosser theorem of formal logic.

ISWIM itself was intended as an abstract description of a family of programming languages and it remained unimplemented. However, at the end of the 1960s, two early functional programming languages turned the ideas from ISWIM into reality. They were the PAL language, created at MIT in 1968, and the GEDANKEN language, created at Argonne National Laboratory in 1969. Both languages followed the same

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

structure as ISWIM with a core functional sub-language based on function application. However, both PAL and GEDANKEN were missing features typically associated with types in functional languages today, including static checking of types, type inference and data structures such as records and unions.

Landin was directly involved in the implementation of the PAL language, having spent a year from 1966–67 at MIT. The initial Lisp implementation was written by Landin together with James H. Morris who was a PhD student at MIT at the time. Morris was inspired by Landin's 'incisive analyses of programming languages'.[54] His work is particularly interesting for our story because he extended the correspondence between programming languages and the lambda calculus from just the applicative structure used in ISWIM and PAL to the structure of types.

5.7 Types Are Not Sets

James H. Morris is one of the computer scientists who illustrate the fact that individuals can bridge and contribute to multiple cultures of programming. When Morris came to MIT for his PhD, he joined the Project MAC. This was the ARPA-funded home of the 1960s hacker culture at MIT that I talked about in Chapter 3. Morris worked on the interactive programming system OPS-3 and made it faster by a factor of 25 to 200 by replacing an interpreter with an (on-line) compiler,[55] a feat that would appeal to the practically minded hackers who care about making computer systems better. However, his PhD thesis 'Lambda-calculus models of programming languages'[56] is a clear contribution to the mathematical culture of programming. Not long after his PhD, Morris joined Xerox PARC, where he contributed to the Modular Programming Language Mesa and the Cedar programming environment. Both projects had a strong engineering ethos. This is clear from the description of the Cedar environment, which is a 'software equivalent of the kind of machine shop needed by an engineering laboratory'.[57]

Morris' PhD thesis is analytical in the same sense as Landin's earlier papers. His aim is not to design a new programming language. Instead, he aims to explain two constructs used in programming languages, recursion and types, using the lambda calculus as a model. In a chapter on recursion, Morris relates recursion in programming languages to fixed point combinators in the lambda calculus. In a chapter on types, he presents a simple type system for a lambda calculus extended with operations for basic arithmetic. The extent to which this was directly inspired by Church's simply typed lambda calculus is not clear. The formalism has a striking similarity to the system described by Church, but Morris does not make an explicit reference to it. He cites the untyped lambda calculus and, in passing, a review of work on combinatory logic that mentions types.

Nevertheless, the material on types in Morris' thesis would be familiar to a contemporary programming language researcher. He proves 'sufficiency' of the type system, which guarantees that a well-typed program does not result in `ERROR`, a special kind of value produced by the formal model when an invalid program is evaluated. Morris

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

also presents an algorithm for type assignment (type inference), which finds a type for a program without explicit type annotations. The structure of the algorithm is similar to that of the Hindley–Milner algorithm from 1978, but simpler, because Morris' type system is not polymorphic. In a brief final section of the chapter, Morris extends his language with pairs and unions, inspired by McCarthy's data spaces and Algol 68 modes. He also adopts the now familiar notation of $\mathtt{T}_1 \times \mathtt{T}_2$ and $\mathtt{T}_1 + \mathtt{T}_2$ for pairs and unions, respectively.

What is also interesting about the work of Morris is that he understands types in a new way. Despite making a reference to McCarthy's set-theoretic model of types when talking about pairs and unions, his view of types is closer to that of Russell in the context of formal logic. Morris does not discuss what types are or how they should be modelled. Types are just a formal construction for identifying invalid programs. This new way of thinking about types in programming languages is clearly captured by a paper 'Types are not sets'[58] that Morris wrote while working at Xerox PARC, four years after completing his PhD thesis. The paper was shaped by his PhD research, but also by his experience working on the Modular Programming Language (MPL), a predecessor of Mesa, which used types for program structuring. Morris thus brings together the mathematical perspective, derived from the lambda calculus, with an engineering perspective, focused on what types are good for in modular programming languages.

Morris first summarises the dominant thinking about types in the mathematical culture at the times, which focuses on what types are. He notes that 'there has been a natural tendency to look to mathematics for a consistent, precise notion of what types are. The point of view there is extensional: a type is a subset of the universe of values. While this approach may have served its purpose quite adequately in mathematics, defining programming language types in this way ignores some vital ideas.' The vital ideas arise not from thinking about what types are, but what types can be used for: 'rather than worry about what types are I shall focus on the role of type checking. Type checking seems to serve two distinct purposes: authentication and secrecy.'

The problem of *authentication* is to guarantee that only values of the correct structure are admitted for processing. Type checking can, for example, ensure that only Boolean values (true and false) are passed to an operation that expects a Boolean. Thinking of types as sets models this problem well. We can see types as a mechanism for ensuring that only an appropriate subset of all possible values is passed to an operation. The problem of *secrecy* is to ensure that 'only the [given] procedures can be applied to objects of [a certain] type'. In this context, type checking should ensure that programs do not unnecessarily depend on the particular representation that a programmer chooses for their objects. Such representation should be hidden from most of the program. The representation should only be visible to a small number of privileged procedures, which are used by the rest of the program and can be easily modified in case the representation needs to change.

The second problem put forward by Morris is the use of types for what was starting to be called information hiding. The concept appeared in the engineering culture in the 1970s as an approach to modular programming. Information hiding was introduced by David L. Parnas, first in a technical report in 1971 and later in an influential paper

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

'On the Criteria to Be Used in Decomposing Systems into Modules'.[59] In the later paper, Parnas proposes a way of structuring programs into independent modules that expose a limited number of operations and hide the representation of data they use. This makes a system easier to change and understand and it also enables independent development of individual components. Morris has been thinking of a protection mechanism that could be used to support information hiding in programming languages. He published a paper on this very topic[60] less than a year before 'Types are not sets'. Morris does not explicitly cite Parnas in either of the papers, but he clearly suggests the use of types and type checking to ensure correct hiding of information in a modular programming language. Interestingly, Morris first describes a dynamic type system that seals and unseals data into or from opaque representations at runtime. He only later notes that some of those checks could be done statically, that is, before the program is executed.

The notion of a type fully acquired its new meaning as a mechanism for information hiding when Barbara Liskov and Stephen Zilles introduced the concept of 'abstract data types'[61] and described its implementation in the Clu programming language. Liskov and Zilles use the term 'type' to relate the mechanism to built-in types in programming languages. An abstract data type is a mechanism for introducing abstract values that are defined by the operations they support rather than by their representation. Liskov and Zilles marry the engineering motivation of Parnas with the use of types advocated by Morris and they reference both of the strands of prior work. In Clu, a programmer can define an abstract data type, which is an abstract description of a module in terms of its public operations. An abstract data type can then be implemented using a structure called a *cluster*, which gave the name to the language. As in the work of Morris, the abstraction in Clu was initially in part enforced by dynamic run time checks. However, Liskov and Zilles also cite personal communication with John C. Reynolds, noting that his recent work 'indicates that complete compile-time type checking may be possible'.

The mention of Reynolds was likely referring to his paper that appeared a year later 'Towards a Theory of Type Structure'.[62] Similar to the research presented by Morris in his PhD thesis, Reynolds' paper brings together the development of types in programming with types developed in the logic tradition. Both Morris and Reynolds used a method that many theoretical programming language papers would use over the next 40 years. They presented a simple extension of the typed lambda calculus and then proved that the extension had a certain desirable property.

In the case of Morris, the property was that a well-typed program evaluates to a value, which is a precursor of the type soundness property that became widespread in follow-up work over the next several decades. Reynolds' paper proves a formal property that he calls the 'representation theorem'. It states that the value obtained by evaluating a program does not 'depend upon the particular representations used to implement its primitive types'. In other words, Reynolds talks about information hiding, rather than about values produced by well-typed programs as is the case in Morris' thesis and in much of the present-day theoretical programming language literature. The work of Reynolds again highlights the value of interactions between different cultures of programming. Although the style and contributions of his paper are strongly rooted

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

in the mathematical culture of programming, the program property that he proves originates in the engineering culture of programming.

5.8 In Search of Types

The story of types told so far is that of a concept shaped by diverse commercial needs, theoretical insights and practical constraints. Different programming languages used types in different ways and different cultures of programming approached them from different perspectives. It may even seem that different authors and different languages are really working with different things. The term 'type' only became popular over time. The precursor of types in FORTRAN was called 'mode', COBOL defined the structure of data in terms of files and records and the early mathematical theory proposed by McCarthy referred to 'data spaces'. In later works, most authors talk about types, but they often further refine the terminology. Algol 68 distinguishes between primitive 'types' and composed 'modes' while the programming language feature to support modular programming was not called just a 'type' but consistently used the qualified term 'abstract data type'.

Yet, treating the multiple notions of type as separate things would hide interesting interactions between the different cultures of programming that shaped them. We have been two notable influences across the cultural boundaries so far. First, the idea of record types for structuring data in data processing appeared in the context of commercial computing, which I associate with the managerial culture of programming. A record simply moved from a filing cabinet into an electronic computer system. The need for richer data structuring was soon recognised by other programmers, leading to mathematical models of records and later to engineering work on supporting records in concrete programming languages including Algol W, Algol 68 and Pascal. The second notable influence across the cultural boundaries proceeded in the opposite direction. Much of the work on types and lambda calculus was done in logic, even before computers existed. Thanks to Christopher Strachey, Peter Landin and Mervyn Pragnell's underground reading group, those ideas entered the mathematical culture of programming and influenced James H. Morris who brought the ideas along when moving from the mathematical culture of his PhD research to the engineering culture at the Xerox PARC team working on modular programming languages.

The story of types is not unlike the story of the definition of a polyhedra told by Lakatos,[63] but the forces that shape a concept in programming are even more numerous. The main forces that shape the definition of a mathematical definition are proofs and counter-examples. In the case of types and programming concepts more generally, some of the forces originate from the corresponding mathematical models, but other forces are also rooted in practical commercial requirements and implementation constraints.

Even though the history of types is tangled, there are two different emerging themes in how types can be defined and used. The two themes have been identified by Stephen Kell.[64] Interestingly, Kell's divide is orthogonal to the boundaries between different

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

cultures of programming. On the one hand, we have 'data types' that primarily exist to structure data, define the representation of data in a computer memory and ensure that it is accessed correctly. Data types exist in COBOL in the form of records; they can be mathematically modelled as sets of values and are supported in programming languages through concrete mechanisms for defining records, unions or variant records in Pascal. On the other hand, we have 'types' as known from logic. The mathematical models of those refrain from stating what types are and instead focus on specifying how to check types. This perspective is derived from work on formal logic and was eventually adopted by Morris and became particularly useful for thinking about information hiding and abstract data types. The distinction between 'data types' and logical 'types' is something that many computer scientists glance over. However, to some, it was very clear as early as the 1960s. In a note about logical types that will play an important role in the next section, first circulated informally in 1969, Dana Scott writes 'the first confusion we should avoid is that between *logical* types and what we might call *data* types'.[65] James H. Morris was less explicit, but his problem of authentication is associated with data types, whereas the problem of secrecy is associated with types in the logical sense.

It may seem that, in the 1970s, the notion of a type was bound to diverge into multiple different concepts. There was the divide between 'data types' and 'logical types'. There were many differences between how individual cultures thought about and used types. Types in programming languages for business data processing were quite different from types in scientific programming languages. Yet, against all odds, the opposite happened.

5.9 The Definition of Standard Types

Robin Milner was another influential British computer scientist who attended at least one or two of Mervyn Pragnell's underground reading group meetings. Like many others in the group, Milner was interested in the semantics of programming languages, that is, in using formal mathematical methods to specify the meaning of programs. After graduating from Cambridge, Milner spent the late 1960s teaching at City University in London and as a research assistant in Swansea. During this time, he got interested in verification and automated theorem proving. As he recalled later, he was particularly inspired by the double relationship between 'the idea of a machine proving theorems in logic, and the idea of using logic to understand what a machine was doing'.[66]

This interest led Milner to work on various forms of machine-assisted theorem proving. When he moved to Swansea, Milner started by trying to create a fully automatic theorem prover. He soon 'became shattered with the difficulty of doing anything interesting in that direction' and, consequently, became 'more interested in amplifying human intelligence than … in artificial intelligence'.[67] Milner got a chance to shift his focus when he joined John McCarthy in Stanford in 1971–73 when two interesting developments came together.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The first development was practical. At the time, most work on theorem provers focused on automatic systems. This included the Boyer–Moore theorem prover, mentioned in Chapter 3, which was developed in early 1971 in Edinburgh. In those systems, you would specify a theorem, run the system and wait until it produced a proof or (more often) exhausted computer resources. Milner was well aware that finding proofs automatically was very hard. It was also the only thing you could reasonably try to do using a computer that was operated in the batch-processing mode, by handing a stack of punched cards to an operator and getting a result back the next day. Time-sharing systems that I discussed in Chapter 3 appeared at the end of the 1960s and allowed a more interactive use of computers. Users could connect to a computer via a terminal, enter commands and get results immediately. This also made it possible to build interactive theorem provers.

The second development was theoretical. The aforementioned privately circulated note by Dana Scott[68] introduced a formalism for reasoning about recursively defined functions. The formalism made it possible to prove equivalence of partially defined functions, which are structures that often occur in work on programming language semantics. Milner referred to the formalism as Logic of Computable Functions (LCF) and used it as the basis of his theorem prover of the same name.[69]

The first version of the prover, sometimes referred to as Stanford LCF, was created in 1971. The goal of the prover was to assist with proofs about programs and programming languages using the formalism developed by Dana Scott. A typical motivating example for the work was proving the correctness of a compilation algorithm.[70] The process of proving was interactive, meaning that you first entered some definitions and a goal, which describes the theorem that you wanted to prove. You then issued commands to transform the definitions and goals. Those include splitting the goal into sub-goals, applying rewrite rules to the definitions, or storing proved theorems into a library for later use. Although LCF was a proof checker rather than an automatic theorem prover, it included an automatic simplification algorithm that made completing proofs somewhat easier.

Constructing proofs in Stanford LCF was still a tedious process. One reason was that the range of commands that one could use to transform definitions and goals was fixed and limited. After moving to Edinburgh in 1973, Milner and his group started working on a 'metalanguage' that would make using LCF easier by allowing users to create their own proof construction operations. ML was a functional programming language that made it possible to write scripts that construct LCF proofs by manipulating the terms of the underlying object language. The design of ML was inspired by a number of programming languages, including Lisp, which was used to implement Stanford LCF and Landin's ISWIM.

To get a sense of how ML was used in the Edinburgh LCF, we can look at a brief example from a paper accompanying an invited lecture at the Mathematical Foundations of Computer Science conference in 1979.[71] The excerpt in Figure 5.6 proves that $X = X \vee X$ in a Boolean logic theory defined earlier in the paper. The code constructs a term `t` that we use as the starting point. It then fetches two different collections of axioms, `s1` and `s2`, and simplifies the original term. This is done by calling the built-

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

```
let  t = "X + (X * (¬ X))" ;;
let s1 = ss(map (AXIOM `BA`) [`andinv`; `oride`] );;
let t1,th1 = simpterm s1 t ;;
let s2 = ss(map (AXIOM `BA`) [`ordist`; `orinv`; `andide`]) ;;
let t2,th2 = simpterm s2 t ;;
let th = TRANS(SYM th1, th2) ;;
```

Figure 5.6 ML script that derives the proof that $X = X \vee X$ (writing + for logical disjunction and * for conjunction) in a Boolean algebra. (Source: Milner (1979))

in `simpterm` function with those two different sets of axioms as the first argument. The theorems assigned to `th1` and `th2` after those operations are `t==X` and `t==X+X`. The final line uses built-in symmetry and transitivity to construct a theorem `X==X+X` represented by `th`. It is worth stressing that ML was not restricted to simple lists of instructions such as those in the above example. It enabled users to define their own functions, or *tactics*, that implement different strategies of searching for proofs. To a contemporary functional programmer, the tactics would appear very much like combinators of a functional domain-specific language.

The challenge for the design of ML was to make it easy to write different strategies for constructing proofs, but prevent users from accidentally constructing non-theorems. To achieve this, ML represented a theorem as an abstract data type `thm`. The only pre-defined theorem values are axioms of a theory and the only operations that a programmer can use to create new theorems are the inference rules of the theory. This means that an ML program, no matter how sophisticated, can only ever produce valid theorems. The result may not be the theorem you wanted, but it will never be a non-theorem.

Unlike earlier languages such as Clu that initially relied on runtime checks, the correct use of types in ML was checked by a static type system. Using the terminology of James H. Morris, types in ML ensure the secrecy of abstract data types. There is no way for the ML code to inspect the structure of the `thm` type and create a new value from scratch.

The ML type system developed by Milner[72] is interesting for three reasons. First, the type system is polymorphic, meaning that types of operations can include type variables. For example, the type of a function that returns the first element of a list in ML is α `list` $\rightarrow$ α. Here, α is a type variable indicating that the operation can work on lists containing any elements. The function can be applied to, for example, a list of theorems `thm list` and it will return the first theorem `thm` from the list. Second, the system does not require users to write any explicit type information. It infers types automatically, based on the structure of the code. The algorithm is reminiscent of that described by Morris[73] but it is more sophisticated as it also handles polymorphism.[74] Finally, the paper also formally proves the soundness of the type system, which is captured by the slogan 'well-typed programs cannot go wrong'. This phrase has, in fact, an exact formal meaning. Milner defines how programs evaluate. Invalid operations

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691



Figure 5.7 The first volume of the Polymorphism newsletter contained a brief report on the November 1982 meeting and a range of contributed articles.[77]
(Source: Courtesy of Luca Cardelli and Dave MacQueen. Reproduced with permission.)

produce a special value, written as 'wrong' and the proof guarantees that well-typed programs never produce this value.

The ML in Edinburgh LCF supported functions, tuples and unions, but lacked richer mechanisms for working with data, including records and discriminated unions. Those were first added in the HOPE language that was based on ML and in an ML implementation known as 'ML under VMS' after the VAX/VMS system that it ran on. The 'ML under VMS' system was implemented by Luca Cardelli, who was working on his PhD thesis, focused on hardware verification, at the same time. Despite being developed by an academic computer scientist, 'ML under VMS' had some notable characteristics of the hacker culture. First, it generated VAX machine code and was significantly faster than the original that used a Lisp interpreter.[75] Second, the main early source of information about the project was a plain text file 'mlchanges.doc', included with the distribution. The 'ML under VMS' was also later ported to other platforms, most notably UNIX.[76]

By 1982, there were a number of incompatible versions of ML, including the original LCF/ML, HOPE, 'ML under VMS' and also 'ML under UNIX'. In November 1982, a meeting hosted by the Science and Engineering Research Council (SERC) brought together 20 programming language researchers interested in ML. The SERC organisers recognised that the ML family of languages was becoming a centre of research activity and were keen to promote collaboration. As reported in the first issue of the Polymorphism newsletter (Figure 5.7), the meeting included a discussion on the differences between ML and HOPE. The attendees could not yet envisage the ultimate functional language and thought that a 'variety and experimentation, rather than standardisation, were needed at this stage'.[78] Yet, standardisation is exactly what happened over a series of meetings in the early 1980s and by 1985, informal drafts for 'Standard ML' were already circulated among the group and the efforts focused on producing more formal mathematical definition of the language. This eventually resulted in 'The Definition of Standard ML',[79] published by MIT Press in 1990.

The ML language originated from the mathematical culture of programming. Many of the contributors were, at some point, affiliated with Pragnell's underground logic reading group. The language itself emerged from work on a proof assistant and the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

ML definition was written in a formal mathematical language. However, the ML type system brings together ideas from many different cultures. It uses types to track how values are stored in memory and uses those for efficient compilation in a way that dates back to the hacker notion of mode in FORTRAN. It supports data types such as records and discriminated unions, which are descendants of records in FLOW-MATIC and were further developed at the intersection of engineering and mathematical cultures. The abstract data type `thm` from the original LCF/ML links the ML notion of types to the engineering work on information hiding. It also later evolved into a support for modules that made it possible to define custom abstract data types and assist large engineering efforts.

The notion of types in the ML language finally brings together the 'data type' notion that emerged from the needs to represent different kinds of data with the notion of a 'type' that appeared in mathematical logic, but existed largely independently of programming languages until the 1970s. It would be tempting to use the Kuhnian perspective on scientific revolutions and see the era before the definition of types in the ML language as the pre-scientific era and the great unification of ideas on types that happened with ML as the birth of a normal science. This would not be entirely unjustified: there was no consensus on a particular theory of types in the era before ML and the birth of the ML paradigm defined a set of methods and the kinds of questions that they can answer. The ML paradigm became hugely influential and has a large following in both the mathematical and the engineering cultures of programming. Yet, the definition of types in ML was not the end of the story and we will soon see the concept developing in multiple different directions, under the influence of the engineering culture and the logical tradition of the mathematical culture. Before that, I briefly recount the line of work on types that emerged from ML.

5.10 Types as a Lightweight Formal Method

The ML approach to types has two key characteristics that enable it to work as a research paradigm: it is both clearly delineated and sufficiently open-ended. On the one hand, the definition of Standard ML sets out a specific way of working with types. Types should be checked at compile-time and should prevent runtime errors. They should be defined formally using a mathematical language and the definition of Standard ML shows how to do so conveniently using inference rules. A programming language with type checking should provide guarantees about the execution of well-typed programs, captured by the slogan 'well-typed programs cannot go wrong'. On the other hand, the phrase 'cannot go wrong' does not say what kind of wrong behaviours should be eradicated using types. The formal methodology based on inference rules, as pioneered by the definition of Standard ML, is easy to extend. It can be used to talk about memory safety, correct implementation of network protocols, thread safety and many other properties. These two characteristics of ML types enabled a large number of theoretical computer scientists to follow suit and contribute new notions of type, addressing a wide range of programming problems.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The research paradigm defined by the ML notion of types caters primarily to the mathematical culture of programming. It shifts focus from issues such as implementation or commercial applications to theoretical work. There is an implicit assumption that the theoretical work is the first step towards developing practical programming languages. The paradigm is perhaps best captured in the introduction of a textbook *Types and Programming Languages*:[80]

> Modern software engineering recognizes a broad range of formal methods for helping ensure that a system behaves correctly with respect to some specification …On one end of the spectrum are powerful frameworks [that] can be used to express very general correctness properties but are often cumbersome to use … At the other end are techniques of much more modest power—modest enough that automatic checkers can be built into compilers … and thus be applied even by programmers unfamiliar with the underlying theories. … [By] far the most popular and best established lightweight formal methods are type systems, the central focus of this book.

This quote positions the concept of a type unequivocally in the context of the formal mathematical approach to programming. In doing so, it also suppresses the rich multicultural history of the concept with its many alternative interpretations of the concept. The introduction also leaves no doubt that there should be a specification of a program behaviour and that we need a formal mathematical method to ensure that an implementation conforms to this specification. As a good paradigmatic textbook should, the introduction defines what ought to be done with types within the ML paradigm.

Thinking about types as a lightweight formal method stretches the managerial idea of a specification. First, types are typically written by the same programmers who write the implementation. This is in contrast with conventional specifications, which are written by analysts and approved by the managers. Second, checking that an implementation follows a specification is no longer a human activity, as for example in the Cleanroom methodology that I discussed in Chapter 2. Instead, it is a task for an automatic type checker.

In this way, the ML tradition of types provides a new perspective on the critique by DeMillo, Lipton and Perlis that formal proofs about programs lack social process. Social processes involving types are not centred around the question of proof checking, that is, verifying that an implementation matches a specification. The type checker does that automatically. However, programmers actively discuss types in at least two ways.

First, they talk about types of common functions and what they mean. For example, functional programmers will happily spend a lot of time explaining what a library function does by discussing the details of its type. This is especially the case for general polymorphic functions like `map` with types such as `List` $a \to (a \to b) \to$ `List` b. Close reading of this type often provides a good enough understanding of what the operation may be doing. Here, the function accepts a list of values of type a and another function that knows how to turn a value of type a into a value of type b. The `map` function then produces a list of values of type b. The only way to do this is to go over all the values in the input list, apply the given function to each element and collect the results. This is, in fact, what the `map` function does.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Second, programmers often ask their peers 'why is this code not type checking?' when attempting to make sense of a programming error. Especially in languages where types are inferred automatically, programmers attempt to follow the logic of the type checker and see where their intuitive understanding of what their code does fails to match with what the inferred types indicate. Both of these are important social processes which ensure that the use of types as a lightweight formal method is not just a black-box that labels programs as 'correct', but a human process.

Comprehensively documenting the work on types in the ML paradigm is a topic for another publication, but it is worth discussing a few directions. First, there are many ways in which a program can go wrong. Many interesting wrong things can happen when different parts of a program can write to and read from a shared memory or in programs that access external resources such as the file system or network. Type and effect systems[81] annotate type information with an *effect* that the evaluation of an expression has. Effect systems were first used to avoid issues arising from the use of shared memory, but were later also used to check the correct use of resources and in other areas. Region-based memory management[82] made it possible to free the memory used by a program more efficiently, while ownership types[83] address issues that occur when the state of an object in an object-oriented language is modified by multiple other objects. All this work subscribes to the ML notion of types and belongs to the mathematical culture of programming. As is often the case in programming, the ideas found their way to other cultures. The perfect example of this is the Rust programming language, first announced in 2010. Rust has engineering origins and ethos, but uses ownership and advanced memory management techniques, which originate in the mathematical culture of programming, as a basis for the design of a safe systems programming language.

Rust is a somewhat unique case. It adopts two specific techniques for solving engineering issues around memory safety and efficiency. For the Rust designers, this issue is important enough to warrant a direct support in the programming language. Many other extensions developed within the realm of the ML paradigm of types are even more single-purpose. This makes them easy to define using the language of the mathematical culture, but it makes it hard to justify their implementation in an industrial-scale programming language arising from the engineering culture. As the types got more complex, it also became harder to keep the usability afforded by the type inference, which frees users from having to write types explicitly. The type system may require more annotations or explicit type specifications, it may become intractable, as well as hard to understand and use.[84]

A different research direction has been taken by the Haskell language which appeared in 1990. Haskell was initially motivated less by a specific approach to types and more by its evaluation strategy. It uses lazy evaluation, meaning that arguments of a function are not evaluated before the function call, but only when they are actually used. In the late 1980s, there was a large number of experimental functional languages featuring lazy evaluation. Haskell was a community effort to introduce a single common language that could be extended, modified and used for research into language features.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Since its creation, much research related to Haskell focused on types. The initial design of Haskell introduced 'type classes' which can be used to specify that a function argument is of any type that supports certain operations. For example, a function `sqr` that calculates a square of a number has a type `Num a => (a -> a)`. This specifies that the function works on any type `a` as long as that type is numerical and supports numerical operators such as multiplication. Type classes solved a relatively small problem at first, but they inspired further work on types in Haskell. The Haskell designers later described Haskell as a 'type-systems laboratory' and recalled that:[85]

> An entirely unforeseen development—perhaps encouraged by type classes—is that Haskell has become a kind of laboratory in which numerous type-system extensions have been designed, implemented, and applied. Examples include polymorphic recursion, higher-kinded quantification, higher-rank types, lexically scoped type variables, generic programming, template meta-programming, and more besides.

The list of examples would be even longer if it included extensions implemented after 2007 when the quoted paper was written. Although Haskell originated within the mathematical culture of programming and most such extensions are formally described in an academic paper, it adopts a different approach to types than the research projects born from the ML paradigm. Haskell includes a family of general purpose type-level mechanisms that can be used to express more detailed specifications using types. Adopting those often requires what I would classify as unwritten knowledge of the hacker culture. Expressing various constraints on the program logic also often involves tricks that are not well documented, but are known to the community of Haskell users.[86]

There are many kinds of constraint that can be specified in Haskell through type system tricks, but types in Haskell are not used for writing fully general specifications.[87] The language, however, led the way to a more general notion of a type that does make it possible to specify an arbitrary constraint. In order to document this idea, we need to return to the early days of interactive theorem proving, before the appearance of ML.

5.11 Automating Mathematics Using Types

As we saw earlier, the ML language, which defined a new paradigm for types in programming languages in the 1970s, has its origins in the work on the LCF interactive theorem prover. Curiously enough, later work on interactive theorem proving, which evolved in parallel with the work on the ML language, reshaped the notion of types in programming languages yet again, in the early 2000s.

To understand the later developments, we need to return to an even earlier interactive theorem prover. The system was called Automath and was developed by Nicolaas Govert de Bruijn in 1967.[88] Automath was not widely publicised, but it was not entirely unknown. Interestingly, Donald Knuth mentions Automath in his review of Hoare's chapter in *Structured Programming*,[89] and notes that Automath is worth looking at because it 'is the epitome of the concept of type'. Automath uses the logical notion of

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

a type in an even more profound way than the work of Morris on lambda calculus or the work on the ML programming language.[90]

At the core of Automath is the remarkable mathematical equivalence between types and logical propositions that Philip Wadler referred to in the presentation that I discussed in the opening section of this chapter. This relationship is now known as the Curry–Howard correspondence after Haskell B. Curry who observed an early version of the idea in 1934 and William A. Howard, who described the modern version of the idea in 1969, independently of de Bruijn who was already using it in Automath.

The Curry–Howard correspondence captures a relationship between types in the lambda calculus and propositions in intuitionistic logic. The key idea is that logical propositions can be understood as types. To prove a proposition, we need to show that there is a value of the required type (the type is inhabited). For example, let's say that we want to prove that $P \rightarrow (Q \rightarrow P)$. This is not a surprising property, but it will serve well as an example. The proposition uses a nested implication. It states that, from P it follows that Q implies P. This can also be understood as stating that, from a proposition P and a proposition Q it follows that P. This reading should already show that the original proposition is true.

Using the Curry–Howard correspondence, we can view this proposition as a type of a function $P \rightarrow (Q \rightarrow P)$ where P and Q are some arbitrary types. In modern programming terminology, the function is generic (or polymorphic) with two type parameters. It is a function that takes a value of type P and returns a function taking a value of type Q and returning a value of type P. This is easy to define! Using the lambda notation, we can write it as $\lambda x.\lambda y.x$. The same can be written in a programming language like JavaScript as:

```
function(x) {
  return function(y) { return x; }
}
```

The type $P \rightarrow (Q \rightarrow P)$ is inhabited by the above function and so the proposition holds. The Curry–Howard correspondence also helps us understand why proving $P \rightarrow Q$ is impossible. We would need to construct a function that somehow produces a value of an arbitrary type Q using just a value of type P. We do not know anything about those two types. All we have is a single value of type P and so there is no way of constructing a value of type Q. The correspondence also works for conjunction and disjunction, which map to pairs of values and union types, respectively. This was already used in Automath. It means that, for example, a logical proposition $(P \wedge Q) \rightarrow P$, stating that P and Q implies P, corresponds to a function that takes a pair of P and Q and returns the first component of the pair.

More interestingly, the correspondence can also be extended to quantifiers of predicate logic. This idea was developed in systems that followed Automath and I will get to them in the next section. The types corresponding to universal and existential quantification are more subtle than those for functions, pairs and unions and are known as dependent types. A type corresponding to universal quantification is the dependent function type, written as $\Pi_{x:A}B(x)$. This is a type of function whose return type depends on the value of the argument. Given a value x of type A, the return type $B(x)$ is a type

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

obtained based on the value x. The complexity introduced here is that $B(x)$ is itself an expression that calculates the resulting type and needs to be evaluated during type checking. The mechanism, however, makes it possible to express types for functions that return different types of results depending on the value of their input argument. A type corresponding to existential quantification is known as the dependent pair $\Sigma_{x:A}B(x)$ and it is a pair of values (x, y) where the type of the second value depends on the first value. The key idea is that types are no longer just simple types like integers, records or lists of integers, but can include computations that produce types based on values. This makes type checking difficult, but as we will soon see, it has interesting practical applications.

The Curry–Howard correspondence shows that types in functional programming languages are related to proofs in logic, but does it justify saying that functional languages are discovered rather than invented as suggested by Philip Wadler in the talk discussed in the opening of this chapter? Mathematically minded computer scientists who accept this argument do so based on a number of assumptions that are typical for the mathematical culture of programming. The first is the unification of complex engineered software systems, such as programming languages, with their mathematical models. The implementation of types in an actual compiler is always more complex and involved than the formal model given on paper. Programming languages often also have various pragmatic features, which are omitted 'for simplicity' from their models that further complicate their formal properties.[91]

The second assumption of the mathematical culture is that it focuses on current mathematical knowledge, but ignores the broader historical perspective. The Curry–Howard correspondence relies on a very carefully constructed matching pair of logic and type theory that has been obtained by an iterative process of refinements.[92] Moreover, both formal logic and the lambda calculus arose from the same community of the early twentieth-century logic, working on the same kinds of issues in the foundations of mathematics. Given this context, the correspondence between types and logic is perhaps less remarkable on its own. The correspondence did, nevertheless, have a remarkable influence on programming languages and types.

5.12 Programming in Type Theories

A number of interactive theorem provers were more or less directly inspired by Automath and used the idea of Curry–Howard correspondence as their basis. As explained in the previous section, creating a proof in a theorem prover based on the Curry–Howard correspondence is done by constructing a program in a language derived from the lambda calculus. This may perhaps not be obvious when one looks at an example such as that of the ALF theorem prover in Figure 5.8 and so it took some time until the wider programming language community realised that this style of working could be used not just for constructing proofs, but also for constructing programs.[93] The change in thinking happened at the turn of the millennium, mainly in Sweden and France.

In the 1970s, Per Martin-Löf introduced a type theory that included the dependent function and the dependent pair types discussed in the previous section. He developed

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

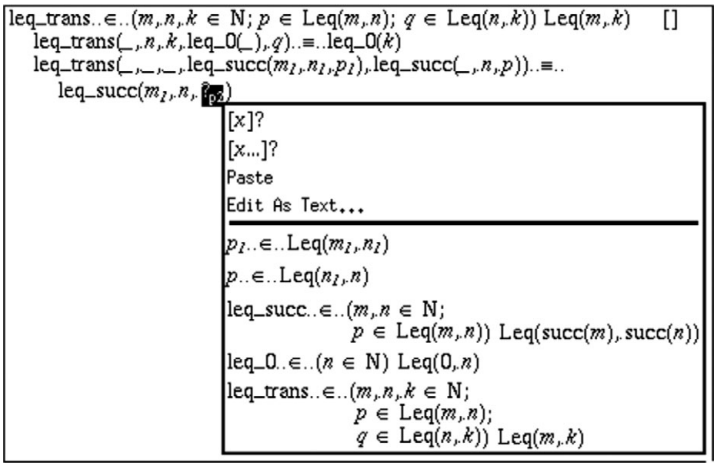


Figure 5.8 Constructing the proof of transitivity of the less than or equal ($\leq$) relation on natural numbers in the interactive theorem prover ALF.
(Source: Altenkirch et al. (1994). Reproduced with permission.)

the theory as part of his work on constructivist foundations of mathematics where existence proofs have to specify how to construct the mathematical entities they postulate. The Curry–Howard correspondence is a perfect fit. A constructive proof created using the type theory can be seen as a program that produces the required mathematical object. Seeing such proofs as programs was, however, a later development. After visiting Martin-Löf in the early 1980s, Bengt Nordström from Chalmers University started writing a book, *Programming in Martin Löf's Type Theory*. The book envisioned a way of using the type theory as a programming language:[94]

> [Per Martin-Löf's type theory] is well suited as a theory for program construction since it is possible to express both specifications and programs within the same formalism. Furthermore, the proof rules can be used to derive a correct program from a specification as well as to verify that a given program has a certain property.

The introduction alludes to the idea that one can write a precise specification as a proposition and, by constructing a proof of the proposition using the type theory, derive a program that implements the specification. Despite having the word 'programming' in the title, the book was more an introduction to type theory with references to functional programming than a book about programming in the conventional sense. The book's example programs can convince only devoted members of the mathematical culture. The most sophisticated one is a program that sorts a sequence of coloured objects. Still, the book inspired the development of a family of theorem provers that eventually brought theorem proving closer to programming, starting with the aforementioned ALF theorem prover,[95] developed at Chalmers University in the early 1990s.

Around the same time, Thierry Coquand introduced the Calculus of Constructions, which was another type theory based on the Curry–Howard correspondence, and

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

```
PrintfType :: String -> #
PrintfType "" = String
PrintfType ('%':'d':cs) = Int -> PrintfType cs
PrintfType ('%':'s':cs) = String -> PrintfType cs
PrintfType ('%': _ :cs) = PrintfType cs
PrintfType ( _ :cs) = Printf Type cs

printf :: (fmt::String) -> PrintfType fmt
```

Figure 5.9 The definition of `printf` function in Cayenne. `PrintfType` is a function that takes a string and returns a type. It is recursively defined by pattern matching over the first characters of the string. For example, if the string starts with `%d`, the result will be a function type taking `Int` and returning whatever type is returned by recursively processing the rest of the list. The `printf` function itself (last line) takes a string `fmt` as an argument and its return type is `PrintfType fmt`, which depends on the value of the format string, given as the first argument.
(Source: Augustsson (1998))

started working on the Coq theorem prover in Paris with Gerard Huet and Christine Paulin-Mohring. The work was inspired by Automath, as well as more theoretical developments around types and the lambda calculus. Both ALF and Coq were primarily theorem provers, even though they were clearly designed with program construction in mind. The authors of the Coq theorem prover soon started experimenting with using it for programming. In the early 1990s, they developed a mechanism for extracting Caml (a language based on ML) programs from Coq proofs.

At the end of the 1990s, two experimental programming languages attempted to bring dependent types into programming in a more conventional way. The languages were Cayenne, created at Chalmers University and Dependent ML, created at the Carnegie Mellon University.[96] Dependent ML took a conservative approach. It allowed natural numbers to appear in a type, as in the type `Vect` n α representing a vector of values of type α of length n. Cayenne supported unrestricted dependent types. A practical motivating example used by the authors was the `printf` function. The type of the function depends on the value of the format string given as an argument (Figure 5.9). For example, the string format `"Hello %s!"` requires a `string` argument and so the type of `printf "Hello %s!"` is a function `string` → `string` whereas `printf "%d"` returns a function `int` → `string`.

Both Cayenne and Dependent ML combined the idea of dependent types, which until then existed only in the mathematical world of theorem provers, with a practical programming language. As the name suggests, Dependent ML extended the ML language while Cayenne was implemented in and inspired by Haskell. Equally, both Dependent ML and Cayenne remained an early exploration of the idea of practical dependent types and were not adopted in practice by a broader programming community.

The next wave of dependently typed programming languages appeared in the mid 2000s and included Epigram, Agda and Idris.[97] The latter two evolved into practical

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

programming languages that have been used not just by academics, but also by some practitioners and have been maintained ever since. At the same time, the Coq theorem prover was also further developed in ways that make it more usable as a programming language and was used, for example, to develop a formally verified compiler for the C language called CompCert. The compiler was started in 2005 and is now available under a commercial licence.[98]

One interesting aspect of dependently typed programming languages is their interactive programming environment. The fact that an interactive programming environment would come from the mathematical cultures is perhaps unlikely. As we saw in Chapter 3, most earlier work on interactive programming was rooted either in the hacker or the humanistic culture of programming. However, interactive theorem provers offer a new kind of mathematical perspective on programs. Rather than seeing programs as mathematical entities that should be proved correct, they see programs as mathematical entities (proofs) that are interactively constructed.

This was not yet the case in Automath where the users would write 'a book' and submit it to be checked.[99] Automath was created before the emergence of time-sharing and personal computers, so this was likely the only way it could have operated. However, all later interactive theorem provers and dependently typed programming languages have an interactive editor that provides immediate feedback.

There are two basic ways of interacting with such a system. In the first style, the user starts with the desired type (theorem) and enters a sequence of commands that invoke various tactics that attempt to construct the program. This approach is used, for example, in the Coq theorem prover and in the ALF editor shown in Figure 5.8. The tactics can be simple rewrite rules, but also more powerful algorithms such as `auto`, which attempts to find a program automatically, using variables in scope (hypotheses) and other standard rewrite rules. This way of programming is akin to using a Read-Eval-Print Loop (REPL). The programmer proceeds step by step, entering commands and observing their effects until they reach the desired goal. In a somewhat different context, this is exactly how hackers work.

In the second style, the user writes the desired type alongside source code of a program in a functional language that has the desired type. This style of working is closer to the typical way of working within the mathematical culture, for example, when writing code in an ordinary functional language. There are two caveats though. First, the programmer starts with a type, rather than starting with the implementation and letting the compiler infer the type as in ML. The type is written first because it is the specification of the program that we are creating. Second, when writing code, the programmer is supported by an interactive editor that offers hints using clever auto-completion tools. It also checks the program in the background and reports mismatches with the desired type.

Naïvely, we could see dependent types as an evolutionary step that follows the work on functional programming languages arising from the ML paradigm. After all, if we view types as a lightweight specification mechanism, then supporting richer types in order to write more precise specifications would be a natural direction. This might have been the motivation for Dependent ML, which was clearly rooted in the ML tradition,

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

but the history is more complicated when it comes to types in languages like Coq, Agda and Idris. All of those depart from one of the basic principles of the ML tradition, which is the fact that types are inferred from code and are checked automatically. By contrast, in Coq, Agda and Idris, types are written up-front and programming is an interactive process in which the programmer constructs an implementation of the right type, supported by various interactive editing tools provided by the programming environment. Seeing dependently typed programming languages just as the next step of the ML paradigm would thus be misleading.

The development of Coq, Agda and Idris partly repeats the meeting of cultures that was necessary for the birth and popularisation of the ML language. The three languages have origins in the mathematical world of theorem proving. However, they combine this with engineering motivations. For example, an early motivation for Idris was systems programming, which motivated a sophisticated system for handling side-effects. To support the interactive development of programs, the languages also required a suitable editing environment. In the early days, those were often implemented as extensions to the Emacs and vim text editors. The two editors are not just the products of the hacker culture, but the rivalry between them epitomises the hacker culture. The latest generation of dependently programming languages is a product of another meeting of cultures, the fruits of which we are probably yet to fully see.

5.13 The Meaning of Types Is Their Use

In the mathematical culture of programming, the primary argument in favour of using types is that they can guarantee partial correctness of programs. In the ML tradition, this was made explicit by the slogan 'well-typed programs cannot go wrong'. However, preventing errors is just one thing that can be achieved using types. Even the proponents of the mathematical culture recognise usefulness of types for engineering reasons. The lecture notes for the theoretical 'Types' course at Cambridge University of Cambridge[100] opens by documenting five uses of type systems: detecting errors, support for structuring large systems, documentation, efficiency and whole-language safety. The three reasons in the middle of the list are chiefly engineering ones. The engineering culture has found many practical ways of leveraging types in the recent history, but it does so using its characteristic approach. That is, by building tools that assist with programming. Many such tools used types and continue stretching the notion of types.

In parallel to the mathematical line of work that I followed earlier in this chapter, types were adopted by object-oriented programming languages, which I will discuss in depth in Chapter 6. The notion of a type in this context has been mainly influenced by the Algol tradition. In the object-oriented language SIMULA, types were used for ensuring that code does not access non-existent object members. The language became an inspiration for Bjarne Stroustrup, who designed the C++ language in 1979. He later acknowledged this influence, but also emphasised what the SIMULA tooling based on types enables:[101]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

> I acquired a great respect for the expressiveness of Simula's type system and … its compiler's ability to catch type errors. …[A] type error almost invariably reflected either a silly programming error or a conceptual flaw in the design.

In the engineering culture, types were initially used for error checking and for efficient compilation of code. They soon also became valuable for the development of sophisticated developer tooling. Much of the early advanced developer tooling originated from programming systems built around Smalltalk and Lisp. In those systems, the tooling could use the powerful reflective capabilities of the programming system to understand and expose the program structure. This approach worked well for interactive programming systems, but not for static programming languages. In the early 1990s, the short-lived Lucid Energize development environment attempted to bring some of the advanced developer tooling to the C++ programming language. The system originated in the Lisp community and its implementation attempted to recreate some of the reflective capabilities of Lisp systems through a system of active annotations.[102] Later systems started to increasingly rely on static information about the program.[103] Types thus soon became an alternative source of information about the program structure that could be used to develop advanced developer tooling. An example of this is Microsoft Visual C++ 4.0 (Figure 5.10), which used type information to offer some of the capabilities pioneered in Lucid Energize.

The engineering culture adopted types as a powerful mechanism for building developer tooling. Most notably, types were used for navigation and for code completion. When invoking a member of an object, the editor would automatically offer a list with the available members and programmers would be able to choose a member from a

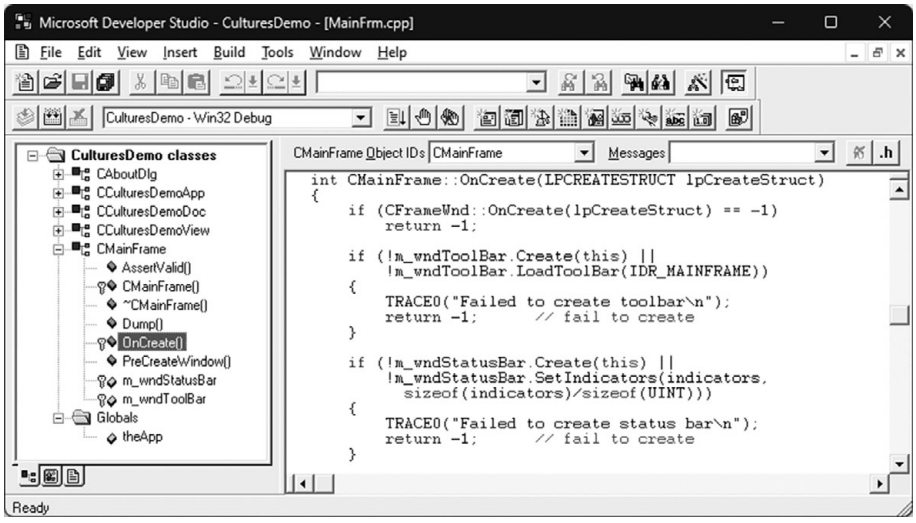


Figure 5.10 Microsoft Visual C++ 4.0, released in 1995, displayed a summary of available types in the 'Class View' panel (left) and allowed programmers to navigate to individual class members.
(Source: Screenshot created by the author, using VC++ 4.0, running on Windows 11)

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

list instead of typing its name in full. In the 1990s, this engineering use of types did not reshape the notion of a type itself. With the growing importance of developer tools and the associated engineering perspective on types, however, engineering trade-offs started to influence the design of types in programming languages.

