A Little Book on Error-Free Software

Mats Danielson

A Little Book on Error-Free Software

Mats Danielson

Sine Metu

Published by Sine Metu Productions, Stockholm, Sweden, 2023
Website: www.sinemetu.se

Front and back cover images: DALL·E 2
DALL·E 2 was prompted to illustrate the fight against software bugs

ISBN 978-91-531-0456-8

First printing, August 2023
Parts first published as a compendium in 2009

The computer is more than a tool. It is a medium in which we think and feel. Its programming languages are materials out of which we can build castles of the mind. Programming becomes a kind of craftsmanship—a process of constructing and debugging that is deeply personal.

Sherry Turkle, 1984, p. 104

Contents

Preface

Most programming books are written for hackers rather than programmers. This book, by contrast, is written with programmers in mind. What does this mean? For the purpose of this book, a hacker is someone who sees programming as a personal hobby. They spend their spare time writing code, often just for the joy of doing so, and tackle a wide range of tasks without necessarily needing a practical reason. Hackers represent a small portion of all software developers, perhaps about one percent. This is the original meaning of the word hacker, and it does not refer to people who break into or damage computer systems for fun or profit. Those individuals are referred to here as criminals. Programmers, on the other hand, make up the rest. They enjoy programming as a creative and intellectually stimulating job, but they choose other things for their unpaid time. Surprisingly, most programming books are targeted at hackers and enthusiasts, apart from the well-known "for dummies" books and other materials intended for complete beginners. Since the vast majority of software developers fall into the category of programmers in this sense, it makes more sense to focus on improving their productivity and skills than to cater to the hacker community.

This story begins in the early 1980s. The author, just out of upper secondary school, like many young people without wealthy parents, looked for work instead of going to university. At that time, entry-level jobs were more accessible to those with only a high-school diploma. Having already worked as a computer operator on a shift schedule during a summer internship, and also during military service in the Navy, it felt natural to look for jobs in the growing computer industry. After applying to several companies, fortune had it that a job offer came from Philips Elektronikindustrier (PEAB) just outside Stockholm. Philips had a Swedish manufacturing plant known as PEAB which had three divisions. The job was in PEAB-T which was part of Philips Data Systems that produced front office equipment for banks, retail stores, and other service providers. This product line was known as Philips Terminal Systems (PTS). At the time, Philips was one of the top three global manufacturers in the banking front office segment, alongside NCR in the United States and Nixdorf in Germany. Together, these three accounted for half the world market, with Philips leading at over twenty percent. The PTS hardware was entirely custom-

designed, including the CPUs, all other electronics, cabinets and peripherals such as printers for bankbooks and vouchers, as well as ATM machines.

By 1980, all major Swedish banks except one used PTS equipment for their front office operations and ATM networks. The same was true for a large number of banks throughout Europe, which was Philips' core market, and also many banks in other parts of the world. The exception was the United States, where the domestic supplier NCR had a strong position. In the rest of the world, Philips had a position as the global leader in banking front-office technology. However, in the mid-1980s the landscape began to shift with the rise of the microprocessor and the arrival of the IBM PC. These developments made smaller, more flexible computer systems widely available at lower cost. At the same time, the Unix operating system started to gain ground in the banking and retail sectors. Philips PTS, despite its earlier success, struggled to adapt to these changes. The custom-built minicomputer models it relied on was losing relevance. In 1989, the division was renamed Philips Financial Business Systems, but it was already facing serious challenges. Two years later, in 1991, it was acquired by Digital Equipment Corporation and merged into a new unit called DEC BCFI (Business Centre Financial Industries). This marked the beginning of the end. When Compaq, a personal computer manufacturer, bought Digital Equipment in 1998, much of the original PTS culture was already lost. By 2001, when Hewlett Packard acquired Compaq, the legacy of Philips Terminal Systems had totally disappeared. A detailed history of the entire Philips Data Systems rise and fall can be found in (Danielson, 2023). In a little over a decade, Philips had gone from being a global leader in front-office banking systems to vanishing from the field entirely. And with it, the programming culture of PTS was nearly forgotten.

But not completely. Some of the people who worked there tried to carry the PTS coding spirit on, and it was only now that it became clear that the spirit had become rather well-known in some programming circles. The author did his best to carry it on in the workplaces he came to work at after PTS, and he tried to write down some of the guidelines for his own preservation. He also tried to keep them updated when he met new languages and new trends such as OOP and design patterns.

The PTS tradition and culture have sometimes been mentioned in writing, for example by Bo Sandén, a PhD working at Philips PTS during its heyday (the early

1980s). Sandén went on to become a professor at various US universities. In his book (Sandén, 2011, p.246), he writes

> *Philips Terminal Systems (PTS), where I worked later, produced teller terminals for banks. Their software had the wonderful property of being correct; once installed, it performed without a glitch. Such were some dedicated systems of old. This one ran a single teller terminal application—albeit with multiple tellers—under the custom operating system TOSS.*

Until now, however, these principles have not been collected into a structured set of guidelines. Yet over the years, they have contributed to software projects that came remarkably close to being error-free. In fact, based on the author's experience across two decades as a software consultant, there has been a clear pattern. The more closely a development team adhered to the PTS rules of thumb, the fewer errors appeared in the final software. Conversely, when such practices were neglected or only loosely followed, faults became more common, often in ways that could have been prevented with better discipline. These observations are not drawn from theory but from real-world projects across a variety of organisations. What became evident over time was that good practices, when applied consistently, make a measurable difference. The PTS rules may seem simple, even obvious at times, but their power lies in their cumulative effect. Following them can raise software quality significantly, and often with less effort than one might expect.

All banking systems sold were turnkey solutions and the software was bundled with the hardware sales. Thus, banking software for installations worldwide was produced at PEAB-T. How did Philips manage to develop and maintain the organisational skill required to produce software that was almost error-free? It was not by hiring large numbers of graduates from computer science programmes, since at the time such were still rare outside the United States. Of course, people with university degrees in other fields were hired, along with those who had no academic background at all. But the real strength of the organisation came from within.

At Philips Terminal Systems, there was a broad system of internal courses designed to teach everything from programming languages and coding practices to operating systems and hardware design. These courses formed a kind of internal university. Each employee's progress through the various topics was monitored, and

so too was their ability to apply the knowledge in real-world projects. As you gained experience and demonstrated skill in a particular area, you were gradually trusted with more complex tasks and greater responsibility. Learning was not confined to any classroom. Much of it took place through exposure to the work of more experienced colleagues. You were given time to study their code, observe how they worked, and engage with them directly. While pair programming was hardly yet invented, and would not have been encouraged even if it was due to gross inefficiencies, skills were transferred in other, sometimes subtle, ways. There was a strong culture of discussion and idea-sharing, both within project teams and across teams. People were encouraged to talk through design decisions, review each other's solutions, and support each other in debugging and problem-solving. This culture created positive feedback loops. The fewer errors that appeared in the software, the more time and mental space the team had to collaborate, explore, and improve their skills. Instead of spending time firefighting defects, they could focus on learning from one another and advancing their crafts. It was this shared commitment to quality and continuous learning that sustained high standards over time.

This is not to say that software project teams at Philips were never under pressure. Of course there were times when deadlines were approaching quickly and teams simply had to deliver. However, such occasions were generally limited in duration and could be managed mostly at the organisational level. One of the reasons this worked well was that coding standards and development practices were shared across different projects and teams. This meant that developers could be reassigned between projects when needed, without a significant drop in productivity. They were already familiar with the way things were done, which allowed the organisation to remain flexible and responsive. Over time, the effectiveness of the software teams became well-known internally. The sales department began to factor this into project planning, gradually reducing the time and budget allocated to the software component of larger projects. In a sense, the continued success of those teams raised expectations, creating an ongoing incentive to refine processes and improve further.

However, the culture that supported this success did not easily transfer to other parts of Philips in other countries. In many cases, it was not well understood, even by those working within PTS. Most developers at PTS were in their first or second

jobs in software, and they quickly developed the impression that software development was simply not that difficult. This belief was often shattered later in their careers when they moved to other organisations and encountered different development cultures. The practices and structures they had taken for granted at Philips were seldom present elsewhere, and it became clear that the apparent ease of PTS development had come from more than just individual talent and devotion. It had grown from a carefully cultivated and disciplined environment.

Perhaps a first hint of how these PTS principles work comes from another passage in Sandén's book. His book is about a specific problem in programming, that of concurrent processes and tasks, but some of his observations are much broader.

> *Treated as skilled artisans (Turkle, 1984), we crafted the software by emulating others and visualised their software architectures as we pored over their code. That's how we acquired the tacit knowledge of the trade (Hoare, 1984). One redeeming trait was that almost every assembly instruction was commented. Seeing how reluctant "real programmers" are to document anything, this was a remarkable concession to understandability.* (Sandén, 2011, p.246)

This should not be taken as an argument against formal training in computer science, but rather simply being an observation that there might be something more to software development than what is being taught in more formal course settings, especially at universities.

Most of the text in the book was written around 2000–2001 when the author was about to change sectors to academia, with additional material drawn from a compendium compiled in 2009. However, the text remained incomplete and was not assembled into a book until the summer break of 2023. Special thanks go to the publisher's English editor who improved the language enormously, to the extent that it almost feels like a new manuscript.

Happy reading!

The author, Stockholm, August 2023

1. What is Programming

Programming is not a branch of engineering, nor is it a subfield of mathematics. To assert that it is reveals a failure to understand either what software is, how it is created, or the unique mental labour required to produce it correctly. The conventional categorisations into which programming is often forced, be it civil engineering metaphors or the rigour of mathematical formalism, are each inadequate in capturing the ontological and epistemic particularity of software construction. Rather, programming must be recognised as a distinctive intellectual craft, shaped by a dynamic interaction between static structure and potential behaviour, between symbol and execution, between abstract form and concrete manifestation. It belongs neither wholly to the sciences nor entirely to the humanities, yet it borrows tools from both. It operates in a realm where the artefacts produced have no weight or size, and yet are required to behave with predictable precision in a wide range of contingent contexts.

The concern is not with algorithmic elegance, nor with theorems about computability, but with clarity and cognitive legibility of code to human readers and maintainers. Complexity is the adversary, not because it cannot be understood in principle, but because its accumulation overwhelms our capacity to reason about behaviour, especially over time and at scale.

Unlike mathematics, where correctness is a question of proof and the artefact is complete upon the derivation of the theorem, programming never guarantees correctness except by exhaustive empirical testing. The testing is not mathematical; it is practical. The artefact is "complete" only in a contingent, tested sense: correct enough, robust enough, under known and bounded conditions. Yet those conditions are rarely static. Code needs to evolve; it is not a static thing. If mathematics is content to assert timeless truths within axiomatic systems, programming must respond to dynamic specifications, moving targets, new inputs, and novel uses. This alone disqualifies the idea that programming is a mathematical act, however much it might borrow notational or logical tools from that domain.

Nor is programming engineering in the classical sense. Physical engineering relies on three-dimensional materials that obey invariant physical laws. Steel, glass,

concrete, you name it. These materials are subject to gravity, fatigue, compression, and thermodynamics. An engineer can stress-test a bridge or a building. A structural flaw can often be seen or measured. But code has no tensile strength. It cannot be weighed. It has no texture. The failures it produces are not cracks but crashes; they are invisible until they manifest, often in interaction with systems or data that were never anticipated by the original programmer. In this respect, software systems are not merely fragile; they are epistemically fragile. That is, our knowledge of how they will behave is necessarily incomplete. Testing becomes not just a tool for error detection but a form of epistemic validation: the generation of knowledge about the artefact through empirical investigation.

Programming cannot be reduced to the execution of rules. It is not enough to know the syntax of a language or the design patterns of an architecture. Programming requires judgement, experience, and critical thinking. The ideal practitioner is not just "a jack of all trades" but someone who thinks about their work in a conscious and who critiques the work without fear. In programming, the constraints are not of material scarcity but of conceptual entropy, of systems growing out of control, of code becoming opaque even to its own authors.

The dualism of programming, the intertwining of static representation and dynamic behaviour, is key to its distinctiveness. No other discipline requires such simultaneous mastery over symbolic abstraction and temporal execution. Data structures are static; they describe the layout, the types, and the potential relationships among elements. But these structures are inert until executed. The execution brings behaviour, motion, and side-effects. And the behaviour of software is not determined purely by its structure. It depends on input, runtime context, and environmental conditions. This is why programs must be tested with extreme data points, corner cases, and degenerate paths. Testing every control path is not an ideal but a practical necessity. And yet, paradoxically, it is also practically impossible in any nontrivial system. This tension between the need for certainty and the impossibility of proof is what gives programming its unique philosophical flavour.

The artefacts of physical engineering are external to the engineer. A bridge, once built, stands or falls by the laws of physics. Its flaws are usually observable and repairable. In programming, the artefact is internalised. It must be *understood* to be

maintained. It must be *read* to be verified. There is no physics of code. There is only the logic of its execution and the quality of its documentation. This is why programming is so often likened to writing. But unlike prose, code must be both readable and runnable, both interpretable by humans and executable by machines. This dual audience further deepens the intellectual demands on the programmer, who must constantly balance clarity with performance, elegance with expediency, and abstraction with transparency.

