4 Software Engineering

Teacher: We talked about mathematical methods and interactive programming and both times we concluded that they are not used as often as they should be. So, how do programmers make sure all the software that is around us actually works?

Pythagoras: Looking at the industry practices, the answer is a mix that includes debugging, testing and quality assurance, rigorous management processes and over-engineering.[1]

Teacher: I propose we start by going through this list in a historical order. What were the first industry practices that helped us produce more reliable systems?

Diogenes: Well, debugging got a lot easier thanks to the birth of time-sharing and interactive computing at the end of the 1960s, because you could explore the state of a program while it was running, but primitive forms of debugging go back to the first digital computers. On EDSAC, you could debug programs by running them instruction-by-instruction and inspecting the state of a part of a memory on a CRT screen...

Xenophon: Such a waste of valuable computer time! It's not surprising that this use of digital computers was soon banned on most installations.

Archimedes: Well, one interesting consequence of the limitation was that it inspired programmers to come up with new approaches. It resulted in the first debugging tools that we use to this day. In the case of EDSAC, those were postmortem dumps and interpreters that printed diagnostic information while running the program.

Pythagoras: I was wondering about the early days of debugging, but it is practically impossible to find any academic literature on debugging techniques!

Diogenes: The debugging tools built at MIT for TX-0 and PDP-1, called FLIT and DDT, are documented well enough in memos published by the MIT Lincoln Laboratory. An academic paper about them would not be very useful. They were clever, but simple tools and hackers learned how to use them through practice...

Xenophon: I see. The black art of programming strikes again! When did debugging turn into a more rigorous discipline?

Archimedes: People started taking debugging more seriously in the 1960s. Once hardware got reliable and high-level programming languages made it possible to write more complex programs, people realised that the number of bugs in programs became the main limiting factor for the use of computers.[2]

Pythagoras: Very well, but what novel debugging capabilities did this new focus give us?

Diogenes: Good question. The focus may have changed, but the debugging tools I was talking about were quite powerful already. You could step through instructions, set breakpoints and even patch your program on the fly. This worked for assembly at first, but in the mid 1960s, similar tools started to appear for Lisp too.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

Pythagoras: This sounds like the debugging tools we have today. See? This lack of progress is surely the consequence of hiding all the information in internal memos and not advancing knowledge gradually through academic publications!

Archimedes: Well, modern debugging tools are more sophisticated. They use efficient incremental compilation rather than interpretation and reverse or time travel debugging makes it possible to step back in time. But the primary improvement is a greater conceptual clarity. In the 1960s, debugging became clearly distinct from testing. As a result, debugging and testing tools started to evolve independently.

Xenophon: I do not see how anyone could confuse the black art of debugging with testing as understood in rigorous software engineering. Around the time of the 1968 NATO conference, testing became a part of the software development process. It was used to certify that a system has been implemented according to requirements of the customer. How comprehensive testing is required was also agreed with the customer.

Pythagoras: Right, but at the NATO conference, Dijkstra also correctly pointed out that testing shows the presence of bugs, never their absence![3]

Archimedes: Full correctness was not the aim. It was a step before handing the system off to a customer and it led to the development of the first automated testing tools in the 1970s. You cannot deny those give you some assurance about software correctness!

Pythagoras: Well, the automation inspired interesting mathematical analyses. One looked how to generate good input data for tests, while another developed a probabilistic theory based on how much of the code was covered by tests.[4] Mind you, this still gives you weaker guarantees than formal verification, but at least it is a rigorous analysis.

Teacher: Let me observe that we seem to have much more to say about testing than about debugging. I wonder why that is the case...

Socrates: It seems to me that a 'test' is an entity that can be shared by multiple people, even if it means different things to them. The consequence is that different cultures of programming develop the idea in different directions. You can find a meaning for 'test' even in my favourite domain of live coding. There, it is different still and is more akin to practice before an art performance!

Archimedes: And we did not even get to the most interesting development which followed when testing turned from a phase in a managerial process into an engineering tool in Test-Driven Development (TDD).

Pythagoras: Isn't TDD using tests just as a poor substitute for formal verification?

Archimedes: No, no, no! In modern use, tests capture the requirements of the customer and serve as a lightweight specification. As a methodology, TDD is a kind of scientific method for building software. You repeatedly add a test and run it (i.e., run an experiment) and then you revise your theory (i.e., your program), until the test passes.[5]

Pythagoras: But using tests as your specification is the wrong way round! You should write a test based on your specification instead.

Archimedes: Except that, in practice, you never have a detailed specification upfront. Writing tests forces you to think about the right behaviour and discuss it with your colleagues and business domain experts. Tests capture the resulting knowledge.

Socrates: Interesting! If tests are something that you discuss with your colleagues and customers, then they provide a focal point for valuable social processes. Exactly the kind of social processes that proofs of program correctness were lacking!

Xenophon: When you put it in this way, I can see how TDD gives you an appealing light-weight software development process. That said, if you have enough time and resources, you should still produce a detailed specification up-front to avoid surprises!

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

Diogenes: Do you suggest we should be using the Waterfall development process? I do not understand how anyone was able to build any software this way!

Xenophon: You are thinking of a caricature of the process. Structured methodologies that divided the process into phases like requirements-gathering, analysis, development and testing were used by the military already in the 1950s. People were also continually improving the process. For example, the oft-cited 1970 paper by Winston Royce[6] is not about Waterfall itself, but about making the process more adaptive.

Socrates: I'm curious how we got from these methodologies to something like the Test-Driven Development. Did Waterfall just get out of fashion?

Xenophon: It worked well enough in the 1970s and 1980s, but in the 1990s, the PC market and the rise of the Internet changed what kind of applications companies needed. It was crucial to build software faster and adapt to changing market requirements.

Diogenes: Is this where the Agile methodologies come from?

Archimedes: It was a more gradual process. Extreme Programming and SCRUM were two methodologies that tried to address the situation, but you are right that the term Agile is the most famous one. The Agile manifesto, written by 17 thought-leaders in 2001 gave those methodologies a common name and popularised them.

Socrates: It is sad that it takes 17 men to re-invent something that has been described more clearly by a woman over a decade earlier!

Archimedes: What do you mean? Please tell us more.

Socrates: In 1987, Christiane Floyd described software engineering methodologies inspired by the Scandinavian idea of 'participatory design' where you work closely with users during a product design.[7] She contrasted the dominant product-oriented approach with an emerging process-oriented approach. The latter is based on a close collaboration with the customer, learning and building from a shared understanding.

Archimedes: That does, indeed, resemble the key ideas of the Agile movement!

Teacher: It seems we have covered testing, debugging and development processes already, so what else was on our list of real-world engineering practices?

Pythagoras: We could talk about exception handling, but that would be missing the forest for the trees. Isn't there some more fundamental theory of software engineering?

Archimedes: Software development is inevitably a human task. You cannot hide the inherent complexity and make simplifying claims. That said, you can identify common patterns, characterise different classes of programs and their general properties and also make rigorous analytical arguments grounded in the understanding of the relationship between the human world and software.[8]

Xenophon: Do you mean observations like Brooks' law, which states that 'adding manpower to a later software project only makes it later'? In my experience, that is the case but I do not have enough data to prove this with certainty![9]

Archimedes: Right, but Brooks' argument does not rely on experience. He explains the 'law' in terms of communication overhead. Look at the two charts in Figure 4.1. If you have a task that can be perfectly divided among team members, the amount of time needed decreases proportionally with the number of people. But programming complex systems requires communication between team members and, in general, you need each member to talk to every other member. So you have to add an overhead of $n*(n-1)$. This is quadratic and so, even for a small amount of communication, it eventually dominates the gains you get by dividing the work.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

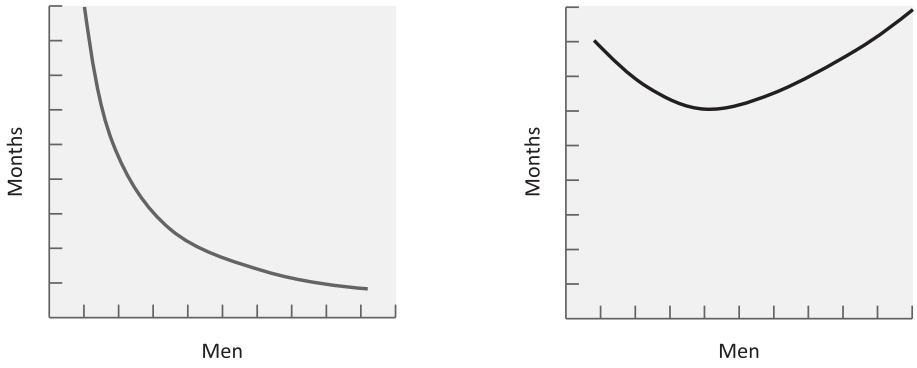


Figure 4.1 Time versus the number of workers for a 'perfectly partitionable task' (left) and a 'task with complex inter-relationships' (right).[10] (Source: Drawing by the author, labels as in the original)

Xenophon: Do you suggest to measure the number of minutes needed for programming and communicating and then compare those based on the team size? That sounds interesting, but I doubt the numbers would generalise well across teams and projects...

Socrates: You do not need concrete numbers. This is a very nice argument, because the logic is convincing without concrete numbers. I wonder what other problems have been studied using this kind of reasoning?

Archimedes: Another example is from the debate about anti-ballistic missile systems (ABMs).[11] In 1969, Joseph Weizenbaum argued that ABM cannot reliably be built because the rate of the change in the environment is greater than the rate of the change we can maintain in the software system. This also relies on a sort of mathematical argument that is convincing enough, even without specific quantities.

Pythagoras: As far as I'm concerned, this is exactly the kind of software that should have a formal specification and a proof that the implementation conforms to it!

Xenophon: That is unrealistic and cannot work in practice. You need a solid process to ensure that system will have the characteristics it needs.

Teacher: We have drifted to the problem of trusting software. What do others think?

Diogenes: If you want to hear my position, then I would like to remind you that software is ultimately built by people. I only trust programs created by people that I can trust.

Socrates: Do not forget, though, that software is written by people for people. I only trust software that is trying to do the right thing, because you can never separate the technical from the social. Getting the social-side of software right has become the limiting factor these days!

4.1 How Did Software Get So Reliable without Proof?

In March 1996, the International Symposium of Formal Methods Europe took place in Oxford. The symposium evolved from a series of events focused on the Vienna Development Method (VDM), a formal method that started with the work on semantics of programming languages in the IBM Vienna lab at the end of the 1960s. The organisers struck a positive note about the industrial applicability of formal methods and a quarter of the presentations at the symposium were reports on industrial use of formal methods.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

The opening keynote was presented by a veteran of the field, C. A. R. Hoare, who developed foundational theories for reasoning about programs that we encountered in Chapter 2. In his talk, 'How Did Software Get So Reliable Without Proof?' Hoare asks a question that may have rightfully puzzled many proponents of the mathematical culture.

The talk started with a reference to an analysis by Donald MacKenzie who attempted to determine how many people had died in computer-related accidents prior to 1992. Only about 30 deaths out of the collected dataset of 1,100 were caused by a computer bug. Of those, 28 were the consequence of a faulty software in the Patriot missile system. The majority of the deaths were instead caused by 'faulty human–computer interaction' such as 'poorly designed interfaces' or 'organisational failings'.[12] Hoare acknowledged that 'formal methods and proof play a small direct role in large-scale programming' and reflected on the situation:

> Twenty years ago it was reasonable to predict that the size and ambition of software products would be severely limited by the unreliability of their component programs. … Fortunately, the problem of program correctness has turned out to be far less serious than predicted. … Similarly predictions of collapse of software due to size have been falsified by continuous operation of real-time [telecommunications] systems. So [a question arises]: why have twenty years of pessimistic predictions been falsified?[13]

Hoare's answer is that software gets reliable through a mix of practical engineering practices, including good project management, rigorous testing, debugging, over-engineering and innovations in programming methodology. As one might expect from a proponent of the mathematical culture, Hoare does not see this as a failure of formal methods. Instead, he argues that many such practices owe much to the theoretical ideas developed 20 years prior to his paper. Formal methods and proof may play a small direct role in large-scale programming, but 'they do provide a conceptual framework and basic understanding to promote the best of current practice, and point directions for future improvement'.

Despite the title, Hoare's keynote was an optimistic one. He concluded that the 20-year gap between theory and practice that he identified 'is actually an extremely good sign of the maturity and good health of our discipline' where basic research conducted by theorists provides fundamental basic understanding and where industry adapts and adopts useful theoretical innovations such as structured programming, informal mathematical reasoning, data types and information hiding.

Hoare's keynote paints an idealised image of a gap between theory, which explores new directions for advancing the state of the art, and practice, which adopts proven research ideas. Using the perspective of cultures of programming that I follow in this book, Hoare would see the mathematical culture of programming as the primary source of new scientific ideas and the engineering and managerial cultures as those who apply scientific ideas in the real world. As we saw when looking at the origins of programming languages in Chapter 2, and as we will repeatedly see in the chapters to come, the interactions between cultures are not so simple. An idea may emerge in any culture of programming, it may be influenced and reshaped by ideas from other cultures and even return back in a different form. Much like the history of programming languages

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

and interactive programming, the history of testing, debugging, error handling and development methodologies that I follow in this chapter is full of such interesting and unexpected twists.

4.2 The Art of Electronic Computer Maintenance

The idea that programming a computer will become a major challenge was completely unexpected when the first digital electronic computers were created. To a contemporary programmer, this may seem naive, but it is less surprising if we recognise that digital electronic computers did not appear out of the blue in the 1940s. In many ways, they were the next evolutionary step from analog computers which used mechanical, hydraulic or electrical quantities to model numerical problems. Analog computers did not require sophisticated programming and their operation was often fully transparent. A wonderful late example of this idea is the hydraulic computer that models the economy, which was created in 1949 and was called MONIAC, 'to suggest money, the ENIAC and something maniacal'.[14] In MONIAC, the flow of money in the economy is modelled as a flow of coloured water. Various mechanisms in the economy are modelled using valves controlled by floats, pulleys and other hydraulic devices as shown in Figure 4.2. The machine had a limited accuracy, but it was unprecedented in aiding the understanding of complex ideas of economics.[15] The user of the MONIAC can always see the full state of the system and noticing that the machine (mis)behaves is just a matter of watching how the water flows through the system.

Electronic analog computers removed this kind of transparency, but kept the simple fixed structure where a small change in the input leads only to a small change in the output. The last traces of this simplicity were finally lost with the advent of digital computers. The operation of digital computers is not just hidden from sight, but it also involves complex 'evolution of a meaning'.[16]

In spite of this complexity, the job of an operator responsible for setting up the computations on the early digital computers was thought of as relatively mundane. Many of the original computer operators of the ENIAC computer were women, recruited from a group of human computers hired to calculate ballistics tables by hand. The setting up of the ENIAC was, at first, done using switches and by connecting individual components of the machine using cables. The operators of the ENIAC had to figure out how to translate a more high-level mathematical plan of a computation into actual wiring of the machine. At least initially, the program on ENIAC was the physical setup of cables and switches.

Why start a discussion about the history of good software engineering practices with something as rudimentary as the ENIAC? As soon as the challenges of programming and debugging appeared, programmers started to look for ways of dealing with those. Many such methods are not unlike the methods that programmers use to program and debug computers today. First of all, different individual programmers approached the problem of programming in different ways. In her memoir, Betty Jean Jennings (Figure 4.3), whom we encountered as one of the first ENIAC programmers

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

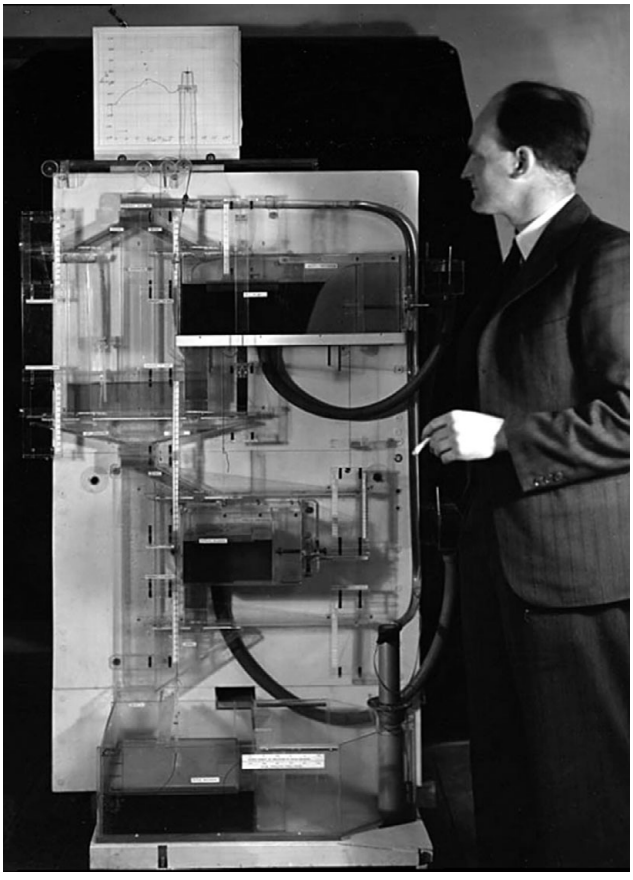


Figure 4.2 Professor A. W. H. Phillips with the MONIAC computer also known as the Phillips Machine. (Source: LSE Library)[17]

in Chapter 2, recalled how her approach to programming differed from that of her colleague Adele Goldstine:

> Adele was an active type of programmer, trying things very quickly. I was more laid back and given to attempting to figure out things logically before doing anything.[18]

Computer programmers never resolved this schism and never settled on one right way of programming. We find the same difference in approaches between the hacker and mathematical culture of programming today. On the one hand, mathematically minded programmers still favour logical reasoning over experimentation. This approach is often taught at universities, in part because it can be more easily written down. On the other hand, hackers often solve problems by quickly iterating on a solution. Being an active type of programmer like Adele Goldstine today may require less formal preparation, but more intuition and experience. A skilled hacker does not try just any

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

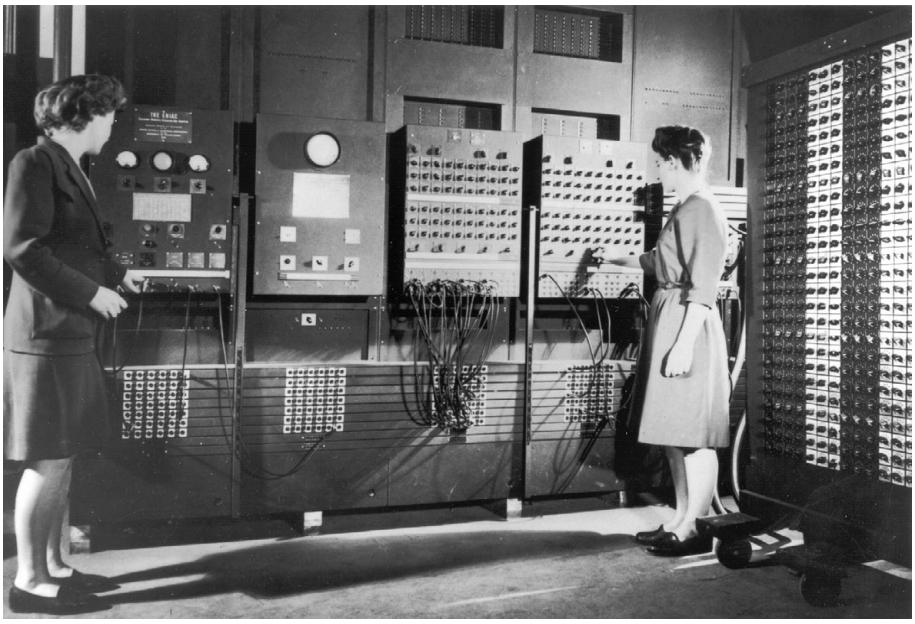


Figure 4.3 Betty Jean Jennings (left) and Frances Bilas (right) operating the ENIAC's main control panel. (Source: US Army/ARL Archives, Courtesy of Mike Muuss)[19]

random solutions. They approach the solution gradually, in a well-chosen sequence of steps.