A prominent recent programming language that makes an engineering trade-off that would not be acceptable in the mathematical culture is TypeScript. The language was released by Microsoft in 2012 and its primary motivation is to enable better tooling for large-scale JavaScript development. TypeScript extends JavaScript with static type checking in a backward compatible way. The design of its type system has been motivated by the patterns that appear in popular JavaScript libraries.[104] TypeScript also makes it possible to annotate an existing JavaScript library with types, through a separate definition file. This allows better developer experience, but the annotations are not (and cannot) be checked, since the implementation is imported from JavaScript.

Annotations for external libraries are one source of potential runtime errors. It may be the one that is important in practice, but not the one that would worry the proponents of the mathematical culture. More worryingly for the mathematicians, the TypeScript type system designers intentionally made a number of design decisions that make the type system unsound. This means that if a type checker determines that a variable has a certain type, this may not actually be the case when a program is executed. As explained by a member of the TypeScript development team, '100% soundness is not a design goal. Soundness, usability, and complexity form a trade-off triangle. There's no such thing as a sound, simple, useful type system.'[105] The creators of TypeScript intentionally chose a simpler but unsound type system design in places where soundness would introduce additional complexity.

A proponent of the mathematical culture would be sure to point out that there is nothing like 99% soundness. In theory, a type system either is sound, or it is not. In practice, a flaw that rarely leads to a problem is still a flaw, but one that is worth accepting if it allows the designers to better fulfil other goals. Since TypeScript compiles to JavaScript, which performs full runtime checking, the lack of soundness does not lead to errors that would make the system unsafe. The TypeScript design follows a set of coherent design principles which are distinct from the principles that are favoured by the ML paradigm. The focus on tooling and producing simple useful documentation for programmers means that it makes sense to sacrifice soundness for simplicity. The approach developed in TypeScript again brings together multiple perspectives developed by multiple cultures of programming. The type system is rooted in the engineering culture, with its focus on simple tooling. At the same time, it incorporates a number of type system concepts developed within the mathematical culture, including discriminated union types from ML and even a form of dependent types. This was added because it was necessary for tackling tricky patterns in existing JavaScript code.[106]

The engineering approach to types departs from the earlier work in a number of ways. It is not worried about formal properties of type systems. Types are useful as long as they have practical benefits and do not get in the way of programmers. The engineering culture is also not concerned with the nature of types and so it does not attempt to define what a type is. Types are not explained as sets or through their logical

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

models. They are a mechanism used in the implementation of various tools. Types in the engineering cultures are defined less by what they are, more by how they are used in the programming systems and tools.[107]

5.14 Stalking the Elusive Type

The meaning of the notion of a type in programming languages shifted dramatically several times throughout its history, but it remained sufficiently stable that a working programmer nowadays can understand most of the different takes on types discussed in this chapter. Different cultures of programming refer to similar things when they talk about types, but think about them differently and look to different origins of the notion. For the hackers, types started as a useful trick to compile programs correctly; for the mathematicians, types import a powerful logical notion for ruling out invalid programs; and for the engineers, types are a mechanism for program structuring and tool development.

Looking at the 70-year history of types in programming makes it clear that there is no simple answer to the question: What is a type?. The engineers and hackers do not seem worried about the lack of a definition and use or implement types in whatever way works. The mathematicians attempt to bring order into the chaos by providing definitions, but those do not stop others using types in yet another incompatible way. A type in Algol 58 was simply one of three pre-defined primitive types, but the idea that programming languages should be extensible and support data as known from business data processing soon changed that and types became sets of values. The logical approach that became ubiquitous in the 1980s shifted the focus from what types represent to how they are checked. This has made it easier to develop new notions of types that do not fit the model of 'types as sets', but it was also less satisfactory to some in the mathematical culture who believed that a type should denote a mathematical object such as a set.

One attempt to again redefine what types are, emerged from work that extends ML types with units of measure.[108] This makes it possible to annotate numerical types with their physical units such as meters or kilograms. The system followed the ML tradition and infers not only types, but also their units. It also does it in a polymorphic way, meaning that units can be abstracted over. For example, a function that calculates the square of a number has a type $\texttt{int}\ u \to \texttt{int}\ u^2$. It takes a number with any physical unit u and returns a number with the same unit squared. Calling the function with a value in meters such as $5m$ results in $25m^2$. The type of this function indicates more than just the fact that it takes a number and returns a number. All functions f of type $\texttt{int}\ u \to \texttt{int}\ u^2$ have the property that $f(k*x) = k^2 * f(x)$ for any constant k. In other words, if we multiply the input by k, the result is multiplied by k squared. The question is, how can this information be captured by the mathematical entity representing the type?

The answer is to represent types not as sets, but as mathematical relations between the input and the output. A relation holds between only some elements of the input and

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

the output sets and so it can, for example, capture the property defined by the units. The idea also proved useful for defining the meaning of types that track the effects that a computation may have. In type and effect systems, the type `int` & {`read` ρ} denotes a computation that produces a number, but it may also read from a memory region ρ. A mathematical relation can capture the fact that values in memory regions other than ρ cannot affect the result of the computation.

This brief example illustrates how the notion of a type evolves and how different cultures of programming can participate in such developments. The use of types for tracking units was motivated by engineering concerns. However, implementing the idea turned out to be at odds with the established mathematical models of types. This counter-example[109] prompted the mathematically minded computer scientists to revisit their definitions of what types are and develop a new formal theory. When the established normal science methods based on modelling types as sets stopped working, mathematically minded computer scientists were forced to reopen the black box of types and develop a new view.[110]

5.15 Pluralism and Scientific Progress

Types are shaped by a complex network of interactions between multiple cultures that interpret and use types differently. The different cultures of programming are, nevertheless, able to exchange ideas thanks to this shared notion and are also able to contribute new aspects to the notion. In a way, types and their concrete implementations in different languages provide a shared language through which the different cultures can communicate. For example, the mathematical culture first realised that types can be fully checked at compile-time. The implementors of abstract data types in the engineering culture used this to eliminate runtime checks and the hacker culture used the idea to improve the efficiency of compilers for languages with types.[111] Types can also play the role of a common language that different cultures can share. For example, abstract mathematical ideas of category theory can be translated to type definitions, which engineers can understand and use for their own purposes.[112]

Types can also become a concept that requires the skills of more than just a single culture of programming. An example would be types as implemented in the programming language Haskell. The language is rooted in the mathematical culture and its GHC compiler has become a lively playground for theoretical research on types. The theoretical ideas are, however, complemented with a substantial engineering implementation. Furthermore, as the extensions to the GHC type system grew, using them started to require more than just theoretical understanding. The only way to learn how to use the variety of extensions is through practical experience, which is a typical attribute of the hacker culture.

The history of types is not linear and does not follow a fixed pattern. Different cultures contribute their ideas, based on different approaches at different points in time. As a result, the development of types is perhaps best explained by the theory of epistemological anarchism.[113] This is not always appreciated by proponents of more

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

rigorous cultures, who believe that having a clear definition would allow a greater collaboration, but there are undeniable benefits of such a structure. The notion of a type remains a living process and even a successful unification, such as the ML paradigm, does not permanently freeze the notion. Talking about types in an integrated way that does not rely on a definition and can account for the mutually inconsistent notions of type used by different cultures can perhaps be best done by considering what can be done using types. If we can use types to produce more efficient compiled code, eliminate bugs or build developer tooling, then we obtain knowledge about types that is independent of a particular definition.[114]

Types in programming are an inherently pluralistic concept. In his reflection on types, Simone Martini generalises the idea to the entire discipline of computer science:[115] 'The crucial point, here and in most computer science applications of mathematical logic concepts and techniques, is that computer science never used ideological glasses …, but exploited what it found useful for the design of more elegant, economical, usable artifacts. This eclecticism (or even anarchism, in the sense of epistemological theory) is one of the distinctive traits of the discipline, and one of the reasons of its success.' The fact that the notion of a type is a multi-cultural mix of ideas that have never been fully integrated might well be the reason behind its success.

Notes

1 Backus et al. (1956).
2 Paraphrasing a letter to the editor of the *Communications of the ACM* (Postley, 1960), quoted by Ensmenger (2012), who discusses it in the context of broader tensions between the data processing industry and academic computer science.
3 Brown (1886).
4 McCarthy (1961).
5 The difficulty of modelling pointers has been pointed out by Priestley (2011). The origins of pointers in high-level languages date to the Address programming language created by Kateryna Yushchenko in 1955 (Yushchenko, 2022). In the mathematical culture in the West, the problem of pointers was dealt with by Hoare (1965) who proposes typed record references, but without discussing their interpretation in terms of sets.
6 Morris (1973b).
7 The concept of information hiding has been described by Parnas (1972) and its usefulness for management is discussed by DeRemer and Kron (1976). The case of information hiding is also discussed as part of a review of the influence of software engineering on programming languages by Ryder et al. (2005).
8 Reynolds (1974).
9 For the multi-cultural origins of types, see Martini (2016) and for different possible answers to this question, see the reflections by Kell (2014) and Petricek (2015).
10 Milner (1978).
11 For these particular examples see the work on session types, coeffects, region-based memory management and ownership types (Honda, 1993; Petricek et al., 2014; Tofte and Talpin, 1997; Clarke et al., 1998).
12 In the context of mathematics, this process, including that of concept-stretching, has been described by Lakatos (1976). Programming concepts like types and tests are subject to similar evolutionary processes.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

13 This is the framing used, for example, in the introduction of a popular textbook on types by Pierce (2002).
14 This is often informally acknowledged in the functional programming community, e.g., 'Being able to just show the code to a client and have him immediately catch domain modelling errors: priceless.' (https://twitter.com/brandewinder/status/570437796113985536, retrieved 6 Oct, 2022) Description of this way of working with types can be found, for example, in a book by Wlaschin (2018).
15 www.typescriptlang.org/docs/handbook/type-compatibility.html, retrieved October 7, 2022.
16 This is a perspective suggested by Kell (2014), who also provides a number of examples of confused claims resulting from the conflation.
17 The talk is based on a paper of the same name (Wadler, 2015). The talk recording is available at: www.youtube.com/watch?v=aeRVdYN6fE8, retrieved October 26, 2022.
18 A popular summary of my own objections can be found at: https://tomasp.net/blog/2018/alien-lambda-calculus/, retrieved October 26, 2022.
19 Documented by Urquhart (1988); broader context is discussed by Coquand (2018).
20 Russell (1903).
21 Russell (1908).
22 Church (1940).
23 Russell (1908).
24 Church (1940).
25 The early history of types and some of their early uses in the context of programming in the context of the lambda calculus has been discussed by Cardone and Hindley (2006).
26 Backus et al. (1956).
27 Perlis and Samelson (1958).
28 Martini (2016).
29 Naur (1978).
30 The history of COBOL has been documented by Sammet (1978).
31 IBM (1960). IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at 'IBM Copyright and trademark information' at www.ibm.com/legal/copytrade.shtml.
32 Sammet (1961).
33 Postley (1960), quoted by Ensmenger (2012).
34 Rochester (1960).
35 Postley (1960).
36 A review of those efforts has been written by Priestley (2011) and includes work on 'plexes' by Ross (1961), work by McCarthy et al. (1962) on list processing in Lisp, as well as the work that I discuss in detail in this section.
37 McCarthy (1961).
38 Also discussed by PROGRAMme (2022).
39 Dahl et al. (1972).
40 Lindsey (1996b).
41 van Wijngaarden et al. (1969).
42 First documented by Wirth (1971), but later historical recollections (Wirth, 1996) provide another valuable source.
43 Schuman and Jorrand (1970).
44 Hoare (1972).
45 McCarthy (1961).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

46 Astarte (2017); Mervyn Pragnell's underground reading group is further mentioned in various personal recollections, including Burstall (2000); Bornat (2009); Hodges (2001); McBurney (2009).
47 Astarte (2017).
48 Burstall (2000).
49 The history has been thoroughly documented by Astarte (2019).
50 Landin (1964).
51 Published as a proceedings paper two years later (Landin, 1966a).
52 Landin (1965a,b).
53 Landin (1966b).
54 Morris Jr (1969).
55 Project MAC (1967).
56 Morris Jr (1969).
57 Teitelman (1984).
58 Morris (1973b).
59 Parnas (1972).
60 Morris (1973a).
61 Liskov and Zilles (1974).
62 Reynolds (1974).
63 Lakatos (1976).
64 Kell (2014).
65 Scott (1993).
66 Milner (2003).
67 Milner (2003).
68 Scott (1993).
69 See Gordon (2000) for a first-hand account of the history of LCF.
70 See, for example, Milner and Weyhrauch (1972) and the general discussion in a later tutorial article, Milner (1979).
71 Milner (1979).
72 Presented by Milner (1978).
73 Morris Jr (1969).
74 The algorithm was first described by J. Roger Hindley in the context of combinatory logic in 1969 and it was independently invented by Robin Milner. It is often referred to as the Hindley–Milner type inference algorithm.
75 It would have been possible to execute ML through a more efficient compiled Lisp. The appeal of generating machine code over more sophisticated use of Lisp is perhaps another link to the hacker culture.
76 Some of the engineering work on ML has been discussed by Abadi et al. (2014).
77 car (1983).
78 car (1983).
79 Milner et al. (1990); the history written by MacQueen et al. (2020) documents the development of Standard ML.
80 Pierce (2002).
81 Talpin and Jouvelot (1994).
82 Tofte and Talpin (1997).
83 Clarke et al. (1998).
84 It is worth noting that the very phrase 'well-typed programs cannot go wrong' has often been paraphrased in subsequent research, for example as 'well-typed programs do not get stuck' (Wright and Felleisen, 1994) or as 'well-typed programs cannot be blamed' (Wadler and Findler, 2009). In this, it follows the more famous 'considered harmful' phrase by Dijkstra (1968).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

85 Hudak et al. (2007).
86 An exception that proves the rule is the paper by McBride (2002), which documents some of the tricks. It does, however, have a somewhat idiosyncratic, personal and lighthearted style that one might expect from a hacker culture of programming.
87 This has been gradually changing with recent developments that focus on extending Haskell with full support for dependent types.
88 For more information about the history and legacy of Automath, see the work by Dechesne and Nederpelt (2012). Automath is also documented by Laan (1997), who puts it in a wider context of the evolution of type theory in logic and mathematics.
89 Knuth (1973).
90 Harrison et al. (2014) documents the technical history of interactive theorem proving, including the different systems and their influences.
91 The fact that Java and Scala type systems are unsound, as shown by Amin and Tate (2016), is just one example where real implementation proved to be more complex than its formal models.
92 As mentioned earlier, this is much like the process of proofs and refutations described by Lakatos (1976) in the context of mathematics.
93 How work on interactive theorem proving influenced programming languages is not well documented, but Altenkirch et al. (2005) provides useful starting points.
94 Nordström et al. (1990).
95 Magnusson and Nordström (1993).
96 Cayenne was described by Augustsson (1998) and Dependent ML by Xi and Pfenning (1999).
97 Epigram by McBride and McKinna (2004), Idris by Brady (2017) and Agda by Norell (2007), which also has a well-documented interactive Emacs plugin Coquand et al. (2006).
98 Leroy (2009).
99 Paulson (1993).
100 Pitts (2011).
101 Stroustrup (1996).
102 Gabriel et al. (1990).
103 The first Microsoft product that featured automatic code-completion was Visual Basic 5.0. Unlike Visual C++, it relied on introspective capabilities of COM and was reportedly more robust. This suggests that the shift from reflective capabilities to types as the information source was more gradual.
104 For examples of this reasoning, see Rosenwasser (2018).
105 Cavanaugh (2018).
106 It remains to be seen whether the use of types in TypeScript can be reconciled with perspectives of other cultures. So far, most proposals coming from other cultures propose to 'fix unsoundness' by making the type system more complex, which goes against the TypeScript design principles (Vekris et al. (2016); Richards et al. (2015)). However, a 2015 article 'In Defense of Soundiness: A Manifesto' by Livshits et al. (2015), published in the *Communications of the ACM* suggests that the engineering culture approach to soundness is gaining acceptance in parts of the computer science community. The article notes that 'the dominant practice is one of treating soundness as an engineering choice' and elaborates on how to best make such trade-offs.
107 In an earlier paper (Petricek, 2015), I suggest this may be related to the 'meaning is use' approach to philosophy of language proposed by Wittgenstein.
108 Kennedy (1996).
109 I say counter-example with reference to Lakatos (1976), who documents how counter-examples in mathematics force mathematicians to revisit their definitions. The difference

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

here is that the counter-example comes from outside of the mathematical culture and so multi-cultural programming concepts like types are likely more susceptible to such changes.

110 This interpretation is inspired by the work of Latour (1987).

111 Using the metaphor introduced by Galison (1997), we can see types as entities that exist in the trading zone between (at least) the mathematical and engineering cultures of programming.

112 The most infamous case of such translation is the notion of a monad (Petricek, 2018).

113 'Anything goes' as Feyerabend (1975) says, but the history of types follows a more refined version of the slogan in that each culture has its own principles that it follows.

114 This view is close to the experimental philosophy of science advocated by Hacking (1983).

115 Martini (2016).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

6 Object-Oriented Programming

Teacher: Object-oriented programming is the dominant programming paradigm today, so I suppose we all have some intuitive idea about it. But it may be useful to make this intuition more explicit. Can someone give a quick summary of what the main concepts of object-oriented programming are?

Archimedes: I hope we will get a chance to talk about the various interesting past takes on object-oriented programming, but there are four main concepts that they all share. Those are dynamic lookup, abstraction, subtyping and inheritance.[1]

Diogenes: Sorry, but this is just too abstract for me! I thought object-oriented programming is about programming with objects and classes. Can you explain what you mean by all those terms?

Archimedes: Certainly! Dynamic lookup means that behaviour is determined at runtime, abstraction means that objects can hide their implementation details, subtyping means that you can use one object in place of another if it has all the required functionality and inheritance means that you can reuse object definitions to define new objects. Object-oriented programming based on classes is one way of implementing these.

Socrates: This is what the textbooks today say, but it is not right. Object-orientation is all about message sending. To build a system that can scale and evolve, it is not useful to constrain what the system is made of. This will need to change anyway as the system evolves. Objects and classes are the uninteresting aspect of object-oriented programming. We need to instead think about the communication within the system.

Pythagoras: Sorry, but I think that the concepts described by *Archimedes* are more comprehensible than this vague idea. They are also consistent with the origins of object-orientation. You can find all of these four concepts in some form in SIMULA 67, which was in many ways the first object-oriented language.

Socrates: SIMULA was great, but it was just a better old thing. It was an improvement over Algol, but it was based on the same basic old ideas. The authors of Smalltalk understood that you can see SIMULA as almost a new thing and started to look for the actual new thing.[2]

Archimedes: This is a nice lofty explanation, but it does not really tell me much. Smalltalk is different from SIMULA in that it is dynamically typed, but it also has objects and classes, so I do not think calling it a 'new thing' is justified...

Teacher: We should be careful about incommensurability in our debate. We know already that the two of you are looking at the problem of programming from a different perspective. To understand each other better, it would help if you, *Socrates*, could explain your thinking about Smalltalk from a broader perspective.

Socrates: Smalltalk was born as the software side of Dynabook, 'a personal computer for children of all ages'.[3] The aim was to make the software highly adaptable, which is the only way to build something that can accommodate the diverse ideas and needs of children and other users. Smalltalk's focus on

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

messaging, reflection and late binding is what makes it possible to adapt it to your needs. It becomes a meta-medium that you can turn into any specific medium you desire.[4]

Diogenes: I can certainly appreciate the self-modifiability of Smalltalk. It is quite amazing. But you do not need objects for that. You can get the same capabilities in Lisp. In a sense the UNIX and C ecosystem also gives you this capability!

Xenophon: There must be something about object-oriented programming that made it so widely popular. I suspect this has more to do with how object-oriented languages can model the real world than with their obscure technical characteristics.

Teacher: Let's focus on this question for a moment. How did object-oriented become so popular? Did it come as a major paradigm shift in how large software is built?

Archimedes: I think it was a much more gradual transition than *Socrates* suggests. Many were looking for better ways of creating large software systems in the 1970s and work on decomposing software systems or abstract data types[5] introduced many of the same ideas as object-oriented programming.

Pythagoras: This makes me think that the textbook definition of object-oriented programming that you started with may well be a historical accident rather than some essential characterisation. Wouldn't it be fair to say that abstract data types are a kind of object-oriented programming?

Archimedes: We could spend the whole remainder of this lecture discussing this question![6] Technically speaking, original implementations of abstract data types were not object-oriented, because they did not support all of the four features from my earlier definition, but I think they share the spirit of object-orientation.

Xenophon: In the late 1970s, what counts as object-oriented was far from settled. The Ada programming language, designed at the US Department of Defense, was seen as object-oriented, even if it did not have inheritance. And the designers had good reasons for seeing inheritance as a dangerous feature for high-integrity systems![7]

Pythagoras: This is an interesting historical observation! I do not think inheritance is necessarily a problem, but it may be pointing at an issue with many object-oriented systems where the behaviour of components is not defined precisely enough. This is because the object-oriented paradigm threw the baby out with the bathwater when it moved away from the mathematical thinking of Algol! Fortunately, the Eiffel language brought this back in the 1980s and made it possible to specify the behaviour of components precisely in the form of contracts.

Socrates: The problem is that 'software engineering' is an oxymoron. It is futile to try to impose rigorous structure on software development because we are still building primitive structures with primitive tools. The only way to build software is to keep the programming system as dynamic as possible, so that you can adapt in response to unexpected needs and changes.

Teacher: It seems that there is an interesting tension between whether we should keep the system more flexible and easy to adapt or more rigid and more precisely specified. Can these two directions be used to frame most of the important developments around object-oriented programming?

Diogenes: There is another notion of flexibility that we should not ignore. It is more concerned with the machine the program runs on. When you know what you want to do, the language needs to let you do that! This is exactly what you get in C++, which extended the C language with object-oriented programming in the 1980s. It is a language for those who take programming seriously and lets you write stable and efficient code that behaves in clearly defined ways. Of course, you need to know what you are doing, but if you do not know that, no tools can help you.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Xenophon: Hearing our discussion, I want to get back to the question that *Teacher* raised earlier, whether object-oriented programming was a major shift in how software is built. I think the answer is that it was not and anyone listening to our discussion can see why! We are talking about language features and technical details, but keep ignoring other aspects of software engineering such as requirements gathering or testing.

Diogenes: Well, we are talking about programming. I don't think your managerial methodologies for 'how to program if you cannot'[8] are relevant here.

Archimedes: To be fair, understanding what the customer needs should be programmer's concern and object-oriented programming does, in fact, help with that.

Xenophon: I'm curious to hear how...

Archimedes: One idea made possible by object-oriented programming is to use English language description of a program as the starting point for your design. If you analyse the description and look for nouns and verbs, nouns will become the classes in your design and verbs will become their operations.[9] This needs to be done more carefully, but it is a powerful guiding principle for object-oriented design.

Xenophon: This is a clever idea, but it still covers only the design and implementation phases of the software development lifecycle. You still need to embed object-oriented design within a more complete process.

Socrates: If you look at how people developed software for the early commercial Smalltalk systems in the 1980s, you will see that they often did not need this kind of heavyweight processes. They were able to leverage the dynamic nature of the system.

Xenophon: How did they manage to get anything to work then?

Socrates: A famous example is the work of Ward Cunningham and Kent Beck at Tektronix research labs. They would come up with an idea, quickly prototype it over a few days or a week, reflect on the prototype and then either spend another week on the project or pursue some other direction.

Xenophon: That sounds like an Agile development methodology to me. Now, I'm not personally a fan of those ideas, but I'm sure they are useful in a more exploratory environment like an industrial research lab. But still, they are processes that control the rest of the application lifecycle!

Archimedes: Mind you, the Agile manifesto was published in 2001 and we are talking about object-oriented programming and software engineering as it was done in the 1980s, so it is interesting that you are making this connection.

Socrates: There is definitely a connection. Many of the authors of the Agile manifesto knew Smalltalk. This includes Ward Cunningham, who I just talked about. It is not surprising that programmers used to the live editing and rapid feedback of Smalltalk systems would want those features in other environments. Methodologies like Extreme Programming are a way of keeping the Smalltalk spirit in languages like Java or C++.

Teacher: Looking at the origins of object-oriented programming was definitely revealing and it prepared the ground for my last question. I will intentionally keep this short. Has object-oriented programming succeeded?

Archimedes: Well, for me, the main idea is that of managing the complexity of software. This is what you find in the early work on decomposing software into components, abstract data types and, eventually, object-oriented programming. In this, I think object-orientation has been a success. The scale of what we are able to build today cannot be compared with the scale of software that we were able to build in the 1980s. And object-orientation played a key role in this step change.[10]

Xenophon: I would like to add that object-oriented programming also sparked a valuable change in the management of software development. We did not talk about this, but object-orientation led to the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

development of the Unified Modelling Language (UML) in the 1990s. UML was not concerned with just architecture and coding. It transformed the entire software development lifecycle through things like the Use Case diagrams.

Socrates: You can define success in different ways. Object-oriented programming succeeded in that it is everywhere. But I think it has not succeeded if we ask whether it achieved its original aims. It was about creating systems that are flexible, can be adapted by their users and qualitatively extend the notions of reading, writing, sharing and publishing of ideas.[11] In this sense, the vision behind object-oriented programming has failed to materialise.

Teacher: Do you have any hopes for the future?

Socrates: It is hard to see how this vision can come true under the current economic conditions, but I still have a hope. There are people who believe in the original vision and are working on making it happen. The Squeak and Lively Kernel projects are two fantastic examples of this![12]

Diogenes: It is remarkable how technically close are those systems to the original Smalltalk implementations from the 1970s. I wonder if the same characteristics can be achieved in other ways, or even in the context of our current computing infrastructure...[13]

6.1 The Computer Revolution Hasn't Happened Yet

In 1997, the object-oriented programming language Java that would soon dominate the industry was publicly available for just two years. It was already talked about so much to cause a reader of a professional magazine to complain about the magazine's contribution to 'all the Java hype'.[14] Meanwhile, C++ was 12 years old and one year away from the first ISO standardisation. The object-oriented programming paradigm was the industry standard and thousands of academics and practitioners regularly gathered at the annual conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).

The keynote speaker at the annual conference in 1997 in Atlanta was Alan Kay, the creator of the programming language Smalltalk that gave object-oriented programming its name. In 1997, Smalltalk was 25 years old, but it had become a very different thing from the Smalltalk of the 1970s that I talked about in Chapter 3. Smalltalk was now available in a range of commercial products with names like VisualSmalltalk Enterprise and the Smalltalk tools market was worth $56 million in 1994.[15] The programming language and environment may not have been so different from that of the 1970s, but its surrounding culture has changed. Whereas object-oriented programming in the Smalltalk of the 1970s was the product of the hacker and humanistic cultures, object-oriented programming in the industry-scale Smalltalk systems of the 1990s was a much more engineering and managerial undertaking.

The keynote delivered by Alan Kay highlights this cultural shift. An oft-quoted passage from the talk condemns much of 'what is actually being done in the world under the name of object-oriented programming (OOP)':

> I've been shown some very, very strange-looking pieces of code over the years by various people, including people in universities, that they have said is OOP code, and

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

> written in an OOP language – and actually, I made up the term object-oriented, and I can tell you I did not have C++ in mind.[16]

In a less frequently quoted follow-up passage, Kay makes it clear that he is not criticising just the C++ language, although the reference certainly elicits much laughter from his audience. Kay admits that he has 'many of the same feelings about Smalltalk'.

Alan Kay remains loyal to the ideals of the humanistic culture of programming. There are two aspects of his keynote that illustrate this well and highlight the cultural difference between the object-oriented programming of the 1970s and 1990s. First, the keynote includes an eclectic mix of references and metaphors, typical for the humanistic culture of programming. Kay uses parallels with molecular biology to suggest how complex systems can emerge. He makes references to the media theorist Marshall McLuhan and the philosopher Arthur Schopenhauer to discuss the adoption and evolution of object-oriented programming. Last but not least, he concludes with a story about a pipe organist E. Power Biggs. When asked how to play a piece of organ music written for some of the largest organs on a 'dinky little organ', Biggs says 'Just play it grand. Just play it grand.' Similarly, Kay wants to inspire his audience to play their systems grand.

Second, Kay's keynote is not about the history of Smalltalk, its successes or particular engineering issues. It is about finding new ways of thinking about computers and building systems that would scale 'by a factor of trillions'. This is not a mere rhetoric figure. Kay believes that we need to think extremely large scale in order to design systems of the future. He quotes one example where this happened. The ARPANET, which became the Internet, was built with the same set of beliefs in mind and was inspired by J. C. R. Licklider's vision of Intergalactic Computer Network. As Kay notes in his talk, the architecture of ARPANET enabled it to expand 'by about a factor of a hundred million' between 1969 and 1997. He encourages the audience to learn from this example. 'That is the metaphor we absolutely must apply to what we think are smaller things'.

Kay uses the issue of scale to point at the 'deepest thing' that he wants his audience to take away from his talk. It is that 'we don't know how to design systems yet'. This point is going very much against the belief, broadly accepted at the time, that object-oriented design is ultimately the right way of building complex software systems. Smalltalk has made its contribution to the search for a better answer, but it was not a particular object-oriented language design. 'Pretty much the only thing' that Kay is proud of is that Smalltalk 'has been so good at getting rid of previous versions of itself'. It enabled the basic iterative search for better answers, but when it went commercial, it ceased changing and stopped being used 'to bootstrap the next thing'.

To a member of the humanistic culture, imagining the future of computing, the keynote is a stimulating inspiration. To an engineer or a manager who thinks about building software systems over the next couple of years or even a decade, Alan Kay remains the same researcher described by Stewart Brand in the 1970s: brilliant, uninterested in conventional goals and with plenty of time for messing around.[17]

Although Smalltalk was not the first programming language that featured some form of object-oriented programming, Alan Kay's 1997 keynote is an appropriate opening

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

for this chapter for two reasons. First, although the term 'object' have been used by others even before Smalltalk, the notion of object-orientation as understood by Smalltalk largely ended up being the one that computer science students are taught today. Second, the keynote illustrates the cultural shifts around object-oriented programming that I will follow in this chapter. But before I get to Smalltalk and its contributions rooted in the humanistic culture, we should start with the mathematical origins of object-oriented programming in work on computer simulations.

6.2 A Language for Describing Discrete Event Systems

One theme that we will encounter repeatedly in this chapter is modelling of the real world. Proponents of programming languages that we now call object-oriented often claim that objects provide a natural way of representing real-world entities in a computer program.[18] This argument suggests a way of thinking about programming that is one step further from the idea of high-level programming languages developed in the 1960s. The first programming systems made it (somewhat) easier to enter instructions for the computer, while high-level programming languages made it possible to model computations. Object-oriented languages, or so it is claimed, make it possible to model not just computations, but the behaviour of the real world itself. The theme of modelling is present directly in the ancestor of object-oriented programming languages, SIMULA. The language, as the name suggests, was explicitly developed for creating computer simulations. The idea of modelling the real world was not a post hoc claim, but a design motivation.

SIMULA was created by Kristen Nygaard and Ole-Johan Dahl at the Norwegian Computing Center in Oslo in the 1960s. Already during his military service at the end of the 1940s, Nygaard carried out numerical computations for Norway's first nuclear reactor, but he later turned his attention to Monte Carlo methods and operations research. This often required creating computer simulations of physical systems that consisted of multiple entities or processes that interact and evolve over time. Such simulations were useful in multiple areas, including science, military research and optimisation of manufacturing processes and there was a growing interest in creating computer simulations throughout the 1950s. The usefulness of dedicated programming languages for creating simulations was only recognised with the turn of what Richard Nance[19] calls 'The Period of Search' into 'The Advent' around the year 1960. As pointed out by Nance, the developments of the early simulation programming languages 'took place concurrently without a clear progression of conceptual convergence'. SIMULA is no different, in that it was developed largely independently of other work in the area.[20]

Nygaard started working on a programming language for writing simulations in 1961. He developed the first ideas on how the programming language should look by the following year, when he described the project progress in a letter to a colleague:

> The status of the Simulation Language (Monte Carlo Compiler) is that I have rather clear ideas on how to describe queueing systems, and have developed concepts which I feel allow a reasonably easy description of large classes of situations. … The work …

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

> could not start before the language was fairly well developed, but this stage seems now to have been reached. The expert programmer who is interested in this part of the job will meet me tomorrow.[21]

The expert programmer that Nygaard was going to meet was Ole-John Dahl, who worked on an Algol-like programming language at the time and joined Nygaard at NCC in 1963. The two became long-term collaborators, working on SIMULA so enthusiastically that a new employee at NCC once worriedly reported that 'two men are fighting violently in front of the blackboard in the upstairs corridor'.[22] In retrospect, Nygaard regarded the letter as naively optimistic. Creating the first version of SIMULA took both serious technical effort and 'entrepreneurial maneuverings'.[23] Two years after writing his optimistic letter, Nygaard managed to secure support from the Univac, which was one of the eight major computer manufacturers of the era. In 1963, the implementation work on SIMULA could finally get started.

What is interesting for this book is the cultural background of the SIMULA programming language. Its origin is firmly rooted in what I refer to as the mathematical culture. Both Kristen Nygaard and Ole-John Dahl received master's degrees in mathematics. More importantly, the SIMULA programming language was based on Algol, a prominent programming language from the mathematical culture of programming that I discussed in Chapter 2. This may have partly been for practical reasons. Algol was popular at the time in Europe and Ole-John Dahl had experience with implementing an Algol-like language, but it was also an intentional design choice. 'The elegant and powerful concepts of this language appealed to Dahl and Nygaard and made it, in their opinion, the perfect match for SIMULA'.[24]

There are a number of other aspects of SIMULA design that align it with the mathematical culture of programming. In a retrospective history of the language, Nygaard and Dahl[25] describe their encounter with SIMSCRIPT, which included unsafe pointers, as a 'cultural clash' that made the design goals of SIMULA clear. It 'had to provide programming "security" to the same extent as Algol 60 itself: Any erroneous program must either be rejected by the compiler (preferably), or by runtime checks (if unavoidable), or its behaviour must be understandable by reasoning based entirely on the language semantics, independent of its implementation.'[26]

Dahl made the link to mathematics even more explicit in his talk 'Programming languages as tools for the formulation of concepts' presented at a Congress of Scandinavian Mathematicians in 1969.[27] He likens 'the art of programming digital computers' to 'the science of constructive mathematics' and argues that the computer plays the role of the set of primitives and axioms. Last but not least, Nygaard and Dahl can also be associated with the mathematical culture through personal connections and collaborations. One of the foremost members of the mathematical culture, Donald Knuth, wrote an introduction for their paper on SIMULA published in the *Communications of the ACM*,[28] starting 'a long and lasting friendship',[29] while Dahl wrote a chapter to a later book on *Structured programming*,[30] with another foremost member of the mathematical culture, C. A. R. Hoare.

The design of SIMULA evolved significantly over time. The two major versions were SIMULA I (originally called just SIMULA), developed between 1961 and 1965 and SIMULA 67, developed subsequently. However, even the original SIMULA I went

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

through a number of designs. I will look at just two of those here.[31] In the first stage, a simulation in the SIMULA language was modelled as a *system*, consisting of a fixed number of active *stations* and a variable number of passive *customers*. Those concepts were also the keywords of the new syntactic structures added to the underlying Algol programming language. In this model, the code associated with stations received customers, checked and updated their state and, eventually, routed them to another station.

Experience with the first stage version prompted Nygaard and Dahl to look for a more general model and they eventually unified the active stations and passive customers under a single notion. The revised version of the language[32] replaces stations and customers with a single language construct called *activity* declaration. The language is still far from being what we might now call a general purpose object-oriented language, but we can already find traces of important object-oriented ideas that would take shape in SIMULA 67:

> The description of a process is called an activity declaration. The concept of an 'activity', which is a class of processes described by the same declaration, is distinguished from the concept of a 'process', which is one dynamic instance of an activity declaration.[33]

An activity declaration is what would later become a *class*. It consists of some processing logic that can be long-running and simulates the activity alongside with attributes that represent parameters and the state of the activity. The description in the paper mentions the term class, but only in its ordinary English-language sense. The paper does not yet talk about objects, but it already uses the term instance. At this stage, processes (i.e., instances of an activity) could be assigned to a special kind of variables called *elements*. For implementing any interesting simulations, processes also need to be able to access the attributes of other processes. In SIMULA I, this was known as 'remote accessing'.

Remote accessing, being an equivalent of member access in modern programming languages, seems surprisingly cumbersome in retrospect. A reference to a process in SIMULA I does not keep track of what activity the process is an instance of. In modern terms, it is untyped. But the authors of SIMULA I wanted to guarantee safety and so remote accessing had to be done using a construct that provides code for all possible kinds of activities:

```
inspect ⟨element expression⟩  when A1 do S1
                              ...
                              when An do Sn
                              otherwise S
```

As the authors note in their historical reflection on SIMULA, try to compute $X.A + Y.A$ using the `inspect` statement![34] Another notable technical idea that made SIMULA I possible was a change in the treatment of code blocks from Algol. Blocks in Algol represent sequences of statements and were a key innovation that enabled structured programming. Blocks correspond to some part of program logic. They can be nested and form the body of loops or conditionals. In Algol, blocks merely structure the program code. The execution entered a block, executed all the statements in the block

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

and then left the block, whose state could then be discarded. However, in SIMULA, newly instantiated activities needed to outlive the call, because they could represent a process that has been started and should continue running. This caused a major headache to the SIMULA designers. There was no hope of implementing SIMULA as an Algol preprocessor as it would involve 'breaking up the blocks, making local variables nonlocal, etc'.[35] Instead, Dahl set out to make a non-trivial change to Algol, replacing the stack-based treatment of blocks with one based on dynamic memory allocation. The allocated blocks also needed to be deallocated when they were no longer used. For this, the authors used a reference counting mechanism with a 'last resort' garbage collector.[36]

SIMULA I included many of the ideas of later object-oriented languages, but in a form that was closely linked to the problem of programming simulations. We can see traces of object-oriented programming in the language, but most would not call it object-oriented yet. But in the summer of 1965, Nygaard and Dahl started thinking and talking about 'new, improved SIMULA',[37] which would eventually became SIMULA 67. Although the language was defined in 1967, it took several more years for the first compilers to appear. In a history of SIMULA written later by the authors, we can see both how the notion of a 'programming language' has became a stand-alone entity (as discussed in Chapter 2) and a reflection on the assumptions of the mathematical culture by someone who went through the tedious process of actually implementing a programming language:

> As language designers we often feel that people believe that one has 'succeeded' and that a language 'exists' as soon as it is defined, and that the subsequent implementation is a mere triviality … In fact it is when the language is reasonably well described and thus the implementation task defined, that the main, tedious, and frustrating part of the struggle for a language's existence really starts.[38]

The design of SIMULA 67 aimed to address a number of shortcomings of SIMULA I. Many of those resonate with what a programmer today may think about SIMULA I. The special variables referring to processes (elements and also sets) were cumbersome and the remote accessing mechanism was hard to use. The authors encountered many 'processes' that did not have their own logic, but represented data with multiple associated procedures and they also felt the need for a mechanism for sharing common data and actions between related processes.

An important inspiration for the new design of SIMULA came from a fortuitously timed meeting. In September 1966, Ole-John Dahl gave a lecture on SIMULA I at a Summer School on programming languages organised by the NATO Advanced Study Institute in Villard-de-Lans. Among the five speakers was also C. A. R. Hoare, whose lecture focused on the topic of 'Record Handling'. Hoare was interested in finding a way of adding a mechanism for representing rich data structures to Algol 60, an issue that I discussed in Chapter 5.

Hoare talks about records representing objects with several attributes. He points out that 'objects of the real world are often conveniently classified into a number of mutually exclusive classes, and each class is named by some collective noun, such as

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

"person" or "bankloan"'.[39] Hoare uses the same model for records, which are 'grouped into mutually exclusive record classes' that determine the fields of the record. For example, a record class declaration for the aforementioned 'bankloan' looks as follows:

```
record class bankloan(integer loan number,principal;
                      real rate of interest)
```

Hoare discussed a number of other noteworthy ideas. He introduced a reference type, such as `reference(bankloan)`, representing a reference to a record of a given class. This makes it possible to access members of a record safely without runtime checks, and without the cumbersome remote access construct of SIMULA. His proposal also makes it possible to split the values of a record into multiple mutually exclusive *subclasses*.

Hoare and Dahl likely discussed the issue of record handling at length at the NATO Summer School. The revised version of Hoare's lecture notes, published after the event, includes an extensive report on how activities of SIMULA I (which have attributes that can be accessed via remote accessing) can be seen as records.[40] Nygaard and Dahl were likely inspired by the meeting and they have reported that 'much time was spent during the autumn of 1966 in trying to adapt Hoare's record class construct to meet our requirements, without success'.[41]

Eventually, they succeeded and produced a design for SIMULA 67 that combined the programming model of SIMULA I with many of the ideas from Hoare's work on record handling. Nygaard and Dahl adopted the term 'class' for the activities and also replaced the term 'process' with a more general term 'object'. They introduced a reference type akin to that of Hoare, making it possible to easily access attributes of an object of a known class. The trickiest construct to support was that of subclasses. This was eventually implemented using a mechanism called prefixing, where a class can be prefixed with another class, inheriting its attributes and procedures. Nygaard and Dahl do not themselves use the term 'inheriting' and, instead, write about 'building the properties of the prefix into the objects defined by the new class declaration'.[42] Figure 6.1 offers a glimpse of what SIMULA 67 programs looked like, illustrating both prefixing and virtual methods, which were a 'last minute extension' to the language.[43]

Looking at SIMULA 67, a present-day programmer would recognise much of the terminology and many of the constructs of a general-purpose object-oriented programming language. But the flexibility of SIMULA had not generally been recognised at the time. In an influential book *Programming Languages: History and Fundamentals*, written by the programming pioneer and co-author of COBOL, Jean Sammet, SIMULA appears in the section IX.3.1.7 in a subsection on specialised languages for discrete simulation. A couple of languages from this group are included, but with some reluctance. Sammet explains the reasons for her relatively limited account of the topic. First, 'usage [of simulation programming languages] is unique and presently does not appear to represent or supply much carry-over into other fields'.[44] Second, 'unless an individual has actually tried to simulate a system or a process, he is not apt to appreciate or completely understand the importance or subtleties of the facilities being provided'.[45]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

```
Process class machine (inq. outq. own crane);
        ref(head)inq. outq;ref(crane)own crane;
        virtual: procedure service:
        begin
        ref (order) served; real setup time;
        procedure service;
            hold (setup time + served.processing
            time);
work:   if inq. empty then passivate
        else begin served: - inq. first;
        served. out; service; served. into(outq);
        activate own crane delay message time end;
        go to work
        end machine;
```

Figure 6.1 Incomplete SIMULA 67 illustration, showing 'how a simple "machine" in a job shop simulation may be described'.
The code defines a class `machine`, using the generic `Process` class as its prefix. 'Since the procedures [sic] `service` is specified as a "virtual quantity" it may be redefined in subclasses to "machines"'.[46]
(Source: Dahl et al. (1968))

It took some time, but the power of the constructs introduced in SIMULA was eventually widely recognised and they have certainly carried over into other fields. The key role in this was, however, played by a quite different culture of programming than the mathematical one from which SIMULA has originated.

6.3 Past Ghosts and Present Spectres

It is now time to slowly return to Smalltalk, the programming language from the opening of this chapter. According to the folk history of object-oriented programming, while SIMULA was the first object-oriented programming language, Smalltalk was the programming language that gave the programming style its name and made it popular. I will question some of those oversimplifications in the next section, but there is no doubt that Smalltalk has played a major role in shaping the object-oriented programming paradigm.

As is often the case with work that has roots in the humanistic and hacker cultures, it is difficult to disentangle exactly the network of influences that has shaped Smalltalk. The early writing about the system is sparse and focuses more on its humanistic aims of building 'a personal computer for children of all ages', than on the formal academic design influences. My starting point are thus the first-person retrospectives, written by Alan Kay and Dan Ingalls,[47] which document the cultural and social context and also technical influences on Smalltalk. According to Kay, the design of Smalltalk was influenced, through his preliminary work on the FLEX machine, by a number

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

of systems that were 'almost a new thing' from the 1960s, including SIMULA, but also Ivan Sutherland's Sketchpad, discussed in Chapter 3, and two systems that Kay encountered as a programmer in the US Air Force.[48] Looking at the publications on the FLEX machine suggests a number of other less widely accepted influences.

I already talked about the ARPA community and Xerox PARC that provided the context from which Smalltalk emerged in Chapter 3. Kay became a part of the community when he joined the graduate school at the University of Utah. While the computing world outside the ARPA community has been dominated by large batch-processing computers and some early terminal-based time-sharing systems, Kay lived in a world where computers were used by children (Papert's Logo system), their role was 'augmenting human intellect' (Douglas Engelbart's and Vannevar Bush's visions), they had interactive graphical displays (Sutherland's Sketchpad project) and were used to build live collaborative environments with video-conferencing (Douglas Engelbart's oN-Line System). Imagining a personal computer 'which could be owned by everyone and could have the power to handle virtually all of its owner's information-related needs'[49] still required a stretch of the imagination, but it was at least conceivable.

The first source of ideas that Kay mentions is a system for transporting files at the US Air Force for the Burroughs 220 computer. The data was stored in a self-descriptive format where the actual records were accompanied with procedures for reading and manipulating the data, that is, using the grouping of data and procedures that is typical for object-oriented programming. Although likely unbeknown to Kay, the Air Force was also an unexpected source of thinking on personal computers. A report on the future computer technology, presented in 1965, envisioned what almost seems as the military version of the Dynabook (Figure 6.2). Fittingly, the envisioned computer was the size of a cigarette package rather than the size of a paper notebook:

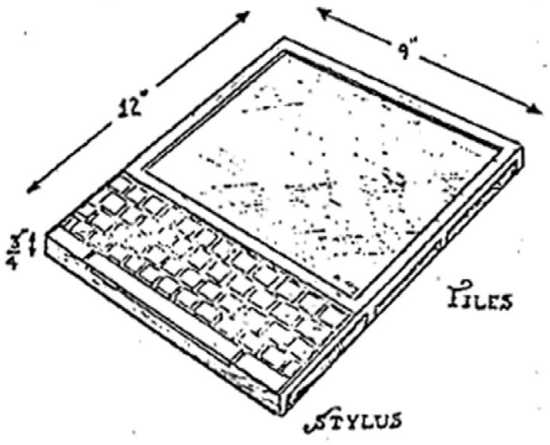


Figure 6.2 A sketch of 'Dynabook' from 'A personal computer for children of all ages'.[50] (Source: Courtesy of Alan Kay. Reproduced with permission.)