Software systems also differ from physical structures in their changeability. Once a building is constructed, modifying it is costly, disruptive, and often constrained by physical law. Software, by contrast, is fluid. It can be copied, refactored, forked, and extended at near-zero marginal cost. But this fluidity is deceptive. Each change risks introducing bugs, altering assumptions, and breaking interfaces. The cost of complexity is cumulative. It is not the feature that kills you, but the interactions among features. The problem is not code in isolation but code in context. Complexity emerges not from any single module but from the interdependencies among modules, the assumptions they encode, and the coupling they imply. Hence the necessity of design principles that foreground isolation, clarity, and layering, principles that resemble heuristics far more than they do the formal rules of science or the deterministic mechanics of engineering.

To understand programming as a craft is not to romanticise it. It is to acknowledge that it resists complete systematisation. Methodologies, languages, and tools may offer guidance, but they do not eliminate the need for judgement. The argument is that good programming emerges not from adherence to formal method alone but from critical engagement with the problem space, the user context, the evolving codebase, and the practitioner's own reasoning. These are not procedural skills but intellectual virtues.

Craft also implies community. A craft tradition is not simply a personal set of techniques; it is a shared culture of standards, mentorship, peer review, and pride of authorship. The code should be traceable to its author. Not to shame or penalise, but to affirm ownership and responsibility. Anonymous code, like anonymous architecture, is often fragile and incoherent. Code with a signature carries intention. And intention, in craft, is what separates function from art. The remarkable autonomy

with which programmers operate in the absence of formal epistemologies, exploring implications for pedagogy, professional culture, and systemic risk

If programming is a true craft, then it is a novel and modern one: dispersed, decentralised, and highly autonomous. There exists no formal epistemology of programming akin to mathematical logic, no universally established method of validation or pedagogy. Yet millions of individuals write code every day, many without formal education in software development, and often with a striking degree of operational independence. This is perhaps the most extraordinary fact about programming: we allow it to be done with so little oversight, so little enforcement of procedure, and so much reliance on the discretion of the individual. Programmers are, in practice, left largely to their own devices, tools aside, to construct artefacts of enormous social, economic, and individual consequences.

The professions and subjects to which programming is most often compared, such as engineering, mathematics and architecture, all feature stringent systems of certification and procedural vetting. To build a bridge, one must be licensed. To prescribe medication, one must be authorised. These professions protect their practices through accreditation, regulation, and peer oversight. Software development, by contrast, exhibits few such constraints. There is no licensure required to build a payroll system, an e-commerce backend, or even safety-critical code running in hospitals or aircraft. The artefacts themselves are not subjected to independent physical testing, because they cannot be; their behaviour must be inferred, simulated, or empirically observed. And so we rely on the programmers' own conscience, standards, and ability to anticipate complexity, isolate faults, and reason about correctness.

This has led to a peculiar kind of professional culture, one that emphasises learning by doing, by tinkering, and even by copying and remixing code found in the wild, for example on online forums. While formal instruction in computer science exists, and is valuable, it is neither a necessary nor sufficient condition for competent programming. This is not a cause for celebration; rather, it demands explanation. What we call best practice is often a matter of oral tradition, informal mentorship, or individual experimentation. This absence of centralised knowledge production does not make programming unprofessional. It makes it artisanal. Knowledge spreads through communities of practice, not doctrinal institutions. It is shared

through example, code review, open-source contribution, and not least admitted failure.

This autonomy is enabled, in part, by the nature of software itself. Code is amenable to version control, rapid iteration, and instantaneous deployment. This allows feedback loops that are far shorter and more intimate than those found in physical disciplines. A civil engineer must wait months or years to see a project realised in the real world. A programmer sees the result of their intervention immediately. If not in a browser, then in a test suite or a debugger. The machine becomes both canvas and critic. This immediacy allows for a kind of solo apprenticeship: through the recursive interaction with the machine, the programmer internalises a sense of cause and effect, of effort and consequence. They develop, over time, an intuition for system behaviour, a feel for abstraction, and a sensitivity to fragility.

And yet this mode of working, independent, improvisational, loosely governed, carries risk. As software systems become more central to public life, the consequences of design failure scale accordingly. The failure of a database may now compromise not a ledger but a democracy. An error in a sorting algorithm may not inconvenience a few users but structurally bias the allocation of loans, benefits, or jobs. In such a context, the reliance on personal integrity, on craft as a virtue, begins to seem insufficient. It is here that the limitations of the craft metaphor reveal themselves. Craft traditions, historically, were slow to scale. They resisted mechanisation. They thrived on direct experience and bodily engagement with materials. Programming, in contrast, can be scaled globally and instantaneously. The same script may be executed a billion times. A single failure can propagate through the cloud at the speed of light. What kind of craft is this, then? One that must remain artisanal, yet produces artefacts whose effects are industrial?

This is the paradox of programming today: it must retain the sensibility of craft while bearing the responsibility of infrastructure. It is neither sufficient to treat programming as amateur tinkering, nor accurate to treat it as a solved engineering science. It is a knowledge practice, but one whose knowledge is often tacit, situated, and difficult to formalise. Polanyi described this kind of knowledge as “we know more than we can tell” (Polanyi, 1966). Programming is replete with such knowing. Ask an experienced developer how they detected a subtle concurrency bug, and they

may tell you a story, not a theorem. They may invoke pattern recognition, experience, and aesthetic discomfort. They may say: "It just did not look right." This is not superstition. It is the residue of accumulated craft knowledge, encoded not in rules but in intuition.

Indeed, much of programming operates at this almost subconscious level. Despite the apparent formalism of code, many decisions are made based on readability, style, or maintainability. Concerns that are social more than logical. Style guides, naming conventions, and indentation rules: these are expressions of culture, shared values, and implicit pedagogy. Complexity is not a metric of cleverness but a tax on future cognition. The good programmer is not the one who writes the shortest code, but the one who enables the clearest reasoning. Again, we are in the domain of intellectual craft, not algorithmic virtuosity.

In this culture of craft, tools matter but not as ends in themselves. No tool is the universally best. Each must be judged in context, selected pragmatically, and evaluated empirically. The emphasis is on adaptability, not adherence. This anti-doctrinaire posture is characteristic of craft traditions, which favour heuristics over rules and exemplars over axioms. The practitioner must know not just how to use the tool, but also when and why, and when to abandon it. Thus the programmer becomes, in its ideal form, a doer: one who constructs from what is at hand, who adapts and combines, who is not bound by purity but guided by function.

This notion of a doer is not the absence of rigour. It is a case of situated judgement. The complexity of modern software systems makes pure formalism insufficient. The search space is too vast. The constraints are too variable. The goals are too conflicting. What is required is not a calculus but a sensibility. This is why programming, properly understood, must be treated as a practice of cultivated intelligence. A craft of the mind, performed through the medium of symbol and logic, in negotiation with both machine constraint and human ambiguity.

But if programming is a craft, then where is its studio? Where is its apprenticeship model? Where are its roots, its shared language of critique, and its pedagogical lineage? In truth, programming pedagogy remains fragmented. University courses focus on algorithms and data structures, computability and complexity. These are necessary but not sufficient. They provide the grammar but not the rhetoric. They teach

how to compute, not how to design. They treat software as output, not as dialogue. And design, as any craftsperson knows, is where the discipline lives. Without design, there is only assembly.

This educational gap has led to the rise of alternative pedagogies: coding bootcamps, open-source mentorships, pair programming, and online communities. These are not substitutes for formal education, but they do fill a void. They provide the studio and apprenticeship model that the academy often lacks. They offer critique, collaboration, and visibility. They allow the transmission of tacit knowledge through joint activity. But they are also unregulated, inconsistent, and exclusionary in their own ways. Not all apprenticeships are created equal. Not all studios are healthy. And so the craft of programming, while vibrant, remains unevenly distributed. Its standards are emergent, not enforced; its virtues are aspirational, not certified.

This, perhaps, is the final irony. Programming is one of the most consequential occupations in the modern world, shaping economies, institutions, and social experience. And yet it remains among the least formalised. Its practitioners operate with enormous autonomy, under conditions of accelerating complexity, without a shared epistemology or pedagogical framework. They succeed, when they do, not because their profession guarantees it, but because they have cultivated habits of attention, reflection, and coping. These are not technical skills. They are intellectual virtues. And they are the foundation of programming as a craft.

Correctness in Programming

The notion of correctness occupies a central yet precarious position in programming. Unlike in mathematics, correctness cannot be proven in most practical cases; and unlike in physical engineering, correctness cannot be observed through physical behaviour alone. The code may run, it may return the right output under known conditions, and still it may harbour fatal errors that might manifest only under obscure input sequences, specific timing interactions, or after scaling beyond its original design parameters. This is not an accidental feature of programming but an essential one. It arises from the dual nature of software as both static and dynamic: the code as written is a static artefact, while its meaning, the way it behaves when executed, is only revealed in motion. Correctness, then, is not a stable property of the artefact

itself, but a relational property that emerges in interaction.

This distinguishes programming from mathematics in the most fundamental way. In mathematics, a theorem is correct if it is derivable from axioms via valid inference. The artefact (the proof) is both the justification and the product. Once the proof is complete, the matter is settled. No further observation is needed. But in programming, a piece of code may compile successfully, it may even be logically coherent within the grammar of the language, and yet behave utterly incorrectly under execution. The gap between syntax and semantics, between the form and the effect, is not merely a theoretical nuisance. An error-free compilation means almost nothing. This is the heart of the problem. It is why even formally verified software, where logical specifications are proved against code, can fail to meet real-world requirements. The specification itself may be incomplete, incorrect, or naïvely conceived. In the world of actual programming, there is no refuge in proof.

Nor is correctness accessible in the manner of physical engineering. When a bridge stands, its load-bearing properties can be modelled, inspected and tested. The structure exists in three-dimensional space and obeys known laws of physics. When it fails, the reasons can often be diagnosed through material analysis, sensor logs, or physical inspection. But when a program fails, when it crashes, produces the wrong output, or hangs indefinitely, the cause is not visible in the static artefact. One must infer it from traces of dynamic behaviour. The code must be examined line by line, the logic mentally reconstructed, the inputs replayed and the state recreated. The failure is semantic, not material. It is a kind of conceptual rot, undetectable by the senses, knowable only through reasoning or simulation.

This epistemic opacity makes testing central to the craft of programming. Testing is not a postscript to implementation; it is the primary method by which the programmer discovers what the program *actually does*. The programmer does not *know* where the flaws are. He can only probe, sample, and experiment. This makes testing less like measurement and more like science: hypothesis, experiment, falsification. Yet this scientific endeavour is constrained by pragmatism. Testing every possible state or path is computationally intractable in all but the most trivial programs. The number of potential states in even a simple interactive system is exponential in the number of input variables and internal branches. Thus, exhaustive testing becomes

impossible, and correctness becomes statistical. We gain confidence in our code not by proving it correct, but by failing to find it incorrect under many plausible scenarios. This is a philosophical compromise. But it is also a practical necessity. The alternatives, to do nothing or to await perfect proof, are paralysis.

This has consequences for how we think about responsibility in programming. In physical disciplines, responsibility is often shared between design and materials. A bridge collapses, and we ask: was the design flawed, or did the materials fail? But in programming, the design *is* the material. There is no intervening substance between idea and execution. The artefact is the embodiment of logic itself. Thus, when the code fails, the fault is conceptual. It lies in the model, the assumptions, or the abstractions. And therefore, it is personal. Complexity is not a neutral state. The programmer is accountable not just for what the code does now, but for how understandable it will be tomorrow. Maintenance is 80% of time spent coding, or more.

The code becomes a signature of intent, of care, of quality. It signals that someone has thought about the edge cases, wrestled with the ambiguity, and stands behind the logic. This stands in sharp contrast to industrial models of software production, where code is often anonymous, commoditised, and subject to rapid churn. In the craft model, code is not just written; it is authored.

But even the most careful author cannot eliminate all uncertainty. In programming, correctness is a moving target. As requirements change, as inputs evolve, and as systems integrate, what was once correct may become incorrect. Acceptable behaviour may become insecure. Valid outputs may become offensive or discriminatory. This makes programming not just a creative activity, but a recursive one. The artefact is never done. It must be revised, retested, re-understood. This evolutionary view further distinguishes programming from mathematics, where proofs remain valid once established, and from physical engineering, where the material constraints remain stable across time. Programming occurs in a fluid epistemic landscape. The platform changes, the user changes, and thus the threat model changes. The artefact must adapt or perish. This is why test coverage, version control, and modularity are not just technical conveniences but existential safeguards. They allow the artefact to change without losing its integrity. They make the artefact self-reflexive, able to contain within itself the means of its own renewal.