The programming of the ENIAC also involved two ideas that are characteristic of the engineering culture of programming, which often strives to find a better or a more reliable method to solve a problem. The first engineering method was used when debugging the program for calculating trajectories. When developing the program, two of the ENIAC programmers first manually calculated a single trajectory in a way in which the ENIAC was supposed to calculate it. When the machine was set up to do the work, it was first run on this test trajectory. If the results differed, the programmers knew something was wrong. They could stop the execution of the ENIAC and run it manually pulse-by-pulse to find the source of the error. This was useful not just for finding errors in the program wiring, but also to identify blown vacuum tubes, which was a common hardware failure of the ENIAC. The idea of using a hand-computed test run as a baseline for testing the program is the kind of robust method that arises from the engineering culture. In fact, the approach is not unlike that of a much later Test-Driven Development (TDD) methodology that we will discuss later in this chapter.

The second engineering technique that is illustrated by the history of ENIAC is the idea to use the computer itself to simplify its programming. In 1948, a reprogramming of the ENIAC changed how the machine operated. The new approach was based on the experience with the ENIAC and also ideas for EDVAC, a subsequent machine that was in the works. The reprogramming wired the ENIAC such that it would execute instructions encoded on punched cards using a '60 order code'.[20] The programming of the

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

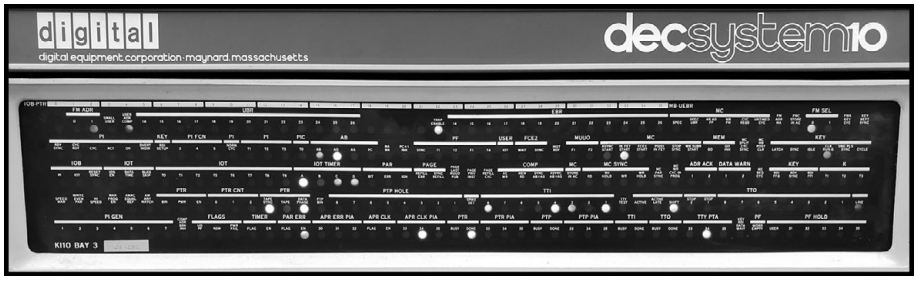


Figure 4.4 Indicator lights at the top of the 'Bay 3' (which also included the main control panel) of the KI10 variant of the PDP-10 computer, released in 1973. (Source: Living Computers Museum, Photo by David Bridgham.[21] Reproduced with permission.)

machine thus no longer required tedious fiddling with cables. The new programmable wiring was kept until the end of ENIAC operation in 1955. Again, using a computer to build a tool that makes the programming easier is an engineering method that we will encounter many times in this chapter.

Despite being electronic and digital, the ENIAC had a certain degree of transparency. The state of the memory was made visible using a grid of light indicators on the control panel, which the inventors used to put on a show during the 1946 public demonstration. Running the machine pulse-by-pulse also made it possible to follow what is going on. A similar degree of transparency was offered by many of the early electronic digital computers (Figure 4.4). An even more curious hacker trick was used to make transparent the later BINAC machine, built by the Eckert–Mauchly Computer Corporation in 1949. One of the engineers discovered that they could connect the output of one of the BINAC circuits to a radio and listen to its sound, much like the hacker-minded narrator in *Zen and the Art of Motorcycle Maintenance*[22] who notices an issue from the different noise of the motorcycle engine:

> Mind you, this weird music wasn't like anything being broadcast on the radio–it was just a sequence or pattern of sounds emitted from the radio as the BINAC was running. Al Auerbach said he'd quickly discovered that he could tell whether the BINAC was running correctly by the sounds coming out of the radio when he played the test program.[23]

In the case of ENIAC, hardware failures were still very common and so debugging using, for example, the manually computed trajectory, often involved looking for both hardware failures and programming errors. However, hardware soon started becoming more reliable, partly thanks to the fact that computers were made of a large number of repeated components. No such repetitive structure exists in computer programs and so programming issues grew in prominence. As mentioned in Chapter 2, one of the computer pioneers who understood the complexity of programming early on was Maurice Wilkes, who designed the EDSAC computer at Cambridge.

The EDSAC computer shared an architecture with many computers built after ENIAC. It was controlled through a program, represented as a sequence of *order codes* that was loaded from a tape or punched cards. The orders, or instructions,

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

were then stored in computer memory and executed. EDSAC recognised 17 different instructions for performing calculations (`AnS` to add the number in storage location `n` into the accumulator), memory access (`TnS` to transfer the contents of the accumulator to storage location `n`), but also for conditional jump (`EnS` to continue executing orders in storage location `n` if the value in the accumulator is greater than or equal to zero).

The practical matter of programming the early computers required 'black art' skills that are characteristic of the hacker culture of programming. Programmers had to invent ingenious tricks, often using the fact that a program itself was stored in memory, to implement operations that seem obvious today, but were not included in the instruction sets of early machines. For example, EDSAC did not have an instruction for indirect addressing that programmers would use today when accessing an element in an array at a specified index. If you wanted to access a storage location based on an index computed by your program, you had to use the transfer instruction `TnS` to first overwrite the part of your program (which was loaded into a known location in the memory) that specified the argument of the next `TnS` transfer instruction. The ACE computer designed by Alan Turing did not even have a conditional jump and programmers had to use the very same trick of overwriting the target location of an unconditional jump.[24]

The first non-trivial program that has likely been subject to debugging is a program to calculate the Airy integral, created by Maurice Wilkes for EDSAC. A detailed account of the debugging process has been given by the historian Martin Campbell-Kelly,[25] who analysed an early tape with the program, finding 'approximately twenty errors in the 126 lines of the program'. Many of those were simple punching errors, but some likely required a notable debugging effort. The unexpected fact that producing correct programs is difficult meant that EDSAC did not initially have any dedicated program debugging tools. As also pointed out by Campbell-Kelly, 'the way to debug a program at Cambridge, and at most other places, was to sit at the console and execute the program manually, instruction by instruction, while observing the registers and memory on monitor tubes'.[26] This tedious process was referred to as 'peeping'. On EDSAC, it was done using the 'Single E.P' button, which executed a single instruction of a program and a CRT monitor that displayed the contents of a portion of the computer memory. Some early computers went even further and provided switches for manually modifying the instructions in memory.

As was the case with the history of ENIAC programming, the history of EDSAC programming also includes a primordial form of engineering programming techniques that later become standard. On EDSAC, the most interesting methods were possible thanks to the fact that the program was stored in memory where it could be programmatically modified. This made it possible to use the computer itself to build tools that make programming and debugging the computer easier.

The first trick simplified the construction of 'large' programs like the Airy integral calculation. Maurice Wilkes and David Wheeler adopted the idea of a subroutine that was proposed in the 'First Draft of a Report on EDVAC' written by the ENIAC team at Moore School. Subroutines were sequences of instructions, eventually stored on tapes in a library, which implemented common tasks and could be added to the program after the main routine. Doing this manually would be laborious, because EDSAC

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

instructions used absolute addressing in instructions that accessed memory and so all addresses in the subroutine would have to be adjusted. To avoid this issue, Wilkes and Wheeler introduced 'initial orders', which was a rudimentary program loader, formed by 40 or so instructions, that loaded subroutines from a tape and replaced relative addresses in its instructions with absolute ones.[27]

A number of other tricks were devised to simplify the debugging of programs. Debugging programs through peeping, by running them instruction-by-instruction on a real machine was not only tedious but it was also taking valuable computer time. As the demands on the machine grew, peeping was recognised as 'extravagant use of computer time'[28] and was outlawed by the designers of EDSAC. Fortunately, the engineering method of using the computer to simplify programming led to two new approaches to the problem of debugging in the new setting.[29]

The first attempt was the development of a 'postmortem dump routine' that was triggered when the program terminated abnormally. The routine printed out a specified region of the computer memory and the programmers could then inspect the state and look for errors offline. The routine is the precursor of the idea of a core dump or memory dump that is produced, for example, when a contemporary operating system crashes.

The postmortem routine did not help when the program worked, but computed an incorrect result. Debugging such issues required a more sophisticated tool. This was implemented by Stanley Gill in 1951. His 'interpretive checking routine' was able to print diagnostic information while the main program was running. To do this, it iterated over the instructions of the main program and executed them one-by-one, simulating the behaviour of the EDSAC. This was about 50 times slower than running the instructions directly, but it provided an invaluable debugging tool. In modern terms, the routine implemented an EDSAC order code interpreter with detailed logging.

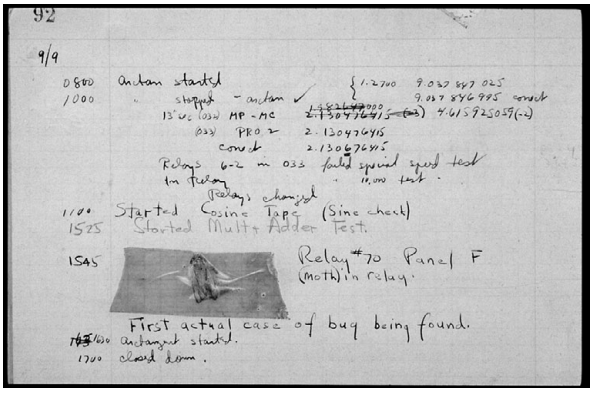


Figure 4.5 'First actual case of a bug being found' – a dead moth removed from the electromechanical Harvard Mark II computer by Grace Murray Hopper. (Source: Division of Medicine and Science, National Museum of American History, Smithsonian Institution)[30]

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

Before we continue, it is useful to review the terminology used to talk about programming issues in the 1950s. This has since evolved and so the historical meaning of the terms differs from the contemporary one. So far, I have been using the term 'debugging' to refer to the work that programmers and computer operators had to do in order to get their programs to run correctly. In the 1950s, this was one of the frequently used terms, although its meaning was broader and included activities that present-day programmers might call testing. The term debugging has its origins in the engineering slang used at the time. Despite a popular tale, the term 'bug' was not introduced into the vocabulary of programmers when Grace Murray Hopper removed a dead moth from one of the circuits of the Harvard Mark II computer and taped it into her logbook (Figure 4.5). The logbook with the dead moth is probably not hers,[31] but Hopper promoted the adoption of the term bug, which has been used by engineers to refer to malfunctions from the late nineteenth century. The use of the term for talking about programming issues was a notable rhetorical move. As pointed out by Kidwell,[32] by using the term 'bug' rather than a 'failure', programmers suggest that the issues are small faults that can be corrected. This proved an ironic naming, given that the estimate cost of work done on fixing the 'Y2K bug' at the end of the 1990s was over $300bn.

Another term that was often used alongside 'debugging' was 'checking' a program or 'program checkout'. An early example that uses this terminology is a paper 'Checking a Large Routine'[33] by Alan Turing from 1949. The paper illustrates that the early terminology around getting programs to run correctly was not clearly differentiated. Turing opens the paper by asking 'How can one check a routine in the sense of making sure that it is right?' He suggests that programmers can help the program checker by 'making assertions about various states the machine can reach'. Program checking in Turing's paper may be read more as an early attempt at proving a program correct than as debugging or testing. In other early systems, 'program checkout' was done, for example, by inspecting log traces produced by a tool such as the EDSAC interpretive checking routine. Throughout the 1950s, the terminology differentiated and the typical view was that you wrote a program, debugged it to make sure it ran and then 'checked it out' to make sure it produced the correct results. The task of 'program checkout' was more akin to what present-day programmers would call testing. Turing's early work that understood program checking more as proving programs correct was not immediately followed. The formal approach to program checking was, likely independently from Turing, reinvented in the mathematical culture of programming in the 1960s.

The shift from direct interaction with the machine to debugging tools was influenced by the business context. The machine time was expensive and postmortem and checking routines enabled programmers to debug their programs while the computer was available for running other programs. From the early 1950s, the operation of EDSAC was so streamlined that a full-time operator to run programs on behalf of the programmers was hired. The combination of managerial requirements and work of the engineering culture thus sidelined the direct approach of the early hackers. However the hacker culture did not have to wait long for another chance at transforming how we program and debug.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

4.3 On-line Debugging Techniques

Running a computer in a batch-processing mode was the norm in the 1950s and so debugging programs using memory dumps and diagnostic logs was the only option for most programmers. The exception from the norm were the MIT hackers who got their hands on the TX-0 computer, which we encountered in Chapter 3, after it was moved to the MIT Research Laboratory of Electronics (RLE) in 1958. The interactive input/output capabilities that were added soon after the move to MIT RLE made it possible to debug programs interactively, although the tools for doing so were initially very limited. The 1958 memo 'TX-0 Direct Input Utility System'[34] first goes to great lengths to justify the very idea of 'debugging at the console' and then acknowledges the current limitations:

> Debugging at the console is not new. In fact, it is the oldest form of debugging. For the past few years it has been regarded as a great professional sin, but like most sins, it still exists, is rarely talked about, and hardly ever admitted. …
> At the present time most utility programs do not provide enough immediate information to enable a programmer to find his error while he is still at the console. And if he can find it, few systems are capable of allowing him to make an immediate change without the preparation of a card, paper tape, or some other secondary input medium, especially if he wants to retain the symbolic language of the conversion program.

The Utility Tape 3 (UT-3) tool, which I mentioned in Chapter 3, made it possible to control the computer interactively, by typing commands on the console, but it was very rudimentary. Among other things, programs had to be typed in octal numerical system. The major advance in interactive debugging was the FLIT utility, written by Thomas Stockham and Jack Dennis. The latter was, in principle, in charge of the TX-0 operation at RLE, but was, in fact, more interested in playing with the machine himself. In a pun typical for the hacker culture of programming, the FLIT utility was named after an insecticide, but was presented as an acronym for Flexowriter Interrogation Tape.[35]

The FLIT utility made it possible to enter programs interactively using a symbolic notation that was very close to the assembly language used when writing the program in the first place. Rather than entering instructions as octal numbers, programmers could work with symbols as in their source programs. FLIT also introduced a debugging technique that remains the bread and butter of debugging to this day. The `break` command made it possible to set a breakpoint at a given location in the user program. When the program was run and reached the location, the control was transferred to FLIT, which printed the contents of registers and gave the user the options to interactively modify the state and program instructions or resume program execution.

Only one TX-0 computer was ever constructed and so the FLIT utility never left the world of MIT hackers. A successor tool was created by another hacker, Alan Kotok, for the DEC PDP-1 computer as soon as the computer was delivered to the MIT RLE in 1961. Following the insecticide-inspired naming tradition, this became known as the DEC Debugging Tape (DDT). The DDT utility was similar to FLIT and let the users modify programs using a symbolic assembler, referring to parts of a program using

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

the symbolic names used in the source code as well as set breakpoints. An interesting characteristic of both FLIT and DDT was their concise language of commands. In DDT, most operations were triggered using a single character and so the tools were very efficient in the hands of experienced hackers.

The appearance of the first time-sharing systems in the early 1960s renewed interest in on-line program development and debugging tools, although much of the work was still happening close to MIT and Cambridge. One such collaboration was between Thomas Evans at Air Force Cambridge Research Laboratories and D. Lucille Darley who worked at Bolt Beranek and Newman (BBN), a Cambridge-based company with close links to MIT that we encountered in Chapter 3. The two wrote an extensive review of existing on-line debugging techniques and created a new tool named DEBUG.[36] The DEBUG tool addressed two issues with FLIT and DDT. First, it made it easier to modify a program by automatically relocating parts of a program to make space for the new code. Previously, this had to be done by inserting the new code in an unused part of memory and inserting error-prone jumps to the right places in the code. Second, DEBUG also made it possible to easily export a clean corrected version of the debugged program. It kept a table of edits performed manually during the interactive session and was able to print out on an 'alter tape' after the end of the debugging session.

In their review of on-line debugging techniques,[37] Evans and Darley remark that time-sharing systems make on-line debugging economically feasible and that facilities for program debugging is an 'area of critical importance for effective utilisation of such time-sharing systems'. The motivation for their review was to document the systems that existed at the time, either in early time-sharing systems or for smaller computers such as TX-0 or PDP-1, which were used interactively at some installations. In doing so, the review also highlights one point that clearly positions work on debugging utilities in the realm of the hacker culture of programming. Evans and Darley noted that 'much of the work in this field has been described only in unpublished reports or passed on through the oral tradition'. Moreover, some of the tools they review 'are far from completely described even in internal memoranda'. In other words, debugging relies on personal knowledge that is difficult to write down and is gained through practice and mentorship. The reliance on oral tradition may also have been a reason why much of the early work on debugging tools was confined to MIT and affiliated organisations.