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

> [We] will be able to build a machine the size of a cigarette package. As in civilian life, all communications will have become digital …. Everyone can have private communications by using a small personal computer for scrambling. The individual may … use communication satellites …. Computers with wideband data links can provide graphical communications, allowing widely separated Air Force elements to discuss plans, documents, maps, etc.[51]

The other, more direct, influences on the Smalltalk design came through systems that Alan Kay encountered during his graduate studies. According to his reflections, he was handed the Sketchpad thesis to read by his advisor Dave Evans when he joined. Soon after, he was given 'a pile of tapes and listings' to get to work. The code was supposedly Algol, but it turned out to be 'doctored to make a language called Simula'.[52] The two encounters certainly influenced Kay's later work, but his writing on the FLEX system suggests that the influence was gradual.

Kay started his studies in 1966 and submitted a master's thesis titled 'FLEX: A Flexible Extendable Language'[53] in 1968 before completing his PhD thesis titled 'The Reactive Engine'[54] a year later. His work focused on the programming side of a small computer developed in the group that Kay and his collaborators called FLEX. The aim of the FLEX project was to build a small computer that an individual could own and use. It was inspired by LINC, which I briefly mentioned in Chapter 3 and is sometimes treated as the first 'personal computer'.[55] In his master's thesis, Kay focuses mostly on the programming language, which is 'intended to be a simple yet powerful and comprehensive notation to express computer-oriented algorithms'. The FLEX system 'follows the traditions set by Algol 60 and several generations of EULER', the latter being an extension of Algol 60 with support for data types developed by Niklaus Wirth. In many ways, the thesis follows the style of the mathematical culture of programming. The language description is written in the style of the language definition of Algol that I examined in Chapter 2. It focuses on the extensible syntax of the language, such as the ability to define new symbolic operators. Notably, the master's thesis does not yet talk about graphical user interface and only reports 'several partially successful attempts' to 'combine [FLEX] compilers with a number of the operating text editors at the University of Utah'. The brief list of references also does not yet include Sketchpad or SIMULA.

In 1968, Kay participated in a graduate student meeting of the ARPA community, saw a demo of the tablet-controlled graphical flowchart programming system GRAIL and visited the Logo group at MIT. He 'realized that the FLEX interface was all wrong'[56] and tried to find ways of fitting some of what he saw into the tiny FLEX machine. His PhD thesis reflects the new ideas. It pays much more attention to the interaction with the user and positions the work using Kay's typical broad-stroke references. In this case, the references are to the 'basic framework of the Indo-European languages' and the philosophical tradition where 'the world has objects which endure over time'.[57] Finally, the thesis also sketches a graphical interface for interacting with the program (Figure 6.3) through a tablet. In the 'Implementations to date' section, the thesis however, does acknowledge failures ('due to the inadequacies of Algol as a

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

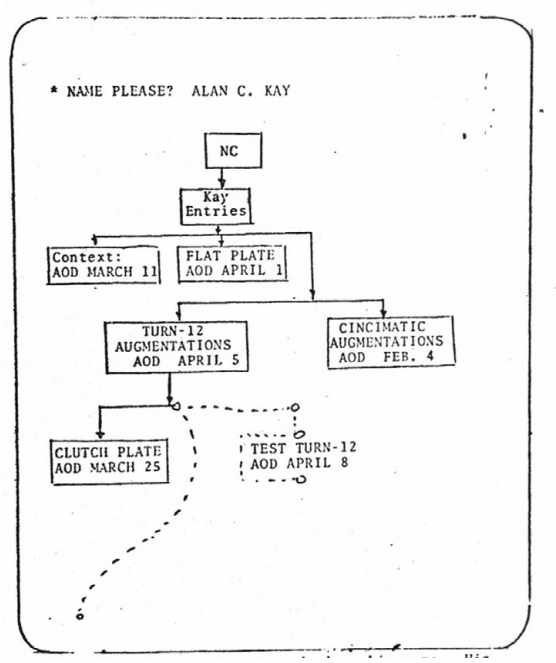


Figure 6.3 'A Sample Program' from the FLEX thesis.[58] The diagram shows the 'previous context' that the user sees after giving their name.
(Source: Courtesy of Alan Kay. Reproduced with permission.)

real-time language') in integrating the FLEX compilers with interactive editors and a graphics system developed at Utah.

The description of the FLEX system does not talk about objects or object-oriented programming, but it exhibits some of the characteristics that will later define the paradigm. The key abstraction is a *property list*, which is associated with a name through a binding. The notion has likely been inspired by property lists in LISP 1.5, where each atomic symbol has an associated property list that stores basic information related to the symbol, such as its name and its raw value or machine language code. The point of departure in FLEX is that property symbols are not primarily internal. They are the mechanism through which the programmer would use the system. The properties stored in the property list can be used in a number of ways, including the familiar dot-notation, although in a prefix rather than postfix form. In FLEX, `age.Bob` refers to the property 'age' of 'Bob' as in the following example, which represents an interactive session (ending with the `type` command, which instructs the system to print the result of an expression):[59]

```
Bob ← ':age, :weight, :height.[if age = 40 then sleep]';
Bob(40, 175, 70);

type age.Bob;
```

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Names with associated property lists, which are the basis of FLEX, are in many ways similar to later objects. The properties associated with a name may include values, but also procedures that may access the associated properties directly without quantification (as when accessing the `age` property in the above example).

Looking at the references in Kay's PhD thesis suggests that his doctoral work was influenced by the mathematical culture of programming at least as much, if not more than, by the humanistic culture. He references Church's lambda calculus and also the work of Peter Landin (discussed in the context of types in Chapter 5) on lambda calculus models of Algol.[60] By contrast, references to SIMULA and Sketchpad are sparse. SIMULA is mentioned in the context of record handling (and only the 1966 paper describing SIMULA I) and the only citation of Sutherland's work is a reference to a specific clipping mechanism.

6.4 A New Medium for Communication

After completing his doctoral studies, Alan Kay joined the Xerox PARC research centre. As I already wrote in Chapter 3, the centre brought together a non-conformist group with strong backgrounds in the hacker, engineering and humanistic cultures, many of whom had earlier links with the ARPA community or Douglas Engelbart's group. The work on Smalltalk and what would later become object-oriented programming was happening amidst the backdrop of interesting hardware developments as well as political struggles against the Xerox management.[61] Kay himself devoted as much attention, if not more, to thinking about small computers for kids and their use in education, as to the Smalltalk language. In this chapter, I focus on the language, although its development is inseparable from this broader technical and humanistic context.

In 1971, Kay created the first version of a new programming language that was, by then, called Smalltalk. The account from Kay's later reflections suggests that his starting point was an interest in building a flexible programming language 'that could handle a variety of styles of programming'[62] in a completely uniform way without built-in language constructs that are treated in a special way. One such construct is the conditional. In Lisp, a conditional is not an ordinary call and needs to be evaluated in a special way. For example, consider an expression `if` (`>` `a` 10) (`setq` `a` 0) that sets the variable `a` to 0 if it is greater than 10. This requires that the second element of the list (`setq` `a` 0) is evaluated only when the first element (`>` `a` 10) is true, rather than being evaluated automatically along with all other arguments as is done for ordinary function calls.[63]

Supporting a variety of programming styles in FLEX required turning different parts of the program into entities that could be passed around and evaluated in different contexts, leading to the idea of objects and messages. Despite making yet another step in the direction of object-oriented programming, the motivating problem would fit well into the ongoing, more academic and mathematical, research on extensible programming languages that I will return to in the next section. At this point, Kay was arguably still solving an 'old problem', but using what was gradually becoming a 'new thing'.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The key ideas of what later became known as object-oriented programming were crystallised in Smalltalk-72. Because of the political fallout from the 1972 Spacewar article by Stewart Brand in *Rolling Stone* magazine, the group was prohibited from publishing until 1975[64] and so a historical account of Smalltalk-72 has to rely on the handbook published in 1976[65] and a handful of internal documents from the era.

Unlike in the earlier publications about FLEX that emphasise many other aspects of the language design, the Smalltalk-72 handbook has a clear focus on objects. This is clear in chapter III, which follows a number of example-oriented walkthroughs and provides 'a direct discussion of the basic Smalltalk concepts: classes, instances, and message sending and receiving'. In particular, the subsequent paragraph very closely captures the design principles described in Kay's retrospective paper (Figure 6.4):

> Every entity in Smalltalk's world is called an object. Objects can remember things and communicate with each other by sending and receiving messages. Every object belongs to a class (which is also an object). The class handles all communication (receiving a message and possibly producing a reply) for every object which belongs to it.[66]

The evolution of the FLEX system (and to some extent Smalltalk-71) into Smalltalk-72 is in many ways similar to the evolution of SIMULA I into SIMULA 67. It is gradual and progresses via reasonable responses to specific design challenges, but it

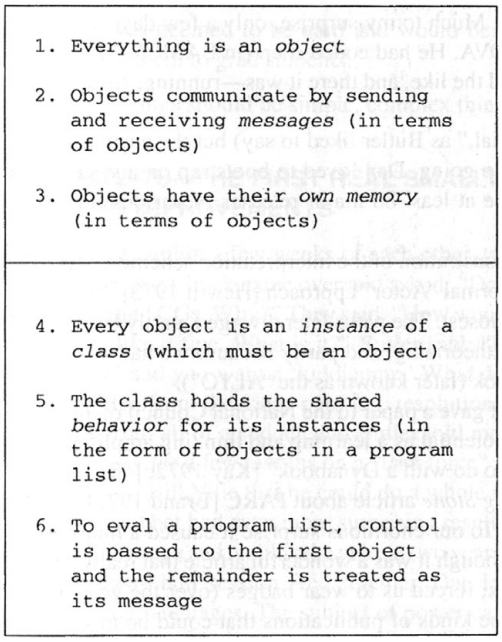


Figure 6.4 Six Smalltalk-72 principles, as reconstructed in Kay's account of the history of Smalltalk:[67] 'The first three principles are what objects "are about" – how they are seen and used from "the outside". These did not require any modification over the years. The last three – objects from the inside – were tinkered with in every version of Smalltalk (and in subsequent OOP designs).'
(Source: Used with permission of ACM Special Interest Group on Programming Languages,[68] permission conveyed through Copyright Clearance Center, Inc.)

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

significantly changes the narrative to one that is much closer to the modern notion of object-oriented programming.[69] It is likely that many of the concepts emerged independently in the two environments, but the terminology of Smalltalk-72 may have been influenced by SIMULA 67. An acknowledgements document, written in March 1973, 'shortly after Smalltalk-72 started working',[70] cites many influences on the language design including Logo and Lisp, but also 'SIMULAs ('65 and '67)'. The latter is recognised for its 'epistemology that allowed a class to have any number of parallel instantiations'. The FLEX language then 'took the SIMULA ideas (discarding most of the AGOLishness [sic]), moved "type" from a variable onto the objects', providing the basis of the model fully embraced in Smalltalk-72.

The fact that a class is also an object in Smalltalk-72 is one of the cornerstones of the Smalltalk language and the treatment of classes provides two nice examples of cultural roots of the language. The first example is the attempt to unify as many concepts in the language as possible. We can see this unification in the treatment of classes, but also in Kay's attempt to unify multiple programming styles in a single language in the FLEX system. Those designs were less motivated by engineering concerns and more by aesthetic aims that are typical for the humanistic (and sometimes mathematical) culture. As Key recalls, 'much of the pondering' during Smalltalk-71 design 'had to do with trying to understand what "beautiful" might mean …'.[71] The second example is the treatment of (to use an anachronistic term) class inheritance. Smalltalk-72 did not adopt the prefixing mechanism of SIMULA 67. Kay recalls that he 'just decided to leave inheritance out as a feature in Smalltalk-72, knowing that we could simulate it back using Smalltalk's Lisp-like flexibility'.[72] In line with the hacker tradition of Lisp, the Smalltalk design gives the programmer enough control to do things in the way they want. Whereas the focus of SIMULA was on security and use of types to prevent potential errors, Smalltalk exposes the internals of the system and lets its users tinker with them.

Another glimpse of the affinity to the hacker culture of programming can be found in the internal Smalltalk memos. The document 'Summary of Smalltalk Message Forms and Intentions',[73] which was prepared to accompany an internal class on Smalltalk, provides a high-level overview of many of the messages that the students may need. Despite being a documentation for a Smalltalk tutorial, the document is focused on source code. It follows the formatting of Smalltalk class definitions, using the actual (non-standard) symbols used in Smalltalk-72 (Figure 6.5). Moreover, in places where the documentation is not complete, the memo makes reference to concrete people from the team (following the individualism of the hacker culture) and the source code of the core Smalltalk library as, for example, in 'Diana and SYSDEFS can tell you the rest'.

The opening comment of the referenced SYSDEFS file[74] is also written in the style of the HAKMEM memo, which I used to illustrate the style of writing of the hacker culture in Chapter 1. An annotated version of the file opens with:

```
HEREWITH A SOMEWHAT WHIMSICAL ANNOTATED VERSION OF
SYSDEFS. ANNOTATIONS ARE IN ITALICS. WHILE IT IS HOPED
THAT THIS WILL
```

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

to turtle z: x y dir pen ink

isnew ⇒ (...)	creates a new turtle
◁ goto ⇒ (▤▤...)	from the current position, traces to x y position: ▤▤. Returns distance.
◁ penup ⇒ (...)	picks the 'pen' up. No ink will flow
◁ pendown ⇒ (...)	puts the 'pen' down. No ink will flow
◁ home ⇒ (...)	take the turtle 'home'; currently x y position 256 256.
◁ ink ⇒ (◁ white ⇒ (...) ◁ black ⇒ (...)	On the CSL graphics color display, ink can be any of 256 colors.

Figure 6.5 A part of the documentation for the 'turtle' class.
(Source: Kay (1974); custom symbols replaced with nearest Unicode alternative)

```
PROVIDE SOME ELUCIDATION OF THE CODE ESCAPES, OBSCURITIES
WILL
NO DOUBT PERSIST. THE ANNOTATIONS ARE INTENDED TO BE BUT
DIMLY
LIGHTED MARKERS ON THE ROAD TO TRUE ILLUMINATION.
```

Smalltalk-72 was developed for the experimental Alto computer built at Xerox PARC at the same time and the combination was used internally at Xerox for a wide range of projects. The Smalltalk team now included Dan Ingalls, who did much of the engineering to make Smalltalk actually work and Adele Goldberg who joined Xerox PARC in 1973 because of her interest in computer education. She built a range of class materials based on Smalltalk, despite later recalling that she initially found Smalltalk-72 an 'extremely odd language' and thought that 'this wasn't going to be teachable to anybody'.[75] The language was also used by various other Xerox PARC members for projects ranging from the Logo turtle library, text editor and music synthesiser to the Pygmalion programming system discussed in Chapter 3.

As Dan Ingalls later noted, much of the team's 'gratification was tinged with a feeling that things were missing or not quite right'.[76] The lack of support for inheritance led to duplication of code, classes and activation records, which were not real objects, the system was not very efficient and would easily run out of space. The dissatisfaction together with interest in building something fresh eventually resulted in a new, cleaner, better engineered and more efficient implementation that became known as Smalltalk-74. One interesting, but invisible, technical innovation in Smalltalk-74 was the OOZE (Object Oriented Zoned Environment). The OOZE was a virtual memory system that made it possible to create more objects than would fit into the limited memory available on the Alto. The basic idea of storing objects that are not actively needed to disk sounds simple, but required ingenious implementation tricks.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

For the history of object-oriented programming the OOZE is interesting for another reason. It offers evidence of the use of the term 'object-oriented' by the Smalltalk team. There is no reference to 'object-oriented' in the publications on SIMULA and Smalltalk is often credited with using the term for the first time, not just by Kay himself. For example, in a paper that tried to characterise object-oriented programming in 1982,[77] Tim Rentsch argued that 'the explicit awareness of the idea – including the term "object oriented" – came from the Smalltalk effort'. It seems likely that the term was in use at Xerox PARC during the 'publication blackout' (1972–75) as there are quite a few publications appearing in the second half of the 1970s that use the term, without feeling the need to define it. That said, the definition of what is object-oriented may not be as straightforward as the popular history suggests and I will return to this later.

6.5 Let's (Not) Burn Our Disk Packs

The two personal retrospectives on the history of Smalltalk, written by Alan Kay and Dan Ingalls and published in the Proceedings of the History of Programming Languages conference,[78] offer two perspectives on the developments after Smalltalk-74. They also illustrate how different cultures of programming joined forces to give birth to the Smalltalk programming language, but then led their proponents down separate paths.

Alan Kay's thinking about Smalltalk is closely aligned with the interests of the humanistic culture. He cared about building a personal computer that could be used by children and saw programming as a basic way of interacting with the machine, much like Seymour Papert treated programming as 'communicating with a living native speaker of mathematics'.[79] Dan Ingalls who led the implementation of Smalltalk invented many of the tricks that made it actually work and his writing on the history of Smalltalk often talks about interesting technical challenges that motivated certain developments. Adele Goldberg joined the team with interest in education, but later became manager of the group in 1979, leading the public release of Smalltalk, and she went on to establish a company, ParcPlace Systems, to develop Smalltalk into a commercial product.

Both Kay and Ingalls express some unease about the state of Smalltalk after Smalltalk-74 became widely used. For Kay, the unease is about losing the original focus. By the end of 1975, he felt 'that "the Dynabook for children" idea was slowly dimming out – or perhaps starting to be overwhelmed by professional needs'. He organised a three-day offsite in January 1976 where he tried to convince the group to 'burn our disk packs' and start afresh. Kay wanted a new machine with a new system. He 'did not see how [object-oriented programming] by itself was going to solve our end-user problems' and he started designing a new language and system called NoteTaker. For Ingalls, the unease about Smalltalk-74 was mainly technical. The 'brewing forces and frustrations came together', during a weekend at the beach in August 1976, 'to suggest an entirely new language and architecture that promised true

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

commercial speed while preserving [the] object-oriented style'.[80] The work resulted in a new language implementation called Smalltalk-76, which became the basis of all 'modern Smalltalks'.

The design of Smalltalk-76 is technically interesting in a number of ways, but most are outside the scope of this book. One remarkable aspect is that it was written in Smalltalk-74, which is one of the cases where Smalltalk 'has been so good at getting rid of previous versions of itself', a quality that Alan Kay praised in the keynote speech discussed in the opening of this chapter. (This required writing a parser and a compiler first in Smalltalk-74, testing it and then transliterating it into the new syntax and structure of Smalltalk-76.) It is also worth looking more closely at a paper about Smalltalk-76 that Dan Ingalls wrote and presented at the ACM POPL symposium in 1978.[81] Although a few publications about Smalltalk started to appear at the time, those were mostly focused on the humanistic visions behind Smalltalk (such as the 'Personal Dynamic Media'[82] paper). Ingalls' paper is, by contrast, a detailed technical account of the language and appeared in a programming language research venue.

The paper keeps some of the humanistic motivations behind Smalltalk. The introduction opens with the purpose of Smalltalk, which is 'to support children of all ages in the world of information'. It argues that the 'SIMULA notion of class and instance is an outstanding metaphor for information structure' and adds the notion of message sending as a similarly powerful metaphor for processing. Following the humanistic introduction, the paper focuses on engineering notions. It describes Smalltalk as 'object oriented rather than function oriented' and talks about modularity, which is that 'no part of a complex system should depend on the internal details of any other part'. The 'class is the natural unit of modularity, as it describes all the external messages understood by its instances'. Smalltalk-76 also (finally) implements inheritance and adopts the terms 'subclass' and 'inheritance'.

The notions of object-oriented programming, classes and inheritance that are familiar to programmers today can easily be found in Ingalls' paper. However, it is perhaps telling that those took final shape only after a redesign and rewrite of the language that was motivated by a mix of hacker and engineering motivations and has played down some of the humanistic visions. Alan Kay himself was, for example, 'not completely thrilled with'[83] inheritance in Smalltalk-76 and as his 1997 keynote makes clear, valued the Smalltalk ability to evolve into new systems more than any specific version of the language design.

6.6 What Is an Object-Oriented Programming Language?

The object-oriented programming paradigm, as understood at the end of the 1970s, was the result of yet another meeting of multiple cultures of programming. It brought together the mathematical culture, from which the work on SIMULA originated, with the humanistic motivations of Alan Kay and a mix of hacker and engineering skills and interests of the other members of the Smalltalk group. The influences were gradual in that the ideas that emerged in one context were later adopted and adapted in

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

another context. As we will see, object-oriented programming also kept its multi-faceted nature after the 1970s. Languages influenced by SIMULA and the mathematical culture emphasise safety and reasoning about programs, whereas languages arising from the hacker and humanistic cultures of Smalltalk emphasise the flexibility of the paradigm. The engineering culture, positioned somewhere in the middle, will interpret object-oriented programming as a pragmatic tool for building software and will look for ways of using it effectively. Before I move to a more recent history of object-oriented programming, it is worth revisiting the origins of some of the terms and ideas of the paradigm and look at parallel ideas that were overshadowed by the later rise of object-oriented programming.

The terms 'object' and 'class' were first used in SIMULA 67 and were motivated by everyday uses of the terms in English. Nygaard and Dahl talked about real-world 'objects' classified into 'classes'. They used the term 'class' in passing even earlier when talking about activity declarations in SIMULA I. The term 'inheritance' has a more interesting history. It was not used in SIMULA, which referred to the mechanism as 'prefixing,' but has already adopted the term 'subclass' for the new class definition created through prefixing. In the context of Smalltalk, the term 'inheritance' is certainly used in Smalltalk-76,[84] but when exactly how it entered the group's vocabulary is hard to trace.

One possible source is the Knowledge Representation Language (KRL) developed by Daniel G. Bobrow (in another research group in Xerox PARC) and Terry Winograd (at Stanford, but consulting for Xerox PARC). The KRL originated in artificial intelligence (AI) research as a language for describing knowledge that could be used by AI tools for reasoning about the represented concepts, but it was surprisingly closely related to the emerging object-oriented programming paradigm. It structured the knowledge in terms of 'entities with associated descriptions and procedures'[85] and cited SIMULA and early Smalltalk as influences for the idea of procedural attachment. Interestingly for my analysis of terminology, KRL provides 'a notion of subclass which allows objects to inherit procedural as well as declarative properties'[86] and KRL also refers to earlier work on 'property inheritance' in the field of AI research. In his retrospective, Kay acknowledges interactions with the KRL designers too, and so it is likely that this particular term came to object-oriented programming through yet another indirect influence.

The last term I want to revisit is 'object-oriented programming'. Not because the history of the terminology itself is particularly noteworthy, but because it will lead us to related ideas developing in parallel with those in Smalltalk. Kay himself suggested[87] that he use the term 'object-oriented programming' to describe his work as early as 1967 during his time at Utah, but the term was not used in publications about the FLEX machine or Smalltalk until 1978. The term was certainly in use by the Smalltalk team earlier, as evidenced by the naming of the OOZE (Object Oriented Zoned Environment). However, what exactly 'object-oriented' stands for was by no means settled by the end of the 1970s. Another contender for the term originated in programming languages built around the idea of abstract data types.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The idea of abstract data types, which I discussed in Chapter 5, is to define entities from which programs are composed by the operations that they support. The concept was introduced by Barbara Liskov and Stephen Zilles alongside a programming language Clu that implemented the idea. Their definition from 1974[88] uses a language that follows much of the object-oriented terminology. According to Liskov and Zilles, 'an abstract data type defines a class of abstract objects which is completely characterised by the operations available on those objects'. They did not call this style of programming object-oriented in the original paper, but a subsequent report co-authored by Liskov from 1976[89] uses the term 'object-oriented languages' to refer to SIMULA 67, Clu and Alphard (another language implementing the notion of an abstract data type). A 1979 PhD thesis supervised by Liskov on 'Machine Architecture to Support Object-Oriented Language' likewise uses the term object-oriented languages for SIMULA and Clu, adding Lisp but still omitting perhaps not yet universally known Smalltalk.

In an email exchange on the topic of object-oriented programming,[90] Kay pointed out that SIMULA has catalysed two main paths. The first being the work on abstract data types. According to Kay, this line of work is a 'better old thing' in that it preserves the programming style with 'bindings and assignments'.[91] The second path is exemplified by Smalltalk, which replaces data and operations with messaging. In an oft-cited (but historically inaccurate) quote, Kay expressed his regrets that he 'coined the term "objects" for this topic because it gets many people to focus on the lesser idea. The big idea is "messaging".'[92] He has also frequently criticised modern object-oriented programming languages for following the former rather than the latter path. It is thus a curious historical accident that the folk history of object-oriented programming today refers so strongly to Smalltalk. The term object-oriented programming has been equally used to talk about programming languages centred around the notion of abstract data type, which are arguably closer predecessors to many modern object-oriented programming languages.

The difference between the two ways of thinking that trace back to SIMULA has not been widely acknowledged in the computer science community and it remains an issue worth discussing. A 2009 essay by William R. Cook[93] opens by pointing out that the typical answer to the question 'What is the relationship between objects and abstract data types?' is 'objects are a kind of abstract data type'. According to Cook's detailed analysis 'an abstract data type is a structure that implements a new type by hiding the representation of the type and supplying operations to manipulate its values'. Whereas 'an object is a value exporting a procedural interface to data or behaviour'. The difference is somewhat subtle, but interesting. Abstract data types implement some precisely specified mathematical interface. This also makes it possible to prove properties of programs that use abstract data types through the interface. Objects are 'autognostic' meaning that they know only themselves. Interaction with any other object is a matter of sending a message, but it is entirely up to the other object to decide how to handle such message.

A close look at Alan Kay's early work on the FLEX machine and early Smalltalk reveals another interesting aspect of the history of programming. Although Kay himself always presents those as early steps towards his vision of Dynabook and

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

object-oriented programming, much of the work on those systems could equally be seen as contribution to the work on 'extensible programming languages'. This now largely forgotten research area emerged at the end of the 1960s and aimed to develop programming languages where 'one can tailor a language facility to very closely mirror the unit data, unit operations, and ways of writing that are "natural" to some problem area'.[94] The research theme covered topics such as definitions of new data types, custom operators, new ways of composing primitives and customisable syntax. What may seem like a hotchpotch of programming language features to a present-day computer scientist, combining work on basic language semantics, syntax and libraries, formed a relatively coherent research field for a number of years.

Early work on FLEX and early Smalltalk fits very well with the theme. The very name, FLEX, is an acronym for 'FLexible EXtendable'. The extensibility applies both to the basic vocabulary of the language and to its syntax. As noted in the FLEX description, 'the system does not distinguish between its own entries and the user's. Therefore, a verb (standing for a subroutine, perhaps) created by the user has as much power, scope and usability as the system-defined verbs.'[95] The syntax can be extended by defining new custom operators, but also by modifying the system itself 'via the compiler-compiler contained in the language'.[96] When designing Smalltalk-71, Kay found it 'annoying that the surface beauty of Lisp was marred by some of its key parts having to be introduced as "special forms" rather than as its supposed universal building block of functions'.[97] He was dissatisfied with the fact that constructs like conditionals could not be fully defined by the user. He later recalls finding an answer in a PhD thesis 'Control structures for programming languages' written by David Fisher at Carnegie-Mellon University in 1970. In his thesis, Fisher cites numerous other works on extensible languages and notes that applications of the work to extensible languages is 'one obvious area for further research'.[98]

Kay attended the Extensible Languages Symposium in 1969, but later recalled that it was, in his view, 'a religious war of unimplemented poorly thought out ideas'.[99] However, he did find inspiration in one of the presented (actually implemented) systems where 'every procedure in the system defined its own syntax'. Kay's reinterpretation of the idea in FLEX (also present in Smalltalk-72 but abandoned in later Smalltalks) was to see 'each object as a syntax directed interpreter of messages sent to it'.[100] In retrospect, Kay was probably fortunate not to attach his work to the area of 'extensible languages' as the area largely lost traction by the mid 1970s. In a somewhat ironic closing of the research agenda, some felt that it has already achieved its goals, while others felt that its goals were 'a bit over-ambitious'.[101]

The birth of object-oriented programming is entangled in curious ways with other work, happening at the time, for which some have used the term 'very-high-level' programming languages. The research relates to the framing that I mentioned when discussing SIMULA. After machine languages, assembly languages and high-level programming languages like FORTRAN and Algol, programmers started exploring ways of bringing the programming languages even closer to the problem domain. SIMULA was (initially) specifically designed to model simulations, but later found use in general purpose modelling of the real world. In extensible languages, the idea

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

was to adapt the language to the problem domain. Similarly, abstract data types aimed to 'ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area'.[102] Out of these three, object-oriented programming is the best known survivor. The notion of abstract data type has became primarily a theoretical idea of the mathematical culture. Extensible languages have been mostly forgotten as a concept, even if many of their technical innovations continue to exist, some as commonplace (data types) and some as sophisticated (macros) programming language features.

6.7 Structured Programming of the 1980s

In the late 1970s and 1980s, object-oriented programming became a household name in the computing industry. There were new object-oriented languages, object-oriented design methodologies and Intel even introduced a processor, Intel iAPX 432, that provided hardware support for object-oriented programming. This rising popularity of the term in various contexts also inspired critical reflections on its meaning:

> What is object oriented programming? My guess is that object oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favour of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.[103]

There was much truth in this prediction. I will not attempt to trace how exactly object-oriented programming gained popularity and widespread adoption. There were likely multiple contributing factors. The perceived industry crisis made developers eager to look for new approaches, object-oriented programming had some early success stories and objects were closely linked to the emerging graphical user interfaces. The technology magazine *Byte* contributed to the popularisation of Smalltalk and object-oriented ideas in 1981, when it dedicated an entire issue of the magazine to the system.[104]

An interesting aspect of the rise of object-oriented programming in the 1980s for my analysis is how the different cultures of programming influenced the subsequent developments. In a way, the developments mirror those of early programming languages that I followed in Chapter 2, where multiple cultures contributed to the idea of a programming language but then developed it in different directions. In the case of object-oriented programming, the origins of the idea also bring together multiple cultures of programming. And similarly, the later developments of object-oriented programming languages take the idea in multiple diverging directions.

The idea of cultures of programming resonates with a distinction that has been made by Peter Wegner during a panel at the ECOOP conference and that was elaborated by Steve Cook in a later conference report.[105] Wegner distinguishes between the Scandinavian and American school of object-orientation. In Scandinavia, 'research seems to be very concerned with putting object-oriented programming on a theoretical

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

footing by making analogies with physics'. This was the case with SIMULA, but multiple languages that followed in the SIMULA tradition adopted the same view. By contrast, 'the American school of thought does not appeal to any fundamental theory … the primary criteria for thinking about object-oriented programming in the USA are pragmatic. What the American school wants to know is, does it work, and if it does, then how do we make it better?'

The so-called American school closely matches the engineering culture of programming, but finding a match for the Scandinavian school is harder. It exhibits a combination of mathematical striving for rigorous foundations with humanistic concerns about the real world. The contrasting views are not merely an interesting historical fact, but they keep shaping discussion about object-oriented programming today. Two of the object-oriented pioneers of the Scandinavian school, Ole Lehrmann Madsen and Birger Møller-Pedersen, wrote an essay in 2022 in response to the 'increasing criticism of object-oriented programming [in] recent years' arguing that there 'are indeed issues with object-orientation as practised by mainstream' and that 'the primary reason for these issues is that reuse is considered the main advantage of object-orientation at the expense of modelling'.[106]

To the dichotomy between Scandinavian and American schools that later became well known, Cook also added the British school. This disappeared from later discussions about object-oriented programming, but it perfectly matches with the mathematical culture as characterised in this book. For the British school, 'the ability to treat programs as mathematical entities is of paramount importance'. Cook points out that, because of its focus on side-effects and sharing, object-oriented programming 'has had something of a hard ride with British researchers'.[107] Perhaps because of the 'hard ride' with mathematically oriented researchers, there are only a few early contributions to object-oriented programming rooted solely in the mathematical culture of programming. The earliest attempts to use formal methods were motivated by the desire to explain how exactly certain practical aspects of object-orientation worked. In the mid 1980s, Luca Cardelli used models based on the lambda calculus to study data abstraction[108] and Cardelli and William R. Cook both used formal programming language semantics to model inheritance.[109] A couple of years later, the book 'A theory of objects'[110] provided a treatment of object-oriented languages that used the standard methods of the mathematical culture and followed the style of the ML paradigm, which I discussed in Chapter 5. However, the mathematical culture was also more directly shaping object-oriented programming in combination with other ways of thinking.

In Scandinavia, Kristen Nygaard kept working on object-oriented programming, collaborating with colleagues from Norway and Denmark, through a series of projects that eventually resulted in the language BETA. Their focus remained on modelling, but more in the sense of modelling real-world concepts than in the sense of numerical simulations. Consequently, BETA was seen both as a programming language and as a modelling language. The designers even had a rule that a language construct had to be useful both as a modelling tool and as a programming feature.[111] The modelling work of the Scandinavian school was, by no means, a dry mathematical activity. Nygaard himself worked with trade unions in Norway and was interested in co-designing sys-

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

tems directly with their users. This way of thinking fits with the humanistic culture of programming and it later resulted in the participatory design methodology that also influenced Christiane Floyd and her work on software engineering, which I talked about in Chapter 4. Nevertheless, many aspects of the BETA programming language design associate it closely with the mathematical culture. One example is the mathematical focus on simplicity and formal elegance. The central feature of the BETA design is a single abstraction mechanism, the pattern, which unifies records, classes, functions and methods. This comes at a usability cost in that the uniform syntax of BETA is 'often claimed [to be] awkward'.[112]

The Smalltalk language also continued to be influential. It did so in multiple ways. After some initial difficulties, it was reimplemented for a new generation of computers and was adopted commercially. I will return to this thread later in this chapter. However, its original ideas also inspired (and continue to inspire) new designs and developments. A programming language that illustrates this influence well is Self. Developed at Xerox PARC by David Ungar and Randall Smith in the mid 1980s, the language aimed to 'improve upon Smalltalk by increasing both expressive power and simplicity, while obtaining a more concrete feel'.[113] Last but not least, the original creators of Smalltalk recovered and open-sourced an implementation of Smalltalk-80, 'finally doing, in September of 1996, what [they] had failed to do in 1980'.[114] The new version, called Squeak, enabled a new wave of interest in Smalltalk in the 2000s, including both research around programming systems in the original humanistic and hacker traditions, as well as a more engineering-oriented development around Pharo, a fork of Squeak created in 2008.

6.8 The Power of Simplicity

Like BETA, the Self programming language also set out to reduce the number of different concepts needed in object-oriented programming. One aspect of this was eliminating variables. Object state was accessed by sending messages to 'self', a design feature that gave the language its name. Another aspect was the use of prototypes instead of classes. In SIMULA and Smalltalk, objects are created as instances of classes, introducing a distinction between a description of objects and objects themselves. Prototypes eliminate this distinction. An object in Self describes its own behaviour, but can refer to another object as a prototype for shared behaviour.

The idea of prototypes in Self relates to its other stated goal, which is a 'more concrete feel'. The language was born when graphical user interfaces were becoming the norm. Smalltalk adopted graphical interface for programming in the form of a class browser, but Self imagined the next step. The designers 'wanted the programming environment's graphical display of an object to be the object for the programmer'. That is, the entities on the screen would be the concrete objects themselves. The idea was implemented in the graphical user interface later called UI1 in 1988/89. The interface represents objects as nested boxes (reminiscent of the Boxer project discussed in Chapter 3) and the prototype-based design of the language made it possible to design

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

an interface where there was 'nothing on the screen (except for pop-up menus) that was not an object'.[115] Because there is no distinction between objects and classes, the prototype of an object is just another object on the screen. Although UI1 achieved the goal of 'concrete' representation of objects, it did not help users build graphical interfaces with elements such as buttons or shapes. This was addressed in a redesigned version of the interface, UI2. Here, the thing displayed on the screen is not a box representing the internals of the object, but a graphical representation of the object called 'morph', a term derived from a Greek word referring to 'physical form'.[116] Morphs could take the form of buttons, shapes and other visual entities that could form a graphical application. As the authors put it later 'every visual element in Self, from the largest window to the smallest triangle is a directly manipulable object'.[117] The visual elements are Self objects and some can be directly modified through interacting with them. To inspect and modify the objects, the user also has the option to display an 'outliner' (accessible through a pop-up menu), which creates a representation of the object similar to that of UI1.

The fact that Self focused on concrete graphical representation of objects and, at the same time, adopted a variant of object-oriented programming based on prototypes is not a coincidence. The same combination existed in other systems that influenced Self. The direct predecessor was ThingLab, developed at Xerox PARC by Alan Borning. ThingLab was a graphical 'simulation laboratory' where users could create graphical objects on screen and specify their behaviour using constraints. ThingLab was itself inspired by the even earlier Sketchpad, which I talked about in Chapter 3. ThingLab used the idea of prototypes in its internal representation of objects, but Borning also envisioned using the mechanism in an object-oriented programming language in 1986, at the time when the Self design was starting, likely directly influencing its design.[118]

The innovative technical characteristics of Self, such as the prototype-based object-oriented model and concrete user interface, can easily overshadow the humanistic vision of the system. This was, perhaps, not emphasised as strongly as in the case of Smalltalk, but a later article by the Self designers makes the vision behind the system explicit:

> [Past papers about Self] have not completely articulated an important part of the work: our shared vision of what programming should be. This vision focuses on the overall experience of working with a programming system, and is perhaps as much a feeling thing as it is an intellectual thing.[119]

The authors explained that 'the aspects of [programmer] experience which are beyond the domain of pure logic' have traditionally been separated from the logic of the programming language. They are the 'domain of the programming environment'. However, just like in Smalltalk, the programmer experience in Self was at least as important as the logic of the language. According to the vision of the authors, 'the Self programmer lives and acts in a consistent and malleable world'. This later reflection makes it clear that Self follows Smalltalk in its allegiance to the humanistic culture of programming. The interesting technical innovations are the logical consequences of a new grand vision (enabled by graphical displays) about what programming should be.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The malleability of the world implies the ability of the user to adapt the environment to their needs and is, to some degree, in contrast with the design requirements typical for other cultures. The malleability does not make it possible to build safeguards required by the engineering culture and it makes the program open to modification, complicating mathematical reasoning about it.[120]

Self has suffered the same curse as many other systems rooted in the humanistic culture. It became influential through its technical contributions, but its humanistic vision was lost along the way. Self is known for its innovative graphical interface, sophisticated implementation techniques and prototype-based object-oriented programming model, but systems that leverage those do not necessarily subscribe to Self's vision of programming. JavaScript would be the prime example. It adopted both the prototype-based object-oriented programming model and also virtual machine implementation techniques that were pioneered in Self. Yet, its links to the humanistic culture of programming are only minimal. The more immediate path to a broader adoption of object-oriented programming led instead through work rooted in the engineering culture.

6.9 Making Programming Enjoyable for the Serious Programmer

Multiple programming languages that emerged in the late 1970s and the 1980s in the USA were developed with the aim of tackling the rising complexity and cost of software development. They paralleled the developments in the field of software engineering, which I talked about in Chapter 4 and that were the reaction to the same perceived industry crisis.[121] To a large extent, the object-oriented languages rooted in the engineering culture followed the approach later identified as the 'American school'. The Clu programming language developed by Barbara Liskov at MIT had academic origins and served as a vehicle for research on data abstraction and abstract data types. It brought together mathematical thinking of academic computer science and managerial thinking about the best development methodologies with the engineering focus on developing specific programming abstractions to support the practical task of programming. As discussed earlier, Clu and abstract data types can now be seen as an approach competing with object-oriented programming, but the authors themselves saw Clu as an object-oriented language. In a 1993 retrospective,[122] Liskov argues that 'Clu is an object-oriented language in the sense that it focuses attention on the properties of data objects and encourages programs to be developed by considering abstract properties of data'. However, she also acknowledges the difference from other object-oriented languages, especially the fact that Clu did not have support for inheritance (a feature she 'would probably try to include' today).

A similar approach to object-orientation can be found in the Ada language, which was developed at the US Department of Defense (DoD) in the late 1970s.[123] Its objectives were to reduce the costs of software development at the DoD and increase the reliability of the developed software. The effort incorporated both managerial and engineering techniques. On the one hand, its funders hoped to reduce costs by having a single language for the entire DoD, reducing training costs and enabling reuse. On the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

other hand, the language implemented numerous features that can help avoid errors, including strong type system, packages and exceptions. Although Ada added further support for object-orientation in a version standardised in 1995, the initial version has retrospectively been also called object-oriented. It was 'designed to support and encourage object-oriented design'[124] through packages. Yet, at the same time, the designers were cautious about a principle that later became a key aspect of object-orientation:

> During the eighties the use of the 'object-oriented' buzzword changed; one now implies 'object-oriented programming' features such as 'inheritance', which are even dangerous in high-integrity systems.[125]

By the turn of the 1980s, the object-oriented 'buzzword' not only gained a more specific meaning, but it also became recognised as a de facto approach to developing software. The programming language that first tried to combine the original object-oriented ideas with the focus on rigorous software engineering was Eiffel. The language was created in 1986 by Bertrand Meyer who had a combination of academic and industry background. Mirroring the background of its designer, Eiffel also had dual origins. It was conceived while Meyer was at UC Santa Barbara, but was developed by a commercial company Interactive Software Engineering that Meyer later established.

The Eiffel language embodies 'the belief that it is possible to treat [the task of software construction] as a serious engineering enterprise'. Meyer did not see his work as creating just a new programming language. The aims of the language 'lead to a new culture of software development, focusing on the reuse of industrial-grade components'.[126] The allegiance to the engineering culture is reflected in the very name of the language and is a reference to Gustave Eiffel and the Eiffel tower. Unlike many software systems of the 1970s, the 1880s tower was built on time and within budget, using a small number of simple principles. The association between a programming language and a new culture is not unique to Eiffel. We saw the same association earlier, when talking about the Logo programming language in Chapter 3. Logo was likewise envisioned as a computer culture, but one aimed at learning and thinking about computers.

The focus on reusability and the development of reusable components became recognised as another tenet of the American school of object-orientation. It also shaped the design of Eiffel in a number of ways. In fact, Meyer claimed that the language 'is nothing else than these principles taken to their full consequences'. Eiffel included many language features that computer scientists associate with good engineering practices and that emerged from the work on software engineering discussed in Chapter 4. This includes structured exception handling, a static type system that eliminates the `null` value and automatic memory management.

In one notable way, Eiffel embraced ideas that were previously the domain of the mathematical culture of programming. A key part of its *design by contract* methodology was a close integration between program code and its formal specification in the form of assertions. To illustrate this, consider the following example, which defines a

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

deferred feature (abstract method in a more common terminology today) that puts a value at the *i*th location of a collection:[127]

```
put_i_th(v : like first; i : INTEGER) is
  -- Put item v at i-th position.
  require
    index_large_enough : i >= 1;
    index_small_enough : i <= count;
  deferred
  ensure
    not empty
end
```

The key parts of the code sample are the `require` and `ensure` clauses. The former specifies the pre-conditions of the operation. It can only be performed if the argument `i` is between 1 and the collection size represented by `count`. The latter specifies a post-condition, which is that the collection is not empty after the value is set. The idea of pre-conditions and post-conditions appeared in the mathematical culture in 1969 in the form of Hoare triples, written as $\{P\}c\{Q\}$. As we saw in Chapter 2, they could be used to prove properties of programs. With Eiffel, the idea takes a new form. Pre-conditions and post-conditions now become a part of the program, are checked during program execution, and their role is to help programmers create 'industrial-grade components' that have a precise specification and behave accordingly. With Eiffel, the pre-conditions and post-conditions thus shift from a mere mathematical tool into an engineering entity, much like proof assistants turned proofs about programs from an external mathematical entity into technical artifacts.

Whereas Eiffel took the engineering approach to object-oriented programming and enriched it with ideas from the mathematical culture, the most influential object-oriented programming language of the 1980s took the engineering approach and added it to a language born from the hacker culture. The language I'm referring to is C++.

The development of C++ very closely reflects the personal experience of its author, Bjarne Stroustrup. During his PhD in Cambridge, Stroustrup used the SIMULA programming language and was 'impressed by the way the concepts of the language helped [him] think about the problems in [his] application'. This was due to the object-oriented concept of a class, which allowed him 'to map [his] application concepts into the language constructs in a direct way', making his code more readable than in any other language.[128] Unfortunately, the SIMULA implementation was inefficient and Stroustrup eventually reimplemented his system in BCPL, a predecessor of the C language. Stroustrup later attributed the inefficiency of SIMULA to a number of its features, including runtime type checking and garbage collection. It is perhaps not a surprise that when Stroustrup left Cambridge and decided 'never again to attack a problem with [unsuitable] tools',[129] he opted to build 'C with Classes' as the tool to use for his next major project. The new language was a 'medium success' and it served as the first step towards a more ambitious project, which was a 'cleaned-up and extended successor to C with Classes'.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

As documented by Stroustrup later, 'C++ was designed to provide Simula's facilities for program organisation together with C's efficiency and flexibility for systems programming'.[130] Stroustrup saw object-orientation as a useful tool for organising large programs, but he was not interested in building a language where everything was an object or where computation was done solely by sending messages. He considered 'computation … a problem solved by C' and had no intention to change the basic model (later remarking that 'Had I wanted an imitation Smalltalk, I would have built a much better imitation. Smalltalk is the best Smalltalk around'). Today, the C++ language is most directly associated with the engineering culture and is seen as a language suitable for systems programming and building large, reliable and efficient systems. In 2018, the US National Academy of Engineering awarded Bjarne Stroustrup the Draper Prize, given for 'the advancement of engineering', joining the ranks of engineers who developed the integrated circuits, turbojet engines or rechargeable batteries. (The list of recipients also includes Alan Kay, although not for his work on Smalltalk, but as part of a team that created Xerox Alto, the 'first practical networked personal computer'.)