Testing, then, is not a chore to be appended to development. It is part of the ontology of programming itself. It is the mechanism by which static intention is transformed into dynamic verification. But it is also incomplete. No test can guarantee that the program does the right thing. It can only show that it fails to do the wrong thing under specified conditions. This is why testing must be complemented by documentation, naming conventions, and readable code. By anything that allows a human to *understand* what the code is meant to do. And herein lies a tension. We often speak of code as if it were formal, rigorous, and deterministic. But our practices betray that illusion. We test empirically. We debug heuristically. We refactor aesthetically. We rely on code linters, code reviewers, and informal conversations. We search Stack Overflow. We experiment. These are not the habits of mathematicians. They are the habits of craftspeople. People who know that materials lie, that tools fail, that the artefact may turn against them if handled carelessly. They respect the medium not for its purity, but for its complexity.

This perspective should not be mistaken for anti-scientific relativism. The logic of code is precise, and bugs are real. But the context in which code runs, and the intentions it embodies, are neither fully specifiable nor fully stable. There is always a residue of ambiguity, of human error, of unexpected interaction. The craft of programming consists of managing this ambiguity without succumbing to it. It is the art of the incomplete proof, the unprovable necessity, and the partially ordered world. The consequence of all this is profound: programming does not produce artefacts that are proven correct, but those that must be *believed* to be correct based on evidence, inspection, and trust. This is a unique epistemic condition. It blends logical reasoning with empirical methods, intuition with formalism, and experimentation with tradition. It cannot be captured by diagrams alone, nor by proofs alone, nor by test results alone. And it is in this mode of practice, in this balance of constraint and creativity, verification and invention, that programming reveals itself as a distinct form of intellectual craftsmanship.

Expressiveness and Interpretation

To speak of programming as a craft is also to acknowledge its expressive dimension.

Code is not merely instruction to machines; it is also communication between humans. The programmer writes not only for the compiler, but for the reader, who may be a teammate, a future maintainer, or the programmer's own self revisiting a system months later. The expressive character of code, then, is not incidental; it is central. It conditions how complexity is managed, how collaboration unfolds, and how understanding is preserved across time. The mistake of many formal treatments of programming is to treat code as pure logic. In truth, code is also language.

This linguistic quality gives programming a dual audience: the machine and the human. The machine demands unambiguous syntax, determinism, and operational semantics. The human seeks legibility, meaningful structure, and economy of expression. These two demands often exist in tension. The shortest code is not necessarily the clearest. The most performant code is not necessarily the most maintainable. It is the job of the craftsperson to navigate these tensions, to produce artefacts that are at once precise and communicative, functional and intelligible. This is why naming matters. Why indentation matters. Why code layout, ordering, and module boundaries matter. They are not cosmetic choices. They are rhetorical devices in a language whose semantics extend beyond the machine.

Clarity is not just an aesthetic preference; it is a cognitive strategy. Code that is easier to understand is easier to verify, easier to modify, and easier to reason about. It reduces the cognitive load on the reader, allowing them to build a correct mental model of the system. This, in turn, reduces the likelihood of bugs, the cost of onboarding, and the risk of unintentional collapse. Clarity, then, is not polishing the end product. It is structural. The pursuit of clarity brings programming into the domain of writing, not writing as a means of verbal expression, but writing as an act of structuring thought. Code is structured thought rendered in a symbolic system, bound by syntactic constraints but animated by semantic intent. Just as writers must consider audience, context, tone, and narrative arc, so must programmers consider interfaces, abstraction, side-effects, and modular cohesion. The best code, like the best prose, invites understanding. It leads the reader. It suggests, rather than obscures, intent.

But this analogy also illuminates a key challenge. Language is inherently interpretive. No sentence is immune to misreading. No text guarantees a single, universal

interpretation. Code is less ambiguous than natural language, but not immune. A function name may suggest the wrong metaphor. A variable name may imply the wrong unit. A nested loop may conceal a performance hazard. Even if the code behaves correctly under execution, its *meaning*, its intent, rationale, and domain logic, may be misread. This is the interpretative burden of programming. One must not only understand what the code does but also why it was written that way.

This interpretative burden is exacerbated in systems that evolve. As software accretes features, fixes, and refactors, the original design intent is diluted. Without clear structure, without expressive interfaces, the code becomes opaque. Still executable, but no longer readable. This is incremental complexity: the slow build-up of structural entanglement that makes each new feature harder to add and each bug harder to trace. The antidote to this is not merely better tools or stronger typing. It is more deliberate writing. It is the insistence that code be readable *as a narrative* and that its structure reflects the conceptual architecture of the domain it represents.

Such deliberateness requires taste. It requires the capacity to recognise when code is "good," in an aesthetic and/or functional sense. This is another hallmark of craft: the presence of aesthetic judgement. The experienced programmer knows when a function is too long, when a class is overloaded with responsibility, or when a dependency breaks encapsulation. These are not issues of syntax but of style and structure. They are perceived, not computed. And they are refined through practice, exposure, and critique. This is why programming, like writing, benefits from peer review. Not just for correctness, but for clarity. Not just for bugs, but for beauty. Beauty in code is not about ornament. It is about proportion, coherence, and expressiveness. A beautiful piece of code is one that says exactly what it means, no more and no less. It is one whose structure mirrors its function, whose boundaries reflect the conceptual seams in the problem space. It is not clever for its own sake. It is elegant because it does much with little and reveals rather than hides. Yet programming differs from writing in one decisive respect: code must run. The artefact is not merely expressive; it is executable. It is not enough for it to be well-structured; it must also do something, and do it correctly. This is what makes programming uniquely demanding. The artefact is both a description and a prescription. It both tells a story and enacts a procedure. It must satisfy two masters: the human and the

machine. No other medium has quite this duality. Blueprints are never mistaken for buildings. Scripts are not confused with performances. But code is both script and performance. It is both notation and action.

This duality also means that code can deceive. It can appear clean while hiding subtle bugs. It can look elegant while violating essential invariants. It can behave correctly under test cases while failing under edge conditions. This is why style is not a substitute for correctness. The code must both read well and run correctly. This demands a level of discipline and self-awareness rarely required in other forms of writing. The programmer must be both author and editor, both dramatist and critic. They must anticipate not only how the machine will execute their code, but how another human will interpret it: under pressure, with limited context, perhaps in a different time zone or cultural milieu.

These demands are rarely made explicit in traditional computer science education. Courses focus on computability, complexity, and algorithmic efficiency. These are important, but they are not enough. They do not teach the student how to name variables meaningfully, how to break a problem into layers, how to balance performance with readability, or how to write for the next person who reads the code. These are rhetorical skills. They require empathy, judgement, and revision. They are closer to design than to science, and closer to prose than to proof. They are, in short, the skills of a craft.

One of the most telling indicators of this linguistic dimension is the very existence of programming idioms. Just as natural languages evolve idioms, common phrases whose meaning transcends their literal syntax, so too do programming languages develop idiomatic forms. These idioms are not enforced by the language. They are conventions learned by reading, mimicking, or participating in a culture. They mark membership in a community of practice. They signal familiarity, trustworthiness, and competence. They are the oral tradition of the programming tribe. Yet idioms can also obscure. What is idiomatic to one community may be cryptic to another. This variability reinforces the need for interpretative flexibility, for the ability to read across styles, adapt to dialects, and understand not just what the code does but how it is situated in its cultural context. This is not a technical skill. It is a hu-

manistic one. It requires the ability to empathise with the author, reconstruct intention, and navigate ambiguity. In this sense, programming is a philological act. It is the interpretation of text under the constraints of logic and effect.

Design is also about shaping interfaces. Not just technical ones, but conceptual ones. An interface, after all, is a boundary across which understanding must flow. If the interface is poorly named, poorly documented, or poorly structured, the boundary becomes opaque. The cost is not just technical; it is cognitive. The downstream reader must reverse-engineer the logic, reconstruct the assumptions, and divine the contract. This is labour that good design should prevent. The goal is not to prevent all bugs since that is impossible, but to prevent misunderstanding. Misunderstanding is the root of error. The best code prevents misunderstanding by being as clear as it is correct.

Programming, then, is not merely the act of getting the machine to do what we want. It is the act of expressing intention in a medium that is both unforgiving and abstract. It is a writing practice in which the reader is often ourselves, months later, or a stranger in a different part of the world, under deadline, under stress. The best programmers know this. They write not only for the CPU but for the next human being. They craft their code as a message across time. And this, above all, is why programming is a craft. Not because it is beautiful, although it often is. Not because it is expressive, although it must be. But because it is authored with intent, interpreted with care, and sustained by a community that values both.

Absence of a Unifying Theory

The refusal of programming to yield to a unified theory of programming is not a sign of its immaturity. It is a structural feature of the medium itself. Programming resists total systematisation because its domain, human intention rendered as computational behaviour, is neither closed nor fully formalisable. While other disciplines develop mature professional frameworks by constraining their object of study, programming remains open-ended. Its object is not just "the program," but the evolving web of interactions, data, interfaces, side-effects, and human needs that define the software system over time. In this sense, the absence of a canonical method is not an oversight; it is a consequence of the terrain.

Why, then, has programming not developed the guild-like institutional trappings of other technical disciplines, accreditation boards, licensure systems, or codes of conduct? Partly this is historical. Programming emerged in the mid-twentieth century as an auxiliary to mathematical computation and military engineering. Its early practitioners were often physicists or engineers by training, and the academic disciplines that housed it were shaped by that lineage. But as programming grew into a global, autonomous practice, its disciplinary structure failed to evolve in tandem. Computer science, as taught in universities, remains focused on abstract formalism: automata theory, computational complexity, and logic. It did not become the foundation for practising programmers. Instead, the practice proliferated in industry, open-source communities, and informal training environments. The higher institutions never fully captured the craft.

The result is a bifurcation: on one side, an academic discipline rooted in the mathematical and theoretical foundations of computation; on the other, a vast, heterogeneous, and largely self-taught population of practitioners building the actual software systems that run the world. This bifurcation explains much of the unease in programming culture: the lack of a unified identity, the tension between science and practice, and the suspicion of academic approaches that seem out of touch with real-world needs. It also explains the persistent rediscovery of insights long known to craftspeople but marginal in the theoretical canon, such as the importance of naming, layering, and interface clarity.

Attempts to bridge this gap often take the form of ideas called methodologies, such as Agile or Extreme Programming (Beck, 1999). These are not scientific theories; they are cultural movements. They codify practices, articulate values, and propose rituals. Some are valuable, some are reactionary, and most are incomplete. What they share is a recognition that programming requires not just tools, but shared discipline, habits of interaction, and norms of code review. But none has succeeded in becoming canonical, because none can be. The variety of contexts in which programming occurs: web applications, enterprise systems, embedded software, simulations, and games, among many others, precludes methodological hegemony. No single set of rituals can govern them all. What is needed instead is a meta-discipline: a way of cultivating judgement about which practices to apply, when, and why.

However, craft as a model has its own limitations. It resists automation. It depends on tacit knowledge. It scales unevenly. This poses a serious challenge in a world where software must be secure, reliable, and compliant at scale. The stakes are higher than ever. Software no longer runs in isolated systems; it orchestrates finance, governance, infrastructure, and identity. The call for "software engineering" is therefore understandable: it promises predictability, standardisation, and auditability. It promises to transform programming from a cottage industry into a mature profession.

And yet, many decades after the phrase "software engineering" was coined, it remains more aspirational than descriptive. The failure is not one of ambition, but of fit. The metaphors of engineering, such as blueprints, load-bearing structures, or stress analysis, fail to capture the dynamics of code. Software is not poured into forms. It is not subject to physical degradation. Its cost is not in materials but in cognition. Its failure modes are not cracks but misbehaviours, often subtle, emergent, or context-sensitive. The assumptions that undergird physical engineering do not hold in the domain of code.

Moreover, the artefacts of software are mutable in a way that physical artefacts are not. One cannot "patch" a bridge in production with a keyboard and a Git commit. But one can, and often must, patch software. One cannot deploy ten thousand copies of a building to run in parallel on different continents. But one can, and often does, deploy software at a planetary scale. This scale and mutability require a different epistemology. They require practices that are robust under change, resilient under uncertainty, and responsive to user behaviour. These are not the qualities of classical engineering. They are the hallmarks of responsive craft.

If there is an analogy to be drawn, it is perhaps to architecture rather than civil engineering. The architect, like the programmer, works with constraints, client needs, budgets, and physical laws, but must also produce something inhabitable, interpretable, and meaningful. The architect's success is not only measured in structural integrity but also in experience. Likewise, a programmer's success is not only measured in correctness, but in usability, maintainability, and conceptual integrity. The blueprint is not the building. The code is not the system. Both must anticipate the future, embody purpose, and mediate ambiguity.

This interpretative burden is what makes programming somewhat philosophical. It is not reducible to physics or logic. It involves reasoning about systems that do not yet exist, in contexts that may change, for users who may misunderstand. It involves choosing between competing goods: performance versus readability, security versus flexibility, or abstraction versus transparency. These are not technical trade-offs alone. They are normative choices, structured by human values and cultural norms. They require judgements, not algorithms. They are shaped by critical reflection, not mere specification.