Tools like FLIT and DDT were a step up from UT-3 and flipping of switches on early machines in that they made it possible to use a symbolic language during their debugging sessions. Their users no longer had to modify the numeric codes representing the instructions, but could instead type assembly language instructions. In 1966 when Evans and Darley published their survey, on-line debugging techniques existed for both assembly languages and for high-level programming languages including Lisp and FORTRAN, although the ones for high-level languages were 'less well-developed and less widely used'.

Programming systems with on-line debugging facilities for high-level programming languages started to develop in the mid 1960s on early time-sharing systems. The time-shared IBM 7044 computer came with a FORTRAN implementation QUICKTRAN

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

and a multi-user FORTRAN also existed for the Berkeley time-sharing system. Those systems generally allowed users to set breakpoints (by inserting appropriate instruction into a program) and inspect the values of variables (using the symbolic name of the variable). QUICKTRAN was based on an interpreter and so it also enabled users to insert and delete statements in the program source code, which was not possible in systems that executed compiled code. The idea of an incremental compilation, which is used by some debugging tools today, already existed in literature, but was not implemented in a real-world system at the time.

The higher-level language with the most advanced interactive programming and online debugging features in the mid 1960s was Lisp. Lisp has been around since the end of the 1950s, but the first implementations were written for batch-processing systems. By 1964, Lisp was available on a number of interactively used machines such as PDP-1 at Stanford, as well as on time-sharing systems including the IBM 7094 computer running the Compatible Time Sharing System (CTSS) at MIT. The first time-sharing implementations of Lisp featured fairly basic interactivity. The time-sharing Lisp running at MIT let users set breakpoints, evaluate Lisp expressions interactively and print the backtrace (recording what functions were being called) when an error occurred, but this was soon about to change.[38]

Thanks to its nature, Lisp has been the perfect language for the development of advanced debugging tools and Lisp hackers soon started to exploit this characteristic. In Lisp both data and code are represented as lists, a feature known as *homoiconicity*. This means that code of a running program can be easily represented and modified during the program execution. As we saw in Chapter 3, a number of Lisp implementations also included editors that let users edit Lisp code not as text, but by directly modifying the list structure representing it. As in the case of FORTRAN, running the code modified interactively required having an interpreter, in addition to a compiler that turned the initial program into efficient instructions. The work on Lisp editing and on-line debugging came together in MacLisp, which ran on the Incompatible Timesharing System (ITS) at MIT. In the aforementioned survey of debugging tools, Evans and Darley describe a scenario illustrating the possibilities of the system:

> With some care, it has been possible, for example, to find a bug while at a breakpoint in running a test case, call the [structure] editor to make a correction, run the program on a simpler test case to verify the correctness of the change, then resume execution of the original test case from the breakpoint.[39]

One interesting aspect of this scenario is how it integrates the different activities involved in getting a program to run correctly. It starts with running a test case, which is an activity we would today refer to as testing; it follows with making a correction in a tool that we would now refer to as the debugger and then verifies this correction, which is something that a present-day programmer would perhaps do interactively using a REPL (Read-Eval-Print Loop) shell. In other words, the on-line debugging techniques of the 1960s do not yet make a clear distinction between development, debugging and testing. More interestingly, error handling is also closely interconnected with the other activities.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

A system that illustrates this well is the BBN-LISP and the corresponding project manual published in 1967.[40] The list of authors reflects the capabilities of the system. It includes Daniel Bobrow, D. Lucille Darley (a co-author of the DEBUG tool and the aforementioned survey), L. Peter Deutsch (who created the structure-based editor for Lisp), Daniel L. Murphy and Warren Teitelman (whom we encountered in Chapter 3 as the author of the PILOT system).

The connection between debugging and error handling is immediately clear in the report, because the two topics are discussed in a single section, 'Error Handling and Debugging Functions'. The section opens by explaining that there are two classes of errors, '`H` errors for which the user can provide Help on the spot; and $\overline{\texttt{H}}$ errors for which no help is possible'. By default, the $\overline{\texttt{H}}$ errors stop all computation, printing functions that have been entered by the failing computation. The error can be handled in code using the `errorset` function, which behaves as an exception handler in modern programming languages. It runs a computation specified as an argument and, if no error occurs, returns its result. If an error occurs, the special value `NIL` is returned.

`H` errors allow the user to fix the mistake and let the program continue. This includes cases where a function or a symbol is undefined (or misspelt). The user can fix such errors interactively by typing the correct name or defining the function during the debugging session. The user can also induce (trigger) an `H` error interactively to break the execution as soon as the next function is entered. In addition to this, BBN-LISP supported more conventional breakpoints, set by invoking the `BREAK` function. This modified a specified function to stop the execution when the function was called, optionally also checking that a certain break condition was satisfied.[41]

By the end of the 1960s, many different tools for on-line debugging, testing and error handling were developed. However, there was still no clear distinction between debugging, testing and error handling and the different tools were often used together. This required practical hacker knowledge held by the practitioners that was more often shared through personal interactions, than through written documentation. However, the ongoing shift from the black art of programming to software manufacturing based on rigorous scientific methods was about to transform the practices of debugging, testing and error handling.

4.4 The Debugging Epoch Opens

While the on-line debugging techniques for the early interactive computers and time-sharing systems were appearing in the 1960s, most commercial programmers were still debugging their programs using the two methods developed over a decade before for EDSAC: the postmortem memory dump and various forms of tracing that printed diagnostic information during the program execution. This was increasingly recognised as a major obstacle for software development and, in November 1965, Mark Halpern made the problem clear in his article for the *Computers and Automation* magazine titled 'Computer Programming: The Debugging Epoch Opens'.[42] In the article, Halpern commends the recent developments in hardware and programming languages

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

and discusses the next major challenge: 'We have machines big and fast enough to execute most useful programs, and software varied and reliable enough to get those programs written; what now? Now: debugging'.

Halpern, who had experience working at IBM and was now based in a research laboratory at Lockheed, saw 'the debugging problem' as the new limiting factor for the growth of the computer industry. He used the term debugging in the broader sense of identifying and correcting programming errors and proposed a number of new debugging techniques. One such technique that remains commonplace to this day was distinguishing program operation in 'Debugging Mode' and in 'Production Mode'. Other techniques he proposed were not unlike some of those that were already becoming possible in Lisp environments developed at the time by the hacker culture. Halpern identified a problem that was becoming increasingly visible at the time and envisioned some tools that would become available later, but he did not anticipate a different kind of response to the problem that was starting to take shape.

At the end of the 1960s, many stopped seeing the difficulty with software development as the debugging problem of eliminating small program flaws and started seeing it as an industry crisis instead.[43] This was a major cultural shift. As discussed in Chapter 2, practical programming in the 1950s was dominated by the hacker culture of programming. It was a black art that relied on personal knowledge and the results were not reproducible or reusable. This state of the art was not acceptable to either the mathematical culture, which sought formal foundations for programming, nor the managerial culture, which wanted to be able to produce software of a predictable quality at a predictable cost.

Tackling the problematic state of computing became the subject of the NATO Software Engineering conference held in 1968 in Garmisch. The idea for the conference came from Fritz Bauer who was a professor at the Technical University of Munich. Bauer was actively involved in the development of Algol 68 and although he considered the language 'too fat',[44] he was not a signatory of the Algol 68 Minority Report that I discussed in Chapter 2.

The conference was organised by Bauer and the NATO Study Group on Computer Science, which was established in the autumn of 1967 by the NATO Science Committee.[45] The organisers decided to structure the conference program along the three themes of design of software, production of software and service of software. They identified group leaders for each of the topics, many of whom were previously involved in the Algol efforts, and invited a limited number of 'carefully chosen leading figures' from different backgrounds and different countries.[46] The attendees included a diverse group of academic and industry programmers, but the attendee list notably lacked representatives of the hacker cultures of programming. The MIT hackers that we encountered in Chapter 3 would likely disagree with many of the assumptions of the conference, but they also did not fit the traditional profile of an academic or industry leader in the field.

Consequently, the conference attendees quickly agreed on the problem with the 'black art' approach of the hacker culture of programming. The agreement on the

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

problem at the 1968 Garmisch conference, however, contrasted with the disagreement on the solution at the follow up conference in 1969 in Rome. The different cultures of programming interpreted the term 'software engineering' differently and were more interested in pursuing their own agenda than agreeing a common approach. According to the mathematical culture, the answer was the development of methods for formal reasoning about programs. The managerial culture saw the answer to the software crisis in scientific management of programming teams. However, the conference also marked the rise of the engineering culture of programming that hoped to address the software crisis by the development of new and better tools and programming practices to assist the programmer.

The development of software engineering brought a more structured approach to thinking about software and its development.[47] A concrete example of this was structured programming that I discussed in Chapter 2 and that soon acquired both engineering and managerial interpretation. However, the structured nature was apparent in the very organisation of the Garmisch conference. The conference program was divided, following the steps of the emerging development process, into sessions on design, production and service. The NATO conference report also shows more differentiation in the terminology used for talking about debugging and testing. It only mentions debugging in a general sense, but includes a number of more thorough perspectives on software testing.

First of all, the authors of the working paper 'The Testing of Computer Software'[48] relate testing to the aim of the conference to provide a scientific basis for software development. The authors note that 'testing is one of the foundations of all scientific enterprise' and argue that software should be 'rigorously tested for conformity with the specification'. They go on to distinguish two different aspects of such testing:

> This testing has two aspects: that carried out by the producer of the software to satisfy himself that it is fit for release to his customers and that carried out on behalf of the customer in order to give him sufficient confidence in the software to justify his paying for it.

The working paper notes that the unfortunate current strategy is that 'most customers accept the software without any form of test' and suggests a testing strategy that includes checking the documentation, system availability, verification of functionality and performance assessment. In other words, they describe the kind of testing that we might today refer to as acceptance testing.

A more fine-grained perspective distinguishing different kinds of tests is from John Nash at IBM whose working paper includes an illustration (Figure 4.6) that distinguishes unit tests, component tests and system tests. As we will see, this kind of division was becoming more commonplace as one aspect of the structured development process that later became known as Waterfall. Finally, another interesting idea from the NATO 1968 report is from a working paper 'Aids in the Production of Maintainable Software'. The author argues that testing should be automated and that 'a collection of executable programs should be produced and maintained to exercise all parts of the

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

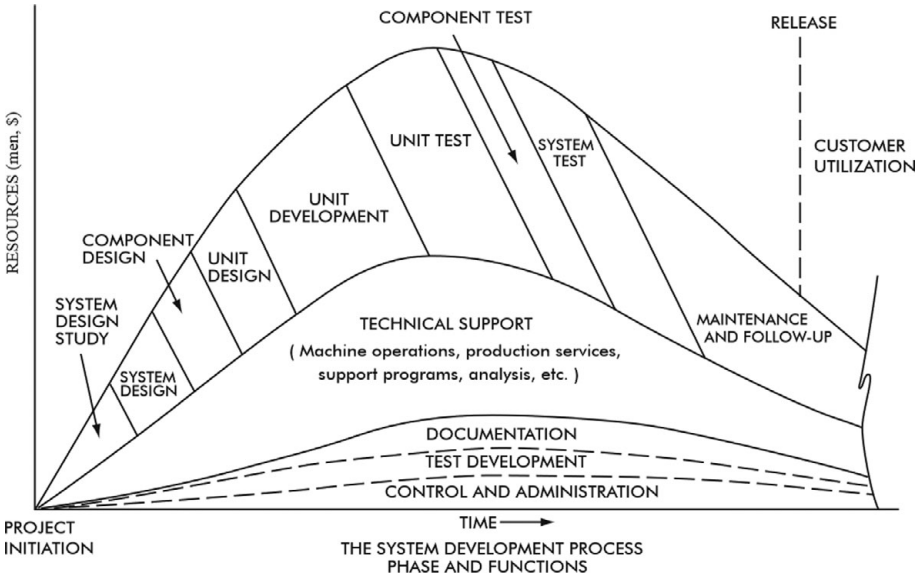


Figure 4.6 Figure 'Some problems in the production of large-scale software systems' from the NATO 1968 Software Engineering conference report. (Source: Drawing by Robert M. McClure.[49] Reproduced with permission.)

system'. This suggests the idea of test automation that will be implemented in a number of systems in the decades to follow.

The NATO conference proceedings show that the task of getting a program to run as required, which was interchangeably called checkout, debugging or testing in the 1950s, started to differentiate into a number of distinct concepts. Most importantly, the notion of testing was becoming a multi-faceted concept that covers the more managerial notion of acceptance testing done by a customer, as well as the engineering idea of automated testing as practised for example by modern unit testing tools.

Testing and debugging also appears in the report from the 1969 Rome conference. Interestingly, most of the discussion appears in the 'Software Quality' section under the 'Correctness' subsection, which covers formal correctness ('how to prevent bugs from ever arising') and debugging ('how to catch them when you have got them'). It is the section on formal correctness that contains an argument in favour of careful testing by Perlis alongside a famous remark by Dijkstra that '[t]esting shows the presence, not the absence of bugs'.

The discussion on debugging in the report is also enlightening. The list of participants of the 1969 Rome conference includes Warren Teitelman, who worked on the on-line debugging tools for Lisp and is possibly the sole clear representative of the hacker culture in the list of attendees. His working paper, 'Toward a programming laboratory' presented much of his work on Lisp, including the PILOT system and interactive error correction. The working paper sparked a number of discussions at the conference, in particular on the relative merits of off-line debugging and on-line

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

debugging. Niklaus Wirth, who worked on the Pascal programming language at the time, expressed his worry about the new developments:

> There has been, since the advance of timesharing and on-line consoles, a very hectic trend towards development of systems which allow the interactive development of programs. Now this is certainly nice in a way, but it has its dangers, and I am particularly wary of them because this conversational usage has not only gained acceptance among software engineers but also in universities where students are trained in programming. My worry is that the facility of quick response leads to sloppy working habits and, since the students are going to be our future software engineers, this might be rather detrimental to the field as a whole.[50]

Wirth's comment comes from the perspective of the mathematical culture, arguing that programmers should first thoroughly, and perhaps even formally, think through the problem before writing any code. The engineering perspective defended by Alan Perlis in response is that 'people don't think like that; conversational systems permit programmers to work from the top down and to compute even with incomplete specifications'. Teitelman unsurprisingly shares this view and argued that conversational systems promote 'a more efficient, systematic approach to debugging and testing'. In the report, Teitelman does not elaborate on what he means by 'systematic', but making that explicit would likely pose a challenge. The ambition of the NATO conferences was to transform programming into a proper engineering discipline. This had almost no effect on debugging, which remained a hacker practice that is learned through experience and relies on the skills of individual programmers. The systematicity of debugging remained rooted in the personal unwritten knowledge of hackers. The shift also did not visibly affect the work on interactive programming, which we followed in Chapter 3. The conferences did however, mark the start of a transformation of exception handling and software testing.

4.5 Orderly and Reliable Exception Handling

When a program running on a 1960s batch-processing system encountered an unexpected situation, it would terminate and optionally produce a postmortem dump. In interactive programming systems, an unexpected situation would break the execution so that a programmer could analyse the situation, correct it and resume the execution. In both cases, the assumption was that the unexpected situation would eventually be handled by a programmer. However, one of the premises of software engineering was to produce reliable programs that can gracefully recover from unexpected situations.

In Lisp, the burden of handling unexpected situations could be moved from the programmer to the program itself. LISP 1.5, the first widely available version, included a function `errorset`, which I mentioned when discussing on-line debugging. When an error occurred inside code wrapped in `errorset`, the function returns `NIL` and the programmer can write code that recovers from the error. The LISP 1.5 manual is careful to say that continuing computation in a different direction when one path results in an error is only possible if the failing code has not modified any variables or

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

caused other damage. In other words, doing the recovery correctly is still a task for a competent hacker.

Another early programming language that contributed to the development of exception handling is PL/I. The language predates the birth of software engineering and was developed at IBM as part of a broader effort to bridge the gap between scientific programming (typically done in FORTRAN) and business data processing (typically done in COBOL). In the 1960s, IBM set out to build a single system to cover all possible applications of computers. It designed the IBM System/360 computer range as well as the operating system OS/360 and also started work on a new multi-purpose programming language. The designers of PL/I believed that the strict separation of scientific and business computing was a false stereotype and that programmers needed a single language to complete all their programming tasks. Some also hoped that having a single language may relieve IBM from the burden of maintaining multiple compilers.[51] Although PL/I was modelled after FORTRAN, it included rich data structuring capabilities and built-in functions for the handling of I/O operations. Those operations for reading data were a common source of program failures, which influenced the design of the language:

> [Languages before PL/I] had few general facilities to define actions which were to be taken when conditions such as reading an end of file, overflow, bad data, new line, occurred. Often the conditions could not even be sensed in the language, and assembly language patches were required. … The [PL/I] designers were determined that entire real applications should be programmable in the language and these programs be properly reliable and safe.[52]

To make programming of reliable and safe applications in PL/I possible, the language designers defined 23 types of error conditions and the `ON` construct which could be used to specify an action to execute in case an error condition occurred. The construct was cumbersome to use, partly because passing information to the exception handler often required using global variables and because the action of registering a handler had to be cautiously explicitly reverted.

Although LISP 1.5 and PL/I introduced much of what is needed for modern exception handling, it took another decade until more sophisticated exception handling started to became commonplace. The developments in LISP 1.5 and PL/I are notable, because they contributed to the differentiation of the various tasks involved in getting programs to behave correctly. Whereas error handling in the 1950s was closely interconnected with testing and debugging and was a matter for hackers, the constructs like `ON` and `errorset` make it a part of the programming language itself. The terminology shows that this was a gradual process. When talking about the `errorset` function, the LISP 1.5 manual[53] talks about 'errors', while the PL/I reference[54] talks about 'exceptional conditions' and introduces the `ON` statement in a section on exception control statements.