In contrast to languages like Eiffel, there is remarkably little direct acknowledgement of engineering concerns or visions in the works that document the history and design of C++. The initial language design certainly shows some preference for reliable engineering, including the use of static typing for the object-oriented programming model and 'systematically eliminat[ing] the need to use [low-level] features except where they are essential'. C++ later evolved according to pragmatic engineering principles based on the needs and constraints of its real-world users. However, the early design shows many more traits of what I associate with the hacker culture of programming. This includes both technical and social aspects. C++ emphasises the efficiency of C, access to low-level operations and ability to deliberately circumvent the type system through unchecked type conversions. The early development process was also closer to what we might find in the hacker culture. In 1986, Stroustrup could still write that 'there never was a C++ paper design; …. There never was a "C++ project" either, or a "C++ design committee".'[131] Ironically, soon after this was written, the process to standardise the language design started and, today, C++ is documented in an ISO C++ standard, whose evolution is governed by the ISO/IEC JTC1/SC22/WG21 committee and its 23 or so sub-groups.

The later thinking of Bjarne Stroustrup also aligns the C++ language more closely with the engineering culture of programming. One example can be glimpsed from a talk that he gave in 1986, when he was invited to present at the conference of the Association of Simula Users in Stockholm. The talk, later published as a paper 'What is Object-Oriented Programming?'[132] is a reflection on the question that I attempted to examine earlier. Stroustrup criticised the tendency to call almost any language object-oriented. ('Could there somewhere be proponents of object-oriented programming in Fortran and Cobol? I think there must be.') and argues that the 'basic support a programmer needs to write object-oriented programs consists of a class mechanism with inheritance'. The feature that the designers of Ada saw as optional and dangerous is now a central feature required for object-orientation. The conclusions of the paper

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

relate object-orientation to data abstraction, an issue that Alan Kay would likely view as worrying about a 'better old thing':

> Object-oriented programming is programming using inheritance. Data abstraction is programming using user defined types. With few exceptions, object-oriented programming can and ought to be a superset of data abstraction.

The history of C++ suggests that the allegiance to a particular culture of programming may not necessarily be fully determined when a new technical development starts. In its early days, C++ was perhaps closer to the hacker culture, but its later evolution attached it firmly to the engineering culture. This development fits with the principle identified by Bruno Latour. To paraphrase: 'The fate of programming languages is in the hands of its later users.'[133] The early technical developments leading to Smalltalk may also fit this scheme as the initial work on the FLEX machine could well be linked to the more mathematical and engineering research on extensible programming languages.

6.10 The Enterprise Age of Visual Smalltalk

In the computer science community, Smalltalk is perhaps best remembered for its pioneering days in the 1970s when it emerged and evolved at Xerox PARC, developing the ideas that would later be seen as the 'pure' object-oriented programming paradigm. But there is also a later era of 'commercial Smalltalk' in the 1980s and 1990s, which is less well known, but equally interesting and perhaps even equally influential. For this era to start, Smalltalk first had to be released from its ivory tower at Xerox PARC. This was not so easy.

First, Smalltalk was developed for a series of workstations created at Xerox PARC, starting with the Xerox Alto. At the start of the 1970s, those were impressive machines, showing that personal computing has the potential to become a reality. At the end of the 1970s, the Xerox PARC workstations were still technologically impressive, but they were also inaccessible and expensive when compared to the technologically basic, but low-cost and increasingly popular microcomputers such as Apple II. Smalltalk utilised the unique hardware capabilities of the Xerox workstations to run efficiently, because much of the virtual machine could be implemented directly using the microcode instructions of the processor.[134] It also used its own character set that included a '•' (bullet) operator for indexing and '←' (left arrow) for assignment. This made releasing Smalltalk for other computers practically tricky.

Second, Smalltalk was not very well publicly documented. After the end of the Xerox publication blackout in 1975, it was described in a couple of academic papers and Alan Kay gave a number of talks featuring Smalltalk. One notable talk was at the banquet of the second West Coast Computer Faire in San Jose, incidentally also the event where the Apple II microcomputer was presented a year earlier. But many who saw Kay's futuristic talks about Smalltalk were confused and left wondering if it actually existed.[135]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Last, but not least, Smalltalk still existed in an industrial research lab that had to decide what it wanted to do with it. At the end of the 1970s, Adele Goldberg became the manager of the laboratory developing Smalltalk as Alan Kay gradually left Xerox PARC after his failed attempt to convince the team to start afresh. Goldberg had been an active member of the ACM and believed all the research done over the last decade or so should be published. With the help of her manager, Bert Sutherland, she convinced Xerox that 'nobody in the company was interested in Smalltalk',[136] meaning that they were free to publish all the research, as long as they did the necessary work. The strategy that Goldberg settled on was to offer access to Smalltalk to a limited number of selected companies (including Hewlett-Packard, DEC, Tektronix, Intel and later Apple) in exchange for their assistance with the publication. The companies were asked for feedback on a book on Smalltalk design and implementation that Goldberg's group would produce and for a contribution to an experience report documenting their attempts to re-implement a basic Smalltalk system.

The initial four companies were carefully selected. The Smalltalk virtual machine was designed to work well on the Xerox workstations and Xerox selected companies that, or so they thought, were capable of producing a similar workstation. However, none of the companies attempted to do that. Instead, they tried running Smalltalk on conventional machines available at the time and all got 'really dismal performance'.[137] The one company out of the four that eventually succeeded and became the epicentre for commercial Smalltalk activity was Tektronix. In the 1970s, the company was a major electronics manufacturer, known mainly as the producer of oscilloscopes. This may seem an unusual background for a computer producer, but Tektronix had a lot of experience with graphical displays and increasingly complex processing units, making it an excellent fit for a Smalltalk system. In 1981, Tektronix also shifted its focus and started developing a personal workstation internally called Magnolia.

Allen Wirfs-Brock and Paul McCullough, who created the initial inefficient Smalltalk virtual machine implementation at Tektronix, transferred to the company's industrial research lab, Tek Labs, and tried to get Smalltalk to run again. This time, they did not directly follow the reference implementation documented in the Smalltalk 'blue book'[138], which they helped to edit. In many ways, the reference implementation followed the idealistic elegant visions behind Smalltalk, such as the principle that 'everything is an object'. So, for example, when sending a message (calling a method) the implementation created a context for the call, which was also an object. However, this was almost never accessed as an ordinary object from the program. The idealistic representation is useful for debugging tools or more advanced programming tricks, but not during ordinary program execution. Wirfs-Brock and McCullough took an engineering-oriented approach, favouring efficiency over uniformity and simplicity. In their new virtual machine, a 'tentative context' was not an object and it only became one if it was accessed as an object. Alongside other similar pragmatic implementation choices, this gave Tektronix a reasonably efficient Smalltalk implementation for the Magnolia.

The new Smalltalk implementation first became available to some 60 researchers at Tek Labs. One of its first users was Ward Cunningham, who also became the most

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

prominent evangelist for Smalltalk at Tek Labs, helping others build Smalltalk projects ranging from mathematical software and expert systems to CAD design and even 3D graphics systems.[139] Many of the new Smalltalk projects were showcased at an internal Tek Labs science fair, convincing the company to turn the system into a product. Aligning with the hype at the time, Tektronix went on to release a series of workstations, starting with Tek 4404, marketed as 'Artificial Intelligence System'. Some 10 years later, commercial Smalltalk users who 'knew that the workstation division [of Tektronix] was really making Smalltalk machines' referred to those systems as 'the first integrated Smalltalk environments, disguised by Tektronix as AI workstations'.[140]

The activity around the Magnolia implementation of Smalltalk at Tek Labs serves well to illustrate the ongoing transition to commercial era of Smalltalk. The system still encouraged a creative exploration of new ideas, but most of the focus has now shifted to building new computer applications and the methodology of object-oriented software design. The humanistic visions of Smalltalk as a medium for augmenting human intellect and a tool through which children would learn became gradually replaced with Smalltalk as a powerful object-oriented software development system. Similarly, concerns about the elegance and simplicity of its object-oriented design principles were replaced by concerns about implementation efficiency and development methodologies.

The new engineering era of Smalltalk was in many ways as fruitful as the initial humanistic era. At Tek Labs, Ward Cunningham was joined by Kent Beck, who had recently graduated from the University of Oregon. The two would form a close working relationship where 'Ward would come up with an idea, and then Kent would go off and implement enough of it to be able to demo the idea'.[141] Their collaboration was the starting point for later developments, including Extreme Programming (XP) and Test-Driven Development (TDD), which I talked about in Chapter 4. Cunningham and Beck also got interested in what would later be called 'design patterns'. In software development, those would only become influential later, but the investigation inspired Cunningham to develop the idea of a wiki. The growing number of people learning and using Smalltalk also got the Smalltalk advocates at Tek Labs to reflect on how to structure object-oriented programs. This led to the birth of new development methodologies, starting with the Responsibility-Driven Design technique created by Rebecca Wirfs-Brock, which I will return to later in this chapter.

Tektronix is a good example of a commercial Smalltalk provider, particularly because of the number of influential people and ideas that emerged from its Tek Labs, but it was not the only company developing a commercial Smalltalk system in the 1980s. In the late 1980s, Goldberg started a company ParcPlace Systems, where she was joined by L. Peter Deutsch, a Lisp hacker we encountered in Chapter 3, who came up with his own way of making Smalltalk run efficiently. Another company, Digitalk, 'delivered what few thought possible' and built a system that ran on a low-cost 8086 DOS machine around the mid 1980s. Smalltalk inspired other commercial object-oriented programming systems, most notably the Objective-C language that added Smalltalk-style objects to C in 1984 and was later adopted by Apple.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

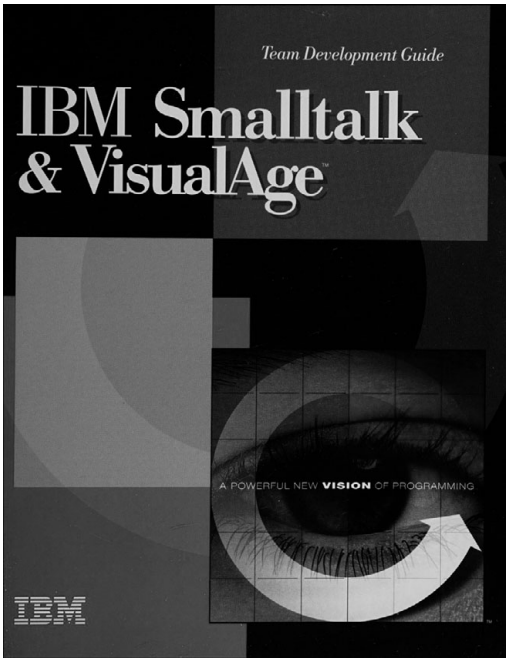


Figure 6.6 A cover of the 'Team Development Guide' manual of the IBM Smalltalk & Visual Age.
(Source: Internet Archive.[142] Reprint Courtesy of IBM Corporation ©)

In the 1990s, the companies producing commercial Smalltalk implementations have merged in various ways. Through a process that perhaps deserves a further investigation, all Smalltalk products have adopted the term 'Visual' as part of their name. The implementation from ParcPlace Systems became 'VisualWorks', the one from Digitalk became 'Visual Smalltalk Enterprise' and IBM joined in with 'IBM Visual Age Smalltalk'. On the one hand, the adoption of Smalltalk by IBM served as a significant endorsement for the commercial use of the technology. It even led some to view Smalltalk as 'the natural successor to COBOL (as opposed to C++) in many organizations'.[143] On the other hand, it also contributed to a further departure from the humanistic ideals and mathematical simplicity of Alan Kay's original vision. This can be illustrated by a quick peek into the 46-chapter-long 'Team Development Guide' (Figure 6.6) of IBM Smalltalk & Visual Age, published in 1994. The guide lists some 15+ different 'development browsers' (evolved from a single browser in Smalltalk-74) including the 'Class Browser', but also the likes of 'Application Changes Browser', 'Editions Manager' or the 'Application Configurations Browser'. The guide discusses browsers, components, applications, sub-applications, component ownership and also team roles and the development process, before showing the first Smalltalk code snippet on page 65. Despite all the complexity, the 1990s Smalltalk products served their purpose. They made it possible to use Smalltalk for commercial software development

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

and also addressed issues such as version control, which were conceptually challenging problems in Smalltalk.

But contrary to the future of Smalltalk envisioned in the mid 1990s, the rise of commercial Smalltalk was soon to stop and the commercial sector initially occupied by Smalltalk was quickly dominated by the object-oriented programming language Java which was announced and released in 1995.

6.11 The Better Way Is Here Now

In the early 1990s, object-orientation was the dominant programming paradigm and the commercial Smalltalk industry was on the rise. Many were thinking how to enable sharing of objects across different systems, leading to the development of 'middleware' systems and standards such as CORBA and the establishing of the Object Management Group (OMG).[144] It is perhaps surprising that the origins of the object-oriented programming language that would soon became the de facto language of enterprise software development had initially very little to do with any of these developments.

An oft-told story about the origins of Java is that a Sun Microsystems software engineer, Patrick Naughton, was about to leave Sun and mentioned this over beers to his hockey teammate, Scott McNealy, who was also the CEO of Sun. McNealy listened to Naughton's criticism of Sun and established a small team with unprecedented freedom to explore new avenues for the company.[145] The members of the resulting project 'Green' noticed that computer chips were appearing in many household appliances and decided to build a 'device that would work as an interface to cyberspace'.[146] To create a prototype, James Gosling developed the Oak programming system. The system was built around a virtual machine, which made it easy to support multiple processor architectures. Gosling utilised his earlier experience, which included building a hardware emulator during his PhD, but he also talked with Smalltalk VM developers, including L. Peter Deutsch.[147] The 'interface to cyberspace' project never materialised, but the technology became useful when the web emerged in the mid 1990s. The team built a Java interpreter embedded in a web browser that made it possible to distribute Java 'applets' through the web. Applets contributed to Java's popularity and enabled further developments that soon turned Java into a multi-platform object-oriented development system that quickly took over the sector previously occupied by commercial Smalltalk.

What preceded the oft-told story is an equally interesting story of how James Gosling ended up at Sun. Bill Joy, who was a co-founder of Sun Microsystems together with Scott McNealy, was interested in developing a new graphical user interface for Sun workstations. He decided to use a Smalltalk-based system and approached ParcPlace Systems, led by Adele Goldberg, to acquire a licence for their implementation. The deal never materialised because of disagreements about the royalties to be paid per machine sold. After the negotiations failed, Joy hired James Gosling to develop the system. The user interface project was ultimately unsuccessful, but the result was that Gosling was around to start working on what would eventually become Java.[148]

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Although Sun occasionally mentioned possible use of Java applets for creative purposes, the programming language was built to solve a specific engineering problem and its marketing also emphasised its engineering qualities. A Java white paper published in 1995[149] first explains the 'burden' of the software developer who is developing applications in C or C++, faces the growth of multiple incompatible hardware architectures and user interface toolkits and now also has to cope with the Internet and the World Wide Web. According to the white paper, 'The Better Way is Here Now'. Java and its runtime system will make 'your job as a software developer' much easier. The introduction talks primarily to the software engineers, uses primarily engineering arguments, but also contrasts Java with C++, rather than Smalltalk which it ended up replacing.

The rising popularity of Java inspired companies and individuals that were actively involved in the rise of the commercial Smalltalk to move to Java. Kent Beck reimplemented some of the tools developed initially for Smalltalk to work with Java, creating for example, the JUnit framework for Test-Driven Development. At the same time, IBM adapted its VisualAge programming environment to work with Java, even though it was still implemented in Smalltalk. It later reimplemented the system, creating the widely used Eclipse development environment. The relatively quick move from Smalltalk to Java is perhaps not surprising given the Java hype I mentioned in the opening of this chapter, but it was also likely supported by the cultural similarity between the two. Both commercial Smalltalk and Java in the 1990s took an engineering perspective on programming. A part of this perspective is also a pragmatic choice of suitable tools.

While the technical side of the Java language and platform has engineering origins, the origins of the development methodologies that were initially used to develop object-oriented solutions in Java are more diverse. The Smalltalk community, including Kent Back and his colleagues at Tektronix, was used to gain immediate feedback from the continuously running interactive system. This led them to highly interactive engineering-oriented development methodologies such as Extreme Programming and Test-Driven Development. Those methods were used by the Smalltalk community, who adapted them to the world of Java. But at the same time, Java quickly entered the managerial world of large enterprise systems development that was concerned about team structure and up-front modelling of the architecture of systems.

6.12 Managing the Object-Oriented Project

There is an interesting interplay between the rise of object-oriented programming, discussed here, and the move to the Agile development methodologies, which we saw emerging at Tektronix, but which I also discussed in Chapter 4. To make sense of it, we can take the inspiration from Peter Galison's analysis of microphysics.[150] Galison argues that dividing the history of physics into distinct overarching periods is too simplistic and instead suggests analysing the history at multiple levels that evolve partly independently. In the case of microphysics, the levels include theory, instrumentation and experimentation.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Each of those evolve independently of the other. When a new instrument appears, it can initially be used in existing experiments to study existing theories. Later, its emergence enables new kinds of experiments and leads to new theories that eventually replace the old ones. Similarly, the development of new programming paradigms seems to proceed somewhat independently of the development of new development methodologies. When object-oriented programming entered the domain of enterprise software development, it was initially integrated with the existing managerial methods. Only later, the new programming paradigm supported a shift to the more lightweight Agile development methodologies. Those new methodologies then, in more recent times, remained intact when some companies embraced the functional programming paradigm.[151]

Throughout the 1970s, many software engineers were occupied with the task of finding good ways to structure large software systems. Structured programming made it possible to organise programs in a more logical way that mirrored their logic, while research on information hiding provided guidance on how to decompose programs into modules or components.[152] However, those ideas were only of limited use in the new object-oriented paradigm, especially in languages like Smalltalk that more significantly departed from the conventional ways of organising programs in structured programming.

The problem was felt clearly at Tektronix. As recalled by Allen Wirfs-Brock, when Smalltalk started spreading from Tek Labs, the researchers 'had to start developing ways to … teach people how do you actually build something using [an object-oriented] language. … Fundamentally, how do you design software using objects'.[153] Thinking about a new object-oriented programming methodology became a major challenge for a group of Smalltalk researchers at Tek Labs. The group included Rebecca Wirfs-Brock, who moved from the graphics group to work on Smalltalk and Ward Cunningham with Kent Beck that we encountered earlier. In trying to formulate the object-oriented design methodology, the group could rely on their experience using Smalltalk, but even they did not have a single widely accepted methodology yet. The key change in thinking they identified was not to think of control structures that direct objects to do things, but instead focus on the interactions between objects.[154] Ward Cunningham took this design approach to the extreme and had an 'actor-oriented approach' with objects playing active roles in the design, producing designs that 'were quite different from others'. Rebecca Wirfs-Brock didn't carry her designs to that extreme, but she also started explaining things in terms of responsibilities that objects had.[155]

The thinking about object-oriented design and experience with internal developer training resulted in multiple publications presented at the conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) that was established in 1986 (and was also the place where Alan Kay delivered his keynote speech some 10 years later). Rebecca Wirfs-Brock wrote a paper that captured her approach and referred to it as Responsibility-Driven Design.[156] The paper also started a long-running tradition of using the format of 'something-driven' in the naming of development methodologies. Wirfs-Brock contrasted the approach with the established

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Data-Driven Design, where the focus was on the different information stored by objects. The work can be seen as another contribution to the debate about the nature of object-orientation that previously contrasted abstract data types (data-driven) with messaging (responsibility-driven). Beck and Cunningham[157] augmented the more abstract approach with the specific tool of Class-Responsibility-Collaboration (CRC) cards which were initially used in teaching, but were also adopted by designers to identify the responsibilities and collaborating classes of a given class.

The reflections on object-oriented design written by Tek Labs researchers had an engineering focus. They were mainly concerned with how to organise code into classes. However, they remained embedded in the broader thinking about development methodologies of the time. This can be seen from Rebecca Wirfs-Brock's book *Designing Object-Oriented Software*[158] that was published a year after her earlier paper and expanded on her design ideas. In the book, she describes how object-oriented design fits with the spiral software lifecycle, which I mentioned in Chapter 4. Here, software is developed in phases well known from the earlier Waterfall methodologies (requirements specification, design, implementation and testing) that are repeated iteratively in a spiral. Rebecca Wirfs-Brock argued that object-oriented design makes the design phase longer and the implementation phase shorter, but the basic lifecycle is not questioned in her book. This may be a surprise today, given that her colleagues at Tek Labs were already using many of the practices that would later become the cornerstones of Agile methodologies, but it is likely a case of 'intercalated periodisation' identified by Galison where a shift at one level (here, development methodologies) does not typically align with shift at another level (here, programming paradigms). The new approach was emerging, and Christiane Floyd has described it already in 1987, but it would take longer for it to fully take place.

Other major contributions to object-oriented design came from a world that was even more remote from the original humanistic origins of Smalltalk. Grady Booch got his undergraduate degree from the US Air Force Academy, had experience building ground support software for the space shuttle, and joined the company Rational Machines (later Rational Software) in 1981 as the Director of Software Engineering Projects. Given his association with the military, it is not surprising that his early work mostly involved the Ada programming language, but he also became a prominent Ada educator. In 1983, Booch wrote a book *Software Engineering with Ada* that has some of the first traces of what would later become an object-oriented design methodology. This includes the idea of looking at English descriptions of systems.[159] In the words of Booch:

> If we examine human languages, we find that they all have two primary components, noun phrases and verb phrases. A parallel structure exists in programming languages, since they provide constructs for implementing objects (noun phrases) and operations (verb phrases).[160]

Booch notes that top-down structured design methods, used in the context of imperative programming, are like trying to communicate with just verbs, while design focused on data structures is like trying to communicate with just nouns. Taking inspiration

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

from Parnas' work on decomposing systems into components,[161] Booch argues for a design that decomposes software systems into objects that capture aspects of both the data structure design (nouns) and the imperative design (verbs). Each object consists of some state, associated actions and a class that determines what kind of object it is. Notably, the term 'class' is used somewhat loosely in the text, because the Ada language at the time did not have an explicit support for what we would today refer to as object-oriented programming constructs.[162] The objects in the book are thus implemented as Ada packages that use local state, type declarations and procedures.

Similar to Rebecca Wirfs-Brock, Grady Booch saw object-oriented design as a part of the broader application development lifecycle. He makes it clear that 'object-oriented development is a partial life-cycle method' that 'focuses on the design and implementation phases of software development'. The design methodology is embedded in the conventional methodology with analysis, requirements gathering, design, coding, testing and maintenance. In the book, Booch illustrated the lifecycle using sketches of the ageing Ada, Countess of Lovelace, with a final sketch of Ada on her deathbed in the 'Operation and Maintenance Phase' section.

The engineering perspective on object-oriented programming dominated the 1980s, but it was not the end of the story. In fact, Rentsch's humorous remark that object-oriented programming was going to be the structured programming of the 1980s had more truth in it than it may seem. Structured programming started as an engineering notion and was later adopted by the managerial culture of programming. Object-oriented programming met the same fate. Over the following decade, approaches built around object-oriented development eventually expanded to other phases of the software development lifecycle and started to be used not just for engineering new systems, but also for managing their development. Following the later work of Grady Booch is a good way to illustrate this gradual transition.

At the turn of the 1990s, Booch wrote an influential book *Object-Oriented Design with Applications*. In contrast to his earlier work, this is not focused on the Ada programming language. It discusses examples implemented (or sketched) in multiple languages, still including Ada, but also Smalltalk, Object Pascal, C++ and the Common Lisp Object System. The broader focus on object-orientation also gained it an endorsement from Bjarne Stroustrup. Like in the earlier work, object-oriented concepts are limited to the design and implementation phases of a conventional lifecycle, this time with reference to the same spiral development model that Rebecca Wirfs-Brock talked about around the same time. The book recognises the difficulty of classifying things and, consequently, the difficulty of identifying meaningful classes and operations. It reviews various approaches, including the one based on identifying nouns and verbs, but offers no single panacea. One important characteristic is that, just like the overall development process, the design process through which the structure of the system emerges needs to be iterative.[163]

To help the process of designing a software system using objects, the book introduces a set of diagrams that can be used to capture the design of the system, including its structure of classes, but also dynamic properties such as state transitions and timing.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

As the introduction to 'The Method' section makes clear, the book is firmly rooted in the engineering culture:

> If you follow the work of any engineer – software, civil, mechanical, chemical, architectural, or whatever – you will soon realise that the one and only place that a design is conceived is in the mind of the designer. As this design unfolds over time, it is often captured on such high-tech media as white boards, napkins, and scraps of paper.[164]

Booch then explains the benefits of using a shared notation over napkin scribbles. It makes it possible to communicate the design to others, 'relieves the brain of unnecessary work' and enables using automated tools for checking the design. The development of unified modelling notation for object-oriented design is exactly what happened over the next couple of years. In the early 1990s, many others were trying to devise methodologies, processes and modelling notations for developing object-oriented systems. This included Ivar Jacobson, who developed the Object-Oriented Software Engineering (OOSE) methodology and James Rumbaugh, the creator of the Object Modeling Technique (OMT). All three methods consisted of a range of notations for capturing the design decisions as diagrams and recommended processes for obtaining the designs. By 1995, Rumbaugh and Jacobson joined Booch at Rational and the three started working on a common language, the Unified Modelling Language (UML), which was adopted by the Object Management Group (OMG) as a standard in 1997. The purpose of the UML was not to replace all the different methodologies for developing object-oriented designs, but merely to provide a common language for the result. However, the 'Three Amigos' at Rational, as they became known, eventually joined forces in developing a single Rational Unified Process (RUP).

Even though the Three Amigos initially approached object-orientation from the engineering perspective, the development of the RUP puts object-orientation clearly into the realm of the managerial culture of programming. The shift in the focus is clear from the first paragraph of the preface of their joint book *The Unified Software Development Process* that describes a non-proprietary version of the process developed at Rational Software:

> There is a belief held by some that professional enterprises should be organised around the skills of highly trained individuals. They know the work to be done and just do it! They hardly need guidance in policy and procedure from the organisation for which they work. This belief is mistaken in most cases, and badly mistaken in the case of software development.[165]

The book goes on to discuss the characteristics of the process, which is iterative and covers the entire lifecycle, starting from the capturing of requirements to testing, release and maintenance. The managerial approach is apparent in that the book no longer tries to structure just the classes of an object-oriented system, but the work of the team delivering an object-oriented project. In the attempts to avoid organising the development around highly trained individuals, it is reminiscent of the motivations that led to the 1968 NATO Software Engineering conferences as discussed in Chapter 2. Some 30 years earlier, the managers similarly tried to replace the reliance on

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

'geniuses and mavericks' and the 'black art of programming' with 'science of software engineering'.

The Rational Unified Process dominated the software engineering world around the year 2000. It was 'in fashion and everything else was considered out of fashion and more or less thrown out'.[166] The process itself was not necessarily heavyweight, but it was often perceived as such. Rational Unified Process was a process framework that could be instantiated in various ways, but most of its instantiations ended up being very complex and that was probably one of the factors that contributed to the 2000s rise of Agile methodologies that restored the more immediate, interactive ways of working that originated at Tektronix in the 1980s.

The clash between the Agile methodologies and processes such as the Rational Unified Process can be interpreted as a struggle between two cultures of programming. Whereas the managerial approach focused on organising teams, the engineering culture preferred focus on individual responsibility. The principle 'individuals and interactions over processes and tools' that was one of the points made in the 2001 Agile manifesto[167] can be read as expressing this exact cultural preference. However, some criticism of the managerial methods came from the Three Amigos themselves. In 2019, Ivar Jacobson referred to the evolution of development methodologies since the NATO Software Engineering conference as the 'Fifty Years' War' of methods. His criticism focused on the fact that 'once you have adopted a method, you get the feeling you are in a method prison controlled by the guru of that method', which is 'the craziest thing in the world'.[168] Even Grady Booch once remarked that he 'cannot be held responsible for all the stupid uses of the UML'.[169] In some ways, much of the criticism of the managerial object-oriented methodologies is nothing new. It is reminiscent of the 1988 remark by Edsger Dijkstra who criticised the software engineering discipline, claiming that it has accepted as its charter 'how to program if you cannot'.[170]

6.13 Object-Oriented Programming, Systems, Languages and Applications

Object-oriented programming is yet another example of a concept that has been influenced and reshaped by multiple cultures of programming. So far, I followed one particular trajectory in this chapter. The basic object-oriented concepts emerged in the context of computer simulations, where they were mainly shaped by the mathematical culture of programming. They were then reinterpreted as new media for communication during the early days of Xerox PARC, primarily from the perspective of the humanistic culture, while others at PARC found Smalltalk a flexible and powerful tool for tinkering with computers in ways favoured by the hacker culture. This diversification of interest in Smalltalk likely prevented a major language redesign in 1976. I then followed the widespread adoption of object-oriented programming, which saw the emergence of new languages that combined the basic concepts with different mathematical, engineering, hacker and humanistic aspects, and we also saw how Smalltalk itself turned from a humanistic vision into a practical commercial tool and the birthplace of many new engineering approaches. As with structured

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

programming a decade earlier, the managerial culture also found its way of turning object-oriented ideas into a basis for development methodology that aims to reduce risks in software development by controlling the development process.

By following a single path, I have inevitably left out many important events and directions. One such key event was the birth and the evolution of the ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) where Alan Kay delivered the 1997 keynote that opens this chapter.[171] The first OOPSLA conference took place in 1986 in Portland, which was the hometown of Tektronix and one of the centres of object-oriented technology at the time. OOPSLA was born out of a feeling that the Smalltalk community 'needed some sort of conference'[172] and was co-organised by Adele Goldberg, the president of the ACM at the time, Allen Wirfs-Brock from Tektronix, David N. Smith who worked on Smalltalk at IBM Research and Tom Love who was involved in the development of Objective-C. Already in 1986, the conference attracted some 600 attendees and it grew to over 2,500 in the 1990s.

In its early days, the OOPSLA conference attracted a mix of academic and practitioners. As recalled by Allen Wirfs-Brock, 'there was not that much difference between the academic papers … and the papers that originated from industrial research labs'.[173] The early publications were generally engineering-oriented and aimed to present the developing object-oriented technology to the broader audience. Many of the early OOPSLA publications were presentations of new programming systems. The conference brought together like-minded, but somewhat disconnected communities with different interpretations of object-orientation. In addition to contributions from the engineering-oriented commercial Smalltalk community, which I followed earlier, the first OOPSLA featured multiple application papers, theoretical papers, but also multiple papers presenting implementations of object-oriented ideas in Lisp.

The work on object-oriented programming in Lisp emerged in the context of interactive programming systems, which I talked about in Chapter 3. Similarly to Smalltalk, Lisp was not merely a language, but a stateful, interactive programming environment through which the computer was programmed, as well as used. In the 1970s, there were multiple personal connections between the two communities that originated in the earlier ARPA days. Through those connections, ideas about message passing in Smalltalk influenced a version of Lisp built at MIT for the Lisp Machine project. A new language feature, called Flavors, which added object-orientation to Lisp, was created by Howard Cannon and David Moon and was described in a report in 1980.[174] By 1986, there were multiple competing object-oriented extensions to Lisp including New Flavors and CommonLoops, both presented at the first OOPSLA, as well as other systems such as the prototype-based Object Lisp. One of the notable innovations of CommonLoops over competing Lisp designs was that the object-oriented system itself was defined in terms of meta-classes. As the name suggests, meta-classes controlled the features and the representation of the object-oriented programming system. This way, the definition of the object-oriented system itself leverages the object-oriented extensibility and becomes open to extension and modification.

Throughout the 1980s, there was an ongoing effort to standardise the many dialects of Lisp, which also included developing a single object-oriented Lisp extension. At the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

end of the 1980s, this resulted in the Common Lisp Object System (CLOS), which was mainly modelled after New Flavors and CommonLoops. Although the eventual Common Lisp standard does not specify the details, most CLOS implementations adopted the CommonLoops idea of using a meta-object protocol, which makes the resulting object system customisable. Through the meta-object protocol, CLOS realises one of the visions of Alan Kay who argued that the only way of handling the complexity of software systems, at least until 'proper engineering methods' are found, is to have extreme late binding and extreme flexibility so that systems can evolve over time in response to new needs and requirements.[175] The fact that main-stream object-oriented languages lack this flexibility was also one of the points that Kay raised in his 1997 keynote where he recalls looking at Java and thinking '[my] goodness … how do they hope to survive all of the changes, modifications, adaptations, and interoperability requirements without a meta-system'.[176]

The meta-object system in CLOS was the starting point for a 1991 book *The Art of the Metaobject Protocol*[177] that described the basic idea, alongside with its concrete simplified implementation. In his keynote, Kay claimed that he recalled the authors saying:

> This is the best book anybody has written in ten years, but why the hell did you write it in such a Lisp centric, closed club centric way?[178]

According to Kay, the book is very hard to read if one does not already know how CLOS is implemented, but it has 'some of the most profound insights about, and the most practical insights about OOP'. The fact that the OOPSLA conference brought together different communities thinking about object-oriented programming was thus apparently not enough to fully bridge the gaps between their individual peculiar interpretations and ways of talking and, perhaps surprisingly, this was the case even between the Smalltalk and Lisp communities, which both treated programming as interaction with a stateful, living system and had similar cultural origins.

The OOPSLA conference can be also used to illustrate another important shift that occurred in the history of object-oriented programming and, perhaps more broadly, programming. Many of the systems that were discussed in the early days of OOPSLA followed the view of programming as interacting with a stateful, living system that was shared by the Smalltalk and Lisp communities. This is the view emphasising interactivity that I retraced throughout the history of programming in Chapter 3. The view is not strictly linked to a particular culture of programming, although it is easier to reconcile with the hacker mindset, focused on direct engagement with the machine, and with the humanistic vision of augmenting human intellect. Throughout the 1990s, the focus of OOPSLA has gradually shifted from discussion of engineering aspects of new systems to more academic theoretical research. But alongside came a shift from interactive programming systems to static programming languages. The shift alienates some of the earlier members of the community, who felt they could no longer read most of the papers.[179] Richard P. Gabriel interpreted the shift more strongly as a Kuhnian paradigm shift. In a 2012 paper 'The Structure of a Programming Language Revolution',[180] he argued that there is a degree of incommensurability between the earlier 'programming system' and the later 'programming language' perspective.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Through the topic of object-oriented programming, we have completed the full circle. I started the book by retracing the origins of two major different ways of thinking about programming. In Chapter 2, I followed the efforts of the mathematical culture to mathematise programming and establish programs and programming languages as formal mathematical entities. In Chapter 3, I then shifted focus to the influences of the hacker and humanistic cultures that viewed programming as interaction with a stateful system. Object-oriented programming has been reconceptualised in major ways, for better or worse, through the shifts between these two views and their respective cultures.

Notes

1 Mitchell (2003).
2 The exact phrasing used here, including the terms 'almost a new thing', 'old thing' and 'new thing' are due to Kay (1996).
3 Mitchell (2003).
4 Kay and Goldberg (1977).
5 Parnas (1972); Liskov and Zilles (1974).
6 The most detailed recent contribution to the discussion has been written by Cook (2009).
7 Whitaker (1993).
8 Dijkstra (1988).
9 First described by Abbott (1983), but made broadly known by Booch (1983).
10 Paraphrasing an answer given by Allen Wirfs-Brock to an audience question about 'the failure of object-oriented programming' at a conference. Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
11 Paraphrasing Alan Kay's answer from an interview by Greelish (2013).
12 Ingalls et al. (1997, 2008), but note that these are large projects with multiple contributors beyond their original creators.
13 The idea of bringing some of the visions behind Smalltalk to UNIX and C has been explored by Kell (2018).
14 Quoted from a letter to the editor in the Dr. Dobb's Journal (Passani, 1996).
15 Wirfs-Brock (2020).
16 Kay (1997).
17 Brand (1972).
18 It is possible that this claim is rooted in broader Western culture, where objects play a crucial role as philosophical entities, but examining this assumption and its alternatives is beyond the scope of this chapter.
19 Nance (1993).
20 Nygaard and Dahl (1978) report that they were familiar with the language SIMSCRIPT, which provided some inspiration for list processing, time scheduling mechanisms and the procedure library in SIMULA. They were aware of GPSS, but have not studied it closely.
21 Letter to Charles Salzmann, dated January 5, 1961, quoted in Nygaard and Dahl (1978).
22 Nygaard and Dahl (1978).
23 The social and political aspects of the development of SIMULA have been described by Holmevik (1994).
24 Holmevik (1994).
25 Nygaard and Dahl (1978).
26 Nygaard and Dahl (1978).
27 Dahl (1969).
28 Dahl and Nygaard (1966).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

29 Nygaard and Dahl (1978).
30 Dahl et al. (1972).
31 Nygaard and Dahl (1978) identifies four different stages. What I refer to as the 'first stage' is their stage 2 and what I refer to as the 'second stage' is their stage 4.
32 Presented in a Communications of the ACM paper (Dahl and Nygaard, 1966).
33 Dahl and Nygaard (1966).
34 Nygaard and Dahl (1978).
35 Nygaard and Dahl (1978). It is worth pointing out that this is exactly what the implementation of async/await (Syme et al., 2011), in modern programming languages such as C#, JavaScript or F# does.
36 Nygaard and Dahl (1978).
37 Nygaard and Dahl (1978).
38 Nygaard and Dahl (1978).
39 Hoare (1965).
40 Compare the lecture notes 'to be delivered' Hoare (1965) with the published proceedings (Genuys, 1968); the likely influence resulting from the meeting is also suggested by Black (2013).
41 Hoare (1965).
42 Dahl et al. (1968).
43 Dahl and Nygaard (2002), also quoted by Black (2013).
44 Sammet (1969), referenced by Nance (1993).
45 Sammet (1969).
46 Dahl et al. (1968).
47 Kay (1996) and Ingalls (2020), although, I do not accept those retrospectives uncritically and also aim to find supporting or contrary evidence in the available primary sources. A useful high-level account of the history of Smalltalk has also been written by Priestley (2011).
48 The title of this section refers to those influences and ongoing technology changes that make them possible. It is also a quote from a November 1971 panel discussion abstract, Kay (1972a).
49 Kay and Goldberg (1977).
50 Kay (1972b).
51 Ware (1966).
52 Kay (1996).
53 Kay (1968).
54 Kay (1969).
55 The term 'personal computer' is used anachronistically here. The term was used by Fisher et al. (1975) in reference to the LINC computer. It appears earlier, in the 1965 Air Force report, and then in a paper by Kay (1972b), but it was also used around the same time for talking about programmable calculators, for example, by Tung (1974).
56 Kay (1996).
57 Kay (1969).
58 Kay (1969).
59 Kay (1969).
60 Landin (1965a,b).
61 Hiltzik et al. (1999) offers a glimpse of those, although not entirely unbiased.
62 Kay (1996).
63 Using modern programming language terminology, the `if` construct requires lazy evaluation (or call-by-name strategy) rather than eager evaluation (or call-by-value strategy).
64 According to the timeline given in the Appendix IV of the published version of Kay (1996).
65 Goldberg and Kay (1976b).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

66 Goldberg and Kay (1976b).
67 Kay (1996).
68 Kay (1996).
69 Correspondingly, writing on SIMULA 67 and Smalltalk-72 is much easier to read for a modern computer scientist than writing on their immediate predecessors.
70 Kay (1996).
71 Kay (1996).
72 Kay (1996).
73 Kay (1974).
74 https://bitsavers.org/pdf/xerox/smalltalk/ALLDEFS_Apr75.pdf, retrieved May 16, 2024.
75 Goldberg (2002).
76 Ingalls (2020).
77 Rentsch (1982).
78 Kay (1996) and Ingalls (2020).
79 See Chapter 3.
80 Ingalls (2020).
81 Ingalls (1978).
82 Kay and Goldberg (1977).
83 Kay (1996).
84 For example, in the paper by Ingalls (1978).
85 Bobrow and Winograd (1976).
86 Bobrow and Winograd (1976).
87 Kay (2003).
88 Liskov and Zilles (1974); As pointed out by Cook (2009), a paper by Zilles (1973) discussed the ideas around data abstraction slightly earlier and was closer to the perspective of Smalltalk. It mentions the term ‘object’ but not ‘object-oriented’.
89 Jones and Liskov (1976).
90 Kay (2003).
91 Kay (1996).
92 Kay (1998).
93 Cook (2009).
94 Quoted from the chairman’s introduction of the May of 1969 Extensible Languages Symposium, (Cheatham, 1969).
95 Kay (1969).
96 Kay (1968).
97 Kay (1996).
98 Fisher (1970).
99 Kay (1996).
100 Kay (1996).
101 Petricek (2023).
102 Liskov and Zilles (1974).
103 Rentsch (1982).
104 The rise of object-oriented programming is contextualised in a retrospective on Objective C by Cox et al. (2020), while the irony of the recurring feeling of a crisis in a financially successful industry has been aptly pointed out by Ensmenger (2012).
105 Cook (1988).
106 Madsen and Møller-Pedersen (2022).
107 Later, the meeting of the object-oriented and British mathematical world was one of the starting points for the development of the F# programming language, as discussed by Syme (2020).
108 Cardelli and Wegner (1985).
109 Cardelli (1984); Cook and Palsberg (1989).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

110 Abadi and Cardelli (1996).
111 As discussed in the first-hand retrospective by Kristensen et al. (2007).
112 Kristensen et al. (2007) acknowledges this, but also points out that this view often comes from people not using BETA.
113 Ungar and Smith (2007).
114 Ingalls et al. (1997).
115 Ungar and Smith (2007).
116 Ungar and Smith (2007).
117 Smith and Ungar (1995).
118 Borning (1986) and also an earlier publication on programming aspects of ThingLab (Borning, 1981).
119 Smith and Ungar (1995).
120 The Self authors do, however, acknowledge the influence of the mathematical culture on the design of the language in its striving for simplicity in the language design in Ungar and Smith (2007).
121 For the analysis of the crisis narrative in the field, see the excellent work of Ensmenger (2012); Ryder et al. (2005) relates a number of developments in programming languages to parallel work on software engineering.
122 Liskov (1993).
123 The overall project history is documented in a retrospective by Whitaker (1993). Technical rationale for the design can be found in the extensive 200+ page report by Ichbiah et al. (1979).
124 Whitaker (1993).
125 Whitaker (1993).
126 Meyer (1992).
127 Meyer (1992).
128 Stroustrup (1996); The author has also written more about the design of the language (Stroustrup, 1994), and its more recent evolution (Stroustrup, 2007).
129 Stroustrup (1996).
130 Stroustrup (1996).
131 Stroustrup (1986); incidentally, also the book containing the quote that gave title to this section.
132 Stroustrup (1988).
133 'The fate of facts and machines is in the hands of latter users' (Latour, 1987).
134 Thacker et al. (1982).
135 Goldberg (2002), Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
136 Goldberg (2002).
137 At Tektronix, the performance was about 1% of the Xerox workstation and their first prototype Smalltalk system took an hour to render the welcome screen, Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
138 Goldberg and Robson (1983), nicknamed after the colour used on the book cover.
139 Wirfs-Brock (2013) and Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
140 Thomas (1995).
141 Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
142 https://archive.org/details/ibmsmalltalkvisu0000unse. IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at 'IBM Copyright and trademark information' at www.ibm.com/legal/copytrade.shtml.
143 Hunt (1997).
144 PROGRAMme (2022).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

145 There is no definitive account of the history of Java. I draw on multiple sources including Thagard and Croft (1999); Bank (1995) and archived articles published by Sun, English (1998); Byous (1998).
146 Bank (1995).
147 Allman (2004).
148 Richard P. Gabriel (personal communication, 17 February 2024).
149 Gosling and McGilton (1995).
150 Galison (1997).
151 An example of this is the Domain-Driven Design methodology that has been initially developed in the object-oriented context, but has been swiftly adapted for functional programming, as described by Wlaschin (2018).
152 The key theoretical reference was the work of Parnas (1972), which later inspired the development of language features such as modules in Ada.
153 Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
154 'Most people tend to think of control structures and directing the objects to do things as opposed to having objects interacting', Rebecca Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
155 Rebecca Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
156 Wirfs-Brock and Wilkerson (1989); the paper was co-authored with Brian Wilkerson.
157 Beck and Cunningham (1989).
158 Wirfs-Brock et al. (1990).
159 The idea was inspired by a paper by Russell Abbott from the same year (Abbott, 1983).
160 Booch (1983).
161 Parnas (1972).
162 Ada made it possible to define subtypes and derived types, but those had a somewhat different meaning than in later object-oriented languages. A subtype defines a new name for a constrained type, i.e., values of another type that satisfy some condition (restricting the range of its values). A derived type is a new distinct type, but with the same structure as another type.
163 The account draws on the excellent overview article Lee (2021).
164 Booch (1990).
165 Jacobson et al. (1999).
166 Jacobson et al. (2019).
167 Beck et al. (2001).
168 Jacobson et al. (2019).
169 Hohpe (2005).
170 Dijkstra (1988).
171 Another part of the history of object-oriented programming that is missing in this book is the development of design patterns that became famous in the form of a book by Gamma et al. (1995), but that also inspired reflection by Gabriel (1996), which is perhaps closer to the origin of the idea. The history, alongside other links between computer science and architecture, has been documented recently by Steenson (2022).
172 Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
173 Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
174 Weinreb and Moon (1980); for more information about the history of object-orientation in Lisp, see the first-hand account by Steele and Gabriel (1996b) and also the earlier incomplete draft of the paper, Steele and Gabriel (1996a).
175 See Alan Kay's appendix 'Is "Software Engineering" an Oxymoron?' (Smith et al., 2002).
176 Kay (1997).
177 Kiczales et al. (1991).
178 Kay (1997).
179 Rebecca Wirfs Brock and Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
180 Gabriel (2012).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

7 Conclusion: Cultures of Programming

Teacher: You may remember that I was initially sceptical about the claim that we all think about programming in different ways. I admit that our discussion over the previous six lectures has convinced me. Our understanding of what a program is differs and each of us has a different perspective on how programming should be done...

Pythagoras: Clearly, it is not just the past that is a foreign country. Other cultures of programming are foreign countries too. They do things differently there...

Diogenes: Sorry, but what are you trying to say?