And this is why programming, even in its most industrial forms, remains irreducibly human. We may speak of outsourcing, coding automation, or recently of AI-assisted development. But behind every developing effort is a choice about what to automate. Behind every abstraction is a choice about what to hide. These choices are not made by tools, they are made by people. People who bring with them assumptions, biases, intuitions, and beliefs. The code they write is not neutral, however much their employer would like that. It is an inscription of these values into a system of behaviour. This inscription may be subtle, but it is consequential. It shapes what the system allows, forbids, enables, and precludes. It defines what users can see, can do, and can imagine.

Thus, the refusal of programming to become a fully standardised profession is not a failure of institutionalisation. It is an acknowledgement of its epistemic condition. Programming is a practice that operates in the gap between specification and implementation, between intention and behaviour. It is a form of design that must be both correct and communicative, both efficient and expressive. It is a form of knowledge production that resists full formalisation because its context is never fixed. The best we can do is to cultivate practitioners who can think in this space, who can tolerate ambiguity, reason across layers, and balance competing constraints. We can teach principles, not recipes. We can share cases, not commandments. We can pass on habits of mind, not mandates of doctrine.

This is the core of an intellectual craft. It is not about artisanal elitism. It is about recognising that good work requires good judgement, and that good judgement is cultivated through experience, reflection, and dialogue. It is about building a culture

that values clarity, wisdom, and experience, not because they are efficient, but because they are durable. In the long run, the systems that endure are not those that are merely optimised, but those that are well-understood. And understanding is not a by-product. It is the work itself.

Philosophy and Culture

To insist that programming is a craft is not to diminish it. It is to dignify it rightly. In a landscape dominated by metaphors of science and industry, it is tempting to frame programming as an immature engineering discipline, awaiting its equivalent of Newton or Gauss. Or, conversely, to treat it as a misapplied form of mathematics, whose tools are half-finished and whose proofs are merely implicit. But programming resists both characterisations. It is neither a subset of physical design nor a variant of formal reasoning. It is its own kind of knowledge production. It is a practice of shaping symbolic artefacts whose reality is behavioural, not spatial; whose correctness is partial, not proven; and whose purpose is evolving, not fixed.

Programming is not the act of commanding machines. It is the act of expressing intentions in a form that both humans and machines can interpret and act upon. This expressive constraint is not a weakness. It is what gives programming its creative and intellectual character. The programmer does not simply solve problems; they articulate systems of possibility. They design grammars of action. They render the fluid demands of human thought and institutional structure into the crystalline logic of execution. The elegance of a function, clarity of an interface, or modularity of a system are not incidental features. They are the signatures of a mind grappling with complexity, communicating across time, and reasoning under uncertainty.

And always, the work is dual. The artefact itself is static, but its life is dynamic. The program, in source code, is inert. But in execution, it lives. It consumes input, produces output, responds to stimuli, reacts to edge cases, and fails under pressure. The dualism is not just metaphysical. It is methodological. It is why we test, debug, simulate, refactor. It is why correctness cannot be asserted, only supported. It is why formal verification, though powerful in principle, remains rare in practice. The terrain is too wide. The assumptions are too fragile. The interactions are too many.

It is this gap between the code as written and the code as lived that makes programming distinct. Physical engineering has its materials; mathematics has its axioms. Programming has neither. Its materials are abstractions, and its axioms are provisional. The only test is behaviour. The only validation is empirical. The only guarantee is vigilance. And yet, it works. Complex systems are built, maintained, and evolved. Failures occur, but successes abound. Code runs in phones, planes, hospitals, farms, museums, laboratories, or just about anywhere. It shapes the world, not through brute force or aesthetic metaphor, but through the orchestration of logic and contingency.

This orchestration is a craft because it cannot be reduced to rule-following. It is not the mere application of templates. It is not the automation of prior knowledge. It is the careful negotiation between constraint and possibility. Between what is desired and what can be encoded. Between what is specified and what will be interpreted. The good programmer is one who sees this negotiation for what it is: not a burden, but the substance of the work. They do not seek to eliminate judgement, they seek to refine it. They do not seek to replace human intuition with static rules. They seek to train that intuition through disciplined experience.

This experience is communal. It is accumulated not in isolated minds, but in a culture: a culture of shared tools, practices, debates, and conventions. Programming culture, for all its fragmentation, is rich in informal pedagogy. Open-source repositories are textbooks of idioms. Style guides are expressions of commitment. Forums, issue threads, and documentation are sites of epistemic struggle: What does this do? What should it do? Why was it done this way? The work is not just in the artefact itself. It is in the discourse around it. It is in the code comments and commit messages, in the pull requests and post-mortems. These are the annotations of a living craft.

The craft is living because the systems are never finished. Unlike bridges or books, most software systems are under constant revision. Requirements change, libraries evolve, and platforms shift. The very fluidity that makes software powerful also makes it unstable. This requires a mindset not of finality but of care. The programmer must assume that what is written today will be read, modified, misunderstood, and extended tomorrow. They must write with that future in mind. This is not

taught in textbooks. It is taught in pain. In late nights spent tracing bugs. In awkward meetings explaining why a feature broke. In moments of insight when a better abstraction makes everything simpler.

To elevate programming as a craft is not to deny the value of science or engineering. On the contrary, it is to locate programming alongside them, as an equal but different form of rational engagement with the world. Science discovers what is. Engineering constructs what can stand. Programming constructs what can *behave*. That behaviour is not visible in space. It unfolds in time. It is not constrained by gravity but by complexity. It is not tested with callipers, but with test suites and simulations. It is not made robust by stronger steel alloys but by clearer logic, tighter interfaces, and better naming.

The intellectual demands of this work are immense. They combine symbolic reasoning with empirical investigation, social awareness with logical structure, and creative synthesis with critical review. Few other domains ask as much breadth from their practitioners. Few others entrust so much responsibility to the individual. Programming remains largely autonomous. Millions of practitioners write software with little formal oversight. They are trusted not because the process guarantees correctness, but because the culture encourages care. This is both a strength and a risk. But it is the condition of the discipline. It is what makes the craft fragile, and what makes it essential. And so programming must be taught, not only as syntax and control flow but as philosophy and judgement. Not only as data structures and algorithms, but as writing and design. Not only as a job but as a form of thinking. It must be framed as a craft because only that term captures its epistemic heterogeneity. The craft of programming is not what one does after the "real work" is done. It is the real work. It is the place where abstraction becomes action, where desire meets constraint, and where logic comes alive.

Finally, is there a built-in tension between regarding programming to be a craft or an art, on the one hand, and on the other striving for error-free software? At first glance, it might seem so. If programmers are let loose like artists painting impressionist visions, surely it will end up as ad-hoc codebases? The key here is to realise that the artistic freedom lies at the conceptual level, not the code level. Read on, and you will be enlightened.

2. A True Craft

This chapter is written a year later than Chapter 1 and addresses the identity of programming from a different viewpoint. As argued in the previous chapter, programming is an intellectual craft, not a form of engineering or a branch of mathematics. While programming shares elements with both disciplines, it transcends both those categories. The art of writing software code is not the application of prefabricated formulas to well-bounded problems, nor is it the derivation of abstract theorems. Instead, programming is a creative and at the same time scholarly pursuit that blends precision and ingenuity to produce complex, dynamic artefacts that are at the same time conceptual and operational. It is a craft in the truest sense: requiring specialised skills acquired through experience and reflection, and yielding outcomes judged not only by correctness but also by elegance and maintainability. This view was also held by early computing pioneers, not only modern practitioners. Donald Knuth (1974) said that *"computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty"*. Yet the term "art" in this context is similar to its older meaning of skilled practice (from Latin *ars*, meaning skill), much like the arts of carpentry or architecture, which combine creative vision with practical execution. This is the same concept of art as in KTH Royal Institute of Technology's (the author's alma mater; Danielson, 1997) motto "Science and Art", in which art should be taken as the engineers' craftsmanship. In contemporary language, one might say *craft*. Indeed, good programming exhibits the hallmarks of craft: it combines beauty with utility and creativity with discipline. As Knuth suggested, a programmer who approaches the task as a craftsperson, taking pride in the creation of a well-made software artefact, will enjoy what he does and will do it better (Knuth, 1992).

Consider how the perception of programming has evolved. In the 1950s, programming was a discipline often practiced by mathematicians and electrical engineers. Early computers were programmed in machine code or at best assembly language, and programming was often viewed as a low-level, almost clerical activity with teams of human "computers," often women, hand-calculating or hand-coding

algorithms. However, as projects grew in scale, for example, IBM's OS/360 operating system in the 1960s, which required hundreds of person-years (Brooks, 1975), the need for a more systematic approach became evident. The term "software engineering" was introduced at a NATO conference in 1969 to provoke thinking about applying engineering principles to software development (Buxton and Randell, 1970). Yet even at that conference, participants debated whether programming could truly be engineered or whether it was inherently a craft. Over the following decades, various movements in software methodology swung between formalism and flexibility. The 1970s brought structured programming and attempts to prove programs correct, with advocates like Dijkstra and Hoare promoting rigorous reasoning (see, e.g., Dijkstra, 1971). In contrast, the 1980s and 1990s saw a surge of practical techniques emphasizing manageability and reusability, such as modular design, object-oriented programming, and design patterns. By the late 1990s, frustration with heavyweight, plan-driven processes led to the Agile movement, essentially reasserting the importance of human adaptability and craft over strict processes (Beck et al., 2001). This historical back-and-forth illustrates a central built-in tension: on one hand, a desire for predictable, assembly-line software production, on the other, the reality that programming is an exploratory, creative endeavour that resists too much rigidity. As Kernighan (1984, 1988) once said, controlling complexity is the essence of computer programming. What has emerged in practice is a hybrid view: we need engineering discipline in areas like version control, testing protocols, and project management, but at the coding keyboards, programming remains a creative craft. The best results come when individual programmers are empowered to make decisions, experiment, and iterate, i.e. when they are treated as craftspeople who take ownership of their work.

Thoughts from 2001

It is necessary to distinguish this concept of programming-as-craft from the more rigid notion of programming-as-engineering. The software industry has long borrowed the language of traditional engineering: we speak of "software engineering," of architects and blueprints, of construction and maintenance. These metaphors were adopted in the late 1960s in response to the first "software crisis" (the second came

in the 1990s, leading to i.a. the agile movement) reflecting a desire to bring order and predictability to software development. Indeed, large software projects, like large civil engineering works, demand teamwork, planning, and sound methodology. However, the engineering paradigm does not preclude individual craftsmanship within the larger effort. As Hunt and Thomas (1999) observe in a recent, untraditional book on programming, even monumental structures like the great medieval cathedrals, which took decades and teams of thousands to build, ultimately depended on the mastery of individual artisans. The stonecutters, carpenters, carvers, and glass workers on those projects were "*all craftspeople, interpreting the engineering requirements to produce a whole that transcended the purely mechanical side of the construction*". Their personal skills gave the final structure qualities beyond what any blueprint could prescribe. The builders' mindset was captured in the so-called Quarry Worker's Creed: *"We who cut mere stones must always be envisioning cathedrals"* (Hunt and Thomas, 1999). In software development, similarly, while overarching design and requirements provide a framework (an analogue to structural engineering), it is the creativity and care of individual programmers that determine the difference between a merely functional system and an elegant, robust one. Within the structure of a software project, there is ample room for individual skill and judgement. It is important for developers to seriously "care about your craft," treating each coding task as an opportunity for creative expression rather than a rote assignment (Hunt and Thomas, 1999). Over time, the sum of these individual contributions yields software that reflects both solid engineering principles and the intangible imprint of craft.

Unlike a bridge or a building, a software program has no physical form that can be directly inspected or stress-tested in the traditional sense. Indeed, as Brooks (1975) noted, the programmer *"works only slightly removed from pure thought-stuff,"* able to *"build… castles in the air, from air"* through imagination alone. This tractable medium of pure logic is immensely flexible since the program's constructs "move and work," producing tangible outcomes in the real world. Paradoxically, this power of abstraction also makes software elusive to grasp in full, because there is no concrete artefact that can be mapped in its entirety. Software's reality is not inherently embedded in space, hence it remains *invisible* and *unvisualisable* in ways

physical structures are not. An architect can draw blueprints and construct scale models; an aerospace engineer can test a wing in a wind tunnel. But a software design, no matter how detailed on paper, cannot be fully understood until the program runs. Any attempt to visualise a complex program results in "not one, but several, general directed graphs, superimposed one upon another," representing control flow, data flow, module dependencies, and so on. These overlapping dimensions resist any single, static representation. The absence of a tangible model is not merely a cosmetic concern but an epistemological one: it impedes our ability to reason about the system's correctness and performance with the kind of certainty we have in classical engineering. We cannot *see* the whole of a software system at once; we must mentally synthesise its behaviour from reading code and observing runtime outputs. This is a key reason why programming must be approached as a craft of managing complexity in thought, rather than as straightforward construction. One of the earliest software engineering lessons noted this cognitive challenge: The craft of programming involves structuring our invisible designs in ways that our finite human minds can control. This stands in stark contrast to traditional engineering, where once a structure is built its physical behaviour often speaks for itself through direct observation or instrumentation. Software's invisibility means that the programmer must play the dual role of both designer and experimental scientist, continually probing and refining the intangible system to ensure it meets its requirements.