Internalising error conditions into the programming language moves the problem of error handling into the realm of the engineering and even mathematical culture. PL/I is a good example. The language originated from engineering efforts at IBM. It was partly inspired by Algol, but the PL/I design lacks the focus that would be typical for the mathematical culture. It is not done with reference to a theoretical basis and does not

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

emphasise the ease of proving correctness. However, PL/I was also extensively studied at the IBM Vienna lab, which produced multiple formal definitions of the language. For some of the formalisation techniques, Algol served as a testbed and this resulted in the Algol specifications that I covered in Chapter 2.[55]

Programming language work on exception handling has been inspired by a more general focus on reliability in the software engineering community that started to emerge in the wake of the NATO Software Engineering conferences.[56] In 1975, John B. Goodenough published two papers that made major contributions to exception handling and testing. We will return to the latter in the next section. Goodenough was working at a software consultancy developing software engineering tools and solutions, so he came from a markedly engineering background. Yet, his proposal for an exception handling mechanism was presented at the ACM Symposium on Principles of Programming Languages (POPL) where it appeared among talks mostly rooted in the mathematical culture of programming. It was then revised and published in the widely read *Communications of the ACM*.[57]

The exception handling design proposed by Goodenough shares many aspects with the modern exception handling mechanism as known, for example, from Java. Exception handlers are associated with blocks of code and can specify code to run for various exceptions that may be triggered during the execution of the enclosed code block. There are a number of built-in exceptions, but users can also define their own. Furthermore, procedures are annotated with exceptions that may be raised and Goodenough argues that the compiler should check that all possible exceptions are handled. In a feature that goes beyond what is supported by most programming languages today, the handler can also resume the execution of the code that raised the exception. There are three different constructs for raising exceptions: `ESCAPE` requires termination, `NOTIFY` forbids termination and `SIGNAL` where the termination behaviour is decided by the exception handler.

Goodenough's paper was a detailed proposal for an exception handling mechanism. It was a theoretical publication and did not come with a specific implementation, but it influenced the design of two programming languages from the 1970s, Clu and Ada, that embody the ideals of software engineering in the form of a programming language. Clu was created by Barbara Liskov at MIT and was motivated by the work on programming methodology and information hiding. Its key innovation was the design of abstract data types, which I cover in Chapter 5, but Liskov also wanted to support 'robust or fault-tolerant programs … that are prepared to cope with the presence of errors'.[58] For this reason, Clu adopted an exception handling mechanism that was inspired by both Goodenough's paper and the mechanism in PL/I. The designers of the Clu language made two design decisions related to exception handling that are common in programming languages today. First, they referred to the mechanism as 'exception handling' and, second, they only implemented the termination model, that is, there was no way to resume the program from the original location after an exception.

The development of the Ada language was led by the US Department of Defense (DoD) and was the result of a managerial 'Software Initiative' to reduce the 'High Costs of Software'. The designers believed that 'a common language for use through-

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

out the DoD would be a first step to building an engineering discipline of software, to supersede the then current "black art"'.[59] The design of Ada was a product of thorough and detailed analysis and was described in a 250-page long 'Rationale for the design of the Ada programming language'.[60] John B. Goodenough was directly involved in the Ada design process and the Rationale makes a reference to his work, as well as other ongoing research on exceptions. Just like Clu, it adopts the terminology of exception handling and also the termination-only mode of exceptions.

Whereas the development of exception handling in PL/I was focused on solving the specific problem of handling of specific error conditions related to input and output issues, much of the later work influenced by the rise of software engineering had a broader realm. Already in Lisp, the exception handling mechanism based on `errorset` was used 'not to trap and signal errors but for more general control purposes (dynamic non-local exits)'[61] and MacLisp later added a pair of primitives `catch` and `throw` so that the original exception handling primitives could be reserved just for error handling. More interestingly, Goodenough also saw the proposed exception handling mechanism 'in general, as a means of conveniently interleaving actions belonging to different levels of abstraction'.[62] This was the case especially with exceptions that are intended as notifications for the caller who then resumes the execution of the code triggering the notification. Similarly, in Clu, exceptions were seen as 'a general way of conveying information from a called procedure to its caller'.[63]

In the case of Lisp, the LISP 1.5 manual indicates that `errorset` can be used to 'continue computation in a different direction when one path results in an error',[64] but does not indicate that the mechanism can be used for purposes not related to errors. The more general use of `errorset` for control flow was likely the result of the ingenuity of the hacker culture, which was able to exploit the mechanism for originally unintended clever programming tricks. Goodenough's paper presented a cultural shift and exceptions were seen as a language mechanism that enabled multiple good software engineering practices, including both error handling for reliability, but also good structuring of code into multiple levels of abstraction. However, Clu, Ada and later Java again constrained the use of the mechanism to just error handling. In a perhaps surprising plot twist, the trend has again been reversed in the 1990s in research on algebraic effect handlers that are, yet again, an exception-inspired programming language construct for interleaving actions across multiple levels of abstraction.[65]

4.6 System Structure for Fault Tolerance

The work on exception handling in programming languages provided a way of recovering from errors and suggested a more general program structuring mechanism. It also served as the basis for over-engineering, another practical engineering practice that Hoare later saw as contributing to software reliability. In the analysis of exceptions that Goodenough presented in his paper, he classified exceptions into two categories. A *range failure* 'occurs when an operation … finds it is unable to satisfy its output assertion' and *domain failure* 'occurs when an operation's inputs fail to pass certain

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

tests of acceptability'. In other words, if something goes wrong when performing the operation, it is a range failure, but when the operation is invoked incorrectly, it is a domain failure.

The former category of program errors is inevitable. A program cannot prevent real-world situations such as a corrupted file. However, in a perfect program imagined by the mathematical culture of programming, a domain failure would never happen. A program would be written so that the acceptability of inputs is ensured by the callers of the operation and, in an ideal world, even guaranteed by a formal proof. The idea that an operation should test acceptability of inputs is thus a basic form of over-engineering, or, pre-empting errors that should not happen in theory, but does happen in practice. In modern programming terminology, the practice of checking for invalid input is widely adopted as part of the *defensive programming* methodology that also dates back to the mid 1970s.[66]

The software engineering research on software reliability inspired a number of other programming language designs that would simplify over-engineering. One example that illustrates possibilities beyond those that are supported in main-stream programming languages today came from Brian Randell. Randell was a signatory of the Algol 68 Minority Report, attended both of the NATO conferences, was the editor of the proceedings for the first one, and dedicated much of his career to work that contributes to the engineering culture of programming. In 1969, he moved from IBM to Newcastle University and started working on software reliability. In 1975, he published an influential paper 'System Structure for Software Fault Tolerance'[67] that introduced *recovery blocks*.

Recovery blocks are a programming language feature designed for building of fault-tolerant systems that can recover from design inadequacies. A recovery block is similar to an exception handler, but the body is guarded by an acceptance condition (a check for a range failure). If the condition is violated by the primary implementation, the block proceeds to execute one of the alternative, possibly simplified, implementations, until it obtains a result that satisfies the acceptance condition. Another mechanism implementing the idea of over-engineering that emerged from the software engineering community in the mid 1970s was to build systems with a self-checking mechanism.[68] Such systems would actively monitor themselves to detect range and domain failures. While none of the mechanisms going beyond exception handling appeared in widely used programming languages at the time, they normalised over-engineering as a way of tackling programming issues and introduced a mechanism that was, in various forms, implemented in later programming systems such as Erlang.[69]

Following the birth of software engineering, exception handling moved from a manual task for a programmer into a condition to be handled automatically, by the program itself. Consequently, it also turned from a feature of an interactive programming system into a programming language construct. Those working on exception handling started to come less from the hacker culture and more from the engineering and mathematical culture of programming. Testing, another aspect of reliable software development, was also soon to be transformed by the birth of software engineering.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

4.7 Professionalisation of Testing

New efforts to organise the community around software testing emerged soon after testing became recognised as a separate activity. In 1972, Bill Hetzel organised the Computer Program Test Methods Symposium at the University of North Carolina that brought together many of those interested in testing. Mirroring the narrative about the state of software development as a whole, Hetzel complained that testing 'is all art' and lacks an 'established discipline to act as a foundation'.[70] The symposium and the book 'Program Test Methods',[71] which was edited and published by Hetzel following the symposium, were an attempt to put testing on a rigorous basis.

Many saw Hetzel's efforts as a promising step. According to a report written for the ACM by one of the symposium attendees, David J. McGonagle, 'the programming industry took another step toward maturity'[72] with the symposium. This step was more in the ambition of the symposium than in its contents and the presentations at the event illustrate that there was not yet a coherent testing community. The presented papers include managerial reports on testing done for large software systems, engineering reports on new testing tools, proposals for standardising software testing practices, but also a number of mathematical talks on proving program correctness. As reported by McGonagle, the event 'served to bridge the gap between the proof of correctness people on one hand and the large systems users on the other'. Although McGonagle doubted that the event 'convinced any of the non-believers of the critical nature of this area of computer systems work',[73] the symposium was the first step. It was followed by a series of events on testing that, over the next two decades, established software testing as a recognised software engineering discipline.

The organiser of the 1972 symposium later co-founded a consultancy focused on software testing together with David Gelperin. The two played an instrumental role in promoting software testing and co-authored a paper 'The Growth of Software Testing'[74] , which also further helped to establish testing as a discipline by providing its origin story. In the paper, Gelperin and Hetzel recognise the early 1970s as the era when the level of professionalism in software testing significantly increased. In this period, the first software testing conferences took place and testing became the topic of an increasing number of academic papers. Companies started to recognise software test engineering as a speciality and started to hire specifically for positions such as 'test manager' or 'test technician'.

In the 1970s, testing became clearly distinguished from debugging. The goal of the latter was to 'make sure the program runs' while the goal of the former was to 'make sure the program solves the problem'. As the positive phrasing suggests, testing was seen as a way of showing that the software satisfies its requirements. As such, it had the same ultimate objective as the work on proving programs correct, but it utilised engineering rather than mathematical methods. This led to a debate between the proponents of the engineering culture and the mathematical culture, which questioned the extent to which testing can guarantee program correctness.

At the 1969 NATO Software Engineering conference, C. A. R. Hoare argued that one can easily show the 'ultimate futility of exhaustive testing'. In the report, this is

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

contrasted with a point made by Alan Perlis that '[m]uch of program complexity is spurious and a number of [well-chosen] tests will exhaust the testing problem'.[75] The two quotes illustrate a debate about the limits of testing between the mathematical culture and the engineering culture. The proponents of the former argued that testing is inferior to proofs, while the proponents of the latter saw testing as a promising practical tool. Interestingly, the most convincing arguments trying to show that testing can guarantee the absence of bugs came from engineers who utilised mathematical methods to counter the arguments made by the proponents of the mathematical culture by using their own means.

Three prominent examples of this approach appeared in the mid 1970s.[76] The first was from J. C. Huang whose PhD was in switching and automata theory, but who also had practical experience as an engineer.[77] Huang acknowledged that doing exhaustive testing, for a very simple program that takes two 32-bit integers as inputs and completes in 1 millisecond would take 50 billion years. His answer was to not treat the program as a black box, but analyse its structure and find a set of tests such that each branch in the program would run at least once (Figure 4.7). John B. Goodenough, who we encountered in the section on exception handling, developed another method for choosing test inputs.[78] He also indirectly responded to Hoare's comment claiming that 'properly structured tests are capable of demonstrating the absence of errors in programs'. Finally, at the same time, Thomas J. McCabe, who was employed by the National Security Agency and later established his own software consultancy, published the 'cyclomatic complexity' measure.[79] The measure is defined in terms of the number of branching points, paths and loops in a program flowchart. Although this was later used without reference to testing, McCabe suggested using the metric as a basis for a testing strategy. If the number of tests was lower than that required for a program of a certain complexity, either more tests would need to be added or the program structure would need to be simplified.

The next period of software testing that Gelperin and Hetzel refer to as 'destruction-oriented' started in 1979 with the publication of the textbook *The Art of Software Testing* by Glenford Myers.[80] It is ironic to see 'art' appear in the very title of the book, but the author does not advocate the earlier 'black art'. Instead, the title is intended to suggest that the book takes a practical rather than theoretical perspective. The author cautions the readers that 'many subjects related to testing such as the idea of mathematically proving the correctness of a program were purposefully excluded'. The book adopts an engineering perspective that draws on methods for test selection developed throughout the preceding decade. Yet, it still recognises the hacker heritage of software testing. In a section on 'Error Guessing', Myers admits that some people seem to have a knack for smelling out errors and that their error guessing technique is largely an intuitive and ad hoc process.

The book is a significant milestone not just because it is the first book solely dedicated to software testing, but because it shifts the perspective on the purpose of testing. Meyers sees testing as 'the process of executing a program with the intent of finding errors'. This way of thinking about testing presents a shift from the earlier view which saw testing as a mechanism for showing program correctness. The new perspective is

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

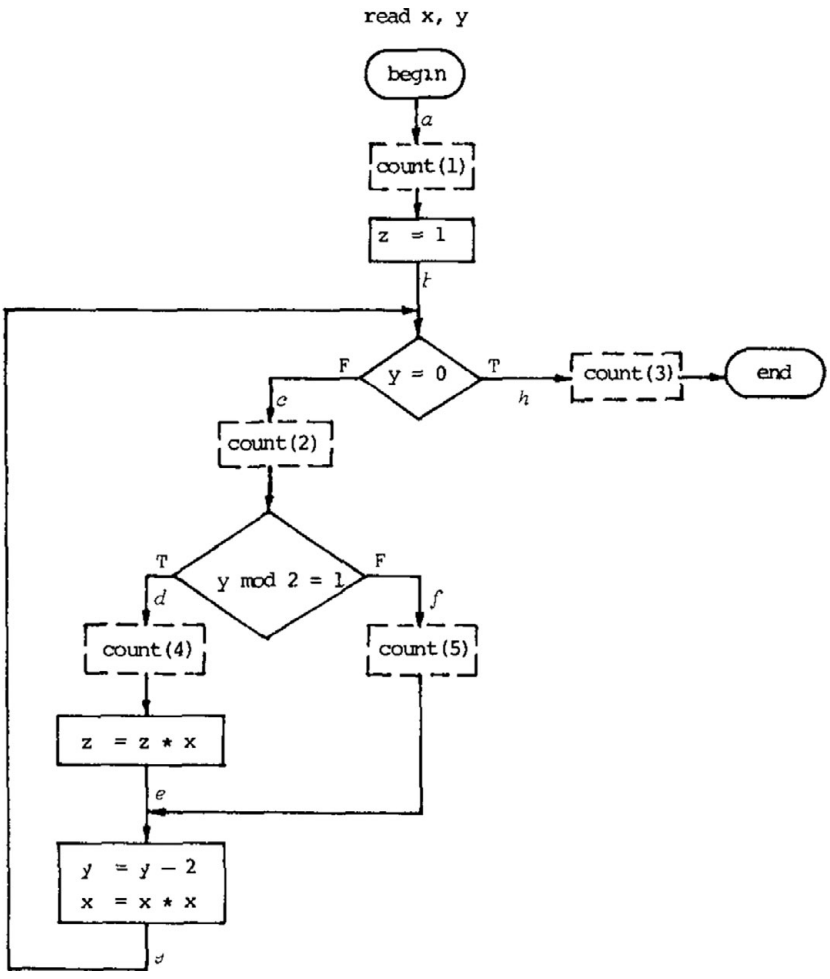


Figure 4.7 A flowchart for a sample program. Each branch (arrow) is annotated with a counter `count(n)` with different number `n` for different counters. A good set of tests will make sure that each counter is incremented at least once. (Source: Used with permission of ACM,[81] permission conveyed through Copyright Clearance Center, Inc.)

a shift from a managerial thinking of testing (acceptance testing done by a customer as part of the development process) to an engineering thinking (testing done by engineers to help them with their job). The perspective happens to be more practically effective as it encourages test selection that is more likely to find faults. It prompts testers to look for edge cases that might not be considered if the aim of testing is to make sure that a program works as required. Finally, the new perspective on testing aligns it with other approaches to fault detection. For example, Myers' book talks about code inspections and code walkthroughs and labels those as non-computer-based (or human) testing.

The hallmark method of the engineering culture is to create new tools that assist with solving the problem at hand. Such tools tend to be less well documented in the aca-

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

demic literature, but there are a number of good examples for the destruction-oriented period of software testing. The idea for one such tool occurred to Richard Lipton in 1971 when he was a student, long before he became known as a co-author of the paper 'Social Processes and Proofs of Theorems and Programs', which I talked about in Chapter 2. The idea was termed 'mutation testing' and it was published both as a paper and as a tool presentation in 1978.[82] The idea of mutation testing is to automatically change some aspect of the program logic and see if this change is detected by the tests. For example, a program that sorts a list of numbers will contain a comparison `X<=Y` to check if one element is less than or equal to another element. A mutation testing tool might change this check to, for example, `X!=Y` testing for an inequality instead. A well-designed test suite can detect such changes in logic. If the tests fail to detect that the modified program is incorrect, it is a hint that more tests need to be added.

In parallel to the engineering developments that I followed in this section, testing also existed in the context of the managerial culture, where it was one of the phases in the application development lifecycle. To document this, we first need to return to the end of the 1960s and follow the managerial response to the industry crisis.

4.8 A Slightly Ominous Note for Information Processing Management

For the managers working in the commercial computing industry, the 1960s started on a high note. The industry was growing at an impressive annual rate of 27 per cent and the number of computers installed in the USA grew from 5,400 to 75,000 during the decade.[83] Yet, many were also increasingly aware of the limitations and issues with the use of computers. There was a growing body of anecdotal evidence of problems with complex computer systems that experts in the field were familiar with.[84] The Mariner 1 spacecraft was lost due to a programming error in its guidance system in 1962. The IBM OS/360, delivered in 1967, was delayed by nine months and cost the company half a billion dollars, four times the original budget. Fred Brooks Jr., the project manager of the delayed IBM OS/360 system later likened the development of large software systems at the time to a tar pit struggle:

> No scene from prehistory is quite so vivid as that of the mortal struggles of great beasts in the tar pits. … Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems – few have met goals, schedules, and budgets. Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty – any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion.[85]

The reasons for the issues were both technical and managerial. The rapid growth of the computing industry meant that computers were used to solve increasingly large and complex problems. It was no longer possible for a small group of highly skilled hackers to keep the structure of an entire system in their head while building it, and software development started to require large teams which, in turn, meant that much more coordination and communication was needed. The programming languages at

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

the time provided only basic mechanisms for structuring and organising large code. The fact that a 'subroutine library' referred to a filing cabinet with punched cards is merely testimonial to that.

For the engineering culture, the NATO Software Engineering conferences are often seen as the pivotal moment. The conferences brought together participants from universities, industrial research labs, computer manufacturers, computer software and service firms as well as governments. But as pointed out by Thomas Haigh[86] a more careful look reveals that most attendees were involved in the development of technically interesting software such as operating systems, programming languages and compilers. Notably absent were discussions of application software, which is the bread and butter of programming such as travel booking or payroll systems.