Pythagoras: Sorry for paraphrasing an unfamiliar quote.[1] The point was that the different 'cultures' we are talking about are like different scientific paradigms. They are also rooted in different basic assumptions, rely on different methods and adopt different ways of understanding the world.

Teacher: Interesting. Can you elaborate on what this would imply about programming?

Pythagoras: This way of looking at the past suggests that there are different eras based on different scientific paradigms and that those paradigms are, to some degree, incommensurable. This means that scientists trained in one paradigm cannot comprehend the theories rooted in another paradigm.

Archimedes: I don't think this is quite the right way of looking at cultures of programming. We may disagree about our basic assumptions, but with some effort, I can understand thinking based on different principles. For example, I understand the consequences of treating programs as mathematical entities, even if I do not agree it is the most useful perspective.

Socrates: Moreover, Kuhnian paradigms cover different historical periods, whereas different cultures of programming exist in parallel. You need a more refined philosophical notion. You could look at cultures of programming as systems of practice[2] that consist of different activities that are used to achieve some goal. This is also compatible with the fact that some activities are shared by different cultures.

Xenophon: This is an interesting debate, but I think you are obsessing over the details. Of course we can understand each other, because we can always describe programming problems using English! This is also why software specifications should be written in a natural language.

Pythagoras: But ordinary natural language is not precise enough for talking about formal entities like computer programs. For that, you need an exact mathematical notation...

Teacher: It sounds like writing a natural language specification and writing a formal mathematical description would be two activities of different systems of practice and also two activities typical for the managerial and mathematical culture of programming.

Diogenes: As I said multiple times, the context in which you do programming differs, but ultimately, you are creating programs and code. The activity of writing code is surely shared by all of us and if you focus on that, most of your confusing epistemological problems disappear.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Archimedes: Except that you can never ignore the context! It is like saying that all architecture is the same, because it is done using bricks and cement!

Teacher: Maybe so, but perhaps we can learn something by looking more closely at code. It really is shared by all cultures of programming and it certainly expresses richer structures than bricks in housebuilding.

Archimedes: I would cautiously agree that code is, alongside with natural language, something that we all share. But code itself is not all that interesting. It does, however, manipulate higher-level entities that we all understand and share.

Diogenes: What 'entities' are you referring to?

Archimedes: We talked about them in the previous lectures. Objects, types, tests and other such entities that programs are built from.

Socrates: Those entities are certainly shared by multiple cultures, but we have also seen that each culture interprets them in a different way. Types, for example, are a data structuring tool for some and a lightweight formal methods for others. It may help to see them as boundary objects[3] that make it possible to exchange ideas, but without a complete consensus between cultures.

Xenophon: This seems to work for entities that exist in code, but what about practices?

Diogenes: I agree. Informal practices like debugging are equally important, but I do not think you can view them as boundary objects.

Pythagoras: Well, maybe that is the problem. If certain programming practices are elusive and cannot be defined or captured in any form, that may be why they have not evolved much since the 1960s...

Socrates: There has certainly been a lot of work on debugging in the context of Smalltalk systems, but you may be right that this has failed to transfer to other cultures of programming for some reason.

Teacher: This brings me to a question originating in the philosophy of science that I wanted to ask. How do different cultures of programming accumulate and share knowledge?

Pythagoras: Computer science is a scientific discipline, so knowledge is accumulated through academic publications.

Archimedes: This is certainly one component, but I think you have to include a broader range of materials. Much of the practical engineering knowledge has been documented in books and, more recently, recordings of conference talks.

Xenophon: For managing software projects, the knowledge is more often distilled in process specifications or international standards, but broadly speaking, we all seem to agree that knowledge is accumulated in written or otherwise recorded form.

Pythagoras: International standards are a good addition, but I do not like the idea of including random videos that have not been peer-reviewed and lack academic rigour!

Socrates: But there are so many important ideas you can only illustrate! Take 'The Mother of All Demos' that we talked about in Chapter 3. This was an eye-opening demo showing the new possibilities of computers. You would not be able to convey the same message equally powerfully in writing.

Diogenes: Formats like videos or screen captures are also interesting, because they let you watch how others actually work. This way, you can learn from experts in ways that, a couple of years ago, were only possible by sitting next to them and watching over their shoulder.[4]

Socrates: In your beloved hacker culture, there seems to be a lot of practical expertise that is learned through practice and guidance. This is otherwise known as tacit knowledge.[5] But I would also insist that this does not make the culture any less scientific. So much of science also relies on tacit knowledge!

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Teacher: Oh well. We identified five different cultures of programming and so I seem to get five different answers to every question I ask!

Xenophon: That may be so, but they are remarkably consistent over time. The answers we find by examining the past seven decades are very similar to those we get by asking the question today.

Socrates: And I'd say that there is yet another consistency at an even more general level. If you look at the history, there are a couple of patterns of interactions between the cultures that repeat over and over again.

Diogenes: For example?

Archimedes: If I understand you correctly, I can think of some examples. For example, we encountered a number of cases where multiple cultures contributed to a single shared idea, making the whole more than the sum of its parts.

Pythagoras: I admit that there were multiple cultures involved at the origins of concepts like programming languages, types or objects. This is a historical fact I do not deny, but I would say this happens at the pre-scientific phase, before the notions are described rigorously.

Socrates: You are just illustrating another, equally common pattern!

Pythagoras: Huh?

Socrates: We have seen many cases where cultures struggle for control over a certain notion. For example, do we treat programs as formal mathematical notions as you would have it, or are they socially constructed entities?

Teacher: It seems that the disagreements are unavoidable, but does it help to acknowledge them? Does it help that we are now, at least, aware of our own biases and assumptions?

Socrates: Certainly. Being aware of your own basic assumptions is the first step towards being able to critically reflect on them.

Diogenes: Yeah, well, we know by now that *Pythagoras* is obsessed with treating everything as a formal mathematical problem...

Archimedes: That may be so, but I fail to see how analysing different cultures of programming helps me, as a practising programmer, build better software more effectively.

Diogenes: Perhaps the critical reflection can also suggest new directions. For example, the case of testing illustrated the value of exposing an entity to other cultures. I imagine that we could get better at programming if the notion of debugging, which mostly exists in the realm of the hacker culture, became available to others.

Teacher: My tentative optimistic conclusion is that we should talk and collaborate more.

Archimedes: Seeing all the cases where different cultures contribute to a certain idea definitely makes me more attentive to ideas from outside of my typical circles...

Diogenes: Please, just do not forget that what we are talking about is, at the heart of it, just programming. You can look at it from multiple perspectives, but fundamentally, you have to learn how to write code.

Socrates: Perhaps, but this happens in a complex socio-political context you cannot ignore!

Xenophon: Do you mean aspects like business needs or legal constraints?

Socrates: I mean at a more fundamental level. For example, take the idea of a more direct, interactive approach to programming that we followed in Chapter 3. It envisions empowering individuals and makes programming more accessible and fun. But it also keeps getting ruined by commercial needs and engineering desire for control and efficiency.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Teacher: I was hoping to end on a positive note of encouraging collaboration and mutual understanding, but perhaps the precarious aspects that arise from interactions between cultures of programming are equally unavoidable. Looking at the 70 years or so of history of programming, it seems clear that the different cultures will not suddenly go away, but perhaps, we can at least respect each other more now that we better understand our different positions.

7.1 The Most Powerful Tool Available to Human Intellect

The role of abstraction in both academic computer science and professional software development cannot be overstated. Many of the computer scientists and programmers that we encountered in this book highlight its importance. According to Hoare, abstraction is 'the most powerful tool available to the human intellect [in] the development of our understanding of complex phenomena',[6] while Dijkstra argued that 'the effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer'.[7]

In programming, the purpose of abstraction is to manage the complexity in a software system. In the view of Barbara Liskov, 'what we desire from an abstraction is a mechanism which permits the expression of relevant details and the suppression of irrelevant details'.[8] Dijkstra goes even further and argues that the purpose of abstraction is 'to create a new semantic level in which one can be absolutely precise'.[9] Computer programmers are advised to introduce abstractions in their programs when they encounter multiple variations of the same code. In the Agile software development, 'The Rule of Three' advises programmers to introduce an abstraction when they write similar code for the third time:

> The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor.[10]

Of course, some of the cultures of programming view abstraction as more central than others. The mathematical and engineering cultures see it as vital, while others may raise objections. To hackers, excessive abstraction creates an obstacle between the machine and the programmer and introduces performance overheads, while the proponents of a humanistic culture may object to abstractions that rely on a particular educational background. However, all programmers and computer scientists utilise abstraction and learning how to do so is a major part of their education.

7.2 Abstracting the History of Programming

The prologue on abstraction is not an attempt to overwhelm the reader with yet another case study of how cultures of programming interact. Its purpose is to frame the ideas put forward in this book. The book is written to shed new light on the history of programming and is more a contribution to documenting history than a contribution

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

to computer science. Yet, it is written by a computer scientist who has expertise and training in utilising abstraction, if sometimes reluctantly. The notion of a 'culture of programming' that I introduce in this book is fundamentally one such abstraction.

To paraphrase the prominent computer scientists quoted earlier, cultures of programming is a notion that aims to provide a finite piece of reasoning that can cover a myriad of cases.[11] It 'arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate on these similarities, and to ignore for the time being the differences'.[12]

As with any abstraction, the key question is whether the concept of cultures of programming truly suppresses the irrelevant details and highlights the relevant ones. To judge this, we can learn from a philosopher of science, Hasok Chang, who faces the problem of justifying the abstraction of 'epistemic iteration' that he introduces in his fascinating account of the invention of temperature. Chang argues that an abstraction 'needs to show its worth in two different ways'. First, 'its cogency needs to be demonstrated through abstract considerations and arguments', and second, 'the applicability of the idea has to be demonstrated by showing that it can be employed in the telling of various concrete episodes in instructive ways'.[13]

Before assessing the notion of a culture of programming using Chang's criteria, it is useful to point out that my abstraction partly overlaps with a number of other abstractions that identify different approaches, subgroups or subcultures within computing. Looking at the origins of computing, Primiero[14] identifies mathematical, engineering and experimental foundations for the discipline. Similarly, Tedre[15] identifies mathematics, engineering and science as three viewpoints on computing. Looking at coding education, Halvorson[16] identifies multiple hobbyist, hacker and professional programming cultures, while the hacker culture is a part of the computing folklore and has been the subject of multiple studies.[17] Several perspectives on the history of computing also appear in the work of Michael Mahoney, a pioneer scholar of the history of computing. At a broad level, Mahoney identifies the 'tripartite nature' of computing, which 'shows up in the three distinct disciplines that are concerned with the computer: electrical engineering, computer science, and software engineering'.[18] He also looks at numerous specific communities, affiliated with domains such as management, data processing, military command and control, human augmentation, computational science and others.[19]

Each of the above classifications focuses on a different aspect of computing and serves a different purpose. My focus has been on identifying different approaches to programming. The key factors for what constitutes a culture of programming have been the basic way the proponents of the culture understand what a program is, what methods for program construction they see as the most appropriate, as well as how they believe programming should be taught and what are the necessary skills of a good programmer. In contrast to some of the above accounts, I am less concerned with the historical origins of the discipline or specific communities of people in computing. Instead, I focused on ways of thinking that can be found repeatedly over the history of programming.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

As an abstraction, cultures of programming satisfy Chang's requirement of cogency in that they are compatible with many of the aforementioned views and, more importantly, each culture of programming is associated with a coherent set of beliefs about programming that I examined in the preceding chapters. (The humanistic culture is an outlier in that certain incoherence is in its nature.) That cultures of programming also satisfy Chang's requirement of applicability has hopefully been illustrated through the many situations discussed in this book. The notion has served well to explain clashes around structured programming, program verification, software engineering, the evolution of types and objects, as well as the discussion between programmers that I encountered at the NDC conference in Oslo in 2014 and mentioned in the opening of the book.

In some cases, we may be tempted to identify a particular historical actor with a specific culture of programming or even self-identify with a particular culture ourselves. However, this would be inaccurate and often misleading. Cultures of programming are a post hoc abstraction. They are fictional constructs that help us make sense of interesting developments and disagreements in the history of programming. They aim to provide accurate a broad strokes account of such interactions by highlighting aspects that play crucial roles in such interactions, but they hide many other aspects and developments. Many of the historical actors that we encountered in this book made significant contributions to multiple cultures of programming. I mentioned John McCarthy as an example of this in Chapter 1. We encountered him as the author of pioneering mathematical papers in Chapter 2, as well as the creator of Lisp and 'Uncle John' of the MIT hackers in Chapter 3. Many others combine aspects of multiple cultures. Edsger Dijkstra was a strong proponent of mathematical methods, but he was also a signatory of the Algol 68 Minority Report that recognised the failure of those methods and an active participant in the NATO Conference on Software Engineering, where he advocated ideas that align well with the engineering culture emerging at the time. The framework of cultures of programming can prompt a critical examination and reveal new perspectives, but it is not intended to reduce the rich history of programming into a simple narrative. Individuals that can bridge multiple cultures of programming may well be essential in facilitating the collaboration between cultures that was essential in multiple developments discussed in this book, ranging from the birth of the idea of a programming language to the conceptual development of objects.

Thinking of cultures of programming as an abstraction, we may also ask how well the notion separates the relevant details of the situation it describes from the irrelevant details. To answer this question, I will contrast cultures of programming with established notions from the philosophy of science that have been used as a lens for looking at the history of programming. Perhaps the most commonly used philosophical concept is a scientific research paradigm, developed by Thomas Kuhn. Scientific paradigms have been used to look at a number of developments in the history of programming.

Computing Historian Mark Priestley uses the notion of a scientific research paradigm to discuss the mathematisation of programming associated with the development of the Algol 60 programming language. He argues that 'what changed the

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

face of programming was not simply the Algol 60 language, but rather a coherent and comprehensive research programme within which the Algol 60 report had the status of a paradigmatic achievement'.[20] Drawing on Priestley's work, I discussed the importance of the Algol 60 report in the context of the mathematical culture in Chapter 2. However, the development of Algol 60 happened concurrently to other work on programming, including, for example, work on interactive programming in Lisp explored in the context of the hacker culture. Algol 60 thus did not define an overarching scientific paradigm for all of programming. Work in the Algol 60 paradigm also interacted with work in many other cultures, for example when importing data types from the managerial culture of COBOL. The Algol 60 paradigm was thus also not entirely incommensurable with other views on programming.

Another notable use of the notion of a scientific research paradigm in looking at the history of programming comes from Richard P. Gabriel's reflection on the 'programming language revolution',[21] which I mentioned in Chapter 6. Gabriel argues that in the 1990s, the dominant way of thinking about (primarily object-oriented) programming systems changed from an engineering perspective focused on programming systems to a more formal perspective focused on programming languages. Gabriel emphasises incommensurability between the two paradigms, which is justified in the sense that researchers working in the 'languages' paradigm misinterpret earlier texts. They treat as essential features that the original authors would see as accidental and vice versa. A reference to scientific research paradigms serves well in this case in that it lets Gabriel explore misunderstandings arising from incommensurability, even if it is not incommensurability in the strongest sense. Yet, the shift to programming languages is not overarching and the engineering perspective, again, continues to exist in parallel to the mathematical one. It is pushed outside of the particular academic conference that Gabriel discusses, but this can arguably be better seen as a struggle for control between different cultures of programming.

Looking at the history of programming through the perspective of cultures of programming highlights the pluralism of the field. Throughout much of its history, there were different parallel developments, pursued predominantly by different but interacting and sometimes overlapping groups of people. This pluralism may have been easy to explain in the early days of programming, when practitioners came from multiple adjacent disciplines including mathematics and logic, electrical engineering, natural sciences and psychology. The fact that the same ways of thinking about programming can be identified throughout its over 70-year-long history suggests that this is not an accidental aspect of an immature discipline, but an essential aspect of computer programming. Cultures of programming provide one possible abstraction that reflects this fact.

In the history of science, a number of abstractions that embrace some degree of scientific pluralism have been proposed as alternatives to the overarching Kuhnian scientific research paradigms. In his classic account of the history of experimental microphysics, Peter Galison[22] identifies two subcultures within the discipline. The image tradition is focused on capturing singleton 'golden events', while the logic tradition justifies its claims through statistical demonstrations. Galison's analysis is relevant

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

to the analysis of cultures of programming in that it also explores the mechanisms through which the different subcultures interact and exchange knowledge. Galison introduces the idea of a trading zone, which is a domain in between the subcultures within which multiple groups can work and communicate using shared (but limited) language. Such trading zones certainly exist between multiple cultures of programming and documenting those would be an interesting addition to the analysis done in this book. Source code in a concrete programming language is likely one example.[23] However, in the history of microphysics, the image and logic subcultures eventually converge and find a common language, which has not been the case in the history of programming.

It is also worth noting that the term 'cultures of programming' that I use has connotations because of its name. In contrast to Galison who carefully justifies his use of the term 'subculture', my use of the term 'culture' is somewhat loose.[24] I use it to highlight the fact that different cultures of programming have particular beliefs, values, assumptions and practices, but the concept of a culture in its original sense is, of course, much stronger. One notable discrepancy between the established use of the term culture and my use in this book is that cultures of programming do not always rely on direct social transmission. Typically, a culture of programming has some shared social venues, be it conferences, journals, publishers, mailing lists or affiliated university departments or laboratories. However, works that are aligned with a particular culture of programming can also emerge without direct social connection. For example, the hacker culture that emerged around microcomputers in the 1970s had only little direct interaction with the 1960s MIT hackers, yet they shared similar values, perspectives and even practices. The similarity in practices, in particular, was likely the result of similar thinking about programming and a similar technological context.[25]

7.3 Theories of Knowledge

Thinking of cultures of programming as an abstraction raises the question of which details of the reality of programming the abstraction highlights and which details it hides. To answer the former question, we can try to identify the cultural markers that characterise different cultures of programming. The first and foremost characteristic of a culture of programming is how its proponents think about programming and the nature of computer programs.

For the mathematical culture, a program is a formal mathematical entity. It also happens to be represented in code, on a real physical medium, and executable by a real physical machine, but as we have seen in Chapter 2, the difficulties arising from this are often ignored. A program is an expression in a language that can be fully formally defined and so the meaning of a program should be unambiguous and can be formally checked. The mathematical culture is less concerned with how programs come to an existence. Just like with mathematical proofs, programs may well come to you in a dream. By contrast, the managerial culture is less concerned with what programs are, but cares mainly about how they are constructed. Programs exist to solve

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

a particular (business) problem and have to be constructed in a way that ensures they will do so well. As programming is a social activity that involves teams of people that have to collaborate, the culture emphasises the processes and methodologies that are used to control the collaboration. The structure imposed by managerial methodologies is reflected in the software structure, either unintentionally or explicitly by design when software is organised in components that enable suitable structuring of teams, as in some of the managerial takes on object-oriented programming discussed in Chapter 6.

The engineering culture exists somewhere in between the two extremes. This position is consistent with its origins discussed in Chapter 4. The engineering culture was born out of a certain dissatisfaction with purely mathematical and managerial attempts to tackle the 1960s software crisis. The culture sees programming as an engineering discipline, that is, the application of scientific and mathematical principles to design and develop solutions to practical problems. It acknowledges that programs often need to solve complex problems that may not be known in full. It also acknowledges that the application of mathematical principles to practical problems bridges a gap that imposes limits on how well the scientific and mathematical principles can be used when building software systems embedded in a real-world environment.

The hacker and the humanistic cultures are more self-centred in that they are less concerned with solving external problems and more with problems that they themselves find important. For the hacker culture, programming is about testing the limits of computers, achieving as much as possible with the available resources, but also about developing better understanding of how computers work. Consequently, programming requires direct access to the machine and the underlying system. The humanistic culture shares an emphasis on learning and enabling humans to do more with computers, but it takes a broader perspective. Programming and interaction with computers more generally is seen as a powerful addition to literacy and human cognitive capacity. Consequently, the proponents of the humanistic culture often emphasise that programs need to be understandable and transparent to their users. The humanistic culture thus also shares the belief of the hacker culture that 'information wants to be free'.[26] The work that arises from the humanistic culture is often interdisciplinary, crossing the boundaries between computing, art, education, psychology and other disciplines. This results in another characteristic of the culture, which is that its conception of programming is often eclectic and takes a variety of different, and sometimes mutually inconsistent, forms.

Looking back at the developments discussed in the past five chapters, we can identify a number of other characteristics that serve as cultural markers for the individual cultures of programming that I have discussed. One that is apparent in the references to the various sources used throughout the book is the different approach to recording and sharing knowledge. Many of the sources that I referred to when discussing the work of the mathematical culture were formal academic publications. The culture has a strong academic affinity, so this is not a surprise. But more importantly, the format of an academic paper is suitable for the kind of knowledge the culture considers important. A paradigmatic example of such knowledge that we encountered were algorithms and programming language specifications.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Like the mathematical culture, the engineering and managerial cultures also believe that most knowledge about programming is explicit and can be clearly articulated and documented in a written form. In the case of the engineering culture, the materials often take the form of professional books written for practising software engineers. The longer format reflects the fact that knowledge in the engineering culture often takes the form of collections of best practices and guidelines and that their abstract description is complemented with concrete worked examples. The references that we encountered when discussing managerial culture were somewhat more diverse. They included publications in professional business journals and books akin to those of the engineering culture. However, the paradigmatic example of a recorded knowledge developed in the managerial culture would be one of the IEEE software engineering standards discussed in Chapter 4, especially the kind that does not prescribe a particular process of software development, but specifies that an organisation must define its own process that satisfies certain criteria and defines responsibilities for following the standard.[27]

The means for sharing knowledge in the humanistic culture are the most diverse. They include conventional academic publications, although they are often found in disciplines adjacent to computer science such as journals on education. They also include accessible articles in magazines intended for a broad readership, such as the 'As We May Think' essay by Vannevar Bush published in *The Atlantic*. Another typical characteristic of the culture is that it often pursues experiments with different new kinds of media. 'The Mother of All Demos' presentation delivered by Douglas Engelbart would be one typical example, while recent work in the humanistic culture includes experiments with computational media, hypertext and interactive essays.

The hacker culture departs more radically from other cultures of programming in that it often questions the very nature of programming knowledge. Hackers would argue that true mastery of programming cannot be learned by studying explicit written accounts. Instead, programming relies on tacit knowledge, gained through practice and through learning from other hackers in the community. Many of the sources from the hacker culture that I referenced in the book were internal memos that recorded important facts for those who are already a part of the hacker culture and know the relevant context. The HAKMEM collection of notes, mentioned in Chapter 1, is not only a paradigmatic example, but it also makes this status of the notes explicit. It acknowledges that the included 'items and examples are so sketchy that to decipher them may require more sincerity and curiosity than a non-hacker can muster' and it adds a caveat that some 'of this material is very inside – many readers will have to excuse cryptic references'.[28]

7.4 Aesthetics and Cultural Pointers

The five cultures of programming differ not only in how they conceive of programs and how they deal with knowledge, but also in what they consider elegant and beautiful. To investigate this aspect, I draw on the extensive analysis of the aesthetics of source code

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

```
float Q_rsqrt( float number )
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number * 0.5F;
  y  = number;
  i  = * ( long * ) &y;        // evil floating point bit level hacking
  i  = 0x5f3759df - ( i >> 1 );            // what the fuck?
  y  = * ( float * ) &i;
  y  = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
//y  = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration, this can be removed
  return y;
}
```

Figure 7.1 Source code of a function that estimates inverse square root from the video game Quake III Arena, developed by id Software in 1999. (Source: id Software GitHub[29] (file q_math.c))

by Pierre Depaz.[30] In his work, Depaz studies the role of aesthetics in understanding source code, but he also identifies what is considered elegant code by four different groups of programmers: software developers, hackers, scientists and poets. The first two correspond to the engineering and hacker cultures as used in this book, while poets are one of the many subcultures of the humanistic culture in my classification. This group of scientists includes computer scientists, who would match closely with the mathematical culture, but also other scientists who write code and are harder to align with my classification.

Many of the observations made by Depaz are directly relevant for understanding how different cultures of programming perceive the aesthetics of source code. Depaz points to the differences between what I refer to as cultures of programming when he notes that 'hacker code and engineer code look different from each other, achieve different purposes than poetic code, abide by different requirements than scientific code' and that within 'each of these conceptions, a judgement of what looks good will therefore be different'.[31]

One aspect of this different judgement is the ambiguous meaning of certain properties ascribed to code. For example, the terms 'clever' and 'obfuscated' would be seen as negative in the engineering culture of programming. The engineering culture values simplicity and directness of code. Its proponents take for granted that there are trade-offs to be made between readability and other characteristics of code such as efficiency. They also believe that readability is often preferable. Such an approach is against the core value of the hacker culture, where 'clever code' is the hallmark of a good hacker. One example of such code would be the inverse square root function used in the Quake III game, shown in Figure 7.1. The function relies on the knowledge of intricate aspects of bit representation of floating point numbers. As is typical for the hacker culture, it also does not use comments to guide the reader in understanding the code.

The humanistic culture, represented by code poets in Depaz's work, may even appreciate code that is explicitly obfuscated. This is illustrated by the code poem in Figure 7.2 which uses code obfuscation to play with the double meaning of code. On the one hand, the poem is executable C code that generates a rain-like effect. On the other hand,

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

```
#define submerge const char*_=O%239?" ":"\t;\t";O*=2654435761;int
#define _cOb8(...) int s,on,__VA_ARGS__;int main(int O, char**Q)

  _cOb8(o_,     _oO8ocQOcOb,       _ocQbo8oo,      _oO8ocOb_
    ){ ;          { ;;;             ;;}             ;{
      ;;            ;{               ; }             {;;}
     }   float   the;;  static things ;; for (;;){ us :;;
    ; ;  break; the;  ;; long grass  ;unsigned squall  ; }
   { } ; while (1){soft:;  submerge us;;    in: sleep (0) ;
    ; ;  printf    (_);   quietly :on  ;;   the; soil:; };
   {{ };           ; ; ;;             ;{ ;            }; {
  {  ;   shake: time (1) ;register   *_, the =clock(s  );
    ;} ; volatile    *_,  winds     ; ;  double wills ;{
      ;  char the    ,*   fire     ;; short companion,*_;}
    ; {  union    {}*_,  together  ;; ;   void *warms  ;}
   } ;;            ;{;              ;} ;              ;;
    ; ;  if (1) wet  :;    raise    (1);  struct{}ure  ;; ;
   ; ;   free (0);for(;;){  newborn :; ;     daughter :; ;
   ;{ ;  extern al,  **     world   ,*re;const ructed  ;};
 ;  ; ;  continue;on:;;    floods   :; ;    of: water :;};}
; ;{ ; ;          ;; {  ;          ; }  ;             }  ; ;  }
```

Figure 7.2 Code poem `water.c` by Daniel Holden and Chris Kerr. (Source: Holden and Kerr (2023))

the code can be read as a textual poem with layout indicating the flow of water. The dual nature of code, as an executable program and as a human-readable text, can be equally important in non-artistic uses within the humanistic culture. For example, work that aims to make programming more broadly accessible, such as the Logo language discussed in Chapter 3, emphasises making code as readable as possible. Aligning the two meanings of code can help this. Reading code as text should evoke, as much as possible, the behaviour that the code represents.

The aesthetic qualities appreciated by the mathematical culture are twofold. First, there is an aesthetic quality that code possesses when it clearly expresses a beautiful mathematical idea, that is, when the overhead of expressing a beautiful mathematical idea in a concrete programming language is as low as possible.[32] Second, there is an aesthetic quality that comes from the structure of the code itself. Especially a programming language is considered beautiful when it achieves a great expressivity in terms of a small number of orthogonal language features. Out of the languages that feature prominently in this book, Algol 60, Lisp and Smalltalk would be examples of programming languages that achieve this quality, while COBOL, PL/1 or C++ would be three suitable counter-examples.

Finally, the five different cultures of programming each has a set of typical references, phrases to use when discussing programming and terminology that indicate their cultural allegiance. These can be seen as *cultural pointers*, a term that has been introduced by Brian Lennon in his analysis of the etymology of the metasyntactic (placeholder) variables foo, bar, baz.[33] Lennon documents the history of the names,

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

but also points out that they serve as references to a culture of the early ARPANET (the predecessor of the Internet discussed in Chapter 3) from which they emerged. In the case of foo, bar, baz, the reference is to what I would classify as the hacker culture. By analogy to the programming language notion of a pointer, Lennon uses the term cultural pointer for such references. He includes the 'Hello world' program as another example, which is a cultural pointer shared by multiple cultures of programming.

In the preceding chapters, we encountered several examples of cultural pointers. Up until the early 1990s, the members of the hacker culture originating from MIT and the early days of the ARPANET maintained the 'jargon file' which served as a glossary of terms and slang used by hackers.[34] References to the terminology from the jargon file, as well as to the file itself, is an unambiguous link to the hacker culture of programming. Similarly, the HAKMEM memo[35] and its content are characteristic references that also serve as cultural pointers for the hacker culture.

The cultural pointers that identify the humanistic culture of programming include both particular terminology used when talking about programming and specific references. When referring to programming, the proponents of the culture often use the term 'augmenting human intellect', which is an allusion to the 1960s work of Douglas Engelbart, as well as the term 'tools for thought', which refers to a 1985 book of the same title[36] that envisions the future of 'mind-expanding technology'. A more recent example of work that utilises the cultural pointers of the humanistic culture of programming is the conference talk 'The Future of Programming' delivered by Bret Victor in 2013.[37] Victor delivers his presentation as if he were talking about the future of programming at a conference in 1973, referring to ongoing research on programming tools at the time and envisioning what the future based on those ideas might be. Of course, the envisioned future is much more exciting than what happened in reality. The references used in the talk are the typical cultural pointers that are frequently anecdotally mentioned in work that belongs to the humanistic culture of programming, including Sketchpad, Englebart's NLS and Smalltalk.

The cultural pointers that can be associated with the other cultures are perhaps less recognisable and unambiguous, but we encountered a number of them too. For the engineering culture, one cultural pointer is the phrase 'considered harmful', which refers to Edsger Dijkstra's 1968 letter 'Go to Statement Considered Harmful'.[38] The phrase has been used to criticise a range of programming practices that are seen as undesirable, ranging from global variables and comments to the concept of software lifecycle. Although used predominantly in the engineering culture, the phrase has also found uses outside programming and 'Considered Harmful' essays were themselves also considered harmful.[39] Another term that serves as a cultural pointer in the engineering culture of programming originated from the naming of the 'Responsibility-Driven Design' methodology, introduced by Rebecca Wirfs-Brock in 1989.[40] The naming was the starting point for a long series of '-Driven Design' and '-Driven Development' methodologies, culminating with the 'Test-Driven Development' of the early 2000s.

For the mathematical culture, the cultural pointers are less subtle. The proponents of the culture use the mathematical notation more often than any of the other cultures. Writing on types, for example, may use a Greek letter with a subscript such as $\tau_1 \times \tau_2$ in

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

a place where the literal program source code will contain a plain-text equivalent such as `T1 * T2`. This style dates back to the Algol 60 programming language definition,[41] which includes multiple 'levels of language'. At the level of the 'Publication Language', which is intended for communication, the specification 'admits variations of the reference language according to usage of printing and handwriting (e.g., subscripts, spaces, exponents, Greek letters)'.

7.5 Exchanging Ideas in a Pluralistic Discipline

One of the conclusions that can be drawn from this book is that programming is an inherently pluralistic discipline. The values, beliefs and methods of the five different cultures of programming have been remarkably stable over the history of the discipline and there is no reason to believe they would merge into a single unified paradigm. But if programming is pluralistic and involves different cultures with incompatible views on the nature of programs, how are those different cultures able to exchange ideas and develop new concepts?

To understand the information exchange between different cultures of programming, we can again learn from the work of Hasok Chang,[42] who talks about different *systems of practice* during the eighteenth-century chemical revolution. Systems of practice are similar to cultures of programming in that they can exist in parallel and exchange some results and practices, even though their practitioners have different basic understanding of chemistry. In the analysis of the information exchange between different systems of practice, Chang distinguishes between semantic and methodological incommensurability.[43] The former refers to the fact that the meaning of the terms employed by the competing theories varies, whereas the latter means that there are no neutral standards for the evaluation of the competing theories. In the chemical revolution, there was a minimal degree of semantic incommensurability in that proponents of different systems of practice could find a common neutral language. There was, however, a significant degree of methodological incommensurability. Different systems of practice used different *epistemic activities* to produce and improve scientific knowledge.

In the history of programming, we have seen a number of cases where different cultures of programming contributed to a single shared entity. They may have different theoretical framing for the entity, but can usually find a common language for talking about it. Different cultures of programming use different epistemic activities when developing the shared entity and this leads them in a very different direction. We encountered examples of this development throughout the entire book. They include the development of the concept of a programming language and structured programming, tests and testing as well as types and objects.

The case of types illustrates well the exchange of ideas between cultures. In the 1960s and the early 1970s, different cultures of programming had a very different theoretical conceptualisation of what a type is. To put it in somewhat simplistic terms, the hacker culture saw types as a clever compilation trick, the managerial culture saw

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

them as an electronic version of paper records, the engineering culture saw them as a language mechanism to support information hiding, while the mathematical culture saw types either as sets or as an abstract program checking mechanism. The incompatible conceptualisation did not prevent the different cultures of programming from exchanging their insights on types. As with the different systems of practice during the chemical revolution, there was a shared common language that enabled the cultures to overcome semantic incommensurability. At the same time, the different cultures of programming had different and often incommensurable methods for working with types. Mathematical proofs about types were of no interest to the managerial culture, which was more concerned with how data in records should be formatted (the PICTURE clause). Yet, the different cultures of programming were able to exchange research results that enabled new uses of types. For example, the proofs about type systems developed within the mathematical culture enabled the engineering culture to eliminate runtime checks from their implementation of abstract data types.[44] In doing so, the different cultures of programming have to overcome some degree of incommensurability, but they were generally able to co-opt practical achievements of other cultures and use them for their own purposes.[45]

Looking at the history of programming, it seems that what enables knowledge exchange between different cultures is having a shared tangible programming entity. An entity such as a programming language, a type, an object or a test may be interpreted differently by different cultures and it may be examined and further developed through different incommensurable methods. But the results of such developments often lead to specific practical outcomes that can be shared with other cultures. This way, the pluralistic mix of cultures that constitutes programming can achieve more than a single unified discipline would.

The analysis of how knowledge has been shared across cultures of programming also sheds light on cases where this did not happen. Discussing what has not happened may lead us down a slippery slope, but we encountered a number of ideas that curiously remained within the sphere of influence of a single culture of programming. One example is debugging techniques discussed in Chapter 4. Looking at descriptions of debugging tools from the 1960s, it is remarkable how those resemble standard tools that are available to programmers today. One possible explanation for this would be that debugging was never built around an entity that could be shared with other cultures, interpreted differently and developed using methodologies available to other cultures. At its heart, debugging remains a hacker practice and debugging skills remain an informal tacit knowledge, as illustrated by the fact that debugging is rarely taught in a formal manner at universities. The various technical developments around debuggers and debugging tools never attracted the attention of other cultures, perhaps because they did not provide specific entities around which formal proofs or new development methodologies could be constructed. Many other concepts and methodologies initially developed for building interactive programming tools and systems that I followed in Chapter 3 have suffered a different curse. Even though they were broadly adopted by other cultures of programming and continued to evolve outside of the hacker and humanistic cultures, their use for humanistic or hacker ideals remained confined to

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

the limits of their respective cultures. Examples range from mixins developed in the context of object-oriented programming in Lisp,[46] to user interface concepts of Smalltalk and icons from Pygmalion. They gave us new object-oriented abstractions and modern user interfaces, but did not make programming more interactive and concrete. As I showed in Chapter 3, interactive programming also keeps getting reinvented in new contexts rather than developing a stable form that could be embraced by other cultures. This may be one of the reasons why it is rare to find contributions to the vision from other cultures.

7.6 Collaborations and Struggles for Control

The discussion about how cultures exchange ideas hints at another interesting aspect of history that is revealed through the cultures of programming abstraction. We have seen that the same pattern, such as a collaboration around a shared entity, can be found repeatedly throughout the history of programming. Although building one abstraction over another risks the danger of losing even more of the subtle details of the intricate developments, I believe highlighting some of the common patterns of interaction can reveal one or two interesting aspects of the history of programming.

The first pattern of interaction is a collaboration around a shared entity. Two examples that we have encountered in this book are the birth of the concept of a programming language and the new interpretation of types that appeared at the end of 1970s in the ML language. In both cases, multiple cultures of programming contributed ideas that either gave rise to a new shared entity (programming language) or give a new meaning to an existing term (type). Referring to this interaction as a collaboration between cultures is somewhat misleading, because the term suggests a direct interaction. In both of the aforementioned cases, the collaboration was less direct. The contributing ideas were either 'in the air' or they developed gradually in publications. In both cases, however, contributions of multiple cultures were essential. In the case of programming languages, the managerial culture brought the need of portability, the hacker culture contributed early implementation techniques and the mathematical culture was able to give the idea of a programming language formal background and new description methods. In the case of ML types, multiple ideas converged in work done primarily in the mathematical culture, but their origins included managerial interest in representing records, engineering need for information hiding and mathematical motivation in the context of theorem proving.

A more common development around a single shared entity or an idea is that the entity or an idea is primarily developed by one culture of programming, but it is then co-opted for a different purpose or reinterpreted in a different way by another culture of programming. The examples of this pattern were plenty in this book and all cultures of programming can be found on both the giving and the receiving end of the development. Structured programming, which originated at the boundary between the mathematical and the emerging engineering culture was co-opted by the managerial culture as a team-structuring mechanism. Testing has gone through multiple devel-

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

opments, but its 1970s managerial interpretation as a stage in a software lifecycle gave way to the engineering interpretation in TDD. The Lisp programming language, which was developed primarily in the mathematical context as a symbol-processing language for AI was soon co-opted by the hacker culture and became the prime tool for interactive programming, while interactive graphical user interfaces developed in Smalltalk under the influence of the humanistic culture were adopted by managerially minded developers of desktop office systems. Finally, the notion of the object was co-opted even twice, first by the humanistic culture and then by the engineering culture. In most of these cases, the co-optation is accepted with reluctance on the giving end. This is illustrated by Dijkstra's remark about managerial structured programming ('How to program if you cannot')[47] and Kay's remark about the engineering take on objects ('it represents a real failure of people to understand what the larger picture is').[48] Despite such remarks, I believe that such co-optations can often be beneficial to the programming discipline as a whole, as long as the new interpretation does not completely overshadow the old one. The old interpretation certainly remained present in the case of Lisp, structured programming, testing as well as some of the later co-optations of the notion of a type. Those developments embrace the pluralistic nature of programming. The new interpretation enables new developments, using methods particular to the receiving culture, that can then enrich the concept as used by other cultures.

The third pattern of interaction between cultures that we have also encountered multiple times can be characterised as a struggle for control over a certain concept or an idea. This happens when a concept is exchanged by multiple cultures of programming, but the interpretation of the concept is important enough that the cultures struggle over which interpretation will become the broadly accepted one. Two examples of this are the struggles for control over the meaning of software engineering and interactive programming. Software engineering emerged in the wake of the 1968 NATO Software Engineering conferences as a substitute for the 'black art' of programming that overly-relied on the individual skills of hackers, but there was a struggle to decide whether the term would have managerial meaning (focused on organising teams and development processes), mathematical meaning (focused on formal methods), or the newly developing engineering meaning (focused on developing best practices, methods and tools). The struggles started at the follow-up NATO 1969 conference and continued throughout the subsequent decades.

Interactive programming can be seen as another concept that has been subject to repeated struggles. In its interpretation by the hacker and humanistic cultures, ranging from the 1960s MIT hackers to the Xerox PARC days and the early era of microcomputers, it gives users great control over how they use and program their computers. However, the later developments of the idea driven by managerial and engineering needs often restricts those freedoms, prompting the hacker and humanistic cultures to find new ways of pursuing their visions.[49]

A struggle for control seems to develop more easily around a general idea or a methodology than a specific technical entity. This can be well illustrated using the case of object-oriented programming. Although different cultures of programming

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

have a different interpretation of what an object is, they do not seem to mind if others use the same term in other ways. As discussed above, this kind of co-optation enables collaboration and new developments. But which culture owns the term 'object-oriented programming' is a subject to contentious debates. It suffices to look at two quotes from Alan Kay and Bjarne Stroustrup, who both argued for their respective interpretations of the term. In 1988, Stroustrup attempted to define the term by writing a paper 'What is object-oriented programming?'.[50] A couple of years later, in the 1997 keynote that I used in the opening of Chapter 6, Kay claims control of the term and says that 'I made up the term object-oriented, and I can tell you I did not have C++ in mind'.

Finally, the last pattern of interaction between cultures of programming that we have encountered in this book is when a basic conceptual disagreement provokes a heated debate. Unlike the cases I talked about so far, where the different cultures of programming overcome the partial incommensurability of their different ways of thinking, the conceptual disagreements have incommensurability at their core. They often result from different basic assumptions taken for granted by the different cultures of programming. The first example, explored in Chapter 2, was the debate around the possibility and the limits of formal verification of software. To the mathematical culture, which views programs as formal mathematical entities, formal verification was possible because of the very nature of programs. Different understandings of what a program is, however, led to multiple objections and counterarguments. The engineering culture highlighted the limits of formally verifying hardware, or the limits of human checking of lengthy mathematical proofs. Those counterarguments attacked the basic accepted assumptions of the mathematical culture and so they were vehemently rejected (as 'a political pamphlet from Middle Ages' or 'doing disservice to the cause of the advancement of the science of programming'). Another case of a similar conceptual disagreement that we encountered were the debates about the possibility of effective anti-ballistic missile systems in Chapter 4. There, the engineering culture developed conceptual objections to both the mathematical assumptions (a program can be proved correct), but also to the managerial assumptions (it is a device that can be precisely specified and handed out to a contractor for production).

The four different patterns show that cultures of programming provide a useful lens for looking at the history of programming. The notion lets us make sense of new sudden innovations, evolving entities and contentious debates. At the same time, it is important to keep in mind that cultures of programming are an abstraction, a simplification of a much more complex reality. As such, they highlight some aspects of the history and hide others. In particular, the way I characterised cultures of programming is primarily based on the technical side of history. I highlighted different ways of thinking about what a program is and different methodologies for working with programs, but de-emphasised the social side of the history. I did not talk about socio-economic factors and power structures that shaped the history of programming at least as much as the technical aspects. The two sides are, however, interconnected and I believe that the perspective I outlined in this book productively complements works that highlight the social aspects of the history of computing.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

For example, Nathan Ensmenger provides a detailed look at how 'computer people' shaped the history of computing in the excellent book *The Computer Boys Take Over*.[51] Ensmenger talks about how the managers in the 1960s used the rhetoric of 'software crisis' to establish control over the programming process or how focus on theory made it possible for early computer scientists to establish computer science as a respected discipline within the university. Both of these have corresponding technical developments that can be well explained as interactions between different cultures of programming. Similarly, the history of anti-ballistic missile defences by Rebecca Slayton[52] which I referred to in Chapter 4, documents the social and political aspects of the development of the disciplinary repertoire of software engineering. The disciplinary repertoire served as a tool to establish authoritative scientific voice, but it was also rooted in particular novel technical perspectives on the nature of programming that I presented as the basic assumptions of the engineering cultures.

While the technical side and social side often correspond, following one of the sides can often reveal aspects of the history of programming that may not be easy to see from the other perspective. On the one hand, social history reveals important issues that do not leave an easily discernible technical trace. The consequences of gender bias, documented for example by Mar Hicks[53] and Janet Abbate[54] do not have a direct technical counterpart in my account of the history of programming.[55] On the other hand, looking at the many cases where multiple cultures of programming contributed to a single technical idea reveals intriguing collaborations. Their social side may not seem remarkable in isolation, but would deserve to be explored and documented if we are aware of its technical consequences.

Technical history of programming can also enrich our understanding of the past if we look for technical developments that may not appear remarkable as singular events, but are noteworthy as repeated patterns. One example is the struggle of the hacker and humanistic cultures to keep programming systems open, giving users full control over their programs. The fact that this way of thinking about programming repeatedly lost to a more managerial or engineering perspective is noteworthy and can serve as a prompt to historians to study the social counterpart of those developments and, perhaps, even as a prompt to practitioners to think how this curse of interactive programming can be overcome.

7.7 Why Cultures of Programming Matter

The idea of overcoming the curse of interactive programming serves as one good example of why looking at the history of programming through the lens of cultures of programming is worthwhile. It is a situation where a fresh look at the history prompts us to think about new possible directions for the future of programming. Indeed, one of my motivations for writing this book is that I believe the perspective of cultures of programming can help us not just better understand history and uncover hidden assumptions about the discipline, but also inspire new technical work that fills the gaps that the historical perspective reveals.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

To reiterate, the first goal of this book is to offer a new perspective for looking at the history of programming. Some of the developments I have written about have been documented well by others, while some parts of this book documented history not written in detail before. But putting the wide range of diverse historical developments side by side justifies the five cultures of programming as useful abstractions. However, my interest is not just in making the history of programming accessible to a broader readership or providing a background for current computer science research. The second goal of this book is to bring clarity to and prompt new ideas within the discipline of programming. In this, I follow the complementary science method advocated by Hasok Chang. The method aims to use the history of science 'to generate and improve scientific knowledge where current science itself fails to do so'.[56] This includes critically questioning established assumptions, recovering past ideas that were abandoned and extending some of those forgotten past ideas.[57] With those two goals in mind, the abstraction of cultures of programming can be useful in multiple ways.

The first use of the concept of a culture of programming is as a tool for making sense of the history of programming. There are many questions about the history of programming that may look puzzling at first, but where cultures of programming provide a useful framework for an answer. The 1968 and 1969 NATO Software Engineering conferences have long captivated historians,[58] but why was there so much agreement about the problem at the 1968 conference and so little agreement about possible solutions at the follow-up event in 1969? Looking at the history of programming languages, how come that the notion of data types which was already around in COBOL at the end of the 1950s only made it into Algol a decade later and why is COBOL rarely acknowledged in academic writing or taught at universities?[59] Both of these questions have complex multi-faceted answers, but seeing them as interactions between multiple cultures of programming is a possible first step towards their demystification.