Furthermore, the construct of the software is not only invisible when static; its dynamics when in operation are as well. A program's true nature lies in its execution, which can vary with different inputs and conditions. The duality of static code and dynamic behaviour is at the heart of what makes programming unique. The source code is a static textual artefact (comparable to a blueprint or a music sheet), but when executed it produces behaviour in time, somewhat comparable to a functioning machine or a performed song. Yet the analogy breaks down immediately. An architect can envision the completed building and the skilled musician can envision the recorded or performed music, while for a programmer there is nothing as tangible to imagine. This fusion of the static and dynamic means that a programmer's work product is simultaneously a description of a process and the process

itself instantiates from that description. The correspondence between the two is exact. The running program does precisely what the code says (unless the underlying compiler, systems software or hardware are broken). Yet, comprehending that correspondence is nontrivial. This makes debugging an intellectual task rather than a checklist activity. Every bug is a case of a static mistake causing a potentially large dynamic misbehaviour. And unlike in mathematics, where a proof can establish with certainty that a theorem follows from axioms, in programming there is seldom a feasible formal proof that a program's behaviour matches the desired intent. As a result, the validation of software leans heavily on empirical testing and iterative refinement. Already in 1969, Dijkstra stated that program testing can be used to show the presence of bugs, but never to show their absence (Dijkstra, 1970). This highlights the hard truth: no amount of testing can conclusively guarantee correctness. It can only increase our confidence by failing to find counterexamples. In practice, developers acquire a sense of correctness by exercising software under many scenarios and edge cases, writing unit tests and integration tests as probes into the program's behaviour. They simulate special conditions, explore corner cases, and use runtime assertions, in effect performing *experiments* on the software. Through this process, knowledge about the software's reliability is built inductively rather than deductively. This is more akin to the empirical approach of experimental sciences rather than the certainty of formal logic. One can perhaps say that each program is a hypothesis about how to achieve certain outcomes, and testing is the experiment that probes that hypothesis. When tests are passed, the hypothesis survives; when a test fails, the programmer must study the results, refine the approach, and try again. Just as a craftsman might test the integrity of a physical piece by applying pressure in various ways, a programmer tests a module by running it through various inputs and states. But where the craftsman's feedback is immediate and tactile, the programmer's feedback is abstract, such as numbers on a screen, log messages, or user bug reports, requiring interpretation and careful reasoning. This underscores that the knowledge embodied in a software system resides not only in its code but also in the mind of the programmer. It is a personal mental model of how the program works, refined through continual experimentation and learning. The programmer builds a particular kind of theory in their head about the program's behaviour (Naur,

1985). Expert developers often internalise techniques that allow them to write simpler, more adaptable code, yet they may find it hard to articulate exactly what they do differently. This is tacit knowledge in the sense of Polanyi's "we know more than we can tell" (Polanyi, 1966).

While engineering metaphors highlight the practical construction aspects of programming, comparing programming to pure mathematics leads to other discrepancies. Computer science theory provides a foundation for algorithms and formal languages, and one might think of programming as just applied mathematics or formal logic. After all, programs execute logical operations and many foundational results in computing, such as algorithm correctness proofs or formal grammars, are mathematical in nature. And many theoretical computer scientists do all in their power to convince the world that "true" computer science is a branch of mathematics. But in practice, programming diverges from mathematics in several ways. Mathematics deals in eternal truths and proofs that, once verified, remain valid for all time while programming deals in contingent solutions that must work in a specific context, under changing requirements and environments. A mathematical proof, once accepted, does not need retesting while a program, on the other hand, can pass all tests today and fail tomorrow if its operating environment or inputs change. Moreover, the specifications that programs implement are often written in natural language or informal diagrams, not in the strict language of axioms and inference rules. There is always ambiguity and interpretation involved in translating requirements into code. Even when formal specifications exist, the complexity of real software often makes full formal verification infeasible. This does not make programming a lesser intellectual task, merely a different one. Formal program proofs have limited practical value unless they become part of a *social* process of validation; otherwise, a proof can be as inscrutable as the code itself (DeMillo et al., 1979). In reality, few developers attempt to prove programs correct with respect to comprehensive formal specs; instead, they use a mix of informal reasoning, peer review, and testing to gain confidence in correctness. Thus, the "truth" of a program is not an abstract certainty but a pragmatic consensus built through evidence. Furthermore, mathematics strives for minimal, elegant solutions, whereas in programming there are often many accepta-

ble solutions with different trade-offs in memory space, execution time and maintainability, the latter often being the most important. For example, one sorting algorithm might be provably optimal in asymptotic complexity, yet a less optimal algorithm could be preferable in a given software context due to ease of implementation or better real-world performance on typical data. Such decisions are not purely mathematical; they involve judgement about context and priorities, again reflecting the craftsman's mindset. In sum, while programming demands logical thinking and occasionally employs formal methods, it is not reducible to mathematics. The programmer's task is less like proving a theorem and more like solving a puzzle that has multiple possible solutions, and then continuously adjusting that solution as the problem itself evolves over time.

The unavoidable complexity of real-world software is another reason programming cannot be reduced to rigid formulas. Software systems are inherently more complex than any other human artefact of similar size, and that complexity is an essential property of software, not an accidental one (Brooks, 1987). Unlike physical machinery, where many parts might be identical or standardised, in software almost no two parts are alike at the detail level. A large program may have an astronomically large number of possible states and execution paths, far beyond what any individual can comprehend. As systems grow, the interactions and edge cases grow combinatorially. The craft of programming thus focuses on taming this complexity by finding ways to organise and reduce it so that our minds can manage the remainder. Dealing with complexity is the most difficult challenge in software design. Good programmers develop heuristic techniques and design principles to keep complexity at bay by dividing systems into modules, establishing clear interfaces, limiting interdependencies, refactoring out duplication, and so on (Parnas, 1972). Each of these techniques is less a scientific law than a craft guideline, a distillation of experience about what tends to work. For example, Parnas' principle of information hiding (1972) advised designers to encapsulate details likely to change behind stable module interfaces, thereby reducing the cognitive burden on anyone using or modifying that module. This was not derived from a theorem but from insight and experience in managing change in complex systems. Likewise, the use of design patterns

in software (recurring solutions to common design problems) arose from practitioners noticing how certain arrangements of classes and objects repeatedly proved effective (Gamma et al., 1994). These patterns became part of the shared experiences of the craft, a way to transfer hard-won knowledge without needing formal proof. Where classical engineering relies on laws of nature and well-understood material properties, software design relies on a growing set of patterns, practices, and principles that skilled programmers learn and apply. This "oral tradition" of software engineering, often passed on in code reviews, wikis, and conference talks, is evidence of its craft nature: knowledge is often transmitted through examples, analogies, and narratives as much as through specifications. At the same time, it highlights why you cannot learn the craft at a university alone. Despite more than 50 years of programming[1], there has been surprisingly little conversation about how to design programs or what good programs should look like, with the core issues of software design remaining largely absent in formal education. Instead, much design wisdom lives in the minds of veteran developers or is scattered across books. The large variation in productivity and quality among programmers attests to this fact. One of the earliest empirical studies found as much as a 10-to-1 difference in performance between individual programmers solving the same problem, underlining the huge influence of personal skill (Sackman et al., 1968). This gave rise to the "10 X" myth still alive to this day. While true under very specific circumstances, it does not mean that certain individuals are ten times as valuable to a software project. Such individuals often lack other capabilities, such as writing maintainable code or documenting. In fact, the most useful programmer is the "0.1 X" individual, i.e. the one that makes one-tenth the mistakes of his or her peers. Expert developers (not the ten-Xers) often internalise techniques that allow them to write cleaner, more adaptable code, yet they may not be able to fully explain what they do differently. This is because a lot of their know-how is tacit and experience-based. This is why novice programmers typically must undergo a fairly long apprenticeship of trial and error to acquire mastery, a defining characteristic of a true craft. However, the making of good programmers, indeed 0.1X-ers, can go much faster by adopting the guidelines of Chapter 4.

[1] Counting generously: since the first von Neumann-architecture machine, the Manchester Mark I in 1949.

Another distinctive aspect of software is its malleability over time. Physical products like bridges or engines undergo wear and tear and eventually fail, whereas software does not rust or crack. If untouched, it can theoretically run forever. However, software does age in a different sense. As user needs evolve and other system components change, software must be modified, and those modifications can introduce new problems. Lehman's laws of software evolution says that a successful software system is in continual change and growth, and as it evolves its complexity tends to increase unless proactive efforts (like refactoring) are made to reduce it (Lehman, 1980). This is sometimes called software entropy or software rot, as degeneration accumulates over time and gradually disorders the structure. In essence, while engineers face metal fatigue, programmers instead face design decay. Managing this entropy is typical of craft-type knowledge. It requires understanding not only the code in isolation but also its context and history of changes, forming what Naur (1985) calls the evolving "theory" of the program in the programmer's head. A classic example is the incremental and sometimes swift degradation of a codebase when quick fixes ("hacks") are applied to solve urgent issues without cleaning up the underlying design. Over time, such software becomes both brittle and difficult to extend. The remedy is not found in any physical repair, but in intellectual restructuring: rewriting or refactoring parts of the code to restore simplicity. That is why good programmers emphasise keeping code clean and continually improving its structure. The broken windows theory borrowed from urban sociology makes the same point (Wilson and Kelling, 1982). Leaving "broken windows" (messy, flawed code) unrepaired in a codebase tends to invite further neglect and deterioration, whereas a culture of constant small fixes prevents such rot (Hunt and Thomas, 1999). In traditional engineering, maintenance is often about repairing material faults. In software, maintenance is more about reconciling the program with new knowledge and requirements. The cost of not treating programming as a craft becomes evident here. If developers do not deeply understand and respect the design of a system (the way a craftsman knows their creation), ill-conceived and ill-applied maintenance changes can quickly erode the structure. On the other hand, a software system that is nurtured by its creators, with periodic redesigns and documentation of rationale, can remain robust and serviceable for decades. This longevity through care is another sign of software's crafty nature. The lack of such a long-term focus is most often not due to the programmers

themselves but rather comes from management decisions. Since the actual state of quality of code artefacts is hard to measure from the outside (i.e. from a management level), it is required of management to stay in close contact with the development teams in order to monitor the quality of codebases that likely are central to the organisation's future.

The cultural and professional milieu of programming further underscores its identity as a craft. Unlike civil or electrical engineering, software development has not historically required a license or formal degree to practice. Millions of programmers around the world enter the field through various paths. Some are formally trained in computer science, but many are self-taught or learn primarily on the job. In the absence of a regulatory framework, software developers exercise a great deal of autonomy in their daily work. A team of programmers might operate with looser oversight compared to engineers in other fields. There is no equivalent of the architectural stamping of blueprints or the mandated external review processes that, say, bridge designs undergo. Instead, the software industry has developed its own self-regulatory practices: code reviews by peers, open-source communities that enforce standards through collective scrutiny, and an emphasis on testing and continuous integration to catch problems early. These practices function similarly to the apprenticeships and peer critiques of traditional crafts. Within a healthy programming team, junior developers learn from senior ones by reviewing code together or pairing on tasks, much as an apprentice woodworker learns by observing a master's technique. Knowledge transfer is often informal and experience-based. One consequence of this informality is that the quality considerations fall heavily on the individual programmer's sense of professionalism. There is a growing recognition that programming autonomy must be coupled with responsibility. For example, both the ACM and IEEE have called for codes of ethics for software engineers, and there are periodic calls for stronger professionalisation of the field (Gotterbarn et al., 1999). Still, the consensus in the community tends to favour approaches that elevate craft competence over bureaucratic control. McBreen has argued that software development should embrace a model of software craftsmanship rather than imitate traditional engineering, emphasizing personal skill, pride in workmanship, and the men-

torship of less experienced developers (McBreen, 2002). They emphasise the creative and human-centric aspects of coding, the fact that writing good code involves style, taste, and judgement that cannot be reduced to a set of checklists.