The managerial perspective on the growing problems with the rise of computers was best captured in a report 'Unlocking the Computer's Profit Potential'[87] (Figure 4.8) published by McKinsey & Company in 1968. The report was widely discussed in computer magazines after its publication. The 'Editor's Readout' of the *Datamotion* magazine from August 1968 labels the report as 'a slightly ominous note for information processing management', while the republication of the report in the *Computers and Automation* magazine in April 1969 ensured it reached an even broader audience. The authors of the report studied the use of computers at 36 companies. Despite the technological advances, from a profit standpoint, 'computer

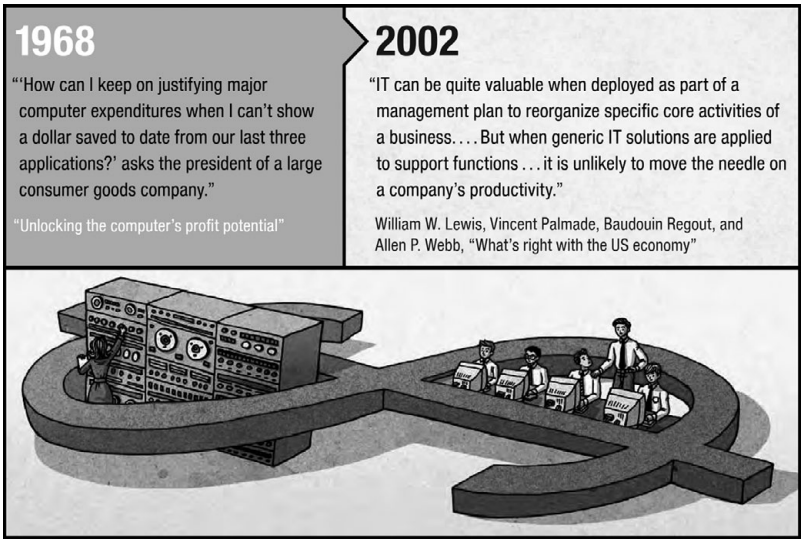


Figure 4.8 A quote from the 1968 report 'Unlocking the Computer's Profit Potential', reprinted alongside with a 2002 quote under a title 'Some things haven't changed much...'. (Source: Exhibit from 'Management, the next 50 years', *McKinsey Quarterly*, 2014 3, www.mckinsey.com. Copyright (c) 2024 McKinsey & Company. All rights reserved. Reprinted by permission.)

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

efforts in all but a few exceptional companies are in real, if often unacknowledged, trouble'.

The basic problem, according to the report, is that most companies completed 'conversions of routine administrative and accounting operations to computer systems', but are now finding it difficult to effectively apply computers in other areas. The computer staff who completed the conversion projects are now leading computerisation efforts, but are 'seldom strategically placed (or managerially trained) to assess the economics of operations fully or to judge operational feasibility'. The McKinsey recommendation was for the top management team to take a more active role in the management of computer projects:

> Many otherwise effective top managements, however, are in trouble with their computer efforts because they have abdicated control to staff specialists … Only managers can manage the computer in the best interests of the business. The companies that take this lesson to heart today will be the computer profit leaders of tomorrow.

The report coincided with a shift in thinking about the nature of programming jobs.[88] Before the end of the 1960s, programming was seen as a creative activity that required unique skills and often also unique treatment. Towards the end of the decade, the literature aimed at top management stopped talking about programmers as weird but ingenious hackers and started seeing them as any other kind of worker. The managerial approach to turning programming into a 'proper engineering discipline' was thus modelled after industrial factories and industrial production lines. The managers hoped that such a model could lead to a reliable and predictable software production, without relying on particular skills of individual programmers. They started looking for a methodology that could control the development process, not so much by appropriately controlling and structuring the code being written, but by appropriately controlling and structuring the team writing it.

Many different attempts at managing programmer teams soon appeared. In Chapter 2, I discussed the Chief Programmer Team methodology and the Cleanroom software engineering approach, which were modelled on the structure of a surgical team and were developed by Harlan Mills at IBM throughout the 1970s. Other approaches include egoless programming,[89] which was a democratic management structure focused on collaboration with a rotating leadership. However, the dominant approaches were top-down development methodologies inspired by the engineering idea of structured programming. The system had to be specified, decomposed into parts and then developed according to the specification by individual teams of programmers. The most prominent structured approach to software engineering became known as the Waterfall development model.

4.9 Production of Large Computer Programs

The software development model that became known as Waterfall in the 1970s was, in fact, already in use in the 1950s during the development of the Semi-Automatic

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

Ground Environment (SAGE) system that I briefly mentioned in Chapter 1. SAGE was a system of computers to process live data from multiple radar sites across the USA that was responsible for identifying a Soviet nuclear strike and coordinating the response. It was an effort of unique scale. The core development team 'grew from 125 personnel in 1956 to 1,400 in 1962' and the organisation behind SAGE 'trained more than 700 programmers between 1955 and 1957'.[90]

The development methodology that was eventually used for the development of SAGE has been presented in a 1956 paper 'Production of Large Computer Programs' at the Symposium on Advanced Programming Methods for Digital Computers in Washington DC, but without any specific reference to SAGE, which was not yet publicly announced. The paper came perhaps too early and was largely forgotten, until it was reprinted a quarter of a century later.[92] According to the paper, the development of the main SAGE control program proceeded in nine phases, illustrated in Figure 4.9. The operational plan defines the design requirements and is used to produce operational specifications. This should be sufficiently detailed so that programmers can prepare the program using only machine and operational specifications. Coding specifications then describe the structure of the program with enough detail that makes it 'possible to predict precisely the output of [each component] subprogram for any configuration of input items'. The coding is followed by two-phase testing. In parameter testing, each component is tested according to its specification in isolation, in 'an environment that simulates pertinent portions of the systems program'. This is then

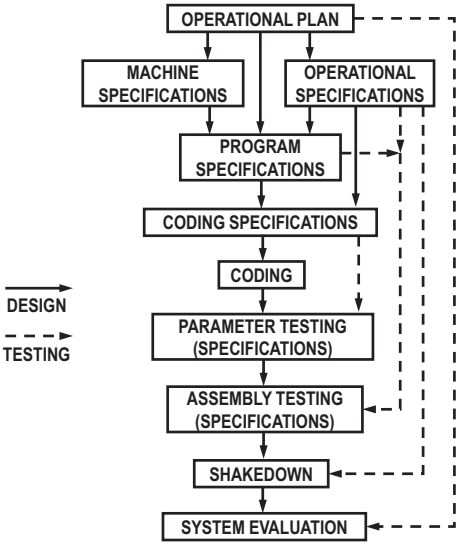


Figure 4.9 'Production of a large program system proceeds from a general operational plan' which defines broad design requirements for the complete control system and is jointly prepared with the user of the system. The dashed line indicates that testing is guided by a specification, not by the coded program.[91]

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

followed by a testing of the assembled program, first using simulated and then using live data.

The development methodology of SAGE may not have directly influenced later managerial development methodologies, but it has many aspects that are characteristic of methods that became widely used in the coming decades. Most importantly, it splits the process into a number of phases that are undertaken in sequence. Testing is treated as a separate phase and tests are based on the developed specifications, rather than being based more directly on system implementation. The goal of testing in the SAGE methodology is to show that the system satisfies the specification, that is, it falls into the 'demonstration-oriented' period of testing as classified by Gelperin and Hetzel. The author of the write-up is, however, also anticipating the later debate about the viability of this approach:

> It is debatable whether a program of 100,000 instructions can ever be thoroughly tested – that is, whether the program can be shown to satisfy its specifications under all operating conditions.

The individual phases of the SAGE development methodology are remarkably similar to those of the Waterfall development model that became widely known and used from the start of the 1970s. The Waterfall model is sometimes attributed to Winston Royce, who describes it in a paper 'Managing the Development of Large Software Systems' published in 1970,[93] although without using the term 'Waterfall'. The model, illustrated in Figure 4.10 starts with requirements gathering and analysis, follows with program design, actual coding, and testing, and concludes with program operation. However, Royce describes the basic model only as a starting point for further discussion. He 'believes in this concept', but points out that the basic method is risky and invites failure.

Royce starts with a naïve version of the process that proceeds only in the forward direction, but soon notes that the methodology needs to be iterative. The chief difficulty, according to Royce, is that we may sometimes need to go back by multiple steps. For example, the testing phase, which only occurs after coding is done, may reveal timing, storage and input/output issues. Those are not precisely analysable upfront and may require design changes, which then, in turn, require new implementation. The result is 'up to a 100-percent overrun in schedule and/or costs'. Royce goes on to suggest five ways in which the process can be improved, including two that will become major themes in later developments that I will revisit later in this chapter. The first is to arrange matters so that the version finally delivered is actually a second version. This is an approach that Fred Brooks recommends in his later 'Plan to Throw One Away' essay.[94] The second is to involve the customer prior to the final delivery. This was a method used in participatory design, which emerged in Scandinavia in the 1970s, but it is also the essence of a shift in software development that started in the late 1980s and eventually led to the Agile development methods.

Yet another way in which the process can be improved focuses on testing and is documented with a diagram shown in Figure 4.11. Here, Royce recommends involving a dedicated team of 'specialists in software product assurance' who should test the

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

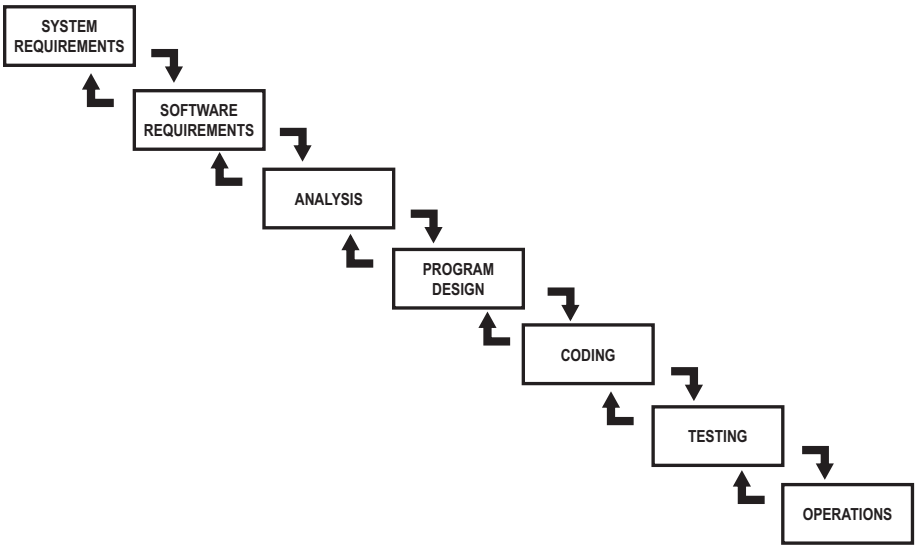


Figure 4.10 'Implementation steps to develop a large computer program for delivery to a customer' as proposed by Royce with iterative interactions, hopefully confined to successive steps.[95]

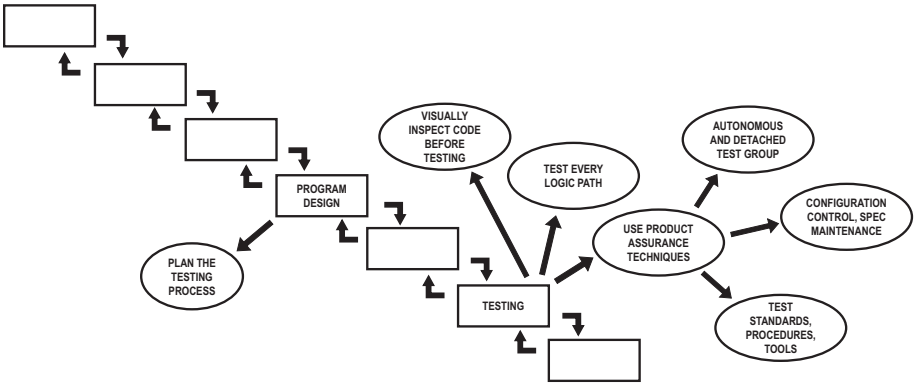


Figure 4.11 Additional aspects for testing outlined by Royce in a section 'Plan, control and monitor computer program testing'.[96]

system based on a detailed documentation, performing manual inspection of code to catch simple errors and manual checkout for identifying more complex bugs. An interesting proposal is to 'test every logic path in the computer program at least once with some kind of numerical check'. This is an informal sketch of the testing approach that was to be developed more formally over the next five years in work on test data selection, which I discussed in the previous section.

Software development methodologies like Waterfall impose a structure on the development process. They organise the work into phases, each of which results

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

in outputs such as documentation, specification or code. This makes it possible to define milestones and monitor project progress. In other words, such methodologies make the opaque hacker process of building software legible to a managerial outsider. The early work on development methodologies illustrates how the managerial culture approaches the problem of programming. In order to professionalise software development, the proponents of the managerial culture also saw the need to standardise the development methodologies and capture the best practice as an industry standard.

There are three notable organisations that contributed to the standardisation of software development processes and the role of testing in those. The Institute of Electrical and Electronics Engineers (IEEE) established a task group on software engineering in 1979. The task group did not, at first, attempt to standardise the process or the best practices for testing. It started by developing a standard for test documentation,[97] which should be used for documenting test activities that occur as part of the software development process. This was followed by two more specific standards, specifying the process for ensuring that products meet their requirements (verification) and that they satisfy user needs (validation),[98] as well as a detailed process for testing individual software components (unit testing).[99] At the same time, the National Bureau of Standards (NBS) published its 'Guideline for Lifecycle Validation, Verification, and Testing of Computer Software',[100] which was intended primarily as a guideline for federal agencies developing new software.

Gelperin and Hetzel chose the 1983 publication of the NBS guideline as the beginning of the 'evaluation-oriented' period of software testing. In this era, tests are used not only to find bugs, but also to assess the project's progress. The NBS guideline recommends a number of different kinds of testing throughout the software development lifecycle. Unit testing is to be done at the end of the programming phase, integration testing and system testing is done at the end of the testing phase, while acceptance testing is done manually, by the customer, at the end of the software installation phase. The guideline also includes regression testing, in order to prevent the reintroduction of errors during the operation and maintenance phase. The presence of multiple separate test phases in the process further increase the legibility of the process, because managers can use it to easily track the project's progress.

The standards published by IEEE and NBS were merely recommendations, but several organisations soon started requiring that software providers follow a specific process. Most notably, the US Department of Defense made its own standard of the software development process a requirement. The standard was published under the code DOD-STD-2167A in 1985. A contractor for the Department of Defense was required to follow a development process consisting of a number of activities such as system and software requirement analysis, preliminary and detailed design, coding, as well as unit testing, integration testing and system testing.

Despite the standardisation efforts, the question of how exactly testing should interact with other activities in the development process remained open. An interesting perspective came from Gelperin and Hetzel, who were the authors of the 1988 paper 'The Growth of Software Testing'.[101] The two were software engineering consultants,

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

who established a joint consulting business in 1986, started organising a conference series on software testing and published a well-regarded book on the topic in 1988.[102] Their retrospective on testing was not written just to document the history of testing, but also to define the new trends. After discussing the 'evaluation-oriented' period, the authors turn to an advocacy of their own approach which they label as 'prevention-oriented' and which purportedly started a year before the publication of their paper and the book.

In the envisioned 'prevention-oriented' period, test planning, test design and development and testing is a ubiquitous activity that happens in parallel with all other steps of the software development lifecycle. Testing is no longer done at certain stages of the process, but throughout the development. Gelperin and Hetzel captured this idea in their consulting company motto, 'Test, then code', which they wore proudly on a blue pin at the Fourth International Conference on Software Testing in 1987.[103] Gelperin and Hetzel discussed their ideas using the vocabulary of the managerial culture. They talked about development phases, testing lifecycle and outputs such as the 'Master Test Plan'. Yet, some of their ideas are closely related to Agile development methods that we will encounter later in this chapter. But first, I want to discuss a more conceptual change that followed the birth of software engineering.

4.10 Disciplinary Repertoire of Software Engineering

The engineering culture of programming that grew in prominence in the 1970s shaped new development methodologies and was behind new approaches to testing and error handling. More importantly, though, it also established a new way of thinking and reasoning about software systems. The new way was distinct from the formal reasoning of the mathematical culture and organisational reasoning of the engineering culture. In her account of the history of missile defence, Rebecca Slayton coined the term 'disciplinary repertoire' of software engineering to refer to this emerging kind of argumentation.[104] The disciplinary repertoire made it possible to look for fundamental laws and principles of software engineering and use those to shape public debates about software systems, such as the missile defence documented by Slayton.

As we saw in Chapter 1, the central notion of debates in the mathematical culture of programming became that of an *algorithm*. The disciplinary repertoire for reasoning about algorithms include primarily formal mathematical methods. For example, the undecidability of the halting problem is a mathematical theorem with a proof. It states that it is impossible to construct an algorithm, which would determine whether an arbitrary computer program will finish running or continue running forever. For the mathematical culture, this has wide-ranging theoretical implications about what kinds of programs can never be constructed. But to the engineering culture, it seems like a theoretical curiosity that tells little about practical limits of building large-scale software applications. The rise of software engineering gradually led to the recognition that a software *system* has a very different nature than a computer *algorithm* and that a different language for talking about each is needed.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

The difference between an algorithm and a software system was also crucial for the critique of program verification by DeMillo, Lipton and Perlis that I discussed earlier. When questioning whether program verification can scale up to complex systems, they argue:

> [T]here is no continuity between the world of FIND or GCD and the world of production software, billing systems that write real bills, scheduling systems that schedule real events, ticketing systems that issue real tickets.[105]

The fundamental difference is that 'the specifications for algorithms are concise and tidy, while the specifications for real-world systems are immense, frequently of the same order of magnitude as the systems themselves. The specifications for algorithms are highly stable, stable over decades or even centuries; the specifications for real systems vary daily or hourly. These are not differences in degree. They are differences in kind.' In other words, the disciplinary repertoire for talking about algorithms may work well for algorithmic programs, but it is not applicable to large software systems.

