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.