In the early days, influenced by engineering, many sought to impose rigorous, linear methodologies such as the waterfall model, where development proceeded through discrete phases (requirements, design, implementation, testing) with formal sign-offs at each stage. This approach mirrored the assembly-line discipline of manufacturing or construction. However, experience soon showed that the waterfall model rarely worked well for software development because of its intrinsic invisible complexity. It proved impossible to foresee all problems in a large system during a single upfront design phase. Important issues and insights only emerged during coding and integration, when one confronts the reality of the code. The initial design for a software system will inevitably have flaws that only will become apparent once implementation is underway, at the earliest. As a collective response, more iterative and flexible methodologies were developed, such as agile development. The very recent Agile Manifesto (Beck et al., 2001) explicitly puts individuals and interactions over processes and tools and favours responding to change over following a plan, clear acknowledgements that software development is an evolving, exploratory activity. Agile methods treat a project not as a fixed blueprint to execute, but as a living process that adapts through frequent feedback. Techniques like short development sprints (short time-boxed development cycles), continuous refactoring, and close customer/end-user collaboration all suggest that building software is as much a craft of discovery as an engineering execution. As a result, design and implementation co-evolve. Programmers refine the architecture as they implement features, adjusting to feedback from tests and users. This dynamic fits naturally with a craft perspective: a potter may start with a vision for a vase, but adjusts the shape continuously on the wheel; a programmer likewise refines the design as code is written, tested, and observed in action. Far from being a sign of weakness, this adaptability is a source of strength in software. It recognises that in an intangible medium governed by logic and complexity, foresight is limited and humility is warranted. The best programmers approach design with a willingness to reconsider, which is why

practices such as refactoring (improving code structure without changing its behaviour) are deeply ingrained in the culture. This is the kaizen (continuous improvement) mindset applied to code, a term borrowed from Japanese manufacturing but also suitable for programming. However, refactoring is a double-edged sword. It might become a perfect excuse for not doing things well in the first place, and it gambles on resources being available to afford to do the work twice. Further, with changes in both management and team members over time, refactoring ambitions tend to be forgotten or postponed, resulting in unnecessarily bad software quality. Instead, good practice is to fix any "broken windows" (small flaws) as soon as they are discovered to prevent a build-up of software entropy, reflecting an artisan's attention to small details to uphold overall quality. Thus, modern methodologies acknowledge that writing software is not a linear procedure of assembling prefabricated components, but an iterative craft of discovery, where feedback and revision are integral to converging on a successful solution.

Equally important are the tools of the trade. Every craft relies on quality tools, and programming is no exception. A software developer's basic toolkit can be compared to a woodworker's set of rules, saws, planes, and chisels, chosen carefully and used skilfully (Hunt and Thomas, 1999). Programmers make use of editors, compilers, debuggers, version control systems, build automation, testing frameworks, and many other tools to amplify their effectiveness. Such a basic set of tools is often mandated by the employer or other management structures. But beyond using basic tools, proficient developers often create or tailor tools to fit their needs. Writing small scripts to automate repetitive tasks, refining build systems, or developing new utilities for common problems is considered part of the craft. This self-made tooling is analogous to a craftsman forging a custom instrument for a specific job. It reflects an intimate understanding of the work and a desire to improve the process itself. In the software world, this has led to a culture of sharing tools and automation techniques. The ability to bend one's working environment to one's needs is a hallmark of craft mastery. The practitioner is not a passive user of tools, but an active shaper of them. The flexibility of the digital medium makes this possible and indeed commonplace. Many popular development tools, from text editors to testing frame-

works, started as one developer's side project, later evolving into broadly used solutions. Think of the Emacs editor as an early example (Stallman, 1985). Such organic growth of tooling underscores how the software craft is in constant evolution, driven by practitioners themselves rather than top-down mandates.

An important aspect of programming as a craft is that code must communicate to two audiences at the same time: the machine and the human. While the computer will faithfully execute any syntactically correct sequence of instructions, no matter how convoluted, the long-term success of software depends on its readability and clarity to other programmers (and likewise to the future self of the original author). This is why programming has an inherent literary dimension. Abelson and Sussman, in the world's arguably best programming course ever, MIT 6.001,[2] advised that *"instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do"* (Abelson and Sussman, 1985). In other words, code is also a medium of communication among people since it serves as documentation of intent. This also underlies Knuth's concept of literate programming (Knuth, 1992), where a program is seen as written as an explanatory essay intertwining code and prose, as well as the widespread emphasis on code style and clarity in professional practice. Unlike a mathematical proof which can function while being opaque as long as it is logically correct, a piece of software is far more valuable when it is clean and understandable because it will inevitably be modified or extended by others. Here again, the craft analogy holds: a finely crafted chair is not just sturdy; its joinery and structure are also evident and elegant, making it easier for another carpenter to repair or build upon it. When programmers refactor code, it is often to make its structure more transparent, i.e. to make the *invisible* more comprehensible, a goal shared with designers, another craft. This is a tacit acknowledgement that programs live in a social context where humans must trust and work with each other's code. The intellectual activity of programming thus includes an element of narrative and style. Just as authors develop a voice, programmers develop a coding style, and part of the mentorship in software teams is transferring conventions that make contributions coherent.

[2] The course, featuring Abelson and Sussman themselves, was filmed and distributed on VHS cassettes to other universities. The author watched the films, which showed another programmer-as-craft dimension, in 1987.

Code reviews often focus not just on correctness but on clarity and simplicity, values that have both a functional and an aesthetic character. We commonly judge code as "beautiful" or "ugly" based on qualities like simplicity, consistency, and expressiveness, reflecting shared cultural aesthetics in the programming community. These are qualities that cannot be measured easily, but experienced developers sense them, much as a seasoned chef can taste a sauce and know whether it needs more salt. The presence of these aesthetic and communicative aspects further establishes programming as a craft where the human element is central.

One of the most powerful demonstrations of programming-as-craft is the open-source software communities. Open-source projects like Linux, Apache, or Python have been built by globally distributed communities of programmers, often volunteers, without a rigid top-down management structure. The "cathedral and the bazaar" metaphor coined by Raymond (2001) contrasts two styles: the cathedral model of software built by a small, tightly controlled team (analogous to traditional corporate engineering) versus the bazaar model of software evolved in the open, with contributions from many independent individuals. The success of the bazaar approach (for example, Linux is now the backbone of servers and Android devices worldwide) suggests that software can grow through a highly decentralised, craft-driven process. In such projects, individual contributors exercise autonomy over small parts of the system and their contributions are integrated through a culture of peer review and continuous refinement. Linus Torvalds, the creator of Linux, likened this process to natural selection (Raymond, 2001). This observation, known as Linus' Law, implies that with many independent craftspeople examining and experimenting with the code, defects and design flaws will be discovered and corrected in an almost evolutionary fashion. The open-source model relies on personal initiative and pride. Developers contribute because they want to make something better or because the software directly serves their own needs. There is often no formal chief architect for many projects (Linux being an exception). Instead, design emerges from the collaboration of many minds, guided by shared principles and the curation of a few maintainers. This is very much the dynamic of a set of craftsmen working in a "bazaar", learning from each other and collectively producing a com-

plex artefact. It shows that the craft approach can scale (at least under the right conditions) to very large systems. At the same time, open-source communities enforce quality through social norms. For instance, insisting on coding standards, requiring descriptive commit messages, and sometimes rejecting patches that do not meet the project's standards of design or cleanliness. The fact that such communities can maintain coherence and produce high-quality software without formal hierarchy speaks to the power of treating programming as a collaborative craft. It reminds of the way scientific communities operate with peer review and replication, but here the "experiments" are software patches and the "replication" is others running and extending the code.

It is worth noting that the balance between rigorous engineering and freestyle craft in programming can vary with the application domain. In safety-critical software such as avionics, medical devices, or nuclear control systems, there is a much greater emphasis on upfront analysis, formal verification, and process. These domains borrow more heavily from classical engineering because the cost of failure is catastrophic. After all, the fact that an aeroplane's software was due for refactoring the next week would be of little consolation if the plane did not reach its destination safely. For instance, aviation software standards (like DO-178C) require evidence of exhaustive testing and traceability from requirements to code, and some aerospace software undergoes formal proofs of certain properties. In these contexts, the autonomy of the individual programmer is deliberately constrained by checklists, peer review, and quality assurance procedures. And yet, even in such fields, the final reliability often comes down to the expertise of individuals making wise choices. Leveson and Turner's (1993) analysis of the Therac-25 radiation therapy accidents, where software bugs in a medical machine led to patient deaths, revealed not just process failures but also a lack of software craftsmanship such as poor design decisions (like inadequate fail asserts and insufficient consideration of concurrency) and insufficient testing of complex interactions (see Chapter 3). The analysis showed that following a process is not enough without deep understanding. Similarly, NASA's Mars Climate Orbiter was lost in 1999 due to a simple software unit mismatch between teams (one used S.I. metric units, the other U.S. units). A failure to catch this discrepancy in testing caused the spacecraft to deviate and crash upon

arrival at Mars. The post-mortem blamed a breakdown in communication and verification, again demonstrating that even in a highly structured project, the small-scale craft of careful coding and checking was needed (also see Chapter 3). Conversely, in consumer internet software (say a social media app or e-commerce site), a "move fast and break things" culture might prevail, with the craft aspect being dominant and the formal processes minimal. Changes are deployed quickly and developers rely on monitoring and quick rollbacks if something goes wrong. This can work because the cost of failure is lower (a glitch might annoy users, not endanger lives) and because the human is very much in the loop. Still, as soon as such systems scale to millions of users or handle sensitive data, they too should incorporate more disciplined practices to manage complexity and risk, such as careful code review, automated testing pipelines, and gradual rollouts of new features. The conclusion is that programming involves aspects of both craft and engineering, but the mix shifts heavily with context. Recognising programming as an intellectual craft does not mean rejecting all structure. Rather, it means empowering practitioners to apply the right techniques at the right times. Craft-aware programmers know when to rely on intuition and quick experimentation and when to apply rigorous analysis or formal methods. In high-stakes situations, they apply their craft in controlled, systematic ways, much as an expert surgeon (another high-skill craftsperson) follows stringent protocols during an operation, yet still relies on experience and judgement for the unexpected. In more exploratory software projects, the same programmer might take bolder creative leaps. This adaptability is itself part of the craft. Knowing the context and constraints and adjusting one's approach accordingly.

Another insight from decades of experience is that adding more people to a software project does not linearly increase productivity; in fact, it might decrease. Brooks' Law, formed in the context of developing IBM OS/360, states that "*adding manpower to a late software project makes it later*" (Brooks, 1975). The rationale behind this is that software tasks are not easily divisible. Work done by one developer cannot simply be partitioned among five developers without significant communication and coordination overhead. In a factory assembly line, more workers can rather linearly produce more widgets, but in software, five programmers working

independently on interrelated parts may slow each other down unless carefully managed. This again emphasises the intellectual and conceptual nature of programming. Communication overhead grows nonlinearly with team size (each pair of developers potentially needs to stay in sync), and the difficulty of integrating many pieces of code can rise sharply. Brooks advocated keeping the teams small and fostering conceptual integrity, which is the idea that a system's design should be unified, ideally by the vision of one or a few minds (Brooks, 1975). This is akin to a work of art that bears the stylistic coherence of its creator. In practice, large successful systems often achieve conceptual integrity by partitioning into components that are owned by small sub-teams, each acting with a degree of autonomy (a bit like separate craft workshops contributing to a cathedral). The lesson is that building software is not a brute-force endeavour. It relies more on talent and tight-knit collaboration than on sheer numbers. This runs counter to a naïve industrial view but aligns with the craft perspective that what matters is the quality of the artisans and the communication among them. It also explains why small start-up teams can out-innovate larger competitors. The efficacy of a few skilled craftspeople with a clear vision can surpass a big group mired in coordination. Good managers of software projects, therefore, act less like foremen directing workers on an assembly line and more like facilitators who enable developers to do their best work. Protecting their time and energy, providing them with good tools, and ensuring that knowledge flows freely within the team. This management approach, sometimes called *servant leadership* in agile contexts, acknowledges that programming is a creative craft at its core and tries to set the conditions for craftsmanship to flourish rather than attempting to micromanage each technical detail. On a larger scale, online communities like Stack Overflow and countless developer forums serve as informal societies where programmers ask questions and disseminate tips and idioms, a modern incarnation of the apprenticeship model at a global level.

Recognising programming as a craft has practical implications for how we train software developers, not least at universities. It suggests that education should go beyond classroom lectures on algorithms and include more studio-like practice and apprenticeship. Just as a budding architect learns by designing and critiquing build-

ings under a mentor, student programmers learn deeply from actually writing programs and receiving feedback from experienced developers. A few universities have begun to incorporate more project-based learning to simulate such an apprenticeship model, but much learning still happens informally in internships and open-source contributions. This is not to say that the rather commonly employed pair programming concept in university settings constitutes a step in the right direction. Pair assignments of any kind and in any subject have mostly been introduced to save money in the hope that less teaching time and less grading effort can go into such assignments. This constitutes a miserable indifference to the vastly different learning styles students have, and can in no way substitute the learning in teams also containing at least one more experienced member.

The craft perspective validates hands-on learning pathways. It also emphasises mentorship within organisations: knowledge transfer in software is often best achieved by code review sessions and informal design discussions, rather than solely by documents. Companies that foster a culture of craftsmanship, for example by encouraging engineers to refactor code when needed, share best practices, and devote time to technical excellence, often produce more maintainable, resilient systems in the long run. Conversely, organisations that treat developers as interchangeable resources and focus only on the processes can stifle the creative problem-solving needed to tackle complex software problems. The craft approach also highlights the importance of diversity in perspectives: a team with varied experiences can bring a richer set of heuristics and ideas to a problem, much as a guild benefits from masters of different specialties. By viewing developers as craftspeople, we encourage pride and ownership in the code they produce, which can lead to higher quality. A programmer who identifies as a craftsperson is likely to care deeply about the product's integrity and the impact on users, rather than just meeting minimum requirements. In an era when software increasingly mediates society's critical functions, from healthcare to transportation and finance, nurturing this sense of responsible craftsmanship is not just an aesthetic preference but a societal need.