The large software systems that software engineering is concerned with are not 'composed of nothing more than algorithms and small programs'. According to De Millo, Lipton and Perlis, 'the colorful jargon of practicing programmers seems to be saying something about the nature of the structures' that programmers work with. The implementation of any such large systems is made of 'patches, ad hoc constructions, bandaids and tourniquets, bells and whistles, glue, spit and polish, signature code, blood-sweat-and-tears, and, of course, the kitchen sink'. From the late 1960s, software engineers started to figure out what kind of fundamental laws and principles may apply to software systems constructed in such a way.

4.11 Software Aspects of Strategic Defense Systems

The key debates that shaped the disciplinary repertoire of software engineering were reflections on concrete software systems. The focus on concrete problems and systems is characteristic of the engineering culture of programming. We saw that the work on software development methodologies started in the 1950s with the reflections on the development of the SAGE system and that exception handling mechanisms were a response to the concrete problem of dealing with input/output errors. The laws and principles discussed in this section will similarly arise from concrete problems and systems.

The arguments about the anti-ballistic missile (ABM) defence, documented by Slayton,[106] were forming the disciplinary repertoire of software engineering from the the end of the 1960s until the late 1980s. The disciplinary repertoire started taking shape at the time of the NATO Software Engineering conferences, although the most active participants in the discussion about ABM were absent from the two conferences. The USA has been developing various forms of the ABM defence since the 1950s, but the announcement of the new Safeguard system in 1969 raised public profile of the program and triggered a new debate. The Safeguard anti-ballistic missile system

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

was intended to protect US nuclear launch sites from limited forms of enemy attack. A number of policymakers as well as computer scientists opposed it. This was often for political reasons, but opposers understood that a more effective argument needs to be based on technical reasons why such a system cannot reliably work.

The first contribution to the debate came from J. C. R. Licklider, who was instrumental in the development of time-sharing systems and the Internet during his time as a research administrator at ARPA and was then a professor at MIT. Licklider was invited to write a chapter on reliability for a study on the feasibility of the ABM. He soon concluded that the difficulties with the development of similarly complex systems are a common wisdom among the practitioners, but that there is very little well-documented evidence of past failures and their reasons. In his report 'Underestimates and Overexpectations',[107] Licklider resorts to talking about 'common wisdom', which is that 'for a certain class of systems, costs and times tend to be grossly underestimated and performance tends to be mercifully unmeasured'. The tricky class of systems has a number of characteristics. They are complex, work in operating conditions that are not fully under the designer's control, their task changes over time and they are difficult to test in a realistic scenario. Licklider discusses a number of examples, including the SAGE and the American Airlines travel booking system, which both suffered from underestimates and overexpectations. He concludes that:

> [C]omplex software subsystems can be 'mastered' … and made to provide useful and effective service if they can be developed progressively, with the aid of extensive testing … and if they can be operated more or less continually in a somewhat lenient and forgiving environment.[108]

Licklider's analysis raises serious concerns about the possibility of building the software component of the ABM system, which is a complex system that cannot be dependably tested and which operates in an evolving hostile environment. At the same time, Daniel McCracken, who was a consultant and the author of multiple popular programming books, started a more plainspoken effort, the organisation Computer Professionals against ABM (CPAABM). In a press release, the signatories simply stated that the 'Safeguard antiballistic missile system should be abandoned because its control computer will not be able to perform the complex job assigned to it'.[109] This is because the 'pattern of development that must of necessity be followed with Safeguard, is highly unlikely to lead to a successful computer system'. The group argued that there are three main issues with the system. First, its precise task cannot be defined, because the countermeasures developed by the enemy cannot be expected; second, realistic testing of the system is impossible; and third, the Safeguard system cannot be developed through evolution.

The arguments used by the CPAABM are similar to those of Licklider. They focus partly on the inherent complexity of such a system and partly on the properties of the environment in which the system needs to be built and operated. However, the arguments are not based on formal proof or impeccable empirical evidence. They are generalisations based on the somewhat anecdotal evidence about past systems.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

The question of necessary evidence for making such claims sparked a debate among the computing community. Herbert Bright, an ACM council member, was one of those who believed that past experience is not sufficient for claims about future systems. He expressed 'a profound distaste for saying we believe it won't work when the truth is merely that we suspect it won't work'.[110] The system would not violate any known physical or mathematical law and so 'it's unfeasible to develop a sound basis for a proof that the proposed system cannot work'. Joseph Weizenbaum did not feel obliged to provide a mathematical proof: 'My suspicion is strong to the point of being belief. I don't think that my statement as a professional that I hold this belief obligates me to a mathematical proof.'[111] However, Weizenbaum did not base his suspicion merely on personal hunches or the analysis of past failures. His belief, interrogated and summarised by Slayton, was based on an informal, but logical argument:

> Computer systems could only become reliable if 'the environment that they are to control or with which they are intended to cooperate must have a change rate smaller than that of the [computer] system itself'. But since the missile defence 'environment is actively hostile and destabilising by intent', the software could not keep up with its changing requirements: 'convergence is logically impossible'.[112]

Weizenbaum's argument is an example of the developing disciplinary repertoire of the engineering culture of programming. It uses logical reasoning to talk about things such as the 'environment' and a 'change rate', which are crucial for large complex systems, but can hardly be defined in a fully formal way. We will see this approach to reasoning about software systems repeatedly in works such as 'The Mythical Man-Month' by Fred Brooks, but also appeared in subsequent debate on the anti-ballistic missile defence.

The Safeguard system was eventually built at one site and was briefly operated, before being shut down, because the politicians accepted that its utility 'will be essentially nullified in the future'.[113] This was not the end of the US ABM efforts, however, and in 1983, Ronald Reagan's administration proposed a new system, Strategic Defense Initiative (SDI), nicknamed the 'Star Wars program'. The Star Wars program was intended to protect the USA from ballistic missiles by destroying them before they hit their target. The program explored various ways of doing this, ranging from laser beams to projectiles dropped from a satellite (Figure 4.12). Again, the system required the development of a very complex control software that would detect the attack, calculate missile trajectories, coordinate interception and discern decoys from actual warheads.

In spring 1985, the SDI established a 'Panel on Computing in Support of Battle Management' consisting of a small number of computer experts. One of those invited to join was David L. Parnas, a computer scientist known for his software engineering work on information hiding. However, on June 28, 1985, Parnas resigned from the panel and published a series of notes that he wrote while being a panel member. In the notes, republished as 'Software aspects of strategic defence systems'[114] by the Communications of the ACM, Parnas concludes that, 'we will never be able to believe, with any confidence, that we have succeeded' in building a trustworthy software for the SDI

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005



Figure 4.12 An artist's concept of a ground/space-based hybrid laser weapon. (Source: US Army, Wikimedia Commons)[115]

and that 'nuclear weapons will remain a potent threat'. Parnas is clear that he opposes the initiative on technical rather than political grounds. His objections are not against any kind of military software, but rather 'on characteristics peculiar to this particular effort'. What makes the SDI software unattainable is its inherent complexity, the fact that it cannot be tested in realistic conditions prior to its actual use, as well as the fact that there won't be sufficient time for human intervention when the system responds to an attack.

The reason that testing and gradual development is the only way to build reliable software is the inherent complexity of large software systems. In his analysis of 'the fundamental technological differences between software engineering and other areas of engineering', Parnas, like Weizenbaum some 15 years earlier, employs logical reasoning for talking about informal concepts. He explains the reasons for software complexity with reference to analog and digital engineering products.

Analog systems have an infinite number of states within a broad operating range and can be modelled using continuous functions. 'Small changes in inputs will always cause small changes in outputs' and so analog systems 'contain no hidden surprises'. It suffices to ensure, through well-understood mathematical analysis, that 'the system components are always operating within their normal operational range'. By contrast, digital systems have a finite number of states that cannot be modelled as continuous functions. Before the advent of computers, the number of states of such systems was typically small enough that exhaustive testing was possible. This is not the case for digital computers, which have a very large number of states.

To obtain correct and reliable hardware, engineers were able to leverage the fact that hardware is built using many copies of smaller digital subsystems, which can be analysed and exhaustively tested independently. But this cannot be applied to soft-

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

ware systems, which have a much larger number of states and no repetitive structure. According to Parnas, 'this difference clearly contributes to the relative unreliability of software systems and the apparent lack of competence of software engineers'. Most importantly, the logical reasoning used by Parnas in his analysis lets him conclude that this 'is a fundamental difference that will not disappear with improved technology'.

4.12 The Mythical Man-Month

The only commercial company that was developing software systems of the same scale as the US military was IBM. It dominated the computer market and was rapidly growing. By the end of the 1950s, IBM had a complex family of products, including multiple ranges of incompatible low-end and high-end computers. In 1961, the company embarked on a new direction and started building a single family of computers that would share the same software and have interchangeable peripherals. The System/360 (Figure 4.13), as the new family was called, was announced in April 1964 at the company's auditorium and was paralleled with press conferences in 164 US cities and 14 other countries. The announcement was followed by a flood of orders from eager customers, but 'it would take all the resources of the company working 60-hour weeks, and in some cases round-the-clock shifts, for nearly two years before the System/360 would be called a success'.[116]

The development of the operating system for the new family of machines, called OS/360, was led by Fred Brooks Jr., who joined IBM several years earlier, after com-



Figure 4.13 The IBM 360/75 Computer, first released in 1965, at the Johnson Space Center in 1975. (Source: Space Frontiers/Stringer/Archive Photos via Getty Images)

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

pleting a PhD in applied mathematics at Harvard. The OS/360 system included two major features that made the development complicated. It supported running multiple programs at once and the ability to handle multiple users through time-sharing interactive use. It also came with 44 new peripherals that all required software support. As the system became delayed, IBM reallocated some of its programmers working elsewhere to the OS/360 effort. The team working on the system eventually consisted of over 1,000 people and the cost of the development, grew from $40 million to $500 million. When Brooks left the company in 1964, the CEO asked him why managing software projects was harder than managing hardware projects. This planted a seed for a series of essays, published as a now classic book *The Mythical Man-Month*[117] a couple of years later.

Brooks' reflection on the development of the OS/360 includes a number of principles that have became well-known slogans in the software engineering community. The most famous one is known as Brooks' law and is based on his experience with adding programmers to the OS/360 development effort in order to make up for the slips in schedule. The law states that 'adding manpower to a late software project makes it later'. The law that was certainly counter-intuitive to Brooks and others at IBM during the development of OS/360 is not just an aphorism based on personal experience. As with the principles I talked about in the previous section, it is supported by a logical argument involving the informal notion of communication.

The reason for the law is that developing a software system in a team requires communication. Most programming tasks cannot be divided into a number of completely independent smaller parts and so the developers will spend some of their time coordinating their work with others. Brooks expresses this point formally when he explains:

> If each part of the task must be separately coordinated with each other part, the effort increases as $n(n-1)/2$. Three workers require three times as much pairwise intercommunication as two; four require six times as much as two. If, moreover, there need to be conferences among three, four, etc., workers to resolve things jointly, matters get worse yet. The added effort of communicating may fully counteract the division of the original task …[118]

In other words, as the team gets bigger, the communication overhead grows faster than the added programming capabilities and the overhead eventually outweighs the gains. The mathematical argument is clear enough even if we do not know the specific numbers, such as the percentage of time spent on communication.

Fred Brooks' essays are not merely about misconceptions to be avoided. He also documents a number of developments in software engineering that he believes can have a positive effect. Brooks argues for a rigorous practice in writing documentation, but he also welcomes new technological developments [sic] such as high-level programming languages, on-line debugging tools and electronic word processing tools (for maintaining a shared up-to-date documentation). He also endorses the surgical team structure developed by an IBM colleague Harlan Mills.

Brooks' analysis of software engineering did not stop with OS/360 and Brooks' law. In 1986, he wrote a new essay that grew out of his later experience reviewing military

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

software. The essay, titled 'No Silver Bullet: Essence and Accidents of Software Engineering', was reprinted a year later in the IEEE Computer magazine. Brooks makes a bold prediction:

> There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.[119]

Again, this is not a baseless claim and Brooks supports it with a logical argument. He argues that software construction involves *essential tasks* and *accidental tasks*. The former deal with the 'complex conceptual structures that compose the abstract software entity', while the latter are concerned with 'the representation of these abstract entities in programming languages and the mapping of these onto machine languages within space and speed constraints'. Essential complexity is caused by the problem that the software solves. A software for filling out tax returns in 10 different countries has to do 10 different things. By contrast, accidental complexity is caused by the imperfection of our programming tools.

According to Brooks, the hard part of producing a software system is dealing with the essential tasks, that is, the specification, design and testing of the abstract software entity. The accidental complexity has been largely simplified by the development of high-level programming languages, although it is still something that programmers need to deal with. The crux of Brooks' argument is that, unless dealing with the accidental complexity 'is more than 9/10 of all effort, shrinking all the accidental activities to zero time will not give an order of magnitude improvement'. Again, the exact number does not matter, as long as we believe that it is less than 9/10. Even if the ratio between essential and accidental complexity was 1:1, reducing the accidental complexity to 0 would only lead to a two-fold improvement.

In 'No Silver Bullet', Brooks also elaborates on the two sources of the essential complexity that we already encountered in the discussion about anti-ballistic missile defence, that is, the inherent complexity of large digital systems and the complexity of an evolving environment. In the discussion of the former, Brooks shares an argument with Dave Parnas who was, for a time, his colleague at the University of North Carolina at Chapel Hill. Software entities are, according to Brooks, 'more complex for their size than perhaps any other human construct'. They are composed of a large number of distinct, discrete elements and, as software gets larger, the situation only gets worse:

> [S]caling-up of a software entity is not merely a repetition of the same elements in larger size, it is necessarily an increase in the number of different elements. In most cases, the elements interact with each other in some non-linear fashion, and the complexity of the whole increases much more than linearly.[120]

The second source of complexity of software systems is that they are 'embedded in a cultural matrix of applications, users, laws, and [machines]. These all change continually, and their changes inexorably force change upon the software product.' These characteristics of the environment in which software systems operate have been the primary concern of another influential contribution to software engineering from the 1980s.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

4.13 Laws of Software Evolution

Like many of those developing the engineering perspective on programming, Meir 'Manny' Lehman had both academic and industrial experience. After studying at Imperial College London, he worked at Ferranti, Israel's Ministry of Defense and IBM's research division before returning to Imperial College. During his term at IBM, the Director of Research at IBM, Arthur Andersen, came across a paper from Bell Labs claiming that a move to interactive programming improved programming productivity three-fold. Andersen came to Lehman exclaiming that 'Anything that Bell Labs can do, IBM can do better!'[121] and asked him to investigate what projects could improve productivity of IBM programmers. The question changed Lehman's research focus for the next 25 years.

The immediate result of this task was a 1969 report 'The Programming Process',[122] which reviewed individual technical advances like interactive programming and high-level programming language, but concluded that the 'solution of the programming problem is seen in the development of a total system-oriented methodology and the associated support technologies and tools'. The focus on methodology assisted by tools positions Lehman's approach firmly in the engineering culture of programming. So do the references in his report, which include works of Dave Parnas and the report from the 1968 NATO Software Engineering conference.

Shortly thereafter, Lehman started looking at the evolution of the OS/360 operating system, which provoked his interest in the dynamics of software growth and the laws that govern it. Together with a colleague, Les Belady, he started using the term *software evolution*. After moving back to Imperial College, Lehman continued studying software evolution by analysing large real-world software solutions and eventually formulated eight laws of software evolution that were published in an influential paper 'Programs, life cycles, and laws of software evolution'.[123]

Lehman opens his analysis by pointing out that program evolution has been always associated with the concept of large programs, but that characterising 'large' and 'non-large' programs has been challenging. To address this, Lehman introduces an alternative and more insightful classification of programs, distinguishing between S, P and E-programs. The types of programs are distinguished not based on their size, but based on their relationship with the environment and the degree to which they can be fully specified. An S-program (*specification*) is an implementation of an algorithm. The problem it solves does not change and the solution can be fully described by a specification. A P-program (*problem*) solves a problem that can be precisely defined, but the quality of the solution is determined by the effect it has in the real world. Lehman uses a chess-playing program as an example. Chess has a clear set of rules, but the solution will involve complex hard-to-specify algorithm and the program quality will depend on the subjective judgement of the user of the program. Finally, E-programs (*environment*) do not even have a fixed problem specification. They 'mechanize a human or societal activity', but in doing so, they are embedded in the environment and change it, often also changing the problem statement. Examples given by Lehman include operating systems, air-traffic control and stock control.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

The first consequence of the classification is, according to Lehman, for the notion of program correctness. Whereas an S-program can be fully formally specified and its implementation checked against the specification, the validity of an E-program 'depends on human assessment of its effectiveness in the intended application'. Although Lehman somewhat optimistically suggests 'that as programming methodology evolves still further, all large programs (software systems) will be constructed as structures of S-programs', his laws talk primarily about the most challenging kind of programs, the E-programs.

The focus on E-programs, which is typical for many of the discussions in this chapter, is in a stark contrast with the typical focus of the mathematical culture of programming, which often assumes that programs can be fully specified and, by doing so, restricts itself to talking about what Lehman classifies as S-programs. An eloquent advocate of the mathematical culture, Edsger Dijkstra, describes his focus clearly in a position paper on software reliability[124] by contrasting the 'scientific questions of correctness' with 'the unformalized question whether a tool meeting [its] specifications is in such-and-such unformalized and ill-understood environment a pleasant tool to use'. The narrow focus of mathematical methods is a frequent source of clashes between the mathematical and the engineering culture. For example, Peter Naur responded to Dijkstra by pointing out that pleasantness is 'a somewhat euphemistic word to use about such calamities as airplanes colliding in mid air or atomic reactors exploding'.[125]

Lehman's realisation that programs need to evolve in parallel with the continually evolving environment in which they are embedded poses a major challenge for top-down software development methodologies that analyse the requirements at the beginning of the process, but ignore the ongoing evolution. In other words, the advances in understanding of software development in the engineering culture of programming outpaced the understanding in the managerial culture, which was busy standardising the steps of a top-down development process.[126] By the end of the 1980s, engineers started using their understanding to rethink software development processes.