The second use of the concept of the culture of programming is that it reveals basic assumptions that are behind many technical decisions in programming and computer science. One interesting recent example would be the development of unsound type systems in programming languages such as TypeScript. This has met with vehement opposition from many academically inclined programmers. The debate can be easily explained as a clash between mathematical and engineering cultures that reason about programming based on incompatible basic assumptions about the discipline. Thinking about the different patterns of interaction, which I analysed earlier in this chapter, we may also wonder whether there is a potential new way of thinking about types in programming languages that would bring those two cultures closer together, much like the development of the type system for the ML programming language did in the 1970s.[60]

Thinking about programming languages and systems through the lens of cultures of programming can also inspire new perspectives. This is where my work fits with what Chang refers to as the extension function of complementary science. The most fruitful approach enabled by the cultures of programming perspective may be to take an existing important notion, such as the idea of an open or trustworthy software, and

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

view it from a perspective of a culture other than the one that currently dominates the narrative about the notion.

Interactive programming, which I mentioned already, is one example. The idea often originates either from the hacker culture, which values direct access to computers, or from the humanistic culture, which values the quick feedback loop that interactivity enables. But interactivity is at odds with the mathematical view of of programs as static linguistic entities. Could there be a different formal model of programs and programming that would support the interactivity that is valued by the humanistic culture, but that would also let the mathematical culture apply its methodology to these new kinds of programs?

In an opposite direction, we can think about the notion of trustworthy software. This is typically affiliated with the mathematical culture, which gives it a specific technical meaning as software that is proved to satisfy its formal specification. Alas, such formal meaning does not suffice for the humanistic culture of programming, which considers software systems in their broader societal impact. A machine learning algorithm may be formally correct, but still exhibit algorithmic bias. Would it be possible for the humanistic and mathematical cultures to share a notion of trustworthy software that could be used to both describe societally desirable behaviour of software systems, but also to formally guarantee it?[61]

Another case that could be reconsidered from a new perspective is open-source software. The origins of the idea can be traced to the hacker culture and its belief that information should be free.[62] Nowadays, open-source is more often seen as a practical engineering approach to building software systems and even as a managerial way to organise (often unpaid) labour. Corporations embrace open-source because it is a more effective way of building and distributing certain kinds of software. The shift can be dated back to the open-sourcing of Netscape Communicator 5.0 in 1998, which was done 'to accelerate [its] development and free distribution' with the acknowledged business aim of 'further seeding the market for Netscape's enterprise solutions'.[63] What would open software look like if the original freedom 'to study how the program works, and change it so it does your computing as you wish'[64] was interpreted by the humanistic culture of programming instead? Could the resulting programs be open to modification not just by skilled professional programmers with enough free time to delve into technical details, but also by casual users of the software?

7.8 Many Things Go

The central message of this book is that programming is an essentially pluralistic enterprise. The five different cultures of programming I identify represent a basic set of remarkably stable ways of thinking about programming that have been shaping the discipline throughout its history. I believe that this pluralism has often enabled new conceptual developments and advances that would be hard to achieve in a more unified field, but also that computer scientists and practising programmers should embrace this pluralism in the future. The discipline should do more to respect the beliefs of

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

those aligned with other cultures of programming and it should do more to encourage encounters that lead to exchange of ideas across cultural boundaries. After all, those exchanges have been vital for the development of many essential concepts that programmers today rely on. To draw my conclusion from two authors that have deeply influenced this book, the only principle that does not inhibit progress in programming is: Many things go.[65]

Notes

1 The paraphrased quote is 'The past is a foreign country; they do things differently there' from the opening of a novel *The Go-Between* by L. P. Hartley. The quote has been used by Chang (2017) to illustrate how strange past ways of thinking may appear to contemporary thinkers.
2 Chang (2012).
3 Star and Griesemer (1989).
4 But recall the time-sharing programming environment at SAIL, discussed in Chapter 4, which made such looking over the shoulder possible already in the 1970s!
5 Polanyi (1967).
6 Hoare (1972).
7 Dijkstra (1972).
8 Liskov and Zilles (1974).
9 Dijkstra (1972).
10 Fowler (1999).
11 Dijkstra (1972).
12 Hoare (1972).
13 Chang (2004).
14 Primiero (2019).
15 Tedre (2014).
16 Halvorson (2020).
17 Markoff (2005); Levy (2010); Turner (2010).
18 Mahoney (1988).
19 Mahoney (2005).
20 Priestley (2011).
21 Gabriel (2012).
22 Galison (1997).
23 For example, the encoding of a mathematical abstraction such as the functional programming notion of a 'monad' in a programming language enables the mathematical culture to exchange concepts with the engineering culture. For more on this example, see also Petricek (2018).
24 It is perhaps closer to the use of the term in the sociology of mathematics, where Livingston (1999) discusses different 'cultures of proving'. MacKenzie (2005) studies the problem in the context of proofs about computer program correctness.
25 Most likely, the desire to get as much as possible from limited hardware through clever tricks.
26 A phrase associated with the hacker culture of programming, often credited to Stewart Brand.
27 This is the case with the processes for software testing discussed in Chapter 4, but also the Rational Unified Process discussed in Chapter 6 and the Securities and Exchange Commission's rule 15c3-5 mentioned in Chapter 1.
28 Beeler et al. (1972).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

29 https://github.com/id-Software/Quake-III-Arena.
30 Depaz (2023).
31 Depaz (2023).
32 Depaz illustrates this with a number of 'beautiful algorithms' expressed in the Julia programming language.
33 Lennon (2019).
34 This was later published in a book format in multiple editions, starting from Steele (1983).
35 Beeler et al. (1972).
36 Rheingold (1985).
37 Victor (2013).
38 Dijkstra (1968).
39 Meyer (2002).
40 Wirfs-Brock and Wilkerson (1989).
41 Backus et al. (1963).
42 Chang (2012).
43 Hoyningen-Huene and Sankey (2001).
44 As illustrated by the reference to the work of John C. Reynolds in the paper on abstract data types by Liskov and Zilles (1974).
45 Here, the history of programming again resembles that of the chemical revolution. In Chang's account, different systems of practice co-opt results from other systems and share experimental activities, despite having a different theory. Chang argues that such history can be well explained by focusing on operationalisation of concepts, which is an approach that would be equally applicable to the history of programming.
46 Gabriel (2012).
47 Dijkstra (1980).
48 Kay (1997).
49 For example, the Raspberry Pi computer, launched in 2012, was built by a foundation established to 'promote the study of computer science and related topics … and to put the fun back into learning computing'. The project also made numerous explicit references (cultural pointers) to microcomputers (BBC Micro) and the culture of magazines around microcomputers.
50 Stroustrup (1988).
51 Ensmenger (2012).
52 Slayton (2013).
53 Hicks (2017).
54 Abbate (2012).
55 This is not to suggest that such biases do not affect the technology that is being created. They certainly do, but uncovering the effects may require a different approach than the one I follow in this book. One may for example start from alternative historical narratives about some of the history of computing, such as the one told by Rankin (2018).
56 Quoted from Chang (2017); Chang himself follows this approach in his two monographs Chang (2004, 2012).
57 In this book, I focus on critical questioning and recovering past ideas, but I do not aim at extension of past ideas. However, I find this an appealing proposition, especially for computer science where reconstructing past programming systems can be relatively inexpensive, and have followed it in Petricek (2020), a small project focused on Commodore 64 BASIC.
58 There is a detailed account by Pelaez Valdez (1988), while multiple historians of computing refer to it as a pivotal moment including Campbell-Kelly and Aspray (1996) and more critically Ensmenger (2012); Slayton (2013).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

59 Hicks (2020) provides social context and documents a practical consequence of the academic disregard for COBOL.
60 Some recent discussions in computer science do, in fact, point in that direction (Livshits et al., 2015).
61 One may see work on counterfactual fairness (Kusner et al., 2017) as a possible step in this direction.
62 See the excellent account by Tozzi (2017).
63 Quoted in Tozzi (2017).
64 The concept is due to Stallman (1986). I quote the modern wording from the Free Software Foundation (www.gnu.org/philosophy/free-sw.html.en).
65 I paraphrase the quote 'The only principle that does not inhibit progress is: anything goes'. from Feyerabend (1975), but using a slogan 'Many things go' that comes from Hasok Chang's discussion of scientific pluralism (Chang, 2012).

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

References

Aaron, S. (2016). Sonic pi–performance in education, technology and art. *International Journal of Performance Arts and Digital Media*, 12(2):171–178.

Abadi, M. and Cardelli, L. (1996). *A Theory of Objects*. Springer Science & Business Media.

Abadi, M., Gardner, P., Gordon, A., Mardare, R., and (eds.) (2014). Essays for the Luca Cardelli Fest. Technical Report MSR-TR-2014-104.

Abbate, J. (2012). *Recoding Gender: Women's Changing Participation in Computing*. MIT Press.

Abbott, R. J. (1983). Program design by informal English descriptions. *Communications of the ACM*, 26(11):882–894.

Adam, A. (2005). Hacking into hacking: Gender and the hacker phenomenon. In Adam, A. (ed.), *Gender, Ethics and Information Technology*, pages 128–146. Springer.

Agaram, K. (2020). Bicycles for the mind have to be see-through. In *Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal, March 23–26, 2020,* pages 173–186.

Allen, C. D., Chapman, D. N., and Jones, C. B. (1972). A formal definition of Algol 60. Technical Report 12.105, IBM Laboratory Hursley.

Allman, E. (2004). A conversation with James Gosling: James Gosling talks about virtual machines, security, and of course, Java. *Queue*, 2(5):24–33.

Altenkirch, T., Gaspes, V., Nordström, B., and von Sydow, B. (1994). A user's guide to ALF. Technical Report, Chalmers University of Technology, Sweden.

Altenkirch, T., McBride, C., and McKinna, J. (2005). Why dependent types matter. *Manuscript*, 235, available online, www.e-pig.org/downloads/ydtm.pdf.

Amin, N. and Tate, R. (2016). Java and Scala's type systems are unsound: The existential crisis of null pointers. *ACM Sigplan Notices*, 51(10):838–848.

ANSI/IEEE (1983). IEEE standard for software test documentation. Technical Report 829-1983, Institute of Electrical and Electronics Engineers.

ANSI/IEEE (1986). IEEE standard for software verification and validation plans. Technical Report 1012-1986, Institute of Electrical and Electronics Engineers.

ANSI/IEEE (1987). IEEE standard for software unit testing. Technical Report 1008-1987, Institute of Electrical and Electronics Engineers.

Ardis, M., Basili, V., Gerhart, S., Good, D., Gries, D., Kemmerer, R., Leveson, N., Musser, D., Neumann, P., and von Henke, F. (1989). ACM forum. *Communications of the ACM*, 32(3):287–ff.

Armstrong, J. (2007). A history of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, 1–26. New York. ACM Publications.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Ashenhurst, R. L. (1972). Curriculum recommendations for graduate professional programs in information systems. *Communications of the ACM*, 15(5):363–398.

Astarte, T. K. (2017). Towards an interconnected history of semantics. In *Fourth International Conference on the History and Philosophy of Computing*, 1–5, Brno, Czech Republic.

Astarte, T. K. (2019). Formalising Meaning: A History of Programming Language Semantics. PhD thesis, Newcastle University.

Astarte, T. K. and Jones, C. B. (2018). Formal semantics of Algol 60: Four descriptions in their historical context. In De Mol, L. and Primiero, G., (eds.), *Reflections on Programming Systems: Historical and Philosophical Aspects*, pages 71–141. Springer.

Atchison, W. F. (1985). The development of computer science education. *Advances in Computers*, 24:319–377.

Atkinson, B. (2016). Bill Atkinson, interviewed by Leo Laporte. https://twit.tv/shows/triangulation/episodes/247.

Augustsson, L. (1998). Cayenne: A language with dependent types. In Doaitse Swierstra, S., Oliveira, J. N. and Henriques, P. R. (eds.), *International School on Advanced Functional Programming*, pages 240–267. Springer.

Backus, J. W. (1980). Programming in America in the 1950s: Some personal impressions. In Metropolis, N., Howlett, J. and Rota G.-C. (eds.), *A History of Computing in the Twentieth Century*, pages 125–135. Elsevier.

Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., et al. (1960). Report on the algorithmic language Algol 60. *Numerische Mathematik*, 2(1):106–136.

Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., van Wijngaarden, A., and Woodger, M. (1963). Revised report on the algorithm language Algol 60. *Communications of the ACM*, 6(1):1–17.

Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Herrick, H. L., Hughes, R. A., Mitchell, L. B., Nelson, R. A., Nutt, R., Sayre, D., Sheridan, P. B., Stern, H., and Ziller, L. (1956). *Fortran: Automatic Coding System for the IBM 704 EDPM*, IBM Corporation.

Backus, J. W., Herrick, H., and Ziller, I. (1954). *Specifications for the IBM Mathematical Formula Translating System*, Fortran. preliminary report, programming research group. Applied Science Division, IBM Corporation.

Baker, T. F. and Mills, H. D. (1973). Chief programmer teams. *Datamation*, 19(12):58–61.

Bank, D. (1995). The Java Saga. *Wired*. www.wired.com/1995/12/java-saga/.

Barber, R. J. (1975). The advanced research projects agency, technical report published by ARPA, number AD-A154 363. URL: https://apps.dtic.mil/sti/citations/ADA154363. 1958–1974.

Bardini, T. (2000). *Bootstrapping: Douglas Engelbart, Coevolution, and the Origins of Personal Computing*. Stanford University Press.

Bartik, J. J. (2013). *Pioneer Programmer: Jean Jennings Bartik and the Computer That Changed the World*. Truman State University Press.

Bawden, A., Greenblatt, R., Holloway, J., Knight, T., Moon, D., and Weinreb, D. (1974). Lisp machine progress report. Technical Report AIM-444, MIT.

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional.

Beck, K. (2003). *Test-Driven Development: By Example*. Addison-Wesley Professional.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al. (2001). Manifesto for agile software development. URL: https://agilemanifesto.org/.

Beck, K. and Cunningham, W. (1989). A laboratory for teaching object oriented thinking. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 1–6, New York. ACM.

Beeler, M., Gosper, R. W., and Schroeppel, R. (1972). HAKMEM. Technical Report, AI Memo 239, MIT.

Benington, H. D. (1983). Production of large computer programs. *Annals of the History of Computing*, 5(4):350–361.

Bissell, C. (2007). Historical perspectives: The moniac a hydromechanical analog computer of the 1950s. *IEEE Control Systems Magazine*, 27(1):69–74.

Black, A. P. (2013). Object-oriented programming: Some history, and challenges for the next fifty years. *Information and Computation*, 231:3–20.

Blackwell, A. F. and Collins, N. (2005). The programming language as a musical instrument. In *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2005, Brighton, UK, June 29–July 1, 2005*, page 11. Psychology of Programming Interest Group.

Bobrow, D. G., Darley, D. L., Deutsch, L. P., Murphy, D. L., and Teitelman, W. (1967). The BBN 940 LISP system. Technical Report, Bolt Beranek and Newman, Inc.

Bobrow, D. G. and Winograd, T. (1976). An overview of KRL, a knowledge representation language. Technical Report AIM-293, Stanford Aritifical Intelligence Laboratory.

Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer*, 21(5):61–72.

Böhm, C. and Jacopini, G. (1966). Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371.

Booch, G. (1983). *Software Engineering with Ada*. Benjamin Cummings.

Booch, G. (1990). *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings.

Bornat, R. (2009). Peter Landin: A computer scientist who inspired a generation. *Higher-Order and Symbolic Computation*, 22(4):295–298.

Borning, A. (1981). The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):353–387.

Borning, A. H. (1986). Classes versus prototypes in object-oriented languages. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, pages 36–40.

Bosak, R., Clippinger, R. F., Dobbs, C., Goldfinger, R., Jasper, R. B., Keating, W., Kendrick, G., and Sammet, J. E. (1962). An information algebra: Phase 1 report – language structure group of the CODASYL development committee. *Communications of the ACM*, 5(4):190–204.

Boyer, C. (2004). *The 360 Revolution*. IBM. Online at https://archive.org/details/h42_The_360_Revolution_Chuck_Boyer.

Boyer, R. S. and Moore, J. (1983). On why it is impossible to prove that the BDX90 dispatcher implements a time-sharing system. Technical report published by NASA document number: 19840017251. URL: https://ntrs.nasa.gov/citations/19840017251.

Boyer, R. S. and Moore, J. S. (1988). *A Computational Logic Handbook*. Academic Press Professional.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Brady, E. (2017). *Type-Driven Development with Idris*. Manning Publications Company.

Brand, S. (1972). Spacewar: Fanatic life and symbolic death among the computer bums. *Rolling Stone*, (123) December 1972, 52–57.

Brand, S. and Crandall, R. (1988). The media lab: Inventing the future at MIT. *Computers in Physics*, 2(1):91–92.

Brooks, F. (1987). No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19.

Brooks Jr, F. P. (1975). *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley.w

Brown, H. (1886). Receptacle for storing and preserving papers. US Patent 352,036.

Brümmer, L. (2021). *The Hub: Pioneers of Network Music*. Kehrer Verlag.

Budd, T. A., Lipton, R. J., DeMillo, R. A., and Sayward, F. G. (1978). The design of a prototype mutation system for program testing. In Ghosh, S. P. and Liu, L. Y., (eds.), *American Federation of Information Processing Societies: 1978 National Computer Conference, June 5–8, 1978, Anaheim, CA, USA*, volume 47 of AFIPS Conference Proceedings, pages 623–629. AFIPS Press.

Burstall, R. (2000). Christopher Strachey: Understanding programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):51–55.

Bush, V. (1945). As we may think. *The Atlantic Monthly*, 176(1):101–108.

Buxton, J., Randell, B., and Committee, N. S. (1970). *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. NATO Science Committee; available from Scientific Affairs Division, NATO.

Byous, J. (1998). Happy 3rd birthday! Sun Microsystems, archived at http://web.archive.org/web/19990223195009/http://java.sun.com/features/1998/05/birthday.html (Retrieved on 19 Jan 2024).

Campbell-Kelly, M. (1980). Programming the EDSAC: Early programming activity at the University of Cambridge. *Annals of the History of Computing*, 2(1):7–36.

Campbell-Kelly, M. (1992). The airy tape: An early chapter in the history of debugging. *IEEE Annals of the History of Computing*, 14:16–26.

Campbell-Kelly, M. (2004). *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*. MIT Press.

Campbell-Kelly, M. (2011). From theory to practice: The invention of programming, 1947–51. In Jones, C. B. and Lloyd, J. L. (eds.), *Dependable and Historic Computing*, pages 23–37. Springer.

Campbell-Kelly, M. and Aspray, W. (1996). *Computer: A History of the Information Machine*. Basic Books.

Cardelli, L. (1984). A semantics of multiple inheritance. In Kahn, G., MacQueen, D. B. and Plotkin, G. (eds.), *Proc. of the International Symposium on Semantics of Data Types*, pages 51–67, Berlin, Heidelberg. Springer-Verlag.

Cardelli, L. and MacQueen, D. (1983). Polymorphism: The ML/LCF/Hope newsletter. *Polymorphism*, 1(1), http://lucacardelli.name/Papers/Polymorphism%20Vol%20I,%20No%201.pdf.

Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523.

Cardone, F. and Hindley, J. R. (2006). History of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, 5:723–817.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Cavanaugh, R. (2018). Type-checking unsoundness: Standardize treatment of such issues among typescript team/community? [Online: https://github.com/Microsoft/TypeScript/issues/9825\#issuecomment-234115900. Accessed September 10, 2018].

Ceruzzi, P. (1988). Electronics technology and computer science, 1940–1975: A coevolution. *Annals of the History of Computing*, 10(4):257–275.

Ceruzzi, P. E. (2003). *A History of Modern Computing*. MIT Press.

Chandrasekaran, S. K., Leijen, D., Pretnar, M., and Schrijvers, T. (2018). Algebraic effect handlers go mainstream (dagstuhl seminar 18172). In *Dagstuhl reports*, volume 8. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. https://drops.dagstuhl.de/entities/document/10.4230/DagRep.8.4.104.

Chang, H. (2004). *Inventing Temperature: Measurement and Scientific Progress*. Oxford University Press.

Chang, H. (2012). *Is Water H2O? Evidence, Realism and Pluralism*, volume 293. Springer Science & Business Media.

Chang, H. (2017). Who cares about the history of science? *Notes and Records: The Royal Society Journal of the History of Science*, 71(1):91–107.

Cheatham, T. E. (1969). Motivation for extensible languages. *SIGPLAN Notices*, 4(8):45–49.

Chris Brown and John Bischoff (2002). Indigenous to the net: Early network music bands in the San Francisco Bay area. http://crossfade.walkerart.org/brownbischoff/IndigenoustotheNetPrint.html [Accessed December 7, 2021].

Church, A. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68.

Cihon, P. (2019). Standards for AI governance: International standards to enable global coordination in AI research & development. Technical Report, Future of Humanity Institute. University of Oxford.

Clarke, D. G., Potter, J. M., and Noble, J. (1998). Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 48–64.

Cohn, A. (1989). The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139.

Colburn, T. R., Fetzer, J. H., and Rankin, T. L., (eds.). (2012). *Program Verification: Fundamental Issues in Computer Science*, volume 14. Springer Science & Business Media.

Conger, R. A. (1962). Certification of algorithm 58: Matrix inversion. *Communications of the ACM*, 5(6):347.

Cook, S. (1988). Impressions of ecoop'88, *Journal of Object-Oriented Programming*, 1(4):42–43. SIGS Publications Inc., New York.

Cook, W. and Palsberg, J. (1989). A denotational semantics of inheritance and its correctness. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '89, page 433–443, New York. ACM.

Cook, W. R. (2009). On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, pages 557–572.

Coquand, C., Takeyama, M., and Synek, D. (2006). An emacs interface for type directed support constructing proofs and programs. In *European Joint Conferences on Theory and Practice of Software, ENTCS*, volume 2.

Coquand, T. (2018). Type theory. In Zalta, E. N., (ed.), *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2018 edition.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Cox, B. J., Naroff, S., and Hsu, H. (2020). The origins of Objective-c at PPI/Stepstone and its evolution at NeXT. *Proceedings of the ACM on Programming Languages*, 4(HOPL).

Crawford, K. and Paglen, T. (2019). Excavating AI: The politics of images in machine learning training sets. *AI and Society*, 36(4): 1105–1116.

Dahl, O.-J. (1969). Programming languages as tools for the formulation of concepts. In Aubert, K. E. and Ljunggren, W., (eds.), *Proceedings of the 15th* Scandinavian Congress Oslo 1968, pages 18–29, Berlin, Heidelberg. Springer Berlin Heidelberg.

Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., (eds.). (1972). *Structured Programming*. Academic Press.

Dahl, O.-J., Myhrhaug, B., and Nygaard, K. (1968). Some features of the SIMULA 67 language. In *Proceedings of the Second Conference on Applications of Simulations*, pages 29–31. Winter Simulation Conference.

Dahl, O.-J. and Nygaard, K. (1966). SIMULA: An Algol-based simulation language. *Communications of the ACM*, 9(9):671–678.

Dahl, O.-J. and Nygaard, K. (2002). *Class and Subclass Declarations*, pages 91–107. Springer.

Damouth, D., Urbach, J., Mitchel, J. G., and Kay, A. (1971). Parc papers for pendery and planning purposes. Technical Report, Xerox PARC.

Dastin, J. (2018). Amazon scraps secret AI recruiting tool that showed bias against women. In Martin K. (ed.), *Ethics of Data and Analytics*, pages 296–299. Auerbach Publications.

DeMillo, R. A., Lipton, R. J., and Perlis, A. J. (1979). Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280.

De Mol, L. and Bullynck, M. (2018). Making the history of computing: The history of computing in the history of technology and the history of mathematics. *Revue de Synthèse*, 139(3–4):361–380.

Dechesne, F. and Nederpelt, R. (2012). N.G. de Bruijn (1918–2012) and his road to automath: The earliest proof checker. *The Mathematical Intelligencer*, 34(4):4–11.

DeMillo, R. A., Lipton, R. J., and Perlis, A. J. (1977). Social processes and proofs of theorems and programs. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 206–214, New York. ACM.

DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41.

Depaz, P. (2023). The Role of Aesthetics in Understanding Source Code. PhD thesis, Université Sorbonne Nouvelle, ED120 - THALIM.

DeRemer, F. and Kron, H. H. (1976). Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86.

Deutsch, L. P. and Lampson, B. W. (1967). An online editor. *Communications of the ACM*, 10(12):793–799.

Deutsch, L. P. (1967). Preliminary guide to the LISP editor. Technical Report Project Genie Document W-21, University of California, Berkeley.

Di Sessa, A. A. (1985). A principled design for an integrated computational environment. *Human-Computer Interaction*, 1(1):1–47.

Dijkstra, E. W. (1968). Letters to the editor: Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148.

Dijkstra, E. W. (1970). Notes on Structured Programming. circulated privately.

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, 15(10):859–866.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Dijkstra, E. W. (1977). A position paper on software reliability. *ACM SIGSOFT Software Engineering Notes*, 2(5):3–5.

Dijkstra, E. W. (1978). On a political pamphlet from the middle ages. *ACM SIGSOFT Software Engineering Notes*, 3(2):14–16.

Dijkstra, E. W. (1980). America's programming plight. Technical Report EWD 750, The University of Texas at Austin.

Dijkstra, E. W. (1982). How do we tell truths that might hurt? In Dijkstra, E. W. (ed.), *Selected Writings on Computing: A personal Perspective*, pages 129–131. Springer.

Dijkstra, E. W. (1988). On the cruelty of really teaching computing science. Technical Report EWD 1036, The University of Texas at Austin.

Dijkstra, E. W. (1993). In reply to comments. Technical Report EWD 1058, The University of Texas at Austin.

Dijkstra, E. W. (2002). Ewd 1308: What led to 'Notes on structured programming'. In Broy, M. and Denert, E. (eds.), *Software Pioneers*, pages 340–346. Springer.

diSessa, A. A. and Abelson, H. (1986). Boxer: A reconstructible computational medium. *Communications of the ACM*, 29(9):859–868.

Dyer, M. and Mills, H. D. (1981). Cleanroom software development. In *Sixth Annual Software Engineering Workshop*. NASA.

Ellis, T. O., Heafner, J. F., and Sibley, W. L. (1969). The GRAIL project: An experiment in man-machine communications. Technical Report, RAND Corporation, Santa Monica, CA.

Engelbart, D. C. (1962). Augmenting human intellect: A conceptual framework. Technical Report AFOSR 3223, Stanford Research Institute.

English, J. (1998). The story of the Java platform. Archived at http://web.archive.org/web/19990218201604/http://java.sun.com/nav/whatis/storyofjava.html (Retrieved on 19 Jan 2024).

Ensmenger, N. (2010). Making programming masculine. In Misa, T. J., *Gender Codes: Why Women Are Leaving Computing*, pages 115–141. IEEE Computer Society and John Wiley & Sons.

Ensmenger, N. (2012). *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. MIT Press.

Ensmenger, N. (2015). 'Beards, sandals, and other signs of rugged individualism': Masculine culture within the computing professions. *Osiris*, 30(1):38–65.

Evans, T. G. and Darley, D. L. (1965). Debug–an extension to current online debugging techniques. *Communications of the ACM*, 8(5):321–326.

Evans, T. G. and Darley, D. L. (1966). On-line debugging techniques: A survey. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 37–50. ACM.

Fedorkow, G. C. (2021). Recovering software for the whirlwind computer. *IEEE Annals of the History of Computing*, 43(1):38–59.

Fetzer, J. H. (1988). Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063.

Feyerabend, P. (1975). *Against Method*. Verso.

Finney, J. W. (1975). Safeguard ABM system to shut down: 5 billion spent in 6 years since debate. *The New York Times*.

Fisher, D. A. (1970). Control structures for programming languages. PhD thesis, Carnegie Mellon University.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Fisher, K. D., Carr, C. J., and Talbot, J. M. (1975). Computer applications in the biological sciences. Technical Report AD-A012 589, Air Force Office of Scientific Research, Advanced Research Projects Agency.

Floyd, C. (1987). Outline of a paradigm change in software engineering. In Bjerknes, G., (ed.), *Computers and Democracy: A Scandanavian Challenge*, pages 191–210. Brookfield, Gower Publishing Company.

Floyd, C., Mehl, W.-M., Resin, F.-M., Schmidt, G., and Wolf, G. (1989). Out of Scandinavia: Alternative approaches to software design and system development. *Human-Computer Interaction*, 4(4):253–350.

Floyd, R. W. (1967). Assigning meanings to program. In *Proc. Symposia in Applied Mathematics, 1967*, volume 19, pages 19–32.

Forrester, J. W. and Everett, R. R. (1990). The whirlwind computer project. *IEEE Transactions on Aerospace and Electronic Systems*, 26(5):903–910.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Fowler, M. (2001). The new methodology. *Wuhan University Journal of Natural Sciences*, 6(1):12–24.

Fox, P. (1960). LISP I programmers manual. Internal paper, MIT, Cambridge.

Freiberger, P. and Swaine, M. (1984). *Fire in the Valley: The Making of the Personal Computer*. McGraw-Hill.

Gabriel, R. P. (1996). *Patterns of Software: Tales from the Software Community*. Oxford University Press.

Gabriel, R. P. (2012). The structure of a programming language revolution. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 195–214, New York. ACM.

Gabriel, R. P., Bourbaki, N., Devin, M., Dussud, P., Gray, D., and Sexton, H. (1990). Foundation for a c++ programming environment. *Proceedings of C++ at Work*, 90.

Gabriel, R. P. and Frost, M. E. (1984). A programming environment for a timeshared system. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, page 185–192, New York. ACM.

Galison, P. (1997). *Image and Logic: A Material Culture of Microphysics*. University of Chicago Press.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson.

Gelperin, D. and Hetzel, B. (1988). The growth of software testing. *Communications of the ACM*, 31(6):687–695.

Genuys, F., (ed.). (1968). *Programming Languages: NATO Advanced Study Institute*. Academic Press.

Gilmore, Jr., J. T. (1958a). A functional description of the TX-0 computer. Technical Report 6M-4789-1, Massachusetts Institute of Technology.

Gilmore, Jr., J. T. (1958b). TX-0 direct input utility system. Technical Report 6M-5097-1, Massachusetts Institute of Technology.

Gold, B. and Simons, R. (2008). *Proof and Other Dilemmas: Mathematics and Philosophy*. MAA spectrum. Mathematical Association of America.

Goldberg, A. (2002). Oral history of Adele Goldberg, interviewed by Janet Abbate, 3 July, 2002. Technical Report Interview 591, IEEE History Center.

Goldberg, A. and Kay, A. (1976a). *Smalltalk-72: Instruction Manual*. Xerox Corporation Palo Alto.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Goldberg, A. and Kay, A. (1976b). Smalltalk-72: Instruction manual. Technical Report SSL 76-6, Xerox Corporation, Palo Alto.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing.

Goldberg, J., Kautz, W. H., Melliar-Smith, P. M., Green, M. W., Levitt, K. N., Schwartz, R. L., and Weinstock, C. B. (1984). Development and analysis of the software implemented fault-tolerance (sift) computer. Technical Report NASA Contractor Report 172146, SRI International.

Gombrich, E. H. (1961). *Art and illusion*. Pantheon Books.

Goodenough, J. B. (1975a). Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696.

Goodenough, J. B. (1975b). Structured exception handling. In Graham, R. M., Harrison, M. A., and Reynolds, J. C., (eds.), *Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto*, pages 204–224. ACM Press.

Goodenough, J. B. and Gerhart, S. L. (1975). Toward a theory of test data selection. In *Proceedings of the International Conference on Reliable Software*, pages 493–510, New York. ACM.

Gordon, M. (2000). From LCF to HOL: A short history. In Plotkin, G., Stirling, C. P. and Tofte, M. (eds.), *Proof, Language and Interaction*, pages 169–186.

Gosling, J. and McGilton, H. (1995). The Java language environment: A whitepaper. Technical Report, Sun Microsystems Computer Company.

Grad, B. (2007). The creation and the demise of VisiCalc. *IEEE Annals of the History of Computing*, 29(3):20–31.

Greelish, D. (2013). An interview with computing pioneer Alan Kay. *Time Magazine*.

Greenblatt, R. (2005). Oral history of Richard Greenblatt, interviewed by Gardner Hendrie, January 12, 2005. Technical Report X3056.2005, Computer History Museum.

Gresham-Lancaster, S. (1998). The aesthetics and history of the Hub: The effects of changing technology on network computer music. *Leonardo Music Journal*, 8(1):39–44.

Gries, D., (ed.). (1978). *Programming Methodology: A Collection of Articles by Members of IFIP WG 2.3*. Springer-Verlag, Berlin, Heidelberg.

Gschwind, M. (1952). The moniac. *Fortune Magazine*, 100–101.

Gugerty, L. (2006). Newell and Simon's logic theorist: Historical background and impact on cognitive modeling. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 50(9):880–884.

Habermann, A. N. (1969). Prevention of system deadlocks. *Communications of the ACM*, 12(7):373–ff.

Hacking, I. (1983). *Representing and Intervening: Introductory Topics in the Philosophy of Natural Science*. Cambridge University Press.

Haigh, T. (2010). Dijkstra's crisis: The end of Algol and beginning of software engineering, 1968-72. [Online; accessed August 27, 2021].

Haigh, T. and Ceruzzi, P. E. (2021). *A New History of Modern Computing*. MIT Press.

Haigh, T., Priestley, P. M., Priestley, M., and Rope, C. (2016). *ENIAC in Action: Making and Remaking the Modern Computer*. MIT Press.

Halpern, M. (1965). Computer programming: The debugging epoch opens. *Computers and Automation*, 14(11):28–31.

Halvorson, M. J. (2020). *Code Nation: Personal Computing and the Learn to Program Movement in America*. ACM Books.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Harel, D. (1980). On folk theorems. *Communications of the ACM*, 23(7):379–389.

Harrison, J., Urban, J., and Wiedijk, F. (2014). History of interactive theorem proving. *Computational Logic*, 9:135–214.

Henhapl, W. and Jones, C. B. (1978). *A formal definition of Algol 60 as described in the 1975 modified report*, pages 305–336. Springer.

Hetzel, B. (1988). *The Complete Guide to Software Testing*. QED Information Sciences, 2nd edition.

Hetzel, W., (ed.). (1973). *Program Test Methods*. Prentice-Hall.

Hicks, M. (2010). Meritocracy and feminization in conflict: computerization in the British government. In Misa, T. J. (ed.), *Gender Codes: Why Women Are Leaving Computing*, pages 95–114. IEEE Computer Society and John Wiley & Sons, Inc., Hoboken, New Jersey.

Hicks, M. (2017). *Programmed Inequality: How Britain Discarded Women Technologists and Lost Its Edge in Computing*. MIT Press.

Hicks, M. (2020). Built to last. *Logic Magazine*, (11).

Hiltzik, M. A. et al. (1999). *Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age*. HarperCollins Publishers.

Hoare, C. A. (1971). Proof of a program: Find. *Communications of the ACM*, 14(1):39–45.

Hoare, C. A. R. (1965). Record handling. *Algol Bulletin*, 21:39–69.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

Hoare, C. A. R. (1972). Notes on data structuring. In Dahl, O. J., Dijkstra, E. W. and Hoare, C. A. R. (eds.), *Structured Programming*. pages 83–174. Academic Press.

Hoare, C. A. R. (1981). The emperor's old clothes. *Communications of the ACM*, 24(2):75–83.

Hoare, C. A. R. (1996). How did software get so reliable without proof? In Gaudel, M.-C. and Woodcock, J., (eds.), *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 1–17, Springer.

Hodges, W. (2001). A history of british logic. http://wilfridhodges.co.uk/history01.pdf.

Hohpe, G. (2005). Revenge of the nerds – OOPSLA 2005. www.enterpriseintegrationpatterns.com/ramblings/37_oopsla.html.

Holden, D. and Kerr, C. (2023). *./code-poetry*. Broken Sleep Books.

Holmevik, J. (1994). Compiling simula: A historical study of technological genesis. *IEEE Annals of the History of Computing*, 16(4):25–37.

Honda, K. (1993). Types for dyadic interaction. In Best, E. (ed.), 4th International Conference on Concurrency Theory (CONCUR' 93), Hildesheim, Germany, 23–26 August, 509–523. Springer.

Hooker, S. (2021). Moving beyond 'algorithmic bias is a data problem'. *Patterns*, 2(4):100241.

Hoyningen-Huene, P. and Sankey, H., (eds.). (2001). *Incommensurability and Related Matters*. Dordrecht: Kluwer.

Huang, J. C. (1975). An approach to program testing. *ACM Computing Surveys*, 7(3):113–128.

Hudak, P., Hughes, J., Peyton Jones, S., and Wadler, P. (2007). A history of Haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 12–1–12–55, New York. ACM.

Hunt, J. (1997). *Smalltalk and Object Orientation: An Introduction*. Springer Science & Business Media.

IBM (1967). IBM System/360: PL/I subset reference manual. Technical Report S360-29, IBM Systems Reference Library.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

IBM (1960). *General Information Manual: IBM Commercial Translator*. IBM. Online at http://bitsavers.org/pdf/ibm/7090/F28-8043_CommercialTranslatorGenInfMan_Ju60.pdf.

Ichbiah, J. D., Krieg-Brueckner, B., Wichmann, B. A., Barnes, J. G. P., Roubine, O., and Heliard, J.-C. (1979). Rationale for the design of the Ada programming language. *SIGPLAN Notices*, 14(6b):1–261.

Ingalls, D. (2020). The evolution of Smalltalk: From Smalltalk-72 through Squeak. *Proceedings of the ACM on Programming Languages*, 4(HOPL).

Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. (1997). Back to the future: The story of Squeak, a practical Smalltalk written in itself. In Berman, A. M. (ed.), *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, page 318–326, New York. ACM.

Ingalls, D., Palacz, K., Uhler, S., Taivalsaari, A., and Mikkonen, T. (2008). The lively kernel a self-supporting system on a web page. In Hirschfeld, R. and Rose, K. (eds.), *Self-Sustaining Systems: First Workshop, S3 2008 Potsdam, Germany, May 15–16, 2008 Revised Selected Papers*, pages 31–50. Springer.

Ingalls, D. H. H. (1978). The Smalltalk-76 programming system design and implementation. In Szymanski, T. G. (ed.), *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, page 9–16, New York. ACM.

Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley.

Jacobson, I., Ng, P.-W., McMahon, P. E., Goedicke, M., et al. (2019). *The Essentials of Modern Software Engineering: Free the Practices from the Method Prisons!* Morgan & Claypool.

Jakubovic, J., Edwards, J., and Petricek, T. (2023). Technical dimensions of programming systems. *The Art, Science, and Engineering of Programming*, 7(3):13–1.

Johns, A. (2010). *Piracy: The Intellectual Property Wars from Gutenberg to Gates*. University of Chicago Press.

Johnson, J., Roberts, T. L., Verplank, W., Smith, D. C., Irby, C. H., Beard, M., and Mackey, K. (1989). The Xerox star: A retrospective. *Computer*, 22(9):11–26.

Jones, A. K. and Liskov, B. (1976). An access control facility for programming languages. Technical Report AFOSR-TR 76-0886, Massachusetts Institute of Technology, Laboratory for Computer Science.

Jones, C. B. and Astarte, T. K. (2016). An Exegesis of Four Formal Descriptions of Algol 60. Technical Report CS-TR-1498, Newcastle University School of Computer Science.

(abstract). In Silberber, M. Y. (ed.), *Proceedings of the November 16-18, 1971, Fall Joint Computer Conference*, AFIPS '71 (Fall), page 395–396, New York. ACM.

Kay, A. (1972b). A personal computer for children of all ages. In Donovan, J. J. and Shields, R. (eds.), *Proceedings of the ACM annual conference-Volume 1*.

Kay, A. (1974). Summary of Smalltalk message forms and intentions. Technical Report, Xerox Palo Alto Research Center.

Kay, A. (1997). The computer revolution hasn't happened yet (keynote). The 12th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, Atlanta, USA.

Kay, A. (1998). prototypes vs classes was: Re: Sun's hotspot. http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html. E-mail sent to the squeak-dev mailing list, Accessed: July 25, 2023.

Kay, A. (2003). Clarification of 'object-oriented'. http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en. E-mail exchange with Stefan Ram, Accessed: July 25, 2023.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Kay, A. and Goldberg, A. (1977). Personal dynamic media. *Computer*, 10(3):31–41.

Kay, A. C. (1968). FLEX, a flexible extendable language. Master's thesis, Utah University.

Kay, A. C. (1969). The reactive engine. PhD thesis, Utah University.

Kay, A. C. (1996). The early history of Smalltalk. *History of programming languages—II*, pages 511–598.

Kell, S. (2014). In search of types. In Bruegge, B. and Krishnamurthi, S. (eds.), *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 227–241, New York. ACM.

Kell, S. (2017). Some were meant for C: The endurance of an unmanageable language. In Torlak, E., van der Storm, T., and Biddle, R., (eds.), *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward!* Vancouver, BC, October 23–27, 2017, pages 229–245. ACM.

Kell, S. (2018). Unix, Plan 9 and the lurking Smalltalk. In De Mol, L. and Primiero, G. (eds.), Philosophical Studies Series, vol 133. *Reflections on Programming Systems: Historical and Philosophical Aspects*, Philosophical Studies Series, vol 133, 189–213.

Kelty, C. M. (2008). *Two Bits: The Cultural Significance of Free Software*. Duke University Press.

Kennedy, A. J. (1996). Programming languages and dimensions. PhD thesis, University of Cambridge, Computer Laboratory.

Kiczales, G., Des Rivieres, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press.

Kidwell, P. A. (1998). Stalking the elusive computer bug. *IEEE Annals of the History of Computing*, 20(4):5–9.

Kleiman, K. (2022). *Proving Ground: The Untold Story of the Six Women Who Programmed the World's First Modern Computer*. Hurst Publishers.

Klein, G., Andronick, J., Fernandez, M., Kuz, I., Murray, T. C., and Heiser, G. (2018). Formally verified software in the real world. *Communications of the ACM*, 61(10):68–77.

Kleiner, D. (2010). *The telekommunist manifesto*, volume 3. Institute of Network Cultures Amsterdam.

Knuth, D. (1968). *The Art of Computer Programming*, volume 1–4. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Knuth, D. E. (1973). A review of 'structured programming'. Technical Report STAN-CS-73-371, Computer Science Department, Stanford University.

Kotok, A. (2005). Oral history of Alan Kotok, interviewed by Gardner Hendrie, November 15, 2004. Technical Report X3004.2005, Computer History Museum.

Kristensen, B. B., Madsen, O. L., and Møller-Pedersen, B. (2007). The when, why and why not of the BETA programming language. In Ryder, B. and Hailpern, B. (eds.), *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 10–1.

Kuhn, T. (1962). *The Structure of Scientific Revolutions*. University of Chicago Press.

Kusner, M. J., Loftus, J., Russell, C., and Silva, R. (2017). Counterfactual fairness. *Advances in Neural Information Processing Systems*, 30.

Laan, T. D. L. (1997). The evolution of type theory in logic and mathematics. PhD thesis.

Lakatos, I. (1976). *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press.

Lampson, B. (1972). Why Alto? Technical Report, Xerox PARC. www.microsoft.com/en-us/research/wp-content/uploads/2016/11/Acrobat-1.pdf.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Landin, P. J. (1964). The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320.

Landin, P. J. (1965a). Correspondence between Algol 60 and Church's lambda-notation: Part i. *Communications of the ACM*, 8(2):89–101.

Landin, P. J. (1965b). A correspondence between Algol 60 and Church's lambda-notations: Part ii. *Communications of the ACM*, 8(3):158–167.

Landin, P. J. (1966a). A formal description of Algol 60. In Steel, T. B. Jr. (ed.), *Formal Language Description Languages for Computer Programming*, pages 266–294. North Holland Publishing Company, Amsterdam.

Landin, P. J. (1966b). The next 700 programming languages. *Communications of the ACM*, 9(3):157–166.

Latour, B. (1987). *Science in Action: How to Follow Scientists and Engineers through Society*. Harvard university press.

Lauer, P. E. (1968). Formal definition of Algol 60. Technical Report 25.088, IBM Laboratory Vienna.

Leavenworth, B. M. (1972). Programming with(out) the goto. *SIGPLAN Not.*, 7(11):54–58.

Lee, G. (2021). The three amigos, among others. *De Programmatica Ipsum*, (39).

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076.

Lehman, M. M. (1993). An interview conducted by William Aspray, IEEE History Center, September 23, 1993. Technical Report 178, The Institute of Electrical and Electronics Engineers, Inc.

Lehman, M. M. and Belady, L. A. (1985). *Program Evolution: Processes of Software Change*. Academic Press.

Lennon, B. (2019). Foo, bar, baz…: The metasyntactic variable and the programming language hierarchy. In Petricek, T., Durnova, H. and Priestley, M. (eds.), *Philosophy & Technology*. 34(1):13–32.

Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115.

Levy, S. (2010). *Hackers: Heroes of the Computer Revolution*. O'Reilly Media.

Licklider, J. C. (1969). Underestimates and overexpectations. *Computers and Automation*, 18(9):48–52.

Licklider, J. C. R. (1960). Man-computer symbiosis. *IRE Transactions on Human Factors in Electronics*, HFE-1(1):4–11.

Licklider, J. C. R. (1965). *Libraries of the Future*. MIT Press.

Lindsey, C. H. (1996a). *A History of Algol 68*, page 27–96, New York. ACM.

Lindsey, C. H. (1996b). *A History of Algol 68*, page 27–96, New York. ACM.

Liskov, B. (1993). A history of Clu. In Bergin, T. J. and Gibson, R. G. (eds.), *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 133–147, New York. ACM.

Liskov, B. and Zilles, S. (1974). Programming with abstract data types. In Leavenworth, B., (ed.), *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, New York. ACM.

Livingston, E. (1999). Cultures of proving. *Social Studies of Science*, 29(6):867–888.

Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B.-Y. E., Guyer, S. Z., Khedker, U. P., Møller, A., and Vardoulakis, D. (2015). In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Llewelyn, A. and Wickens, R. (1968). The testing of computer software. In Naur, P. and Randall, B. (eds.), *Software Engineering, Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany*, pages 7–11.

MacKenzie, D. (2004). *Mechanizing Proof: Computing, Risk, and Trust*. Inside technology. MIT Press.

MacKenzie, D. (2005). Computing and the cultures of proving. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363(1835):2335–2350.