Today (2023)

A current[3] development that is testing the boundaries of programming-as-craft is the rise of AI-assisted coding tools (such as GitHub's Copilot from 2022 or the more general OpenAI ChatGPT 3.5 from the same year). These tools can automatically generate code snippets or even entire functions based on natural language prompts or context from the developer's current file. This has raised questions whether the automation will eventually reduce the need for human programmers or transform programming into a higher-level supervisory activity. In practice, however, these AI tools hitherto serve more as very junior assistants than replacements. They can today handle boilerplates and suggest solutions to routine sub-problems, but they do not "understand" the overarching requirements or the subtle trade-offs and constraints of a particular project.[4] It still takes a skilled programmer to judge whether a generated piece of code fits correctly, to test it, and to integrate it with the system's architecture, assuming that it was even locally correct in the first place.[5] Therefore, the advent of such tools actually further highlights the craft aspect. The human programmer takes on an even more design-oriented role, curating and refining the output of automated chatbot power. Using AI assistance effectively will become a new skill in the programmer's toolbox, analogous to an experienced craftsman using power tools such as power drills and chainsaws. They significantly speed up the execution of certain subtasks, but the craftsperson must guide the tool and ensure quality. Early studies of Copilot usage seem to indicate that it can accelerate coding for those who know how to prompt it well and critically assess its suggestions, but it can also produce blatant errors and unsafe code if taken anywhere near face value (Pearce et al., 2022). Thus, rather than eliminating the need for craftsmanship, AI may eliminate some plain work while elevating the importance of human judgement, architecture, and creative problem-solving, the very areas where the craft of programming is most evident. It reaffirms that programming, at its core, remains an intellectually driven

[3] In 2023, when this text now receive some renewed attention after having spent many years in a "drawer".
[4] By "understand" is not meant any conscious human activity, but rather the ability to adapt to a larger context.
[5] At the time of writing (2023), many AI-suggested code snippets do not make much sense or are not even syntactically correct. This will surely improve rapidly, and the only wise thing to do is not to make any predictions on the extent or quality of coding assistance provided by such tools in, say, five years' time from now.

activity requiring insight and discernment that cannot be fully automated away.

Ultimately, proposing that programming is an intellectual craft is not to downplay the scientific or engineering facets, but to elevate the importance of human skill, creativity, and responsibility in software. It points out that great software is made by great programmers, not by managers, sales departments, processes or tools alone. By recognising the craft element, we encourage continuous learning and a community of practice where knowledge is shared. The challenges of the future, ranging from artificial intelligence to global-scale complexity, will demand programmers who are not just coders but true craftspeople of logic, able to marry theory with practice in inventive ways. In conclusion, programming is still an endeavour that is intensely intellectual yet deeply practical, constrained by logic yet propelled by creativity. This is not changed by the introduction of AI/LLM tools. Likewise, the need for good and consistent coding guidelines and rules, like those of Philips PTS presented in Chapter 4, still prevails. The human in the loop is still the same old human, and if nothing else, the tools can help pruning the expression space a bit while maintaining the creative craft components, even enhancing them.

3. Software Bugs

Software has become the invisible infrastructure of modern civilization, and its reliability hinges on a paradox: while elegant algorithms and sound architectures are necessary, even the simplest implementation mistake can bring the entire system down. In other engineering disciplines, catastrophic failures often trace back to complex causes or unforeseeable external forces. In software engineering, by contrast, the cause of a disaster is often embarrassingly trivial: a missing minus sign, a mistyped unit, or an off-by-one loop index. These minuscule lapses in attention to detail have repeatedly led to outsized consequences: space missions lost to unit conversion errors, airliners felled by a few lines of faulty code, or financial systems bankrupted by a single unchecked assumption. This pattern suggests that *error-free software is largely dependent on attention to detail* at the coding and testing stages. The best algorithms and architectures cannot compensate for sloppy implementation. Most software-related losses, whether measured in human lives, money, or time, stem from small mistakes that could have been prevented by rigorous "software hygiene", i.e. the disciplined application of best practices and quality checks at every step. This chapter examines that unsettling truth, surveying high-profile software failures attributable to tiny errors, assessing the cumulative toll of everyday bugs, and arguing that a diligent commitment to detail and discipline is needed in software development. Throughout, the discussion is academic in tone yet also reflective and critical, calling for a cultural shift in how we approach software quality. The analysis is informed by both industry insights and academic research (including principles from professional coding standards) to make the case that our biggest enemy is not algorithmic complexity per se, but the simple preventable mistakes we persistently allow to slip through.

Trivial Bugs, Epic Failures

In July 1962, NASA's Mariner I spacecraft veered off course shortly after launch and had to be destroyed, a failure later attributed to a single omitted hyphen in the guidance computer's code. This early cautionary tale prefigured a long history of software failures caused by seemingly tiny mistakes. Over decades, countless missions and systems have been compromised not by fundamental design flaws, but by

lapses so small that one is tempted to label them "bugs" in the original sense of the word. A "bug" implies something gnat-sized, a grain of sand in the eye, a tiny annoyance. Yet in software, such a gnat can cause a hurricane. The following examples illustrate how often the downfall of complex systems has boiled down to a few lines of code or one oversight that better attention to detail would have caught.

Therac-25 (1985–1987) – Concurrency Bug with Lethal Consequences

Not all small software mistakes "only" cost money; some cost lives in the most direct sense. A chilling illustration is the Therac-25 radiation therapy machine incidents in the mid-1980s. The Therac-25, a computer-controlled medical linear accelerator, delivered massive radiation overdoses to patients on at least six occasions, causing severe injuries and three confirmed deaths, essentially frying people. Investigations revealed that the machine's control software had a race condition, a subtle timing bug in which a particular sequence of fast user inputs could bypass safety checks and allow the machine to enter an improper state. Essentially, a tiny synchronization flaw (a shared variable not protected properly or a flag reset too late) was the culprit. This was not a hardware failure or an operator error; it was a relatively simple programming oversight in handling concurrent operations. The Therac-25 software was written in assembly language and reused code from earlier models. Unfortunately, it lacked the interlocks and failsafe-guards of its predecessors, relying purely on software for safety. The bug remained latent for a long time, a testament to how such issues can lurk unnoticed until rare conditions align and trigger them. When they did, the software would erroneously disable the safety beam monitor under certain conditions, allowing the machine to fire its high-powered beam without proper moderation. This is a dramatic example of a "small" bug (just a few lines of problematic code) having deadly results. It emphasises that in safety-critical systems, *no detail is too minor to double-check.* The Therac-25 accidents led to a complete rethinking of software quality assurance in medical devices. They underline how an accurate assembly of software, thoroughly reviewing and testing even the smallest concurrency scenarios, is sometimes literally a matter of life and death. Most importantly, Therac-25 taught the software community that "trusted" code (carried over from previous versions) must still be rigorously inspected in its new

context, and that any assumption that a certain combination of events "cannot happen" should be challenged with thorough attention to detail.

Saab JAS 39 Gripen (1989 and 1993) – Unstable Control from a Software Bug

The JAS 39 Gripen, a Swedish multi-role fighter, offers another example from aviation where small software mistakes had dramatic consequences. During the development of this fly-by-wire fighter jet, two early crashes (one in 1989 at the manufacturer's airstrip and another in 1993 over the capital) were attributed to instabilities in the flight control software. In the 1993 incident, a widely televised crash during an air show over Stockholm city, the test pilot lost control at low altitude due to a *pilot-induced oscillation*, exacerbated by a flaw in the control software. This was the fighter jet's premier in public, which ended as it went down on Långholmen island, the former prison island in the middle of Stockholm city. The author watched this incident live. Specifically, an investigation found "high amplification" of the pilot's quick stick inputs by the flight control system, effectively a feedback loop that grew out of control. In simpler terms, the software's response to large control inputs was not properly damped; it amplified rather than mitigated the pilot's overcorrection. This was traced to a minor mis-tuning or algorithmic oversight in the control software. Essentially a few parameters or lines of code determined how the aircraft responded to rapid stick movements. Once identified, the problem was corrected with a software update and the Gripen program proceeded with additional stability modifications. As with other cases, the error itself was trivial in nature (a control gain issue in a feedback loop). It did not require reinventing the physics of flight; it only required adjusting the software to avoid reinforcing the pilot's inputs so aggressively. Yet until it was fixed, this subtle bug proved dangerous, causing the loss of expensive prototypes and risking lives. The Gripen crashes underscore how cumulative small errors in real-time software (here, the exact values in control algorithms) can tip a system from stable to unstable. Fly-by-wire aircraft rely wholly on software mediation between pilot and plane, so a tiny mistake in that mediation can nullify the plane's engineered stability. This is why modern aerospace software development is notoriously strict about validation. Every coefficient and line in the flight control code must be scrutinised. In Gripen's case, one can view the initial

crashes as a costly but perhaps somewhat constructive lesson in getting those details right. The subsequent success of Gripen (once the bug was fixed and the software matured) highlights that when attention to detail is restored, the system can perform excellently. It also reinforces that pilot training or hardware quality cannot compensate for software mistakes in such systems. The only solution was to fix the code itself through more careful analysis and testing, the kind of painstaking work that should have prevented the bug in the first place.

Ariane 5 Flight 501 (1996) – Overflow and Self-Destruct

When the European Space Agency's Ariane 5 rocket had its maiden flight in June 1996, the launch ended in an explosion barely 37 seconds after lift-off. The cause was eventually traced to a bug in the rocket's inertial navigation software: a data conversion from a 64-bit floating point number to a 16-bit integer overflowed the smaller variable, an overflow error that had not been anticipated. This caused the primary inertial reference system to crash. Unfortunately, the redundant backup system was running the same flawed software, so it crashed too, both systems having effectively "failed on the same bug." With no valid attitude data, the rocket veered off course and triggered its automatic self-destruct. The root of this bug was *software reuse without sufficient re-evaluation*. The software module that overflowed had been carried over from the Ariane 4, in which the particular variable's value never grew large enough to overflow. But Ariane 5's faster trajectory meant the same calculation quickly exceeded the 16-bit range. In essence, an overflow error, one of the most mundane programming mistakes, destroyed a $370 million launch and its satellite payload. The subsequent inquiry noted that the code did not handle the exception because it was assumed to never occur, an assumption invalidated by the new context. This case underscores a crucial point: small mistakes often hide in assumptions. A simple range check or exception catch could have averted the overflow's effects. Indeed, had the developers foreseen the possibility of this one variable exceeding its limit and added a few lines of defensive code or even just disabled that module during the critical launch phase, the mission would have proceeded normally. Ariane 5's failure became a textbook example in engineering schools precisely because it was caused by such a pedestrian error. The lesson, as observed in

an analysis, was that the quality of a device's software must be considered in the context of the entire system. In other words, rigorous attention to detail is needed especially when reusing code. One cannot blindly trust legacy software in a new environment without re-checking all the little assumptions baked into it.

Mars Climate Orbiter (1999) – Metric vs. Imperial

In September 1999, NASA's $125-million Mars Climate Orbiter disintegrated in the Martian atmosphere because of a unit conversion error that went undetected. One engineering team had provided thruster impulse data in pound-seconds while another team's navigation software expected newton-seconds. The mismatch meant that navigation calculations were off by a factor of 4.45. Yes, 445%, not 4.45%. Over the long interplanetary journey, this small discrepancy accumulated until the Orbiter's trajectory was fatally off. A post-mortem by NASA described it as a "simple error" in unit conversion that was not caught due to insufficient systems engineering checks. The director of JPL remarked that their "inability to recognise and correct this simple error has had major implications". In other words, no exotic hardware failure or esoteric algorithm was to blame. The spacecraft was essentially "lost in translation", an undeniably trivial oversight that any attentive review could have caught. Outside observers were blunt in their assessment. A space policy expert said of the loss: "That is so dumb […] There seems to have emerged […] a systematic problem in the space community of insufficient attention to detail." The Mars Climate Orbiter fiasco dramatically showed how a single unchecked calculation (a mundane piece of code) undermined an entire mission. It was a stark reminder that even at NASA, with its highly skilled teams, lapses in basic diligence (failing to double-check units and interfaces) can annihilate vast investments of time and money.

Mars Polar Lander (1999) – A Premature Shutdown

Just a few months later in 1999, NASA lost the Mars Polar Lander, a companion mission, in a similarly frustrating manner. The lander's descent software likely mistook the jolt of the landing legs deploying for a touchdown signal and cut off the engines while the probe was still high above the surface. In essence, a false sensor

reading triggered by a routine event (leg deployment) was not properly filtered out by the software. The code interpreted a transient spike as confirmation of landing and shut down the thrusters, causing the lander to free-fall and crash. Post-incident analyses concluded that an "inadequate software" logic caused the premature engine cut-off. Again, the underlying mistake was trivial in principle: a simple logic check or sensor validation could have prevented the engines from stopping too early. But because this small case was overlooked, arguably an omission in requirements or a missed detail in coding, NASA's lander was destroyed. The Mars Polar Lander and Climate Orbiter failures, coming back-to-back, forced NASA to confront the reality that its faster, cheaper mission approach had skimped on the "hygiene" of rigorous testing and peer reviews. The problem was not that humans made an error (that's inevitable), but that the processes and checks failed to catch it. These Mars mission failures highlight how often a trivial slip. A single conversion or a single sensor event can cascade into mission failure if not diligently caught and corrected.