4.14 Paradigm Change in Software Engineering

Already in 1970, when Royce wrote his critique of the simple i software development model, he was aware of the drawbacks of a process that proceeds forward in phases, from analysis to specification, implementation, testing and deployment. Royce pointed out that more iteration and feedback is needed in the process. He suggested a 'do it twice' method in which a pilot project is completed to understand the problem before building the real system. The same approach was advocated later by Fred Brooks in the 'Plan to Throw One Away' essay, included in his book *The Mythical Man-Month*.[127]

Another suggestion by Royce was to involve the customer not just in the initial requirements gathering stages, but repeatedly throughout the development process. The chief motivation for developing such processes is that fully understanding the problem upfront is almost never possible. This point was also common wisdom in

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

the community. It was pointed out by Dave Parnas when discussing the limits of software engineering methods in the context of anti-ballistic missile software. He argued that 'it is hard to make the decisions that must be made to write [an upfront project specification]. We often do not know how to make those decisions until we can play with the system.'[128]

Throughout the 1980s, a variety of innovations in software development were proposed to address the issue. Newly created development models included *prototyping*, where some part of the cycle is done multiple times to support learning, as well as *incremental* models, where the system is broken down into smaller parts that are developed gradually. A development model that brought many of the preceding ideas together was developed by Barry Boehm. As is typical for the key actors in this chapter, Boehm had an academic background, but spent most of his career in industry. In 1988, he developed the *spiral development model* that is illustrated in Figure 4.14. Rather than proposing a single methodology, the model proposes a way of determining the most appropriate process based on the analysis of risks at each stage. Boehm later described the model

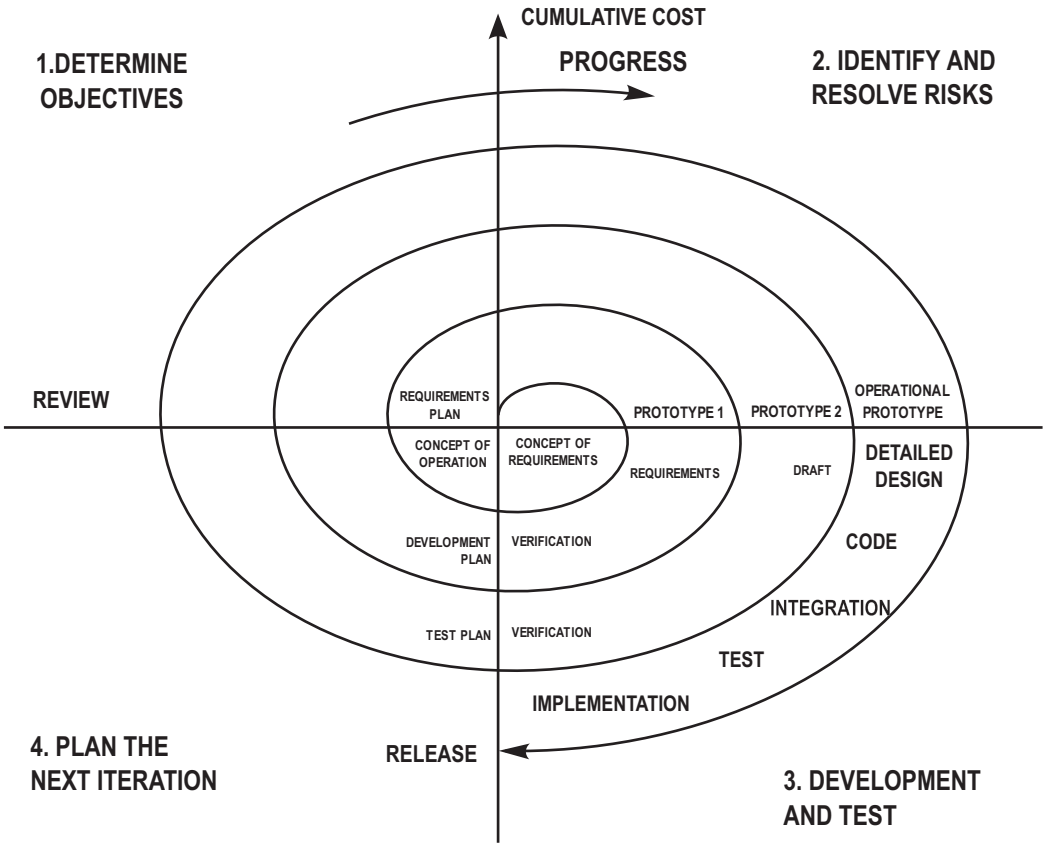


Figure 4.14 A diagram illustrating the Spiral development model. (Source: Adapted by the author from Wikimedia Commons)[129]

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

as a 'process model generator', highlighting the fact that the resulting process can dynamically take a number of different shapes.[130]

Despite the numerous innovations, the work on software development models never fully separated itself from the managerial idea of control over the the development process and the programmers involved. In other words, even the more flexible and adaptive development models still aimed to transform the 'black art of programming' into a rigorous discipline that is reproducible, predictable and does not rely on the skills of individual programmers. The processes do this by wrapping the inherent uncertainty in the programming process with enough additional layers to make software development more deterministic.

A more radical vision of how software should be built came from Christiane Floyd. After completing a PhD at the University of Vienna, she worked in several academic and industry jobs, before becoming the first female professor of computer science in Germany in 1978. Since joining the Technical University of Berlin in 1978, Floyd and her group have been working on a new approach to software development 'which is primarily concerned with the development of software as an adequate tool for users'.[131] The approach transcends 'the conceptions of software engineering prevalent in [Germany]. These see it as being based on fixed requirements of the software products to be developed, ignoring the social context in which they are embedded.'[132]

In pursuing this goal, Floyd closely collaborated with Scandinavian colleagues. Following two field trips to Denmark, Norway and Sweden in 1986, she published a trip report that documents the 'Scandinavian Approach'[133] to software development, which makes the user a more integrated part of the process. In Scandinavia, such an approach emerged in the 1970s under the term 'participatory design'. While in Scandinavia, work on participatory design led to the birth of an active research field on human–computer interaction, Floyd used the participatory design approach as the basis for a software development methodology that she named STEPS (Software Technology for Evolutionary Participative System Development). Perhaps the most visionary write-up arising from Floyd's work appears in a chapter 'Outline of a Paradigm Change in Software Engineering',[134] published in the proceedings of a conference held in Aarhus titled 'Computers and Democracy'. In the chapter, Floyd describes the dominant paradigm that emerged following the 1968 NATO Software Engineering conference as *product-oriented* and summarises the emerging *process-oriented* alternative.

According to Floyd, 'the product-oriented perspective regards software as a product'. It 'considers the usage context of the product to be fixed and well understood, thus allowing software requirements to be determined in advance'. By contrast, the process-oriented perspective 'views software in connection with human learning, work and communication, taking place in an evolving world with changing needs'. This approach reflects the changing environment in which software development happens and the difficulties with producing upfront specifications that were gradually being recognised by many researchers and practitioners of software engineering. In the process-oriented approach, the final product 'is perceived as emerging from the totality of interleaved processes of analysis, design, implementation, evaluation and feedback, carried out by different groups of people involved in system development

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

in various roles'. The process-oriented approach admits that requirements change and understanding is built gradually.

In the product-oriented perspective, process aspects such as requirements definition, quality assurance, user acceptance and software modifiability are considered 'as additional aspects outside the realm of systematic treatment'. Floyd holds that 'this situation is inherently unsatisfactory and can only be remedied if we adopt … the richer process-oriented perspective as our primary point of view'.

The perspective from which Floyd approaches software engineering cannot be easily classified as belonging to any of the individual cultures that appear repeatedly in this book. Like other development methodologies, STEPS can be seen as emerging from a managerial attempt to structure the development process. However, the process-oriented methodology also makes individual engineers more autonomous as they are involved in learning about the problem and shaping the final product. The recognition that software cannot be precisely specified upfront resonates with the insights developed within the engineering culture of programming, including the understanding of E-programs developed by Lehman. However, Floyd's work also emphasises human learning and the Scandinavian participatory design, which connects it to the humanistic culture that we followed in Chapter 3. Her work thus illustrates my claim that interesting ideas often emerge at the intersection of multiple cultures of programming.

When Christiane Floyd wrote about the paradigm change in software engineering, she talked about an 'ongoing controversy between rivalling ideas and attitudes'. She clearly saw a glimpse of an emerging new approach to software engineering, but it took 15 more years before the new paradigm took a form that enabled it to dominate the field of software engineering.

4.15 The New New Approach

At the same time Christiane Floyd was thinking about the Scandinavian approach to software development and about the ongoing paradigm change, a similar idea emerged in the USA in the more general context of product development. Hirotaka Takeuchi and Ikujiro Nonaka had management and operational theory backgrounds and gained experience in both industry and academia in the USA and Japan. In 1986, they published an article 'The New New Product Development Game'[135] in the *Harvard Business Review*, in which they analysed a 'new approach to managing the product development process'.

Takeuchi and Nonaka looked at six specific products including a Honda city car and two Canon cameras and reported that companies are 'increasingly realising that the old, sequential approach to developing new products simply won't get the job done' in a fast-paced, competitive world where speed and flexibility are essential. They liken the new approach of the pioneering companies to rugby 'where a team tries to go the distance as a unit, passing the ball back and forth'. Using the new approach:

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

> The product development process emerges from the constant interaction of a hand-picked, multidisciplinary team whose members work together from start to finish. Rather than moving in defined, highly structured stages, the process is born out of the team members' interplay.

In contrast to the humanistic Scandinavian approach focused on shared knowledge building, the wording here is more managerial and emphasises teams and initiative. Yet, the ultimate vision of a collaborative and a more dynamic approach is closely related. The authors talk about a new product development organisation, but the organisation is nothing like that of the managerial structures of the 1970s. In the new approach, teams have a high degree of autonomy, knowledge is shared and exchanged between team members and the process is controlled by more subtle indirect mechanisms.

Takeuchi and Nonaka talked about product development more generally, but in the early 1990s, the ideas about the 'new approach to product development' served as a motivation for the next step in the evolution of software development methodologies. The SCRUM methodology, developed by Ken Schwaber and Jeff Sutherland, was first presented at a workshop on Business Object Design and Implementation,[136] associated with the OOPSLA conference. By now, the authors took for granted that software development occurs under rapidly changing circumstances and that the development requires maximum flexibility. They contrast SCRUM to other development methodologies, notably Waterfall, iterative and spiral models. In SCRUM, the development is done in short sprints which are always followed by review, which makes it possible to adapt more rapidly to the lessons learned during the development.

SCRUM remains rooted in the managerial approach to programming. It describes how teams and the development process should be organised, rather than, for example, recommending particular tools. It does, however, reflect a shift in the managerial culture. Whereas the managerial approach to software development in the 1970s attempted to eliminate dependence on individual programmers and their skills, the new methods that emerged in the 1990s see programmers as creative and responsible professionals that should be trusted with making technological decisions and empowered to learn and contribute to product development.[137]

4.16 A Programming Environment for a Timeshared System

Before we continue our discussion of software development methodologies, we need to look to another source of ideas for how programming should be done. In the 1970s, the hacker culture that advocated for interactive programming and developed on-line debugging techniques in the 1960s continued mastering their craft. The main vehicle for the continued development was Lisp, which was now seen as one of the most prominent 'AI languages' and was used for research and development of expert systems and systems for natural language processing and knowledge representation. Lisp was used either on time-sharing systems or on personal 'minicomputers' and in particular on Lisp machines (mentioned in Chapter 3), which was a class of minicomputers designed to efficiently execute Lisp.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

The hacker culture continued to develop at MIT on the East coast, but also at Stanford where John McCarthy moved in 1962 to establish the Stanford AI Laboratory (SAIL) and also at Xerox PARC following its founding in the year 1970. There continued to be an active exchange of ideas between the centres, especially between SAIL and PARC which were a mere two miles apart. As is typical for the hacker culture of programming, how exactly programming was to be done was primarily the unwritten knowledge of hackers, but we can get a glimpse of some of the practices and tools from a rare report on the programming environment that was in place at SAIL from the mid 1970s and from personal recollections of some of the participants.[138]

The programming environment at SAIL was built around a time-sharing system and while personal computers were becoming increasingly commonplace at the time, the authors thought that 'it is not necessary to move to personal computers for a very usable programming environment'. Users interacted with the system through a text-based video display and custom-made keyboards. The primary interface for working with the system was an interactive, 'display-oriented text editor' called E, which later influenced the well-known EMACS editor. Unlike the LISP editor created by L. Peter Deutsch in the 1960s, E was text-based and was used to interface with Lisp as well as other subsystems of the SAIL time-sharing system.

Following the hacker way of thinking about programming, the SAIL programming environment was interactive. The E editor was not used merely for editing source code, but for controlling the system. It was possible to use it to edit files in the storage, but it was equally possible to use it in a special EVAL mode 'which allows the user to type text directly to Lisp'.[139] In this mode, output from running the Lisp code was returned to a selected destination in the editor. Notably, this was also the case when the Lisp program encountered the `break` command. The debugging was, again, done through the interface of the E editor. This way of working is reminiscent of modern Read-Eval-Print Loop (REPL) shells, but the SAIL system went even further. The designers believed that 'if a person can do something while sitting in front of a computer, it should be possible to write Lisp code to do the same thing'.[140] The E editor was built around the notion of commands and it could accept commands from the user, as well as from any subsystem. It was thus possible to write and run Lisp code that would control the editor. Despite having a programming language underneath, the programming environment at SAIL was interesting as a stateful, interactive programming system. This is the perspective that is easily recognised by the proponents of the hacker and humanistic cultures, but that is difficult to grasp for those working in the mathematical and perhaps even the engineering cultures.

The way to master the development of complex systems in the hacker culture of programming was thus not by developing a strict methodology, but by having an adaptable programming environment, a community of skilled hackers and a local 'Lisp wizard down the hall'.[141] Learning how to work in such an environment required learning from the local wizards. At the SAIL system, this was made easier by the fact that the system was equipped with a video switch. The switch made it possible for multiple people to interact with the same (virtual) terminal through their own display and keyboard, a practice called 'mapping'.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

The system gave rise to two practices.[142] First, it was possible to share the terminal when tackling a difficult programming problem. A programmer could send a message to the local wizard (through a facility implemented in E) to ask for help. The wizard could then connect to the other person's terminal and control it through his own keyboard, while using an internal phone system to talk. Second, a newcomer could learn how to program and debug in the environment by connecting to the virtual terminal used by an experienced hacker (the local wizard down the hall) and watching them program. This degree of openness was expected by hackers, but it was unexpected for outsiders. As recalled by Gabriel:

> [V]isitors and new professors complained about people being able to 'spy' on them while they were working. In response, the systems people implemented a way to hide a screen (prevent mapping), but the wizards demanded a way to override that, which was also implemented. The privacy freaks demanded that they be notified when such an override was invoked, and the systems people implemented that as well as a way to override the override notification. Information is free.

The way of working that was commonplace in the hacker culture of programming included many aspects that resembled or prefigured new ways of thinking about programming methodologies in the engineering culture. Partly because of personal beliefs, but partly also as a consequence of working in a more experimental problem domain, the hacker way of working rarely needed an upfront specification. Programming was seen as learning and problem exploration. This may have looked like 'black art' in the 1960s, but was becoming an increasingly widely adopted perspective in the 1980s. The programming practices that hackers developed are also precursors of later engineering practices such as pair programming.

To what extent and how exactly hacker practices influenced later developments in engineering development methodologies is difficult to trace precisely. However, SAIL was only two miles away from Xerox PARC where Smalltalk was developed at the same time and, as we will see in Chapter 6, the community that emerged around commercial Smalltalk in the 1980s, included the two pioneers of Extreme Programming. It is likely that at least some of the practices developed by hackers contributed to the subsequent work on programming methodologies.

4.17 Extreme Programming

The need for a new development methodology was in the air at the end of the 1980s and new ideas have emerged from a variety of programming cultures. The Scandinavian process-oriented approach adopted many humanistic ideals, the SCRUM methodology followed a managerial approach, while the hacker practices influenced how practical programming was done in the engineering culture of programming. The mix resulted in a new development method that emerged in the 1980s and gained prominence in the 1990s when it started to be called Extreme Programming (XP). As with the work of Christiane Floyd and the SCRUM methodology, XP tackled the typical issues of its time, that is, the development of software in an environment where requirements are

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

vague and rapidly changing. What makes XP interesting for my story is not the various ways in which it is similar to or differs from SCRUM or the Scandinavian approach, but the fact that it has quite different origins.

Extreme Programming was not developed by academics or management consultants, but by software engineers. The methodology was developed and later publicised by Kent Beck,[143] who nevertheless credits many ideas to 'common knowledge' and also his collaborator Ward Cunningham. Both Beck and Cunningham were consultants, working on commercial object-oriented systems developed using the Smalltalk programming language. They were looking for better ways of building systems and explored a range of ideas, such as building on the concept of pattern languages conceived by the architect Christopher Alexander, which I return to in Chapter 6. They were also a part of the Smalltalk community, which had a long tradition of empowering individuals and allowing them to shape their tools. The Smalltalk community, in turn, had close connections to the hacker culture of programming that was working in their own interactive programming environments such as that developed at the Stanford AI lab.

All of these aspects are mirrored in the XP methodology, which is based on decentralised decision making, rapid feedback and adaptation as well as focus on tools and best programming practices. The grounding in the engineering culture of programming is perhaps best summarised in Erich Gamma's foreword of Beck's book, which opens by saying that 'Extreme Programming nominates coding as the key activity throughout a software project. This can't possibly work!'[144]

The engineering origins of the Extreme Programming methodology shift the attention from managerial issues to engineering, but the methodology still talks about a number of management aspects. In particular, XP identifies a number of different roles in the team. The two most prominent ones are a programmer and a customer, who is now a direct part of the team, in accordance with what Christiane Floyd envisioned. The role of the customer is a challenging one. The customer decides what the software should do, specifies the tests for the required behaviour, while doing their regular job. In the first widely discussed project completed using the XP methodology, the Chrysler Comprehensive Compensation System (C3) payroll system, the customer was able to do all this work, but eventually found the work too stressful and moved on to a different position. The team soon realised that finding another 'XP customer' was more challenging than expected, which resulted in an ominous presentation 'Will Extreme Programming kill your customer?' presented at the OOPSLA conference in 2001.

Extreme Programming shows its engineering origins in that it describes a number of specific programming practices. Many of those are not new. As explained by Kent Beck, 'To some folks, XP seems like just good common sense. So why the "extreme" in the name? XP takes commonsense principles and practices to extreme levels.'[145] Beck lists 12 practices covering areas such as project planning, customer involvement and testing. For example, 'pair programming' requires that all code is written by two programmers at one machine. The practice should not be used, for example, only when solving a particularly tricky problem, but all the time. 'Testing' requires continually written unit tests created by programmers and customer-created tests demonstrating

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

that individual features are completed. In contrast to earlier methodologies, testing in XP becomes the driving force behind development. Another principle is that of 'refactoring', which asks programmers to continually restructure code in order to improve its structure, simplify it and ensure it communicates the intent better.

