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]

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]

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?

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.

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

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

[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

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

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.

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.

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

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

```
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

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

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.

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]

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

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.

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

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

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

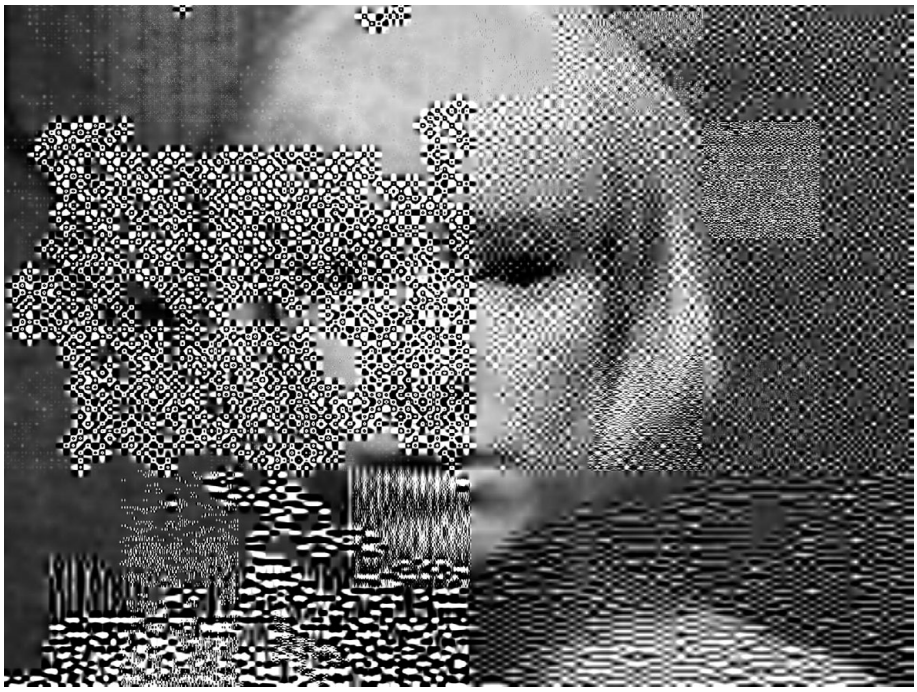

Figure 1.4 Vernacular of File Formats (2009–2010) by Rosa Menkman explore how introducing an error in a compressed file results in different visual effects. The example here is that of the JPEG2000 format. (Source: Rosa Menkman,[63] reproduced with permission)

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

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

'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.

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,

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.

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

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'

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).

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.

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.

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.