MacKenzie, D. (2014). A sociology of algorithms: High-frequency trading and the shaping of markets. https://c.mql5.com/forextsd/forum/169/algorithms25.pdf.

MacQueen, D., Harper, R., and Reppy, J. (2020). The history of Standard ML. In Steele, G. L. Jr. (ed.), *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–100.

Madsen, O. L. and Møller-Pedersen, B. (2022). What object-oriented programming was supposed to be: Two grumpy old guys' take on object-oriented programming. In Scholliers, C. and Singer, J., (eds.), *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward!* Auckland, New Zealand, December 8–10, 2022, pages 220–239. ACM.

Magnusson, L. and Nordström, B. (1993). The ALF proof editor and its proof engine. In Barendregt, H. and Nipkow, T. (eds.), *International Workshop on Types for Proofs and Programs*, pages 213–237. Springer.

Mahoney, M. S. (1988). The history of computing in the history of technology. *Annals of the History of Computing*, 10(2):113–125.

Mahoney, M. S. (1992). Computers and mathematics: The search for a discipline of computer science. In Echeverría, J., Ibarra, A. and Mormann, T. (eds.), *The Space of Mathematics. Philosophical, Epistemological, and Historical Explorations. Berlin: Springer*, pages 349–363.

Mahoney, M. S. (1997). *Computer Science: The Search for a Mathematical Theory*. Taylor and Francis.

Mahoney, M. S. (2005). The histories of computing(s). *Interdisciplinary Science Reviews*, 30(2):119–135.

Markoff, J. (2005). *What the dormouse said: How the sixties counterculture shaped the personal computer industry*. Viking Press.

Martin, W. and Hart, T. (1964). Time sharing LISP. Technical Report AIM-67, MIT.

Martini, S. (2016). Several types of types in programming languages. In Gadducci, F. and Tavosanis, M., (eds.), *History and Philosophy of Computing*, pages 216–227. Springer International Publishing.

McBride, C. (2002). Faking it simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4–5):375–392.

McBride, C. and McKinna, J. (2004). The view from the left. *Journal of Functional Programming*, 14(1):69–111.

McBurney, P. (2009). Guerrilla logic: A salute to Mervyn Pragnell. https://vukutu.com/blog/2009/09/guerrilla-logic-a-salute-to-mervyn-pragnell/.

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

McCarthy, J. (1961). A basis for a mathematical theory of computation, preliminary report. In Bauer, W. F. (ed.), *Papers presented at the May 9–11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, pages 225–238.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

McCarthy, J. (1962). Computer programs for checking mathematical proofs. In Dekker, J. C. E. (ed.), *Proceedings of Symposia in Pure Mathematics, vol 5*. American Mathematical Society.

McCarthy, J. (1963). Towards a mathematical science of computation. In Popplewell, C. M., (ed.), *Information Processing*, pages 21–28. North-Holland Publishing Company.

McCarthy, J. (1964). A formal description of a subset of Algol. In Steel, T. B. (ed.), *Formal Language Description Languages for Computer Programming*, pages 1–12. North-Holland Publishing Company.

McCarthy, J. (1978). History of LISP. In Wexelblat, R. L. (ed.), *History of Programming Languages*, pages 173–185, ACM.

McCarthy, J. (1992). Reminiscences on the history of time-sharing. *IEEE Annals of the History of Computing*, 14(1):19–24.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., and Levin, M. I. (1962). *LISP 1.5 Programmer's Manual*. MIT Press.

McCracken, D. (1973). Revolution in programming: An overview. *Datamation*, pages 50–52.

McGonagle, J. D. (1972). Notes on the computer program test methods symposium. *SIGPLAN Notices*, 7(5):8–12.

McKenzie, J. A. (1974). TX-0 computer history. Technical Report 627, MIT RLE.

McKinsey (1968). Unlocking the computer's profit potential. Technical Report, McKinsey & Company, Inc.

Meerts, J. and Graham, D. (2010). The History of Software Testing. www.testingreferences.com/testinghistory.php.

Meyer, B. (1992). *Eiffel: The Language*. Object-Oriented Series, Prentice Hall.

Meyer, E. A. (2002). 'Considered Harmful': essays considered harmful. https://meyerweb.com/eric/comment/chech.html.

Mills, H., Dyer, M., and Linger, R. (1987). Cleanroom software engineering. *IEEE Software*, 4(5):19–25.

Milner, R. (1972). Logic for computable functions: Description of a machine implementation. Technical Report, Stanford, CA, USA.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375.

Milner, R. (1979). LCF: A way of doing proofs with a machine. In Bečvář, J., (ed.), *Mathematical Foundations of Computer Science 1979*, pages 146–159. Springer.

Milner, R. (2003). An interview with Robin Milner, interviewed by Martin Berger, September 3, 2003. https://users.sussex.ac.uk/~mfb21/interviews/milner/.

Milner, R., Tofte, M., and Harper, R. (1990). *Definition of Standard ML*. MIT Press.

Milner, R. and Weyhrauch, R. (1972). Proving compiler correctness in a mechanized logic. *Machine Intelligence*, 7(3):51–70.

Mindell, D. A. (2011). *Digital Apollo: Human and Machine in Spaceflight*. MIT Press.

Misa, T. J. (2011). *Gender Codes: Why Women Are Leaving Computing*. John Wiley & Sons.

Mitchell, J. C. (2003). *Concepts in Programming Languages*. Cambridge University Press.

Montfort, N., Baudoin, P., Bell, J., Bogost, I., and Douglass, J. (2014). *10 PRINT CHR (205.5+ RND (1));: GOTO 10*. MIT Press.

Moore, J. S. (2019). Milestones from the pure LISP theorem prover to ACL2. *Formal Aspects of Computing*, 31(6):699–732.

Morris, J. H. (1973a). Protection in programming languages. *Communications of the ACM*, 16(1):15–21.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Morris, Jr., J. H. (1973b). Types are not sets. In Ullman, J. D. (ed.), *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 120–124, New York. ACM.

Morris Jr, J. H. (1969). Lambda-calculus models of programming language. PhD thesis, Massachusetts Institute of Technology.

Mosses, P. D. (1974). The mathematical semantics of Algol 60. Technical Report Technical Monograph PRG-12, Oxford University Computing Laboratory, Programming Research Group.

Mugridge, R. (2003). Test driven development and the scientific method. In Cockburn, A. (ed.), *Proceedings of the Agile Development Conference, 2003. ADC 2003*, pages 47–52. IEEE.

Murhpy, E. M. (2013). In the Matter of Knight Capital Americas LLC Respondent. Technical Report SEC Release No. 70694, File No. 3-15570, Securities and Exchange Commission.

Musa, J. D. (1975). A theory of software reliability and its application. *IEEE Transactions on Software Engineering*, 1:312–327.

Myers, G. J., Badgett, T., and Sandler, C. (1979). *The Art of Software Testing.* John Wiley & Sons, pages 22041–3467.

Nake, F. (1971). There should be no computer art. In Metzger, G. (ed.), *Bulletin of the Computer Arts Society*, PAGE 18:1–2.

Nance, R. E. (1993). A history of discrete event simulation programming languages. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, page 149–175, New York. ACM.

National Bureau of Standards. (1984). Guideline for lifecycle validation, verification, and testing of computer software. Technical Report, U.S. Dept. of Commerce, National Bureau of Standards.

National Museum of American History (2024). Log book with computer bug. https://americanhistory.si.edu/collections/nmah_334663.

Naur, P. (1966). Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, 6(4):310–316.

Naur, P. (1978). *The European Side of the Last Phase of the Development of Algol 60*, page 92–139, New York. ACM.

Naur, P. (1993). The place of strictly defined notation in human insight. In Colburn, T. R., Fetzer, J. H. and Rankin, T. L. (eds.), *Program Verification*, pages 261–274. Springer.

Naur, P., Randell, B., Bauer, F., and Committee, N. S. (1969). *Software engineering: report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*. Scientific Affairs Division, NATO.

Neff, G. and Nagy, P. (2016). Talking to bots: Symbiotic agency and the case of Tay. *International Journal of Communication*, 10:4915–4931.

Newell, A. and Simon, H. (1956). The logic theory machine: A complex information processing system. *IRE Transactions on Information Theory*, 2(3):61–79.

Noble, S. U. (2018). *Algorithms of Oppression*. New York University Press.

Nofre, D., Priestley, M., and Alberts, G. (2014). When technology became language: The origins of the linguistic conception of computer programming, 1950–1960. *Technology and Culture*, 55(1):40–75.

Noll, A. M. (2016). The Howard Wise gallery show computer-generated pictures (1965): A 50th-anniversary memoir. *Leonardo*, 49(3):232–239.

Nooney, L. (2023). *The Apple II Age: How the Computer Became Personal*. University of Chicago Press.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Nordström, B., Petersson, K., and Smith, J. M. (1990). *Programming in Martin-Löf's Type Theory, Volume 200*. Oxford University Press, Oxford.

Norell, U. (2007). Towards a practical programming language based on dependent type theory. PhD thesis.

November, J. (2004). Linc: Biology's revolutionary little computer. *Endeavour*, 28(3):125–131.

Nygaard, K. and Dahl, O.-J. (1978). The development of the SIMULA languages. *SIGPLAN Notices*, 13(8):245–272.

O'Neil, C. (2016). *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*. Broadway Books.

Ornstein, S. (2002). *Computing in the Middle Ages: A View from the Trenches*. AuthorHouse.

Papert, S. (1980). *Mindstorms: Computers, Children, and Powerful Ideas*. Basic Books.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.

Parnas, D. L. (1985). Software aspects of strategic defense systems. *Communications of the ACM*, 28(12):1326–1335.

Passani, L. (1996). Java naysayer or realist? (letter). *Dr. Dobb's Journal*, (251):10–11.

Paulson, L. C. (1993). Designing a theorem prover. In Abramsky, S., Gabbay, D. M. and Spurr, J. (eds.), *Handbook of Logic in Computer Science (vol. 2) Background: Computational Structures*, pages 415–475.

Pelaez Valdez, M. E. (1988). A gift from Pandora's box: The software crisis. PhD thesis, University of Edinburgh.

Perlis, A. J. (1978). *The American Side of the Development of Algol*, page 75–91, New York. ACM.

Perlis, A. J. and Samelson, K. (1958). Preliminary report: International algebraic language. *Communications of the ACM*, 1(12):8–22.

Petricek, T. (2015). Against a universal definition of 'type'. In Murphy, G. C. and Steele, G. L. Jr. (eds.), *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 254–266, New York. ACM.

Petricek, T. (2017). Miscomputation in software: Learning to live with errors. *The Art, Science, and Engineering of Programming*, 1(2):14.

Petricek, T. (2018). What we talk about when we talk about monads. *The Art, Science, and Engineering of Programming*, 2(3):12.

Petricek, T. (2020). The lost ways of programming: Commodore 64 basic. Presented at HaPoC 2020.

Petricek, T. (2023). The rise and fall of extensible programming languages. Abstract presented at 7th International Conference on the History and Philosophy of Computing.

Petricek, T., Orchard, D., and Mycroft, A. (2014). Coeffects: A calculus of context-dependent computation. In Chakravarty, M. (ed.), *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 123–135, New York. ACM.

Pierce, B. (2002). *Types and Programming Languages*. MIT Press.

Pirsig, R. M. (1999). *Zen and the Art of Motorcycle Maintenance:An Inquiry into Values*. Random House.

Pitts, A. M. (2011). Lecture notes on types for Part II of the computer science tripos. www.cl.cam.ac.uk/teaching/1112/Types/typ-notes.pdf.

Plauger, P. J. (1993). *Programming on Purpose*. PTR Prentice Hall.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Pleasant, James, C. (1989). The very idea (technical correspondence). *Communications of the ACM*, 32(3):374–381.

Plotkin, G. and Pretnar, M. (2009). Handlers of algebraic effects. In Castagna, G. (ed.), *European Symposium on Programming*, pages 80–94. Springer.

Polanyi, M. (1958). *Personal Knowledge*. University of Chicago Press.

Polanyi, M. (1967). *The Tacit Dimension*. University of Chicago Press.

Postley, J. A. (1960). Letters to the editor. *Communications of the ACM*, 3(1):0.06–.

Priestley, M. (2011). *A Science of Operations: Machines, Logic and the Invention of Programming*. Springer.

Priestley, M. (2017). AI and the origins of the functional programming language style. *Minds and Machines*, 27(3):449–472.

Primiero, G. (2019). *On the Foundations of Computing*. Oxford University Press.

PROGRAMme (2022). *What is a computer program?* In De Mol, E. (ed.), title and publisher to be added at proof stage.

Project MAC (1967). Project MAC progress report III – July 1965 to July 1966. Technical Report DTIC AD-648346, MIT.

Pym, D., Spring, J. M., and O'Hearn, P. (2019). Why separation logic works. *Philosophy & Technology*, 32(3):483–516.

Radin, G. (1978). The early history and characteristics of PL/I. *SIGPLAN Notices*, 13(8):227–241.

Randell, B. (1975). System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232.

Rankin, J. L. (2018). *A People's History of Computing in the United States*. Harvard University Press.

Raymond, E. S. (1996). *The New Hacker's Dictionary*. MIT Press.

Raymond, E. S. (1997). *The Cathedral and the Bazaar*. O'Reilly Media.

Raymond, E. S. (2003). *The Art of UNIX Programming*. Addison-Wesley Professional.

Redmond, K. C. and Smith, T. M. (2000). *From Whirlwind to MITRE: The R&D Story of the SAGE Air Defense Computer*. MIT Press.

Reenskaug, T. M. H. (1981). User-oriented descriptions of smalltalk systems. *Byte, 6*, 8.

Rentsch, T. (1982). Object oriented programming. *SIGPLAN Notices*, 17(9):51–57.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., et al. (2009). Scratch: Programming for all. *Communications of the ACM*, 52(11):60–67.

Reynolds, J. C. (1974). Towards a theory of type structure. In Robinet, B., (ed.), *Programming Symposium*, pages 408–425, Springer.

Rheingold, H. (2000). *Tools for Thought: The History and Future of Mind-Expanding Technology*. MIT Press.

Richards, G., Nardelli, F. Z., and Vitek, J. (2015). Concrete Types for TypeScript. In Boyland, J. T., (ed.), *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37, pages 76–100, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Rochester, N. (1960). Foreword. In Rochester, N. (ed.), *Papers Presented at the December 13-15, 1960, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '60 (Eastern), page 1–9, New York, ACM.

Rosenwasser, D. (2018). Announcing TypeScript 3.0. https://blogs.msdn.microsoft.com/typescript/2018/07/30/announcing-typescript-3-0/.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Ross, D. T. (1961). A generalized technique for symbol manipulation and numerical calculation. *Communications of the ACM*, 4(3):147–150.

Royce, W. W. (1970). Managing the development of large software systems. In *Proceedings IEEE WESCON*, pages 1–9.

Rushby, J. M. and Von Henke, F. (1993). Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23.

Russell, B. (1903). Appendix B: The doctrine of types. In Russell, B. (ed.), *Principles of Mathematics*, pages 523–528. Cambridge University Press.

Russell, B. (1908). Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):222–262.

Russell, S. (2017). Interview with Stephen (Steve) 'Slug' Russell, conducted by Christopher Weaver, January 8, 2017. Technical Report, Video Game Pioneers Oral History Collection, Archives Center, National Museum of American History, Smithsonian Institution, Washington, DC.

Russell, S. R. (1963). Improvements in LISP debugging. Technical Report AIM-10, Stanford Artificial Intelligence Project.

Ryder, B. G., Soffa, M. L., and Burnett, M. (2005). The impact of software engineering research on modern progamming languages. *ACM Transactions on Software Engineering and Methodology*, 14(4):431–477.

Sack, W. (2019). *The Software Arts*. MIT Press.

Sack, W. (2022). Miniatures, demos and artworks: Three kinds of computer program, their uses and abuses. In De Mol, L. and Petricek, T. (eds.), *Fifth Symposium on the History and Philosophy of Programming*, HaPoP 2022, pages 21–24. HAPOC.

Sammet, J. E. (1961). Detailed description of cobol. In Goodman, R., (ed.), *Annual Review in Automatic Programming*, volume 2 of *International Tracts in Computer Science and Technology and Their Application*, pages 197–230. Elsevier.

Sammet, J. E. (1969). *Programming Languages: History and Fundamentals*. Prentice-Hall, Inc.

Sammet, J. E. (1978). The early history of COBOL. *SIGPLAN Notices*, 13(8):121–161.

Samson, P. (2017). Interview with Peter Samson, conducted by Christopher Weaver, January 9, 2017. Technical Report, Video Game Pioneers Oral History Collection, Archives Center, National Museum of American History, Smithsonian Institution, Washington, DC.

Schank, R. and Riesbeck, C. (1981). *Inside Computer Understanding: Five Programs Plus Miniatures*. Lawrence Erlbaum Associates.

Schuman, S. A. and Jorrand, P. (1970). Definition mechanisms in extensible programming languages. In Brown, D. (ed.), *Proceedings of the November 17-19, 1970, Fall Joint Computer Conference*, AFIPS '70 (Fall), pages 9–20, New York. ACM.

Schwaber, K. (1997). Scrum development process. In Sutherland, J., Casanave, C., Miller, J., Patel, P. and Hollowell, G. (eds.), *Business Object Design and Implementation*, pages 117–134. Springer.

Scott, D. S. (1993). A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1):411–440.

Seemann, M. (2021). *Code That Fits in Your Head: Heuristics for Software Engineeing*. Addison-Wesley Professional.

Shapin, S. and Schaffer, S. (2011). *Leviathan and the Air-Pump: Hobbes, Boyle, and the Experimental Life*. Princeton University Press.

Simondon, G. (2016). *On the Mode of Existence of Technical Objects*. Univocal Publishing.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Skelton, M., Pais, M., and Malan, R. (2019). *Team Topologies: Organizing Business and Technology Teams for Fast Flow*. IT Revolution Press.

Slayton, R. (2013). *Arguments That Count: Physics, Computing, and Missile Defense, 1949-2012*. Arguments that Count. MIT Press.

Smith, B. C. (1985). Limits of correctness in computers. Technical Report CSLI-85-36, The Center for the Study of Language and Information, Stanford, CA.

Smith, D. A., Raab, A., Reed, D., and Kay, A. (2002). Croquet: The user manual (draft revision 0.1). Technical Report, Viewpoints Research Institute, Glendale.

Smith, D. C. (1977). *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Birkhauser.

Smith, R. B. and Ungar, D. (1995). Programming as an experience: The inspiration for Self. In Tokoro, M. and Pareschi, R. (eds.), *European Conference on Object-Oriented Programming*, pages 303–330. Springer.

Solomon, C., Harvey, B., Kahn, K., Lieberman, H., Miller, M. L., Minsky, M., Papert, A., and Silverman, B. (2020). History of Logo. *Proceedings of the ACM on Programming Languages*, 4(HOPL):79:1–79:66.

Stallman, R. M. (1986). What is the free software foundation? *GNU Bulletin*, 1(1).

Star, S. L. and Griesemer, J. R. (1989). Institutional ecology, 'translations' and boundary objects: Amateurs and professionals in Berkeley's Museum of Vertebrate Zoology, 1907–39. *Social Studies of Science*, 19(3):387–420.

Steele, G. L., (ed.). (1983). *The Hacker's Dictionary*. Harper & Row.

Steele, G. L. and Gabriel, R. P. (1996a). The evolution of LISP (draft). www.dreamsongs.com/Files/HOPL2-Uncut.pdf.

Steele, G. L. and Gabriel, R. P. (1996b). The evolution of Lisp. In *History of Programming Languages—II*, pages 233–330. ACM

Steenson, M. W. (2022). *Architectural Intelligence: How Designers and Architects Created the Digital Landscape*. MIT Press.

Stockham, T. G. and Dennis, J. B. (1960). FLIT – Flexowriter Interrogation Tape: A symbolic utility program for the TX-0. Technical Report M-5001-23, MIT.

Stroustrup, B. (1986). *The C++ Programming Language*. Addison-Wesley.

Stroustrup, B. (1988). What is object-oriented programming? *IEEE software*, 5(3):10–20.

Stroustrup, B. (1994). *The Design and Evolution of C++*. Pearson Education.

Stroustrup, B. (1996). A history of C++ 1979–1991. In *History of Programming Languages—II*, pages 699–769. ACM.

Stroustrup, B. (2007). Evolving a language in and for the real world: C++ 1991–2006. In Ryder, B. and Hailpern, B. (eds.), *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 4–1.

Sutherland, I. E. (1963). Sketchpad: A Man-Machine Graphical Communication System. PhD thesis, MIT.

Sutherland, W. R. (1966). On-line Graphical Specification of Computer Procedures. PhD thesis, MIT, Lincoln Labs Report TR-405.

Syme, D. (2020). The early history of F#. In Steele, G. L. Jr. (ed.), *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–58.

Syme, D., Petricek, T., and Lomov, D. (2011). The F# asynchronous programming model. In Rocha, R. and Launchbury, J. (eds.), *International Symposium on Practical Aspects of Declarative Languages*, pages 175–189. Springer.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Takeuchi, H. and Nonaka, I. (1986). The new new product development game. *Harvard Business Review*, 64(1):137–146.

Talpin, J. and Jouvelot, P. (1994). The type and effect discipline. *Information and Computation*, 111(2):245–296.

Tan, C. (2020). The poetics of computer code: Tracing digital inscription in Ada Lovelace's England. *Digital Studies/Le champ numérique*, 10(1).

Tedre, M. (2014). *The Science of Computing: Shaping a Discipline*. CRC Press.

Teitelman, W. (1965). EDIT and BREAK functions for LISP. Technical Report AIM-84, MIT.

Teitelman, W. (1966). PILOT: A Step toward Man-Computer Symbiosis. PhD thesis, MIT.

Teitelman, W. (1974). INTERLISP reference manual. Technical Report, Xerox Palo Alto Research Center.

Teitelman, W. (1984). The Cedar programming environment: A midterm report and examination. Technical Report, Xerox Palo Alto Research Center.

Thacker, C. P., MacCreight, E., Lampson, B. W., F, S. R., and Boggs, D. R. (1982). *Alto: A Personal Computer*, chapter 33. McGraw-Hill.

Thagard, P. and Croft, D. (1999). *Scientific Discovery and Technological Innovation: Ulcers, Dinosaur Extinction, and the Programming Language Java*, pages 125–137. Springer.

The Free Software Foundation (2022). What is free software? www.gnu.org/philosophy/free-sw.en.html, Retrieved July 3, 2022.

Thomas, D. A. (1995). Travels with Smalltalk. web.archive.org/web/20130612055149/www.mojowire.com/TravelsWithSmalltalk/DaveThomas-TravelsWithSmalltalk.htm.

Tofte, M. and Talpin, J.-P. (1997). Region-based memory management. *Information and Computation*, 132(2):109–176.

Tomayko, J. E. (1988). Computers in spaceflight. Technical Report, NASA contractor report CR-182505, National Aeronautics and Space Administration, Scientific and Technical Information Division, Washington, DC, USA.

Tozzi, C. (2017). *For Fun and Profit: A History of the Free and Open Source Software Revolution*. MIT Press.

Tung, C. C. (1974). The 'personal computer': A fully programmable pocket calculator. *Hewlett-Packard Journal*, 25(9):2–20.

Turing, A. (1949). Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge. University Mathematical Laboratory.

Turner, F. (2010). *From Counterculture to Cyberculture*. University of Chicago Press.

Ungar, D. and Smith, R. B. (2007). Self. In *Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages*, pages 9–1.

Univac (1957). Introducing a new language for automatic programming univac flow-matic. Computer History Museum, catalog no. 102646140.

Urquhart, A. (1988). Russell's zig-zag path to the ramified theory of types. *Russell: The Journal of Bertrand Russell Archives*, Johns Hopkins University Press, 8(1–2), summer/winter 1988, 82–91.

van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., and Koster, C. H. A. (1969). *Report on the Algorithmic Language Algol 68*, pages 80–218. Springer.

Vekris, P., Cosman, B., and Jhala, R. (2016). Refinement types for TypeScript. In Berger, E. (ed.), *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 310–325, New York. ACM.

Victor, B. (2013). The future of programming. DBX Conference. https://worrydream.com/dbx/.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

von Neumann, J. and Goldstine, H. (1947). Planning and coding of problems for an electronic computing instrument, Part II, vol. 1. Technical Report, Institute for Advanced Study, Princeton.

Wadler, P. (2015). Propositions as types. *Communications of the ACM*, 58(12):75–84.

Wadler, P. and Findler, R. B. (2009). Well-typed programs can't be blamed. In Castagna, G. (ed.), *European Symposium on Programming*, pages 1–16. Springer.

Waldrop, M. M. (2001). *The Dream Machine: J.C.R. Licklider and the Revolution That Made Computing Personal*. Viking Penguin.

Ward, A., Rohrhuber, J., Olofsson, F., McLean, A., Griffiths, D., Collins, N., and Alexander, A. (2004). Live algorithm programming and a temporary organisation for its promotion. In Goriunova, O. and Shulgin, A. (eds.), *Proceedings of the README Software Art Conference*, volume 289, page 290.

Ware, W. H. (1966). *Future Computer Technology and Its Impact*. RAND Corporation.

Weinreb, D. and Moon, D. (1980). Flavors: Message passing in the LISP machine. Technical Report AIM-602, MIT.

Whitaker, W. A. (1993). Ada—the project: The DoD high order language working group. *SIGPLAN Notices*, 28(3):299–331.

Wilkes, M. (1985). *Memoirs of a Computer Pioneer*. MIT Press.

Wilkes, M. V., Wheeler, D. J., and Gill, S. (1951). *The Preparation of Programs for an Electronic Digital Computer: With Special Reference to the EDSAC and the Use of a Library of Subroutines*. Addison-Wesley Press.

Winberg, G. M. (1971). *The Psychology of Computer Programming*. Van Nostrand Reinhold.

Wirfs-Brock, A. (2013). Smalltalk at Tektronix. STIC'13. https://wirfs-brock.com/allen/files/tek/Tek-Smalltalk-history-slides.pdf.

Wirfs-Brock, A. (2020). The rise and fall of commercial Smalltalk. https://wirfs-brock.com/allen/posts/914.

Wirfs-Brock, R. and Wilkerson, B. (1989). Object-oriented design: A responsibility-driven approach. In Bosworth, G. (ed.), *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '89, page 71–75, New York. ACM.

Wirfs-Brock, R., Wilkerson, B., and Wiener, L. (1990). *Designing Object-Oriented Software*. Prentice-Hall, Inc.

Wirth, N. (1971). The programming language Pascal. *Acta Informatica*, 1(1):35–63.

Wirth, N. (1996). *Recollections about the Development of Pascal*, page 97–120, New York. ACM.

Wlaschin, S. (2018). *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#*. Pragmatic Bookshelf.

Wright, A. K. and Felleisen, M. (1994). A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94.

Xi, H. and Pfenning, F. (1999). Dependent types in practical programming. In Appel, A and Aiken, A. (eds.), *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227.

Yau, S. S. and Cheung, R. C. (1975). Design of self-checking software. In Shooman, M. L. and Yeh, R. T. (eds.), *Proceedings of the International Conference on Reliable Software*, pages 450–455, New York. ACM.

Yourdon, E. (1975). *Techniques of Program Structure and Design*. Prentice-Hall.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Yushchenko, Y. (2022). Pointers in programs on the computer MESM. For the European Virtual Computer Museum of the History of Information Technologies in Ukraine. www.icfcst.kiev.ua/MUSEUM/TXT/YuriYushchenko.pdf.

Zachary, G. P. (2018). *Endless Frontier: Vannevar Bush, Engineer of the American Century*. Simon and Schuster.

Zilles, S. N. (1973). Procedural encapsulation: A linguistic protection technique. In Graham, R. M. and Schroeder, M. D. (eds.), *Proceeding of ACM SIGPLAN - SIGOPS Interface Meeting on Programming Languages - Operating Systems*, page 142–146, New York. ACM.

Zmölnig, J. and Eckel, G. (2007). Live coding: An overview. In Jensen, K. K. (ed.), *Proceedings of the 2007 International Computer Music Conference, ICMC 2007, Copenhagen, Denmark, August 27–31, 2007*. Michigan Publishing.

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Index

'The Hub', 19
15c3-5, 7
1977 trinity, 80, 115
3D graphics, 269
60 order code, 139

a posteriori knowledge, 68
a priori knowledge, 68
A-0 system, 34
Aarhus, 177
Aaron, Sam, 124
ABM, *see* anti-ballistic missile system; anti-ballistic missile defence
abstract data type, 193, 211, 212, 215, 230, 237, 257, 263, 299
abstract idea, 60
abstract machine, 207
abstraction, 236, 288, 291–294, 300, 304
academia, 40
academic computer scientist, 195
academic paper, 71
academic publication, 44, 286, 293
acceptance condition, 155
acceptance testing, 149, 158, 165, 183, 186
accidental complexity, 23
accidental tasks, 173
accountability, 28
accounting, 25
ACL2, 6, 65
ACM, 10, 40, 44, 156, 199, 201, 268
ACM Symposium on Principles of Programming Languages, 62, 153
activity, 243
activity declaration, 243, 256
Actor model, 95
actor-oriented approach, 273
Ada package, 275
Ada, Countess of Lovelace, 88, 275
adaptation, 182
Advanced Research Projects Agency, 89, 128, 250, 278
advising, 94
aerodynamic stability, 83
aesthetics, 86, 252, 294
agent-based simulation, 78
Agile, 2
Agile development, 238, 272
Agile development methods, 163
Agile manifesto, 2, 134, 238, 277
Agile methodologies, 124, 134, 277
Agile Software Development, 18
Agile software development, 288
AH-6 helicopter, 11
AI languages, 179
air defence, 13, 84
Air Force, 145
air-traffic control, 174
aircraft, 67
Airy integral, 141
Alexander, Christopher, 182
ALF theorem prover, 222
algebraic effect handlers, 154
algebraic language, 118
Algol, 3, 11, 44, 71
  criticism, 34
  formalisation, 34
  preliminary report, 33
Algol 60 report, 291
Algol 68, 16, 54
Algol 68 Minority Report, 148, 155, 290
Algol report, 45, 94
Algol working group, 54
Algol, compiler, 48
Algorave, 3
algorithm, 4, 33, 40, 72, 125, 166, 174, 248, 293, 305
  in the Communications of the ACM, 47
  trading, 5
algorithmic bias, 4, 24, 305
Algorithmic Language, *see* Algol
algorithms, 26
alienation, 24
Allen, Paul, 115
almost a new thing, 236
Altair 8800, 80, 113, 115, 122
alter tape, 145
Amazon, 4
American Airlines, 168

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

American management philosophy, 59
American school, 260, 263, 264
amplifying human intelligence, 213
An Open Letter to Hobbyists, 115
analog computer, 12, 83, 88, 137
analog system, 170
Andersen, Arthur, 174
anecdotal evidence, 168
animation system, 107
anthropomorphic metaphor, 42
anthropomorphisation, 99
anti-ballistic missile, 171
anti-ballistic missile defence, 166–303
anti-ballistic missile software, 176
anti-ballistic missile system, 135
anti-intellectualism, 59
anything goes, 102, 306
Apollo, 2, 15, 21
Apple, 268
Apple Computer, Inc., 115
Apple II, 115, 119
applet, 271
applicability, 289, 290
application development lifecycle, 275
application software, 160
applicative expressions, 207
applied mathematics, 1
apprenticeship, 39
architecture, 286
Armstrong, Joe, 185
ARPA, *see* Advanced Research Projects Agency
ARPA community, 103, 126, 247
ARPANET, 3, 12, 103, 240, 297
array, 199
art, 122–125, 293
Art and Illusion, 110
art collectors, 97
articulate language, 110
artificial intelligence, 45, 65, 86, 87, 94, 97, 213, 256
Artificial Intelligence System, 269
artist, 18, 87
As We May Think, 18, 88, 100, 294
Aspect-Oriented Programming, 94
assembly, 58
assembly language, 67, 145, 258
assertion, 264
Association for Computing Machinery, *see* ACM
Association of Simula Users, 266
associative trail, 88
assumption, 287, 292, 304
Atkinson, Bill, 121
atomic symbol, 249
Auerbach, Al, 140
Augmentation Research Centre, 101
augmenting human intellect, 78, 87–91, 100–102, 121, 127, 247, 297
augmenting the human mind, 78
authentication problem, 210
auto-complete, 194
Autocode, 34
autognostic objects, 257
automated test runner, 183
automated testing tools, 133
Automath theorem prover, 220, 222
automatic coding, 34, 41
autonomy, 179
axiom, 52, 63
Axiom of Assignment, 52
axiomatic approach, 49
Axiomatic Basis for Computer Programming, 48

backdoor, 116
Backus, John, 16, 38, 198
Backus–Naur form, 50, 203
Ballistic Missile Early-Warning System, 67
ballistics tables, 137
bandaids and tourniquets, 167
bankloan, 245
Bartik, Jean, 36, 137
BASIC, 80, 115–118
basic assumption, 9, 77, 304
BASIC manual, 116
batch processing, 11, 19, 83, 91, 144, 214
Bauer, Friedrich L., 148
BBN, *see* Bolt Beranek and Newman
BBN-LISP, 93, 97, 147
BDFL, 186
beauty, 294
Beck, Kent, 182, 238, 269, 273
Beginner's All-purpose Symbolic Instruction Code, 115
Belady, Les, 174
belief, 290, 292, 298
Bell Labs, 12, 174
bells and whistles, 167
Bemer, Bob, 200
Benevolent Dictator for Life, 186
Berkeley, 100
Bertrand, Russell, 191
better old thing, 257, 267
bias, 24, 287
big technology companies, 79
Biggs, E. Power, 240
binding, 249
Birkbeck College, 206
bit fiddling, 192
bit representation, 295
black art, 16, 43, 55, 115, 141, 147, 148, 156, 157, 177, 181, 183, 184, 186, 198, 199, 277, 301
black art of programming, 16, 132
black box, 157
blackboard, 71
blog post, 7

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

blue book, 268
Blue Marble team, 121
BNF, *see* Backus–Naur form
Bobrow, Daniel G., 97, 256
Boehm, Barry, 176
Bolt Beranek and Newman, 88, 92, 97, 145
Booch, Grady, 274–277
Boolean logic, 214
bootcamp, 26
bootloader, 113
bootstrapping, 240
Borning, Alan, 262
boundary object, 44, 60, 184, 185, 286
Boxer, 109–112, 122
Boxer project, 261
Boyer, Robert S., 65
Boyer–Moore theorem prover, 65, 67, 214
branching point, 157
Brand, Stewart, 104, 114, 240, 250
Bravo editor, 119
break command, 144
breakpoint, 132, 144, 184
Bricklin, Dan, 119
Bright, Herbert, 168
British school, 260
Brooks' law, 134, 172
Brooks, Fred Jr., 134, 159, 171, 175
Brown, Henry, 192
bug, 133, 143
Building 26, 81, 85
Bush, Vannevar, 12, 18, 88, 100, 294
business data processing, 152, 192, 199–202
business journals, 294
Business Object Design and Implementation, 179
business record, 200
business tool, 119
business-oriented programming, 202
business-oriented programming language, 200
Byte magazine, 259
Böhm and Jacopini, 58
Böhm–Jacopini theorem, 35

C with Classes, 265
CAD system, 101
Calculus of Constructions, 223
Cambridge, 141, 213
Cambridge University, 265
Cannon, Howard, 278
Canon camera, 178
CAR, 93
Cardelli, Luca, 216, 260
Carnegie Mellon University, 224
Cartesian coordinates, 98
Cartesian product, 203
catch function, 154
category theory, 230
cathode-ray tube, 81, 83
causal effects, 11
CDR, 93
Cedar programming environment, 209
Certificate in Data Processing, 40
certification, 49, 61
Chalmers University, 223, 224
Chang, Hasok, 289–290, 298
change rate, 169, 186
changing requirements, 169, 182
character set, 267
chatbots, 1
ChatGPT, 28
checking, 143
checking mechanism, 299
checkout, 150
chemical revolution, 298
chess, 39, 174
chief programmer, 59, 64
Chief Programmer Team, 58–60, 62, 63, 161
children, 19, 97–100, 246, 255
chip, 69
Chrysler Comprehensive Compensation System, 182
Church, Alonzo, 191, 196, 203, 206
Church–Rosser theorem, 208
cigarette package, 247
City University, 213
Clark, Wes, 90
class, 236, 238, 243, 252, 262, 275
class browser, 261, 270
Class View, 227
Class-Responsibility-Collaboration (CRC), 274
Cleanroom, 161
Cleanroom methodology, 70, 218
Cleanroom Software Development, 63–66
clever code, 295
cluster, 193, 211
co-opting, 300
COBOL, 44
CODASYL, 200
code, 285, 292, 295
code block, 243
code completion, 227
code coverage, 186
code inspection, 158
code listing, 80
code of ethics, 25, 59
code poem, 295, 296
code walkthrough, 158
coding education, 289
coding vs. programming, 41, 137
cogency, 289, 290
cognitive capacity, 293
Cohn, Avra, 69
collaboration, 293, 300
colorful jargon, 167

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

combinatory logic, 209
COMEFROM statement, 59
command and control, 89, 102, 289
commercial art world, 97
commercial interest, 79, 127
commercial tool, 118
Committee on Data Systems Languages, 200
Commodore PET, 115
Common Business-Oriented Language, *see* programming language, COBOL
Common business-oriented language, *see* programming language, COBOL
Common Lisp Object System (CLOS), 279
common sense, 182
common wisdom, 168
CommonLoops, 278
communication, 100, 159, 177, 251
communication overhead, 134, 172
communication satellite, 248
communication system, 89
Communications of the ACM, 10, 40, 47, 49, 62, 68, 69, 153, 169, 207, 242
Compatible Time-Sharing System, 92, 146
CompCert, 34
CompCert compiler, 71, 225
compilation trick, 298
compiler, 41
compiler-compiler, 258
complementary science, 304
complex computer systems, 159
complex phenomena, 288
complex software, 72
complex software system, 186
complex systems, 167
complexity, 168, 238
  accidental, 23
  of computer programs, 62
component, 237, 273
component subprogram, 162
component test, 149
Composition with Lines, 96
compositional reasoning, 72
compositionality, 48, 57
computational media, 294
computer, 67, 250, 293
  ACE, 141
  Altair 8800, 80, 115
  analog, 83
  Apple II, 115
  BINAC, 140
  Burroughs 220, 247
  business applications, 13
  Commodore PET, 115
  Dynabook, 236
  EDSAC, 132, 140, 147
  ENIAC, 36, 137
  Feranti Orion, 207
  ground control, 29
  IBM 701, 199
  IBM 704, 91, 197
  IBM 7044, 145
  IBM 709, 91
  IBM 7094, 146
  installation managers, 43
  LINC, 90, 248
  Magnolia, 268
  military applications, 13
  MONIAC, 137
  onboard guidance, 21, 29
  operators, 37
  PDP-1, 12, 81–83, 91, 97, 132, 144
  Radio Shack TRS-80, 115
  tablet, 79
  TX-0, 12, 77, 81–86, 126, 132, 144
  TX-2, 89
  UNIVAC I, 41, 199
  user groups, 43
  Whirlwind, 12, 77, 83–86, 88
  Xerox Alto, 253
computer art, 20, 95–97
computer bug, 136
computer culture, 264
computer games, 118
computer industry, 62, 148
computer installation, 132
computer memory, 213
computer music, 3, 85
computer on a chip, 113
computer operator, 81, 137
computer people, 303
Computer Professionals against ABM, 168
Computer Professionals for Social Responsibility, 67
computer program, 285, 292
Computer Program Test Methods Symposium, 156
computer programming sector, 39
computer revolution, 6
computer science, 25, 30, 231, 289, 304
  curricula, 25
computer scientist, 295
computer scientists, 40
computer simulation, 241
computer, TX-2, 19
computer-related accidents, 136
Computers and Automation magazine, 147, 160
Computers and Democracy, 177
computing centre, 43
computing folklore, 9
computing industry, 159, 259
COMTRAN system, 199–201
concept stretching, 193, 218
conceptual flaw, 227
conditional, 250

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

conditional expression, 47
conference, 292
conference talk, 286
conjunction, 221
considered harmful, 297
constructive mathematics, 242
constructive proof, 223
Content Analysis Language, 102
context, 286
continuous testing, 186
contract, 237
control, 287
control structure, 273
control transfer, 56
controversial debate, 72
Cook, William R., 257, 260
Coq theorem prover, 224
Coquand, Thierry, 223
CORBA, 271
core assumption, 70, 72
core dump, 142
correctness, 150, 214
correctness guarantee, 194
correctness proof, 185
counter-culture, 12, 78, 104
counter-example, 51, 212, 230
counterfactual fairness, 4
CPAABM, 168
CPT, *see* Chief Programmer Team
craft, 72
CRC card, 274
creative freedom, 3
creative thinking, 194
creative thought, 110
creativity, 88, 98, 110
critical reflection, 96, 287
cross-machine compatibility, 43
CRT, 83, *see also* cathode-ray tube
cryptic reference, 294
CTSS, 92, *see also* Compatible Time-Sharing System
cultural marker, 292, 293
cultural pointer, 111, 294–298
culture, 292
culture of programming, 8
  interaction, 8, 9
  outputs, 17
cultures
  interactions between, 60
  meeting, 60
cultures of programming, 212, 228, 289, 303–305
  as an abstraction, 27
  bridging, 209
  clash, 118, 175, 239, 277, 287, 300–303
  interaction, 27–28, 66, 72, 95, 125, 178, 185, 193, 212, 217, 228, 230, 255, 259, 267, 277, 287, 300–303
  intersection, 87
cultures of proving, 63–66
Cunningham, Ward, 182, 238, 268, 273
Curry–Howard correspondence, 221, 222
custom operator, 258
customer, 22, 133, 182, 186, 194
customer involvement, 182
customisable syntax, 258
cybernetics, 25, 29, 42
cyberspace, 271
cyclomatic complexity, 157

Dahl, Ole-Johan, 203, 241–246
Damn Warren's Infernal Machine (DWIM), 95
Darley, D Lucille, 145
Dartmouth, 86, 91, 115
data abstraction, 260, 263, 267
data analysis, 81
data description, 200
data division, 200
data privacy, 24
data processing, 40, 201, 289
data processing industry, 39
data space, 192, 203, 212
data structure, 192, 275
data type, 136, 191, 213, 217, 291
database, 119
Datamation, 40, 59, 160
dBase, 120
DDT utility, 132, 144
de Bruijn, Nicolaas Govert, 220
de-orbit burn, 29
dead moth, 143
deadlock, algorithm correctness, 47
DEBUG tool, 145
debugger, 299
debugging, 46, 132, 133, 136–151, 156, 184–185, 268, 286, 287, 299
debugging mode, 148
DEC, 82, 268
DEC Debugging Tape, 144
decentralisation, 114
decision making, 182
decision theorist, 107
deductive argument, 69
deductive methods, 34
defensive programming, 155
degree, computer science, 25
DeMillo, Lipton and Perlis, 35, 61, 64, 68, 70, 72, 167, 218
DeMillo, Richard, 61
demonstration-oriented period, 163
Dennis, Jack, 84, 144
Department of Defense, 89
dependent function, 221, 222
dependent pair, 221, 222
dependent type, 193, 224

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

dependently typed programming languages, 225
deployment, 6, 175
deployment automation, 7
derivation rule, 63
design by contract, 264
design committee, 266
design requirement, 162
desktop metaphor, 79
destruction-oriented period, 157, 183
Deutsch, L. Peter, 92, 104, 269
developer tooling, 226–229
development
  iterative, 1
  top-down, 35
development browsers, 270
development methodology, 1, 185, 263
development process, 15, 55, 133, 149, 158, 175, 270, 301
DevOps, 6
diagnostic log, 144
differential analyser, 12, 88
differential equation, 12, 88
digital electronic computers, 137
digital system, 170
Digitalk, 269
Dijkstra, Edsger, 17, 22, 33, 40, 55, 57, 58, 60, 80, 118, 133, 150, 175, 186, 203, 277, 288–290, 300
disagreement, 287, 290
disciplinary repertoire, 166–175, 303
discipline, 298
discrete simulation, 245
discriminated union, 204
diSessa, Andrea, 111
disjunction, 221
division of labour, 14
Do What I Mean (DWIM), 94
do-it-yourself, 114
doctrine of types, 196
documentation, 149, 172, 194, 226
DoD, *see* US Department of Defense
DOD-STD-2167A, 165
domain failure, 154
domain-specific language, 102, 215
double meaning, 295
Draper Prize, 266
Dynabook, 78, 105, 106, 236, 247, 254, 257
dynamic lookup, 236
dynamic memory allocation, 244
dynamic typing, 236

E (editor), 180
E-programs, 174, 178
East coast, 100, 180
Eastern annual Joint Computer Conferences, 201
Eckert, J. Presper, 36–39
Eckert–Mauchly Computer Corporation, 140, 199
eclecticism, 231
Eclipse development environment, 272
economy, 137
ECOOP conference, 259
Edinburgh, 65, 214
Edinburgh LCF, 214
EDSAC, 34, 38, 56
education, 95, 97, 250, 288, 293
  liberal arts, 4
educational materials, 122
educator, 18, 87
Edwards, Dan, 83
effect handlers, 154
effect system, 219
efficiency, 219, 226, 295
efficient compilation, 227
Eiffel tower, 264
EJCC, 201
Electric Pencil, 119
electrical engineering, 14, 25, 289, 291
electronic brain, 42
electronic circuit, 107, 110
electronic computer system, 212
electronic music, 122–125
elegance, 294
ELIZA, 28
EMACS, 180
Emacs editor, 224, 226
empirical evidence, 168
end-user application, 120
Engelbart, Douglas, 77, 78, 100, 121, 247, 250, 294
engineering, 55, 276, 289
  traditional, 16
engineering culture, 16–18, 22, 25, 41, 54, 57, 67, 72, 139, 156, 166, 167, 174, 205, 210, 212, 217, 219, 226, 227, 230, 260, 263, 271, 274, 293, 295
engineering discipline, 16, 293
engineering enterprise, 264
engineering interest, 127
engineering knowledge, 286
engineering laboratory, 209
engineering perspective, 62
engineering practices, 16, 136
engineering reports, 156
engineering trade-off, 228
engineers, 127
English language, 200, 238, 274, 285
English, Bill, 102
English-like notation, 203
ENIAC, 10, 36
  women, 37
ENIAC programmers, 36–39, 137
ENIAC reprogramming, 139
EnS, 56
ensure clause, 265