Extreme Programming brings back to prominence the practice of software testing that keeps appearing, disappearing and evolving throughout this chapter. When discussing the professionalisation of testing, I pointed out that testing became clearly distinguished from debugging in the 1970s. Companies started hiring 'test technicians' to work on automated software testing tools and, with the start of the destruction-oriented period, testing started to focus specifically on finding errors in programs. In most of the software development processes that I have talked about so far, testing exists as a separate phase to be done at a specific point during the development. Sometimes, the process also draws an explicit distinction between unit testing, which focuses on individual units of functionality, integration testing that focuses on the overall system and acceptance testing to show that the product matches requirements specified by the customer.

In XP, testing took the extreme form of Test-Driven Development (TDD). In this practice, an automated test for a required functionality needs to be created even before the programmer writes any code to actually implement the functionality. The idea was not new. As discussed earlier, Gelperin and Hetzel's consulting company Software Quality Engineering promoted the motto 'Test, then code' as early as 1986, and even put it on their lapel pins at a software testing conference in 1987. Yet, TDD treated the idea in a new way in that it took it as the basis for controlling the development process.

Test-Driven Development, which eventually evolved into a software development methodology of its own,[146] fully completes the shift from the managerial culture of programming to the engineering culture. It turns tests from something that can be done manually to a software artifact that has to be automated. Tests are not used just for ensuring that a certain functionality is implemented correctly, but they serve as a description of the system. In TDD, programmers start by writing tests for a minimal system that could be possibly run and then gradually add tests that describe further functionality, ensure that the tests pass by implementing the necessary logic and then proceed to make the code simpler and more reusable before moving on to the next feature. The tests can also be seen as capturing the understanding of the problem, jointly developed by programmers and the customer. Although proponents of the mathematical culture would find the idea absolutely inadequate, tests can even be treated as a partial executable specification.

TDD shows its connection to the engineering culture in that it is structured around tools, automated test runners. At the same time, it makes another attempt at fulfilling the vision outlined at the 1968 NATO Software Engineering conference, which was to turn the black art of programming into a science of software engineering. Already in 1968, one of the conference participants, A. I. Llewelyn, working at the short-lived UK Ministry of Technology, pointed out that 'testing is one of the foundations of all scientific enterprise'.[147] Test-Driven Development develops the parallel between programming and science in that it outlines a systematic software development process

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

based on testing. But just as actual scientific work is more complicated than its naive image, so is software development even when following the TDD methodology.

4.18 The Debugging Paradox

In 1965, Mark Halpern concluded his article 'The Debugging Epoch Opens' by claiming that what 'is certain is that the debugging problem must be confronted squarely and soon if the computer is to take on some of the critical roles it is presently being cast for'. He warned that 'the danger is real and imminent' and argued that the 'remedy will require the commitment of the programming community's full resources'.

In many ways, Halpern was correct. Eliminating program bugs and, more generally, making sure that programs do what they are intended to do has become a major effort in the proceeding decades and remains as important an issue as ever. In one way, however, his prediction was mistaken. The interesting developments that we followed in this chapter focused on testing and software development processes, but very little on *debugging*. This may be a historical accident, because the meaning of phrases like program checkout, testing and debugging kept shifting and evolving throughout the 1960s and 1970s. Yet, when Halpern wrote his article, debugging was already increasingly associated with 'making sure that programs run' whereas testing referred to 'making sure that it solves the problem'.[148]

So, why didn't I have more to say about debugging? The reason is that the basic practices of debugging today are not unlike that discussed in a review of on-line debugging techniques by Evans and Darley in 1966.[149] Just like when debugging programs using a Lisp system in 1966, programmers today set breakpoints to pause program execution, explore the values of variables, modify the program and then test the result. Unlike in 1966, this works reliably for high-level programming languages, which is a notable technical achievement, but there have been few conceptual innovations.[150] It is difficult to say what the reasons for this are with any certainty, but thinking about the interactions between the different cultures of programming in this chapter offers one possible answer.

In the 1950s, programmers talked about debugging, testing, program checkout and error handling somewhat interchangeably. This was the era of the black art of programming and dealing with errors was a 'private arcane matter'.[151] As testing became distinguished from debugging, the term 'test' slowly started to be used not just as a verb, but also as a noun. A 'test' became a description of a check to be performed, first manually and later automatically. This, however, also made a test into an artifact that could be shared with programmers across cultures. In other words, a test became a boundary object.[152] This may have made it easier for other cultures of programming to contribute to work on testing. Managers could make the running of tests part of their processes, mathematicians could study tests formally and engineers could build new testing tools. Eventually, Test-Driven Development made an executable test into a primary mechanism for controlling the software development process.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

The case of error handling shares similar characteristics with that of testing. In the early days of programming, the handling of program errors was fully within the realm of the hacker culture. In the 1960s, Lisp and PL/I added a programming language construct for working with errors; in the 1970s, the exception handling mechanism appeared and was adopted by Ada and many programming languages that followed. Here again, the vague programming practice results in an artifact, or a boundary object, that could be shared across cultures. Engineers started implementing programming languages with exceptions, mathematicians were able to formally analyse programs using them, as well as to propose new ways of working with exceptions.

Similarly to tests, which gave rise to a new development methodology, exceptions gave rise to a new programming style in the 1980s. While working on the development of fault-tolerant systems for a telephone exchange, Joe Armstrong at Ericsson came up with a new way of handling exceptions for their programming language Erlang. In telecommunications, systems are designed to 'run forever' and must be resilient to hardware failures. As such, other computers must respond to errors. To support this, Erlang programs are structured as sets of lightweight processes, which may execute on separate machines. When an exception occurs, the affected process is killed. The crash will then be handled by a dedicated supervisor process that may restart the process or activate another system. In Erlang, exceptions became a key building block for building reliable fault-tolerant systems.[153]

In contrast to testing and exception handling, debugging remains a hacker practice with no simple boundary objects that could be exchanged with other cultures. As is characteristic of the hacker culture of programming, the knowledge is often unwritten and it remains the case that 'much of the work [on debugging] has been described only in unpublished reports or passed on through the oral tradition'.[154] Even the internal memoranda documenting debugging tools often merely provide a reference manual for the tools, rather than documenting how to actually use them.[155] Building sophisticated modern debugging tools is an engineering challenge and mathematical work on program analysis contributed valuable ideas for debuggers, but the process of debugging remains a private arcane matter.

4.19 Different Ways of Trusting Software Systems

Some of the most interesting developments discussed in this chapter happened when different cultures interact, often by exchanging and contributing to a specific technical concept. Each culture brings its own methods and ways of thinking about programming. While the concept of debugging remained within the realm of the hacker culture, concepts such as testing were broadly adopted and developed. Different cultures of programming are also apparent in the way programmers think about the limits of what software can be built and what they see as sufficient criteria for trusting that a system is correct.

As we saw in Chapter 2, for the mathematical culture of programming, a formal correctness proof is the ultimate goal. A fully trustworthy software system should come

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

with a proof showing that the implementation corresponds to a formal specification and the trust in proofs may be established through a social process or through mechanical checking. In reality, proving an entire system correct is often unachievable. In practice, the proponents of the mathematical culture will also, even if uneasily, trust systems where the formal proof concerns a simplified model or only a core part of the system.

The mathematical culture also recognises mathematical laws as the only limitation of what software can be built. To put it simply, it is impossible to build a system that solves the halting problem, but anything else is doable. The proponents of the mathematical culture believe that there is no fundamental conceptual difference between a compiler and a software to detect and target ballistic missiles. The question whether a sufficiently precise specification of the system can exist is out of the scope for computer science. To make this point explicit, Edsger Dijkstra distinguishes between the problem of correctness (whether a system meets a specification) and of pleasantness (whether the system is one we wanted to have).[156]

The managerial and engineering cultures are more concerned with practical constraints of building complex software systems that lack clear specifications and are built in an evolving environment. The disciplinary repertoire of software engineering made it possible to claim that certain systems are impossible to build, for example, when the change rate of the environment is higher than the change rate at which the system can be built or when progressive development with continuous testing is impossible.

What exactly is required by the managerial and engineering cultures in order to trust a system has been evolving over time. In the managerial culture of the 1970s, a system can be trusted if it was built by correctly completing all the required phases of the selected software development methodology including, most notably, acceptance testing by the customer. The trust is thus mediated through a process agreed ahead of time. In the case of the engineering culture, the trust in software system relies on the responsibility of individual programmers, but it can be supported by tools. For example, in Test-Driven Development, a tool can measure code coverage, which is a metric of what proportion of code has been executed during testing. While some engineers will emphasise that this is a simplistic metric, many are likely to trust systems with a higher code coverage.

Finally, in the hacker culture of programming, the trust is mainly placed in individuals. In the 1950s, those were the masters of the 'black art' of computer programming, but a similar level of trust still exists in highly technical, especially open-source, programming projects such as the Linux kernel or in the development of programming languages. For example, some open-source communities award the Benevolent Dictator for Life (BDFL) title to the project leader who is trusted with making the ultimate decisions about the project.

Donald MacKenzie discusses the issue of trust with reference to historical and sociological research on risk and trust in the societies of high modernity. His summary is that 'in traditional communities face-to-face interactions typically obviated the need for formal structures of objectivity'. In a society of a high modernity which, among other things, relies on increasingly complex computer systems, 'face-to-face solutions

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

are no longer viable. Modernity's "trust in numbers" is a substitute for absent trust in people'.[157]

In the case of software development, the question of trust does not follow a simple progression from traditional communities to a society of high modernity. The hacker culture still relies on the equivalent of face-to-face interactions, whereas the mathematical culture is fully committed to the modernist trust in numbers. The managerial and engineering cultures are somewhere in the middle of the spectrum. Although individual responsibility is important in both engineering and managerial culture, the two cultures rely on processes and tools, respectively, as crucial components for trust.

Notes

1 Based on the answer suggested by Hoare (1996).
2 Inspired by Halpern (1965), who predicted that the 1960s and 1970s would be the epoch of debugging.
3 Buxton et al. (1970).
4 Goodenough and Gerhart (1975), Musa (1975).
5 This recent way of thinking about testing has been documented, for example, by Mugridge (2003).
6 Royce (1970).
7 Floyd (1987).
8 The list is based primarily on works by Lehman (1980), Brooks Jr (1975) and Parnas (1985) that will be discussed later in the chapter.
9 As pointed out by Licklider (1969), lessons learned in the 1950s and 1960s have resulted in 'common wisdom'. His work was one of the first attempts to make such knowledge more explicit.
10 Brooks Jr (1975).
11 Documented by Slayton (2013) and discussed below.
12 MacKenzie (2004).
13 Hoare (1996).
14 Gschwind (1952), quoted in Bissell (2007).
15 Bissell (2007).
16 Using a term introduced by von Neumann and Goldstine (1947). The historical significance of the problem of controlling the evolution of a meaning is discussed in detail in a chapter on notations by PROGRAMme (2022).
17 www.flickr.com/photos/lselibrary/3833724890/.
18 Bartik (2013).
19 https://ftp.arl.army.mil/ftp/historic-computers/.
20 Haigh et al. (2016).
21 http://pdp10.froghouse.org/KI10/.
22 Pirsig (1999).
23 Bartik (2013).
24 Documented by Priestley (2011).
25 Campbell-Kelly (1992).
26 Campbell-Kelly (2011).
27 Initial orders and subroutines have been described in a textbook by Wilkes et al. (1951). A useful historical summary can be found in Campbell-Kelly (2011).
28 Campbell-Kelly (1992).
29 Both presented in Wilkes et al. (1951) and documented by Campbell-Kelly (2011).
30 https://americanhistory.si.edu/collections/nmah_334663.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

31 National Museum of American History (2024).
32 Kidwell (1998).
33 Turing (1949).
34 Gilmore (1958b).
35 As is typical for the hacker culture of programming, the system was only ever described in an internal memo (Stockham and Dennis, 1960), written for the benefit of the MIT hacker community.
36 See Evans and Darley (1966) for the review and Evans and Darley (1965) for the DEBUG tool.
37 Evans and Darley (1966).
38 As usual, the systems have been described in various internal memos including ones by Martin and Hart (1964), Teitelman (1965) at MIT and one by Russell (1963) from Stanford.
39 Evans and Darley (1966).
40 Bobrow et al. (1967).
41 The `EDIT` and `BREAK` functions were first described by Teitelman (1965).
42 Halpern (1965).
43 This interpretation is offered by Ensmenger (2012).
44 Lindsey (1996b).
45 Pelaez Valdez (1988).
46 The history of the conference has been documented by Pelaez Valdez (1988).
47 As discussed in Chapter 2, the extent to which the NATO conferences contributed to the rise of the engineering culture is open to discussion. Ensmenger (2012) talks about 'major cultural shift', while Haigh (2010) argues that the impact is hard to 'square with the actual historical record'.
48 Llewelyn and Wickens (1968).
49 http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.
50 Buxton et al. (1970).
51 The context in which the PL/I language was developed has been documented by Astarte (2019).
52 Radin (1978).
53 McCarthy et al. (1962).
54 IBM (1967).
55 For a detailed account of the history of PL/I and its formalisations, see Astarte (2019).
56 Ryder et al. (2005).
57 Goodenough (1975a,b).
58 Liskov (1993).
59 As recalled in the history of the Ada project written by Whitaker (1993).
60 Ichbiah et al. (1979).
61 Steele and Gabriel (1996b).
62 Goodenough (1975a).
63 Liskov (1993).
64 McCarthy et al. (1962).
65 The idea has been introduced by Plotkin and Pretnar (2009) and a more accessible review of the feature can be found in a report by Chandrasekaran et al. (2018).
66 The modern term for the specific practice is *offensive programming* and seems to have been introduced in the late 1990s, but the term defensive programming dates back to a 1975 book by Yourdon (1975).
67 Randell (1975).
68 A notable example of this is the system designed by Yau and Cheung (1975).
69 Armstrong (2007).
70 Quoted in MacKenzie (2004).

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

71 Hetzel (1973).
72 McGonagle (1972).
73 McGonagle (1972).
74 Gelperin and Hetzel (1988).
75 Buxton et al. (1970).
76 The three examples are referenced by MacKenzie (2004).
77 Huang (1975).
78 Goodenough and Gerhart (1975).
79 McCabe (1976).
80 Myers et al. (1979).
81 Huang (1975).
82 A demo, presented at the AFIPS conference by Budd et al. (1978) and a paper by DeMillo et al. (1978).
83 Ensmenger (2012).
84 Reported by Licklider (1969) and discussed by Slayton (2013).
85 Brooks Jr (1975).
86 Haigh (2010).
87 McKinsey (1968).
88 The shift has been documented in detail in the work of Ensmenger (2012).
89 Introduced in 'The Psychology of Computer Programming' by Winberg (1971).
90 See Slayton (2013), who documents not just the development of the SAGE, but the entire socio-technical context of missile defence software.
91 Benington (1983).
92 Benington (1983).
93 Royce (1970).
94 Brooks Jr (1975).
95 Royce (1970).
96 Royce (1970).
97 IEEE Standard for Software Test Documentation, ANSI/IEEE (1983).
98 IEEE Standard for Software Verification and Validation Plans, ANSI/IEEE (1986).
99 IEEE Standard for Software Unit Testing, ANSI/IEEE (1987).
100 National Bureau of Standards. (1984).
101 Gelperin and Hetzel (1988).
102 Hetzel (1988).
103 This and many other events in the history of testing are documented in the excellent online reference by Meerts and Graham (2010).
104 Slayton (2013).
105 De Millo et al. (1979).
106 Slayton (2013).
107 Licklider (1969).
108 Licklider (1969).
109 Available at https://digitalcollections.library.cmu.edu/node/16631.
110 Quoted in Slayton (2013).
111 Quoted in Slayton (2013).
112 Slayton (2013).
113 Finney (1975).
114 Parnas (1985).
115 https://commons.wikimedia.org/wiki/File:Ground-Space_based_hybrid_laser_weapon_concept_art.jpg.
116 As reported by Boyer (2004).
117 Brooks Jr (1975).

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005

118 Brooks Jr (1975).
119 Brooks (1987).
120 Brooks (1987).
121 As recalled by Lehman in an oral interview conducted by William Aspray (Lehman, 1993).
122 Later republished as part of a joint book (Lehman and Belady, 1985).
123 Lehman (1980).
124 Dijkstra (1977).
125 Naur (1993).
126 The different pace of development in the two cultures of programming is reminiscent of the intercalated periodisation in the history of microphysics by Galison (1997), where instrument, theory and experiment sub-cultures also advance at a different pace.
127 Brooks Jr (1975).
128 Parnas (1985).
129 https://commons.wikimedia.org/wiki/File:Spiral_model_(Boehm,_1988).svg.
130 Boehm (1988).
131 Floyd et al. (1989).
132 Floyd et al. (1989).
133 Floyd et al. (1989).
134 Floyd (1987).
135 Takeuchi and Nonaka (1986).
136 Schwaber (1997).
137 As noted by Fowler (2001), the shift did not happen solely in software development, but is a more general move in product design and manufacturing.
138 Gabriel and Frost (1984) and Richard P. Gabriel (personal communication, 17 February 2024).
139 Gabriel and Frost (1984).
140 Richard P. Gabriel (personal communication, 17 February 2024).
141 Steele and Gabriel (1996b).
142 As recalled by Richard P. Gabriel (personal communication, 17 February 2024).
143 The publication that popularised the idea is a book by Beck (2000).
144 Beck (2000).
145 Beck (2000).
146 Outlined by Beck (2003).
147 Naur et al. (1969).
148 Gelperin and Hetzel (1988).
149 Evans and Darley (1966).
150 Modern Smalltalk-based programming environments are perhaps an exception from this rule, but the innovations developed in those systems rarely leave the particular technical context from which they originate.
151 Backus (1980).
152 Using the terminology introduced by Star and Griesemer (1989).
153 Armstrong (2007).
154 Evans and Darley (1966).
155 Using the terminology of Polanyi (1958), debugging is a kind of personal knowledge.
156 Dijkstra (1993).
157 MacKenzie (2004).

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 11 Nov 2026 at 12:08:10, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.005