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

enterprise software development, 273
entity, 298
environment, 168, 169, 174, 181
  forgiveness of, 23
environment division, 201
epistemic activity, 298
epistemological anarchism, 230
epistemological theory, 231
epistemology, 252
epoch of debugging, 187
Ericsson, 185
error checking, 227
error guessing, 157
error handling, 147
error message, 100
error rate, 36
error-free programs, 21
errorset function, 147, 151, 154
essential tasks, 173
Ethernet, 112
EVAL mode, 180
evaluation strategy, 219, 250
evaluation-oriented period, 165
Evans, Dave, 248
Evans, Thomas, 145
evolution of a meaning, 137
evolving environment, 186
exception, 153, 191, 264
exception handler, 147
exception handling, 151–154, 185, 264
executable specification, 183
exhaustive testing, 156, 170
existential quantification, 221
Expensive Planetarium, 91
Expensive Synthesiser, 91
Expensive Typewriter, 91
experiment, 133
exploration, 126
extensible language, 205
Extensible Languages Symposium, 282
extensible programming language, 205, 250, 258, 267
extension, 220
external problem, 293
Extreme Programming, 17, 134, 181–184, 238, 269, 272

face-to-face interaction, 186
factorial, 48, 57, 63, 111
factorial function, 46
fairness, 4
fault detection, 158
fault tolerance, 154–156
fault-tolerant programs, 153
fault-tolerant systems, 155, 185
Feranti Orion, 207
Ferranti, 174
Fetzer, James, 68
Feurzeig, Wallace, 97
Feyerabend, Paul, 105
Fifty Years' War, 277
file, 200, 205
file system, 219
filing cabinet, 192, 212
financial market, 6
financial regulations, 6
financial software, 6, 119
FIND program, 48
firing tables, 37
First Draft of a Report on EDVAC, 141
Fisher, David, 258
fixed point combinator, 209
fixed point mode, 191
fixed point type, 197
Flavors, 278
FLEX machine, 246
FLEX system, 246–251
flexibility, 237, 256
Flexowriter Interrogation Tape, 144
Flexowriter terminal, 81
flight control, 67
flight simulator, 83
FLIP, 94
Flit debugger, 84
FLIT utility, 132, 144
floating point mode, 191
floating point number, 295
floating point type, 197
floor turtle, 98
flowchart, 84
FLOW-MATIC, 42
FLOW-MATIC system, 199–201
flowchart, 158, 248
Floyd, Christiane, 134, 177, 181, 261
Floyd, Robert W., 47, 62
folk history, 246, 257
folklore, 12
foo, bar, baz, 296
foreign country, 285
forgiving environment, 168
formal definition, 51
formal elegance, 261
formal grammar, 44
formal language, 1, 8, 42, 126
Formal Language Description Languages, 207
formal logic, 191, 196–198, 208
formal methods, 6, 135–137, 218, 286, 301
formal proof, 70, 168, 194
formal reasoning, 166
formal specification, 35, 186, 305
formal system, 195, 196
formal verification, 63, 67, 69, 133, 302
  of financial algorithms, 6

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

format string, 224
Forrester, Jay, 83
FORTRAN, 10, 16
foundations of mathematics, 195–197, 222
Foyles bookshop, 206
Frankston, Bob, 119
free society, 23
free software, 13, 114
Free Software movement, 86
freedom, 301, 305
  software, 24
Frege, Gottlob, 196
functional language, 191, 195, 216
functional programmer, 218
functional programming, 214, 223, 273
fundamental knowledge, 33
fundamental law, 166
fundamental research, 201
funding
  US government, 63
future of programming, 28

Gabriel, Richard P., 94, 180
Galison, Peter, 291
Gamma, Erich, 182
garbage collection, 265
garbage collector, 244
Garmisch, 148
Garmisch, Germany, 55
Gates, Bill, 115
GDPR, 4
Gelperin, David, 156
gender bias, 33
GHC compiler, 230
giant brain (nickname), 36
Gill, Stanley, 142
Glennie, Alick, 34
glitch, 6
Glitch art, 21
GNU project, 13
Go sub statement, 116
Go to considered harmful, 17, 35
Go to statement, 116
Go to statement considered harmful, 57
go wrong, 193, 215, 217, 226
Goldberg, Adele, 19, 20, 106, 253, 254, 267
golden event, 291
Goldstine, Adele, 137
Gombrich, Ernst, 110
Goodenough, John B., 153–157
Gosling, James, 271
GOTO, 35
goto, 53
goto statement, 56, 57
gradual development, 170
Graetz, Martin, 81
GRAIL system, 90, 248

grant application, 71
graphical display, 247
graphical interface, 108
graphical user interface, 100, 110, 248, 261, 271, 301
Greek letter, 297
Green project, 271
Greenblatt, Richard, 94
Guidance and Navigation System Operations Plan, 15
guideline, 294

H-LAM/T, 101
hacker, 3, 12, 77, 295, 301
  microcomputer, 126
  MIT, 126
hacker community, 97
hacker culture, 11–13, 24, 41, 57, 72, 83, 93, 145, 148, 179–182, 198, 220, 226, 230, 246, 252, 265, 266, 277, 286, 289, 291, 293, 295, 297, 300
hacker ethic, 83
hacker practice, 181, 185, 299
HAKMEM, 12, 86, 105, 252, 294
HAKMEM memo, 297
Halpern, Mark, 147, 184
halting problem, 166, 186
hardware, 126, 250, 302
hardware architecture, 272
hardware device, 66
hardware errors, 38
hardware failure, 140, 185
hardware fault, 21
hardware verification, 216
Harvard, 172
Harvard Business Review, 178
Harvard Mark II, 143
heavyweight process, 238
Heidegger, 4
Heideggerian hammer, 29
Hello world, 297
Hetzel, Bill, 156
heuristic, 65, 86
Hewitt, Carl, 95
Hewlett-Packard, 268
hierarchical structure, 59
hierarchy of types, 196
high modernity, 24, 187
high school, 107
high-integrity system, 237, 264
high-level programming language, 172, 174, 184, 241, 258
higher-kinded quantification, 220
Hindley–Milner algorithm, 210, 215
Hingham Institute, 81
historical actor, 290
historical period, 285

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

history of programming, 288–290
history of science, 304
history of types, 229
Hoare triple, 48, 56, 65, 265
Hoare, C. A. R., 47, 52, 60, 70, 135, 156, 203, 242, 244, 288–289
hobbyist, 123, 289
hockey, 271
Homebrew Computer Club, 112–115
homoiconicity, 146
Hopper, Grace, 34, 42, 143, 199
hospital simulation, 107
hostile environment, 168
Huang, J. C., 157
Huet, Gerard, 223
human augmentation, 289
human computers, 10
human factors, 89
human–computer interaction, 136, 177
human-readable text, 296
humanistic culture, 18–20, 23, 24, 87, 100, 105, 107, 120, 178, 180, 246, 250–254, 277, 293, 295, 297, 301, 305
hydraulic devices, 137
hydrogen bomb, 37
HyperCard, 120
hypertext, 88, 294

i, 47
I/O, 54
IAL, 44, *see also* International Algorithmic Language
IBM, 35, 53, 59, 63, 86, 118, 148, 152, 155, 161, 171, 174, 200
IBM 704, 42, 87
IBM OS/360, 159
IBM Research, 278
M Visual Age, 270
IBM System/360, 152
IBM Vienna lab, 153
IBM VisualAge, 272
IBM Visual Age Smalltalk, 270
icon, 110
IDE, *see* Integrated Development Environment
idiosyncratic typography, 205
IEEE, 165
IEEE Computer, 173
If you See What I Mean, 208
IFIP WG 2.3, 54
IFIP Working Group 2.1, 204
image tradition, 291
immediate feedback, 225
imperative programming, 274
Imperial College, 174
implacable sceptic, 65
implementation, 244
incommensurability, 23, 27, 87, 236, 285, 291, 298, 302
Incompatible Timesharing System, 146
incremental compilation, 133
incremental models, 176
indirect addressing, 141
Indo-European languages, 248
industrial factories, 161
industrial production line, 161
industrial research, 54
industrial-grade component, 264, 265
industry, 62
industry crisis, 39, 148, 259, 263
industry practice, 132
industry standard, 165
Infer Static Analyser, 65
inference rule, 193, 215
informatics, 25
information exchange, 298
information hiding, 136, 153, 169, 193, 210, 273, 299, 300
Information is free, 181
information processing management, 160
Information Processing Techniques Office, 89, 102, 128
information science, 25
information systems, 25
Ingalls, Dan, 108, 253, 254
ingenious tricks, 61
inheritance, 236, 237, 252, 255, 256, 260, 263
inheriting, 245
initial orders, 142
Institute of Electrical and Electronics Engineers, 165
instruction-by-instruction, 142
instruction-by-instruction debugging, 132
integer type, 197
integral equations, 60
integrated computational environment, 112
Integrated Development Environment, 78
integration testing, 15, 165, 183
Intel, 113, 268
Intel 4004, 113
Intel 8080, 113
Intel iAPX 432, 259
interaction
  collaboration, 18
  struggle for control, 18
interactive computing, 132
interactive editor, 225
interactive essay, 294
interactive games, 122
interactive programming, 77–81, 174, 179, 225, 291, 300, 301, 303
interactive programming environments, 182
Interactive Software Engineering, 264
interactive terminal, 116

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

interactive theorem proving, 127, 220–226
interdisciplinary, 293
Intergalactic Computer Network, 240
interim Dynabook, 106
interim language, 200
Interlisp, 94, 109, 117
internal memo, 294
International Algorithmic Language, 199, *see also* IAL
international standard, 286
International Symposium of Formal Methods, 135
Internet, 12, 103, 134, 240, 272, 297
interpreter, 41, 51, 146
interpretive checking routine, 142
interpretive routines, 34
intuitionistic logic, 195, 221
invalid program, 210
inverse square root, 295
iPad, 106
IPTO, *see* Information Processing Techniques Office
irrelevant detail, 289
ISO C++ standard, 266
ITS, *see* Incompatible Timesharing System

Jacobson, Ivar, 276
jargon file, 297
Jennings, Betty Jean, *see* Bartik, Jean
job interview, 26
Jobs, Steve, 115
Joint Computer Conference, 100
journal, 292
Joy, Bill, 271
juggling, 100
JUnit framework, 272
Jupyter notebook, 81

Kay, Alan, 19, 78, 105–109, 239–241, 246–250, 254, 278, 300, 301
Kemeney, John G., 115
Kennedy, Andrew, 229
kernel, 67
kilogram, 229
KIM-1 microcomputer, 122
kitchen sink, 167
Knight Capital, 1, 5, 8, 21, 23, 65
Knight, Thomas, 94
knowledge, 39–41, 83, 179, 286, 292–294, 298–300
  practical human, 4
knowledge exchange, 299
Knowledge Representation Language, 256
Knuth, Donald, 40, 220, 242
Kotok, Alan, 81, 144
Kuhn, Thomas, 217, 290
Kurtz, Thomas E., 115

labour crisis, 14, 33, 34
Lakatos, Imre, 61, 212
lambda calculus, 34, 45, 52, 87, 191, 193, 195, 203, 206, 221, 222, 250, 260
Lambda Days Conference, 195
lambda notation, 93
Lambda the Ultimate, 95
Lamport, Leslie, 62
Lampson, Butler, 119
Landin, Peter, 206, 250
language metaphor, 42
language safety, 226
language specification, 293
large corporations, 59
large language models, 28
large software systems, 156, 273
large-scale programming, 136
laser printer, 112
Latour, Bruno, 267
laws of physics, 60
lazy evaluation, 219
LCF, 65, 214
LCF theorem prover, 193, 220
learning, 126, 177, 179, 181, 293
legacy system, 44
legal constraint, 287
legibility, 165
Lehman, Meir, 174
Lensman series, 81
letters to the editor, 62
levels of language, 298
lexical scoping, 95
liberal arts, 4
Library of Congress, 82
library problem, 88
Licklider, J. C. R., 88, 101, 103, 121, 168
lifecycle, 1, 275, 276
light pen, 19, 81, 83
lightweight formal method, 193, 218
lightweight specification, 133, 225
limits of software, 185
line numbers, 117
linear equation, 116
linguistics, 42
Linux, 14
Lipton, Richard, 61, 158
Liskov, Barbara, 153, 211, 256, 263, 288
LISP, 45, 65
Lisp, 92–95
LISP 1.5, 151, 154
LISP editor, 92–95, 126, 146, 180
Lisp machine, 95, 179, 278
Lisp system, 184
Lisp wizard, 180
Lisp-Flexo system, 92
list processing, 45, 86, 92, 98
literacy, 1, 18, 293
live coder, 125

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

live coding, 3, 21, 80, 122–125
live editing, 238
Lively Kernel, 239
Lockheed, 148
logging, 142
logic, 66, 213, 291
Logic of Computable Functions, 214
Logic Theorist, 65, 86
logic tradition, 291
logical argument, 169
logical paradox, 193
logical proposition, 221
logical tradition, 193
logical type, 213, 217
Logo, 19
love, 100
Lovelace, Ada, *see* Ada, Countess of Lovelace
LSD, 78, 104, 121
LSD trip, 104
Lucid Energize, 227

machine, 288
machine checking, 72
machine independence, 8, 43
machine instruction, 57
machine language, 42, 258
machine learning, 305
Machine-Oriented Language, 102
Macintosh, 120
MacKenzie, Donald, 136
macro, 259
Macro assembler, 84
magazine, 80, 294
magnetic core memory, 84
Magnolia, 268
mailing list, 292
mainframe computer, 118
malleability, 263
man–computer symbiosis, 78, 88, 94, 194
man–machine symbiosis, 121, 127
management, 59
management process, 132
management structure, 64
manager, 303
managerial concept, 193
managerial culture, 13–15, 22, 25, 41, 56, 105, 114, 143, 148, 159–166, 179, 212, 275, 278, 285, 291, 292, 300
managerial interest, 127
managerial method, 62, 273
managerial reports, 156
managers, 13, 59, 127
Managing the Development of Large Software Systems, 163
manufacturing, 241
many things go, 306
map function, 219
mapping, 180
Mariner 1 spacecraft, 159
Martin-Löf, Per, 222
masculinisation
  of software engineering, 29
master programmer (ENIAC), 38
mathematical
  entity, 1
  thinking, 1
mathematical abstraction, 67
mathematical analysis, 170
mathematical culture, 10–11, 21, 41, 57, 62, 63, 67, 87, 105, 118, 136, 148, 151, 152, 155, 156, 195, 203, 205, 206, 209, 212, 217–219, 222, 225, 226, 228, 242, 260, 285, 291, 292, 296, 300, 302, 305
  characteristics, 10
  subcultures, 70
mathematical entity, 68, 285, 292
mathematical idea, 296
mathematical law, 169
mathematical logic, 206
mathematical methods, 45, 71, 72, 156, 290
mathematical model, 68, 222
mathematical notation, 285, 297
mathematical object, 192
mathematical problem, 287
mathematical proof, 61, 66, 67, 169, 302
mathematical puzzle solvers, 39
mathematical reasoning, 136
mathematical relation, 229
mathematical rigour, 66
mathematical software, 269
mathematical structure, 202
mathematical theorem, 166
mathematical verification, 64
mathematics, 98, 289
Mauchly, John, 36–39
McCabe, Thomas J., 157
McCarthy, John, 45, 46, 51, 64, 71, 86, 179, 191, 202, 213, 290
McCracken, Daniel, 168
McCullough, Paul, 268
McGonagle, David J., 156
McKinsey, 15
McKinsey & Company, 160
McLuhan, Marshall, 105, 106, 240
McNealy, Scott, 271
meaning, 292
mechanical calculator, 36
mechanical checking, 186
mechanical theorem prover, 64, 70
mechanised proof, 70
mechanised theorem proving, 70
media theory, 18, 107
media,personal dynamic, 19

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

medical systems, 90
medium, 126, 250–254
  for communication, 19
medium for communication, 107
medium for thinking, 18
memex, 88
memory address, 116
memory dump, 144
memory location, 116
memory management, 219, 264
memory region, 230
memory safety, 219
Menkman
  Rosa, 22
Menlo Park, 100, 114
Merry-Shapiro, Diana, 20
message, 250, 261
message sending, 236, 255
messaging, 257
meta-circularity, 87
meta-class, 278
meta-language, 51
meta-medium, 78, 107, 237
meta-object protocol, 279
metalanguage, 214
metaphor, 240
metasyntactic variable, 296
meter, 229
method, 298
methodological incommensurability, 298
methodology, 174
methods
  rigorous, 16
Meyer, Bertrand, 264
Micro-Soft, 115
Microalgol, 51, 71
microcoded CPU, 67
microcomputer, 13, 19, 80, 112–122, 301
microcomputer industry, 119
microcomputer kit, 113
microfilm, 18, 88
microphysics, 272, 291
microprocessor, 67
Microsoft, 228
Microsoft Tay, 28
Microsoft Visual C++ 4.0, 227
Microsoft Word, 119
microworld, 19, 98
Middle Ages, 62
middle school, 107
middleware, 271
MIDI, 124
military, 36, 37, 83, 88, 274
military research, 241
military software, 170
Mills, Harlan, 63, 161, 172
Milner, Robin, 65, 213–217
mind-expanding technology, 297
Mindstorms, 99
minicomputer, 179
Minority Report, 54–56, 205
Minsky, Marvin, 97
missile defence, 166–171
missile trajectories, 169
mission-critical systems, 34
MIT, 3, 11, 14, 77, 81, 83, 88, 168, 180, 209, 248, 263, 278, 290, 301
MIT Artificial Intelligence Lab, 97
MIT Computation Centre, 91
MIT hackers, 24, 126, 144
MIT Lincoln Laboratory, 84, 132
MIT Media Lab, 125
MIT Research Laboratory of Electronics, 77, 84, 144
MIT Tech Square, 12
MITS, 113
mixin, 300
ML (meta-language), 65
ML paradigm, 225, 260
ML type system, 217
ML under UNIX, 216
ML under VMS, 216
mlchanges.doc, 216
mode, 191, 197–199, 205, 212
mode constructor, 205
mode declarer, 205
modelling, 237, 241, 260
modelling language, 260
modernity, 187
Modular Programming Language, 209, 212, 210
modularity, 255
module, 273
module system, 193
modus ponens, 86
molecular biology, 240
Mondrian, Piet, 96
monitoring, 16
Monte Carlo Compiler, 241
Monte Carlo method, 241
moon, 67
Moon, David, 278
Moore School of Electrical Engineering, 39
Moore, J. Strother, 65
Mordor, 24
morph, 262
Morris, James H., 209–212
MOS 6502, 123
Mosses, Peter, 52
mouse, 100
multidisciplinary team, 179
music instrument, 124
music synthesis, 81
music synthesiser, 253
musical actor, 122

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

musical instrument, 20
mutation testing, 62, 159
Myers, Glenford, 157

naive realism, 111
Nake, Frieder, 96
napkin, 276
NASA, 15, 68, 101
Nash, John, 149
National Bureau of Standards, 165
National Institutes of Health, 90
National Security Agency, 157
NATO 1969 Software Engineering conference, 57
NATO Advanced Study Institute, 244
NATO Conference on Software Engineering, 16, 290
NATO Science Committee, 148
NATO Software Engineering conference, 55, 60, 133, 148, 153, 156, 160, 167, 174, 183, 276, 301, 304
NATO Study Group on Computer Science, 148
natural deduction, 195
natural language, 98, 285
Naughton, Patrick, 271
Naur, Peter, 47, 175
NBS, 165
NDC conference, 290
nested boxes, 111, 261
Netscape Communicator 5.0, 305
network, 219
neural network, 4
neurophysiological research, 90
New Flavors, 278
New York Stock Exchange, 5
New York Times, 59
Newcastle University, 155
Newell, Allen, 65, 86
ninth floor, 12
NLS, *see* oN-Line System
No Silver Bullet, 173
Noll, A. Michael, 96
non-computer-based testing, 158
non-hacker, 294
non-linear interaction, 173
Nonaka, Ikujiro, 178
Nordström, Bengt, 222
normal operational range, 170
normal science, 217
Norway, 260
Norwegian Computing Center, 241
notebook, 19
NoteTaker, 108, 254
noun, 238, 274, 275
nuclear launch site, 168
nuclear reactor, 241
nuclear strike, 14, 162
nuclear weapons, 170
numerical analysis, 25
numerical codes, 41
numerical operator, 198
Nygaard, Kristen, 241–246, 260

Oak programming system, 271
Oakland, 122
obfuscation, 295
object, 236, 241, 245, 250, 257, 262, 273, 286, 287, 302
object browser, 108
Object Management Group (OMG), 271, 276
object member, 226
Object Modeling Technique (OMT), 276
object of beauty, 10
Object Oriented Zoned Environment, 253, 256
object-oriented design, 238, 274
object-oriented language, 219
object-oriented programming, 80, 89, 108, 204, 226, 236, 254, 256, 264, 267, 301
Object-Oriented Programming, Systems, Languages and Applications, *see* OOPSLA Conference
Object-Oriented Software Engineering (OOSE), 276
object-oriented system, 182
object-to-think-with, 98
obscurantism, 124
office of the future, 79, 105, 112
office system, 301
ON construct, 152
on-line debugging, 144, 172, 179, 184
oN-Line System, 77, 100, 109
OOPSLA conference, 179, 182, 239–241, 273, 278
OOZE, *see* Object Oriented Zoned Environment
open-source, 186
open-source software, 305
operating system, 116, 174
  OS/360, 152, 171
  UNIX, 3, 12
  verification, 67
operational specification, 162
operations research, 241
operator, 11, 37
OPS-3 programming system, 209
oral tradition, 185
order codes, 140
order of magnitude improvement, 173
organ, 240
orthogonal design, 205
orthogonality, 296
OS/360, 152, 171, 174
oscilloscope, 268
over-engineering, 16, 132, 136, 154
ownership type, 219

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

P-programs, 174
package, 264
pair programming, 17, 124, 181, 182
pair type, 221
Palo Alto Research Centre, *see* Xerox PARC
Panel on Computing in Support of Battle Management, 169
paper record, 191, 299
Papers We Love, 16
Papert, Seymour, 97, 105
paradigm change, 177
paradigm shift, 94, 279
paradigmatic achievement, 1, 291
paradox, 191, 196
parameter testing, 162
PARC, *see* Xerox PARC
PARC Papers for Pendery and Planning Purpose, 112
ParcPlace Systems, 254, 269, 271
Paris, 50
Parnas, David L., 169, 173, 175, 210
participatory design, 134, 163, 177
Passmore, Grant, 6
past, 285
past idea, 304
pattern, 261
pattern of interaction, 287, 300
Paulin-Mohring, Christine, 223
payroll, 200, 202
payroll system, 160, 182
PDP-1, 12, 81–83
Peek and poke statements, 116
peer-review, 286
Pendery, Don, 112
Penrose, Roger, 206
performance, 21, 124, 268
performance assessment, 149
performance overhead, 288
Perlis, Alan, 60, 150, 156
Perlman, Radia, 20
personal computer, 118, 180, 225, 246, 247
personal computer for children of all ages, 78
Personal Dynamic Media, 19, 106–109, 117, 120, 255
personal knowledge, 148
philosophy, 18, 110
philosophy of science, 286, 290
physical law, 168
physical machine, 68
physical system, 241
physical unit, 229
physics, 14
physics textbook, 111
Piaget, Jean, 105
PICTURE clause, 200, 299
PILOT, 92–95
PILOT system, 150
Ping-Pong balls, 36
Pixar Animation Studios, 105
Plan to Throw One Away, 163, 175
planning, 14
pleasantness problem, 22, 175
plotter, 96
pluralism, 23, 28, 73, 193, 194, 291, 298–301, 305
pluralistic discipline, 72
poet, 295
poetic code, 295
pointer, 192, 206, 242
political act, 23
political pamphlet, 62
political struggle, 250
polyhedra, 212
polymorphic function, 218
polymorphic recursion, 220
polymorphism, 215, 229
Polymorphism newsletter, 216
POPL symposium, 62, 153, 255
Popular Electronics, 113
portability, 300
portability problem, 33
Post production systems, 34
post-condition, 48, 57, 265
postmortem dump, 132, 147, 151
postmortem dump routine, 142
Power Peg, 5
power set, 203
power structure, 302
powerful idea, 99
powerful ideas, 26
practical constraint, 212
practical expertise, 286
practical problem, 293
practice, 39, 292
practices
  rigorous, 16
Pragnell, Mervyn, 206, 212, 213
pre-condition, 48, 57, 265
pre-scientific era, 217, 287
predicate, 196
predicate logic, 221
predictability, 177
predictive policing, 24
prefixing, 245, 252, 256
prevention-oriented period, 166
primitive type, 191, 203, 229
Principia Mathematica, 196
Principia Mathematica, 65, 86
principles of software engineering, 166
printf function, 224
privacy freaks, 181
private arcane matter, 184
private communication, 248

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

problem exploration, 181
problem of correctness, 186
problem of pleasantness, 186
procedural attachment, 256
procedure division, 201
process, 243, 294
process model generator, 177
process of refinements, 222
process specification, 286
process-oriented approach, 134, 177
processor, 259
product, 192
product assurance, 163
product development, 179
product-oriented approach, 134, 177
production, 14, 15
production mode, 148
Production of Large Computer Programs, 162
production software, 167
profession, low-status, 40
professional book, 294
professionalisation of programming, 33–36
professionalisation of testing, 183
professionalism, 16, 23
profit leader, 161
program, 67
  correctness, 35
  error-free, 21
  proofs, 60–71
  reasoning, 56–58
  understanding, 23–25
program checkout, 143, 184
program construction, 289
program correctness, 10, 33, 47, 60, 156, 175, 226
program description, 200
program error, 21, 23, 99
program proof, 11, 47
Program Test Methods, 156
program testing, 61
program verification, 61, 63, 167, 290
programmer, 39, 151, 288, 289
  individual, 16
  individual skills, 16
  shortage, 33
programmers, 14
  shortage, 39
programming, 285, 292, 304
  as a craft, 72
  as a mathematical activity, 46
  as a musical instrument, 20
  as a political act, 23
  as translation, 34
  black art, 16
  complexity, 16
  education, 25–26
  engineering discipline, 16
  knowledge, 25
  respectable discipline, 33
  scientific way, 16
  styles, practices and tools, 17
programming by demonstration, 110
programming entity, 299
programming environment, 77, 180, 261, 262
programming error, 227
programming experience, 262
programming gap, 39
programming industry, 156
programming language, 42, 72, 155, 198, 199, 279, 287, 300
  Ada, 153, 184, 237, 263, 274
  Agda, 224
  Algol, 3, 11, 33, 49–54, 71, 109, 202, 226, 242, 248, 304
  Algol 60, 191, 197–199, 290
  Algol 68, 204–205, 212
  Algol W, 191, 212
  Alphard, 256
  as a boundary object, 44
  BASIC, 80, 115–118
  BCPL, 265
  BETA, 260
  C, 3, 12, 295
  C with Classes, 265
  C++, 226, 237, 239, 265–267, 302
  Caml, 224
  Cayenne, 224
  Clu, 153, 193, 211, 256, 263
  COBOL, 44, 191, 200, 212, 304
  COMTRAN, 199–201
  Dependent ML, 224
  Eiffel, 237, 263–265
  Epigram, 224
  Erlang, 185
  EULER, 248
  F#, 65
  FLOW-MATIC, 191, 199–201
  FLPL, 87
  FORTRAN, 16, 115, *see also* QUICKTRAN
  GEDANKEN, 208
  Haskell, 219, 230
  HOPE, 216
  HyperTalk, 121
  Idris, 224
  International Algorithmic Language, 198
  IPL, 86
  ISWIM, 208, 214
  Java, 77, 153, 239, 271–272
  JavaScript, 194, 228, 263
  LISP, 45
  Lisp, 77, 86–87, 92–95, 97, 132, 146–147, 179–181, 184, 278, 290, 300
  LISP 1.5, 249

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

Logo, 19, 26, 95, 97–100, 124, 248, 295
Mesa, 209
ML, 65, 213–217, 304
Objective-C, 269
OCaml, 65
PAL, 208
Pascal, 205, 212
Pharo, 261
PL/I, 152, 184
Rust, 219
Scheme, 95
Self, 261–263
semantics, 49–54
SIMSCRIPT, 242
SIMULA, 203, 226, 241–246, 251, 265
SIMULA 67, 236, 241–246
Smalltalk, 106–109, 182, 236, 239–241, 246–255, 265, 267–271, 286, 300
Smalltalk-74, 254
Smalltalk-76, 255
Squeak, 239, 261
Standard ML, 216
the idea of, 8
TypeScript, 194, 228, 304
programming language concept, 193
programming language semantics, 206–209, 214
programming language theory, 70
programming languages, 125
programming methodology, 54, 136
programming paradigm, 256, 273
programming style, 9, 250, 252
programming system, 94, 110, 179–181, 227, 248, 262, 279, 291
A-0, 34
A-2, 42
Autocode, 34
Boxer, 109–112
FLOW-MATIC, 42
Pygmalion, 109–112
Short Code, 42
programming systems, 125
programming vs. coding, 41, 137
programming, professionalisation of, 33–36
programs
as mathematical entities, 34
progress, 194
Project MAC, 92, 209
project management, 136
project planning, 182
project progress, 165
proof, 49, 64, 66, 193, 212
mathematical, 61
program correctness, 61
reflection, 67
proof assistant, 11, 220–226, 265
proof checker, 214
proof checking, 36, 64
proof of correctness, 67, 156
proofs of programs, 60–71
proper engineering discipline, 161
property inheritance, 256
property list, 249
proposition, 196, 221
Propositions as Types, 195
protection mechanism, 192
prototype, 261
prototyping, 176
psychedelic drugs, 104
psychoacoustics, 88, 101
psychologist, 87
psychology, 291, 293
publication, 300
publication language, 10, 298
pulse-by-pulse, 139
punched card, 11, 41, 81, 214
purpose of testing, 157
Pygmalion, 109–112, 127, 253

Quake III, 295
quality assurance, 132, 178
quality control, 64
quantifier, 221
QUICKTRAN, 145, 191, 197–199

radar, 12, 13
Radiation Laboratory, 12
radio, 140
Radio Shack TRS-80, 115
RAND Corporation, 14
Randell, Brian, 155
range failure, 154
rapid feedback, 182, 238
rate of change, 23
Rational Machines, 274
Rational Software, 274, 276
Rational Unified Process (RUP), 276
Read-Eval-Print Loop, 80, 146, 180, 225
readability, 295
real number, 199
real world, 241, 289, 293
real world effect, 67
real-world software, 63
real-world systems, 167
real-world user, 266
reasoning, 256
reconstructible medium, 111
record, 199–206, 244, 300
record class, 245
record data type, 200
record pointer, 120
record type, 191, 212
recording, 286
recovery block, 155

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

recursion, 209
recursive definition, 203, 206
recursive function, 51
recursive function theory, 45
recursive structure, 192
refactoring, 183
reference, 126
reference counting, 244
reference type, 245
reflection, 227
reflective capabilities, 227
refrigerator, 67
region-based memory management, 219
reinforcement learning, 97
reliability, 132, 263
reliable hardware, 170
Remington Rand, 199
remote accessing, 243
repetitive structure, 171
REPL, *see* Read-Eval-Print Loop
representation, 210
representation theorem, 211
reproducibility, 177
reproduction study, 49
require clause, 265
requirements, 22, 133, 156
requirements definition, 178
requirements gathering, 175, 238
research paradigm, 217
research retreat, 108
Responsibility-Driven Design, 269, 273, 297
reusability, 264
reuse, 263
Revised report on the algorithmic language Algol 60, 50
revolutionary change, 99
rewrite rule, 225
Reynolds, John C., 193, 211
right to explanation, 4
rigorous argument, 134
robot, 42
  turtle, 19
Rolling Stone, 104, 251
Rome, Italy, 56
Ronald Reagan, 169
Royal Signals and Radar Establishment, 69
Royce, Winston, 134, 163, 175
rugby, 178
Rule 15c3-5, 7
Rule of Three, 288
Rumbaugh, James, 276
runtime check, 242, 299
Russell's paradox, 196
Russell, Bertrand, 196, 206
Russell, Steve, 81, 87, 91
Russia, 23
S-expression, 203
S-programs, 174
Safeguard system, 167
safety, 194, 228, 256
SAGE, 13, 41, 77, 84, 88, 167
SAIL, *see* Stanford AI Laboratory
Sammet, Jean, 245
Samson, Peter, 83
San Francisco, 100
scale, 240
scale of software, 238
scaling-up, 173
Scandinavia, 177, 260
Scandinavian approach, 177, 181
Scandinavian school, 260
Scheme, 95
schoolchildren, 124
Schopenhauer, Arthur, 240
Schwaber, Ken, 179
science, 289, 291
Science and Engineering Research Council, 216
scientific code, 295
scientific discipline, 286
scientific enterprise, 183
scientific method, 16, 17, 133, 147, 149
scientific paradigm, 9, 217, 285
scientific pluralism, 28
scientific practice, 49
scientific programming, 152, 200, 202
scientific research paradigm, 46, 290
scientific way, 16
scientist, 295
Scott, Dana, 212, 214
Scratch, 112, 125
screen turtle, 98
SCRUM, 134
SCRUM methodology, 179, 181
SDC, 14
SDI, 169
seal and unseal operations, 211
SECD machine, 208
secrecy problem, 210
Securities and Exchange Commission, 5, 7
sel4 microkernel, 3, 11
self, 261
self-checking, 155
self-consciousness, 96
self-modifiability, 237
self-referential proposition, 198
semantic incommensurability, 298
semantics, 49–54, 242
semantics of programming languages, 213
Semi-Automatic Ground Environment, 13, 84, 161
separation logic, 65
SERC, 216
serendipity, 99, 125
set, 191, 192

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:37, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

set theoretical model, 206
set-theoretic model, 210
SHARE, 43
SHARE user group, 201
shared entity, 300
shared memory, 219
Shaw, Clifford, 86
Short Code, 41
Short-Range Committee, 200
Shoup, Richard, 105
Show us your screens, 124
Simon, Herbert, 65, 86
Simondon, Gilbert, 24
Simonyi, Charles, 119
simplicity, 295
simplification algorithm, 214
simply typed lambda, 209
simpterm function, 214
simulation, 258, 277
simulation laboratory, 262
Simulation Language, 241
Single E.P, 141
Sketchpad, 19, 89, 101, 105, 247, 297
Slug, *see* Russell, Steve
Smalltalk, 19
Smalltalk-72, 108, 110, 117
Smalltalk-72 handbook, 251
Smalltalk-76, 108
SMARS, 5
Smith, Alvy Ray, 105
Smith, Brian Cantwell, 67
Smith, David Canfield, 110, 119
Smith, E. E., 81
Smith, Randall, 261
snapshots, 47
Snyder, Betty, 36
social activity, 293
social control, 193
social history, 303
social process, 35, 60–63, 66, 70, 72, 133, 186, 194, 218
Social Processes and Proofs of Theorems and Programs, 61
social structure, 23
social transmission, 292
societal impact, 305
societies of high modernity, 186
society, 18, 96
socio-economic factor, 302
socio-political context, 99, 287
socio-technical issues, 24
SofTech, Inc., 153
software
  complexity, 23
  error-free, 21–23
  financial, 6
  opacity, 23
software engineering, 55
Software aspects of strategic defence systems, 169
software component, 64
Software Configuration Control Board, 15
software correctness, 61
software crisis, 293, 303
software developer, 295
software development, 288
software development lifecycle, 238
software development methodology, 167, 186
software development process, 72
software engineer, 182, 294
software engineering, 28, 64, 80, 149, 151, 156, 167, 178, 237, 261, 263, 289, 301, 303
Software Implemented Fault Tolerance, 67
Software Initiative, 153
software lifecycle, 301
software manufacturing, 147
software production, 55
software projects, 59
software quality, 64
Software Quality Engineering, 183
software reliability, 155
software structure, 293
software systems, 166, 175
Software Technology for Evolutionary Participative System Development, 177
software testing, 156–159, 183
software verification, 49, 70
Solomon, Cynthia, 20, 97, 98
Sonic Pi, 3, 20, 124
soundness, 228
soundness property, 211
source code, 78, 180, 294
Soviet Union, 14
Spacewar!, 3, 81–83
Spacewar article, 251
spatial metaphor, 111
special form, 258
specification, 14, 22, 64, 67, 133, 149, 174, 175, 194, 223, 285
Sperry Rand corporation, 208
spiral development model, 176, 275
spiral model, 179
spiral software lifecycle, 274
spreadsheet, 80, 119
sprint, 179
standard, 294
Standard ML, 216
standardisation, 156
Stanford, 180, 213
Stanford AI Laboratory, 180
Stanford LCF, 214
Stanford Research Institute, 100
Star Wars program, 169
state transition, 275

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

static type system, 215
station, 243
Steele, Guy, 95
STEPS, 177
Stockham, Thomas, 144
storage location, 141
stored program, 141
Strachey, Christopher, 52, 206, 212
Strategic Defense Initiative, 169
Stroustrup, Bjarne, 226, 265–267, 275, 301
structure editor, 93, 109, 117, 146
structured data, 200
structured programming, 17, 35, 57, 58, 72, 136, 149, 161, 193, 203, 242, 243, 259, 275, 290, 300
Structured Programming Theorem, 58
struggle for control, 301
subclass, 245, 255, 256
subculture, 289, 292, 295
subroutine, 116, 141
subroutine library, 160
subscripted variable, 199
subtyping, 236
sufficiency, 209
Sun Microsystems, 271
Sun workstation, 271
supervisor process, 185
supporting programmer, 59
surgical team, 64, 172
Sussman, Gerald, 95
Sutherland, Bert, 89, 267
Sutherland, Ivan, 19, 89, 246
Sutherland, Jeff, 179
Sven, Doug, 6
Swansea, 213
switching theory, 25
symbiotic system, 94
symbol manipulation, 86
symbol-processing, 301
syntactic structures, 68
syntax, 50
syntax directed interpreter, 258
Sys statement, 116
SYSDEFS file, 252
system availability, 149
system decomposition, 210
System Development Corporation, 103
system of expertise, 24
system of practice, 285
system test, 149
system testing, 165
System/360, 171
systems
  mission critical, 34
  technically challenging, 16
Systems Development Department, 112
systems of practice, 298

tablet, 106
tacit knowledge, 26, 39, 286, 294, 299
tactic, 215, 225
Takeuchi, Hirotaka, 178
tape, 201
tape reader, 113
tar pit, 159
tax return, 173
Taylor, Bob, 101
TDD, *see* Test-Driven Development
team, 293
team role, 182
Team Topologies, 59
Tech Model Railroad Club, 12, 77, 81, 85
technical artifact, 23, 265
technical concept, 8, 185
Technical University of Berlin, 177
Technical University of Munich, 148
Teitelman, Warren, 94, 150
Tek 4404, 269
Tek Labs, 268, 273
Tektronix, 238, 268, 272, 278
tele-conferencing, 100
telecommunications, 185
telephone exchange, 185
teletype, 19, 113
teletype terminal, 117
temperature, 67, 289
template meta-programming, 220
TEN-Mode, 92
tentative context, 268
terminal, 19, 82, 91, 180
terminology, 256
test, 133, 184, 191, 286
test automation, 150
test documentation, 165
test manager, 156
test selection, 157
test technician, 156, 183
test trajectory, 139
Test, then code, 166, 183
Test-Driven Development, 17, 80, 133, 139, 183, 184, 186, 269, 272, 297, 301
testing, 16, 17, 61, 62, 132, 133, 136, 149, 156–159, 175, 182, 238, 300
testing practices, 156
testing tools, 156
testing,, 23
tests
  as a specification, 183
text editing, 100
text editor, 248, 253
text processing, 119
Thacker, Chuck, 105
The Art of Computer Programming, 40

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

The Art of Software Testing, 157
The Atlantic, 18, 294
The Debugging Paradox, 184
The Duke of Marlborough, 207
The Growth of Software Testing, 156
The Hub, 123
The League of Automatic Music Composers, 123
The Mother of All Demos, 77, 100, 286, 294
The Mythical Man-Month, 169, 172
The New New Product Development Game, 178
The Principles of Mathematics, 196
The Telekommunist Manifesto, 23
theorem, 62, 65
theorem prover, 63–66, 213, 220–226
  ACL2, 6
  ALF, 222
  Automath, 220, 222
  Boyer-Moore, 65, 66, 213
  Coq, 223
  LCF, 193, 220
theorem proving, 213, 300
theoretical computer science, 191
theoretical curiosity, 166
theory, 260
theory of computation, 202
theory of data types, 203
Therac-25, 1
ThingLab, 262
thinking
  in children, 97
  in machines, 97
thm type, 215
Three Amigos, 276
throw function, 154
time travel debugging, 133
time-sharing, 56, 77, 91–92, 97, 103, 115, 132, 145, 172, 179–181, 214, 225, 247
timing, 67
Titanic effect, 36
TMRC, *see* Tech Model Railroad Club
Toggle Switch Storage, 84
tools, 16, 226
tools for thought, 297
top management, 161
top-down development methodologies, 161, 175
top-down software development, 59
TOPLAP, 20, 124
trade union, 260
trade-off, 228, 295
trading system, 5
trading zones, 60, 230, 292
training costs, 263
transfer instruction, 141
Tree-Meta, 102
tricks, 141
tripartite nature, 289
trust, 185–187
trustworthiness, 24
trustworthy software, 304
Turing Award, 60
Turing machines, 34
Turing, Alan, 141, 143
Turtle graphics, 19, 98, 109, 111
TX-0, 12, 77, 81–86
TX-0 tools, 85
TX-2, 19
type, 212, 286, 287, 298, 300
  dependent, 193
  fixed point, 197
  floating point, 197
  integer, 197
  nature, 197
  primitive, 191
  unsound, 194
type annotation, 210
type assignment, 210
type checker, 218, 219
type checking, 194, 210, 222
  compile-time, 193
  run-time, 193
type classes, 220
type error, 227
type inference, 209, 210, 219
type inhabitation, 221
type system, 218, 226, 264, 299, 304
type theory, 222
type-systems laboratory, 220
types
  as sets, 192
  formal origins, 196–197
Types and Programming Languages, 218
types as sets, 229
Types course, 226

UC Santa Barbara, 264
UK Ministry of Defence, 69
UK Ministry of Technology, 183
UML, *see* Unified Modelling Language
unchecked type conversion, 266
Uncle John, 290
undecidability, 166
Underestimates and Overexpectations, 168
underground logic reading group, 206, 216
underground reading group, 213
understanding, 295
unexpected situation, 151
Ungar, David, 261
unification, 252
Unified Modelling Language (UML), 239, 276
unified paradigm, 298
union of non-intersecting sets, 203
union type, 221
unit test, 149
unit testing, 15, 165, 183

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691

USA, 263
United States Army, 37
United States Military, 83
units of measure, 229
Univac, 42, 242
UNIVAC I, 41, 199
universal programming language, 43, 202
universal quantification, 221
universe of values, 210
university, 10, 13
university computing centre, 25
university department, 292
University of Cambridge, 38, 69, 226
University of North Carolina, 156, 173
University of Oxford, 52
University of Utah, 247
University of Vienna, 177
UNIX, 3, 14
Unlocking the Computer's Profit Potential, 106, 160
unsound type system, 228
unwritten knowledge, 220
US Air Force, 247
US Air Force Academy, 274
US Census Bureau, 39
US Department of Defense, 153, 165, 237, 263
US military, 171
US National Academy of Engineering, 266
US Navy, 100
USA, 263
Use Case diagram, 239
user, 78
user-defined types, 205, 267
user interface toolkit, 272
UT-3, 84, 144
Utility Decks, 92
Utility Tape 3, 84, 144

vacuum tubes, 38, 84, 139
validation, 165
variable, 261
variant record, 205, 213
VAX machine code, 216
VDL, *see* Vienna Definition Language
vector, 224
verb, 238, 274, 275
verification, 149, 165
verification debate, 11
Vernacular of File Formats, 22
very-high-level programming language, 258
vicious-circle principle, 196
Victor, Bret, 297
video display, 180
video switch, 180
video terminal, 113
Vienna Definition Language, 53
Vienna Development Method, 135
Vienna IBM laboratory, 53
vim editor, 226
VIPER microprocessor, 69
virtual machine, 268, 271
virtual memory, 253
virtual method, 245
VisiCalc, 80, 119
visual effects, 81
visual element, 262
Visual Smalltalk Enterprise, 239, 270
VisualWorks, 270

Wadler, Philip, 195
Wadsworth, Christopher P., 52
warfare, 67
water flow, 137
Waterfall, 134, 149, 161–166, 179, 274
web browser, 271
Wegner, Peter, 259
Weizenbaum, Joseph, 28, 135, 168
well-typed program, 193, 209, 211, 216, 217
West coast, 13, 78, 100
West Coast Computer Faire, 267
Western annual Joint Computer Conferences, 201
Wheeler, David, 141
while statement, 57
Whirlwind, 12, 77, 83–86
Whitehead and Russell, 86
Wiitanen, Wayne, 81
Wilkes, Maurice, 38, 140–141
Winograd, Terry, 256
Wirfs-Brock, Allen, 268, 273
Wirfs-Brock, Rebecca, 273
Wirth, Niklaus, 151, 205, 248
WJCC, 201
women
  exclusion from the workforce, 33
  in the hacker culture, 12
  in the humanistic culture, 20
word processing, 112, 172
World War II, 83, 88
World Wide Web, 88, 272
Wozniak, Steve, 115
WYSIWYG, 119

Xerox Alto, 106, 112, 267
Xerox PARC, 67, 79, 104, 106, 126, 180, 209, 210, 212, 247, 250–254, 267, 277, 301
Xerox Star, 79, 112, 119, 127
XINTF function, 198
XP, *see* Extreme Programming
XP customer, 182

Y2K bug, 143
Yale University, 61

Zen and the Art of Motorcycle Maintenance, 140
Zilles, Stephen, 211, 256
Zurich, 50

Downloaded from https://www.cambridge.org/core. IP address: 187.236.92.223, on 22 Oct 2026 at 13:03:31, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://www.cambridge.org/core/product/075A2D1DE611EE47807A683147B21691