Boeing 737 MAX (2018–2019) – Few Lines of Code, Catastrophic Outcome

Modern aviation is heavily reliant on software, and the Boeing 737 MAX disasters tragically demonstrated how a small coding logic and oversight can bring down state-of-the-art aircraft. Boeing's 737 MAX was equipped with an automated system called MCAS (Maneuvering Characteristics Augmentation System), introduced to adjust the aircraft's pitch under certain conditions due to the plane's redesigned engine placement. The MCAS logic, in essence, consisted of only a few lines of code inside the flight control computer, but it was implemented in a way that relied on a single Angle of Attack (AoA) sensor input. If that lone sensor gave a faulty high reading (as happened on two flights), MCAS would repeatedly push the airplane's nose down, mistakenly "thinking" the plane was stalling. In October 2018 and March 2019, two new 737 MAX aircraft (Lion Air Flight 610 and Ethiopian Airlines Flight 302) crashed, killing 346 people in total. Investigations pinpointed that erroneous data from a failed AoA sensor had activated the MCAS software in both cases, repeatedly trimming the aircraft into an unrecoverable dive. From a software perspective, what is astounding is how minor the implementation of MCAS was: one report noted that the entire MCAS control law was a few lines of code that could

command nose-down trim when triggered by a single sensor failure. In other words, a tiny fragment of the millions of lines on the aircraft became the single point of failure. Boeing's design did not include adequate cross-checks or redundancy for this system. A mere *eight lines of defensive code* could have prevented the disaster, according to one software expert's recent analysis (Hamblen, 2023). Those eight lines (which would compare the readings of the two AoA sensors and disable MCAS if they disagreed beyond a certain threshold) were not in the original implementation which relied on a single sensor not malfunctioning. Such a statement illustrates how extremely small omissions, a few lines not included, can lead directly to catastrophe. The 737 MAX case is especially instructive because it shows that even in a highly regulated, safety-critical industry like aerospace, lapses in software diligence can creep in under competitive and schedule pressures. Boeing initially classified MCAS as a non-critical change to avoid extensive retraining for pilots, which led to it being developed without the utmost rigor a critical system would merit. The outcome was essentially a hidden "trapdoor" in the plane's behaviour. Once again, a chain of minor failures contributed: a faulty sensor, an omitted warning light (another detail that Boeing chose to sell as an optional feature), and pilots not informed of the system's existence all played a role. But at the core was the software's small logic mistake of trusting one sensor implicitly. In the aftermath, Boeing and regulators fixed the software to take input from two sensors and limit MCAS's authority, effectively implementing those few lines of code that were missing. The lesson is that in complex systems, even a short snippet of poorly conceived code can overpower all the sophisticated engineering around it. The 737 MAX crashes became a case study in how not to handle software engineering: they demonstrated the need for thorough unit testing, scenario analysis, and failure-mode consideration for even minor code changes.

Financial Systems and Other Domains – Small Errors, Massive Losses

The sphere of mission- and safety-critical systems is not the only one riddled with examples of little bugs causing big trouble. In finance and business, where software glitches may not kill people but can wipe out fortunes, the story repeats itself. A notorious example is the Knight Capital Group incident of 2012, often cited as one

of the most expensive software bugs in history. Knight Capital, a major financial trading firm, deployed new code for its high-speed trading algorithms that, due to a simple error, accidentally triggered an obsolete function left over from old software. In essence, a flag in the code was mis-set, or an old debug routine was not removed, a very basic version control or deployment oversight. This caused the trading system to enter an erratic loop of buying and selling millions of shares in minutes. In 45 minutes, Knight Capital accumulated about $440 million in losses and was driven to the brink of bankruptcy. It turned out that just one defect in the code was responsible. An internal review later revealed that deploying the new code to some of the servers but not others (an operational oversight) allowed an old unused piece of code, ironically called the Power Peg, to activate on those servers and flood the market with erroneous orders. The root cause was traced to a lack of testing. In addition, the rollout procedures did not catch that one server was left running the old code and the software itself lacked safeguards to prevent such high-volume unintended trades. As analysts noted, this bug was not an intricate mathematical flaw. It was a lapse in simple software hygiene, failing to remove dead code and failing to uniformly update all modules. The lesson from Knight Capital is straightforward: when dealing with high-stakes software (in this case, controlling real money), even one unchecked bug can bankrupt a company. The "glitch", as it was euphemistically called, was entirely preventable through more careful testing and deployment protocols. Industry observers remarked that Knight's fiasco, happening to a highly regulated financial firm, proved that no amount of external audits or compliance can save you if your internal software engineering practices are too sloppy. The fix for Knight's bug was presumably as simple as removing or disabling a few lines of old code, a trivial change that would have averted a multi-hundred-million-dollar loss. Similar scenarios have played out in other sectors. For instance, a 2012 software bug in the NASDAQ exchange's IPO system led to chaos during Facebook's initial public offering, causing tens of millions in losses due to order processing errors. Or consider the AT&T long-distance outage of 1990, where one mistyped line of code in a single switching centre's software update cascaded through the network and brought down AT&T's long-distance phone service nationwide for nine hours. The culprit was a `break` statement in C code placed incorrectly inside an `if` block; this tiny mistake prevented the error-handling code from working and triggered switch

resets that propagated through the network. It cost AT&T an estimated $60 million and huge reputational damage, all for want of a careful code review on one line. These incidents, though in different domains, echo the same refrain: most failures are not due to the fundamental impossibility of building correct software, but due to known, small failure modes that were simply not guarded against. They reinforce the argument that the vast majority of the real work in delivering error-free software lies in the accurate assembly of the system through programming and testing, rather than in the conceptual brilliance of the design. When that assembly process lacks diligence, the smallest imperfection can spoil the result.

Accumulated Toll of "Small" Errors

For every headline-grabbing disaster caused by a software bug, there are thousands of smaller bugs quietly exacting a toll on productivity, profitability, and user satisfaction. Indeed, while the dramatic failures discussed above illustrate the principle in extreme form, the claim that "most software-related losses come from small mistakes" is borne out by industry statistics and studies of software quality. These studies reveal a staggering cumulative impact from the multitude of minor defects that slip into everyday software. Individually, a typo in code might cost an hour of debugging; a missed null-pointer check might cause a minor service outage. But collectively, such issues can cost billions of dollars and untold hours of lost time. This section examines the evidence for the scope of these losses and how they overwhelmingly trace back to preventable mistakes in implementation.

Economic Costs

The cost of poor software quality has been quantified in numerous analyses, and the numbers are eye-opening. A study commissioned by the U.S. National Institute of Standards and Technology (NIST) in 2002 estimated that software bugs were costing the U.S. economy approximately $59.5 billion per year at that time. Notably, the study concluded that over a third of that cost (around $22 billion) could be eliminated by improved testing and earlier detection of defects. In other words, billions were being lost essentially because of a lack of attention to detail early in the software lifecycle, allowing small bugs to persist into production where they became

much more expensive to fix. Fast-forward two decades, and the costs have only skyrocketed with the growing scale of software systems. The Consortium for IT Software Quality (CISQ, 2020) publishes regular reports on "The Cost of Poor Software Quality" in the U.S. Their report estimated that poor software quality cost the U.S. around $2.08 trillion in 2020 alone. By 2022, this figure had grown to $2.41 trillion annually. These figures encompass various sources of loss such as failed IT projects, legacy system problems, operational failures, and cybersecurity breaches, with the common thread that much of it stems from avoidable software defects. For example, of the 2020 cost, the largest component (about $1.56 trillion) was attributed to operational software failures, which include downtime, outages, and security incidents caused by software malfunctions. Such failures often have proximate causes like unhandled exceptions, memory leaks, off-by-one errors leading to crashes, etc. The mundane bugs of day-to-day programming. The CISQ analysis emphasises that these are not "Acts of God" but consequences of poor-quality code and insufficient testing. The fact that the figure is in the trillions suggests that across all industries, the aggregate effect of small software faults is enormous. It dwarfs the losses from headline disasters. To put it in perspective, $2 trillion is about 9% of US GDP or three times the Swedish, an economic drag attributable purely to suboptimal software. If even a fraction of those trillions could be saved by catching bugs earlier or preventing them, the impact would be measured in hundreds of billions of dollars.

Developer Time and Productivity

Another way to measure the toll of software bugs is in the time and effort developers must spend to find and fix them. Numerous studies in software engineering have found that a significant portion of the development cycle is consumed by debugging and rework. One widely cited statistic is that programmers spend anywhere from 35% to 50% of their time not writing new code, but rather debugging and validating existing code. O'Dell (2017) notes in *ACM Queue* that testing, debugging, and verification activities often account for 50–75% of a software project's total budget. This is an astonishing figure: it implies that for every dollar spent on developing software, up to three-quarters might be spent on detecting and correcting mistakes that were introduced along the way. While some level of verification is unavoidable,

the implication is that *a huge amount of effort is essentially waste that results from preventable defects*. If initial coding were flawless or at least much more reliable, the cost and time spent on debugging could be dramatically reduced, accelerating project schedules and reducing expenses. There is also the oft-quoted aphorism: "Debugging is twice as hard as writing a program in the first place. So if you write the program as cleverly as possible, you are, by definition, not smart enough to debug it." (Kernighan and Plaugher, 1974). This humorous observation carries a serious point: clever, complex code invites bugs that are hard to find. Thus, simple, clear, and carefully written code, a result made possible by disciplined, detail-oriented development, makes for easier debugging and maintenance. Too often, however, developers do not pay that rigorous attention at the outset, and the result is a tedious hunt for bugs later. The cost difference between finding a bug during requirements or coding and finding it in production can be enormous. A rule of thumb in software engineering (stemming from early work by Boehm and others) is that a bug fixed during the design phase might cost 10 times less than if fixed during coding, and a bug fixed in coding costs perhaps 10 times less than if found after release. Multiply these factors across hundreds of bugs, and the savings from better initial diligence become clear. Poor "software hygiene" effectively taxes the developers by forcing them into long debugging sessions. In the US alone, it is estimated that over $100B is spent annually on identifying and fixing product defects and that the average developer spends up to 75% of their time on debugging in some capacity. Even a more conservative figure of around 50% would indicate an industry-wide inefficiency of sorts. If half of the developer time is spent on avoidable rework, then any practice that can reduce bug introduction (and thus debugging) can effectively double developer productivity in terms of feature delivery. This is precisely why many organisations invest in test-driven development and continuous integration testing: to catch mistakes earlier when they are quicker to fix. The data overwhelmingly supports the argument that small bugs collectively steal vast amounts of time.

Compounding Technical Debt.

Software bugs and quality shortcuts accumulate over time into what is often termed *technical debt*. This metaphor characterises suboptimal code and known defects as

a debt that the development team owes. One that incurs "interest" in the form of extra effort to work around the issues or fix them later under less optimal circumstances. Just as minor financial debts can spiral with interest, minor software issues can accumulate and interact in ways that greatly increase the difficulty of making changes or ensuring reliability. The CISQ 2022 report estimated that the "technical debt" in the U.S. (meaning the future cost to fix issues left in code) was about $1.52 trillion and growing (CISQ, 2022). Technical debt often consists of those "little" things developers knew were not ideal. A quick hack here, a TODO left there, a set of edge cases not thoroughly tested. Be honest, you have seen it many times. Initially they might not cause a system to fail, but over time they create brittleness. Eventually, some small change will cause the system to collapse under the weight of all those neglected details. High-profile outages in industry often reflect this: a system that worked for years suddenly breaks when a slight increase in load or a new configuration exposes one of these latent bugs that have been lurking. In many cases, each individual issue was minor (and perhaps management consciously deferred fixing it, thinking it was a low priority), but aggregated they pose a systemic risk. The key insight is that quality is not simply a feature that can be added later; it is the result of continuously keeping the codebase clean and robust. The small mistakes that are left "for now" tend to become tomorrow's big production incident. Thus, discipline in fixing even trivial bugs and polishing rough edges is part of reducing technical debt and thereby avoiding exponential costs later.

User Impact and Trust

Another intangible but significant cost of software bugs is the erosion of user trust and the opportunity cost of poor quality. If a word processor crashes and loses a document due to some small memory management bug, or if an e-commerce site suffers a glitch that prevents checkouts, users may lose confidence in the product. While this may not be as quantifiable as direct losses, it translates to brand damage and customer attrition. Security bugs, often literally single-line mistakes such as buffer overruns or unchecked inputs, can lead to data breaches that destroy user trust irreparably. For example, the Heartbleed vulnerability in OpenSSL (disclosed in

2014) was caused by improper input validation. A missing bounds check in the handling of a TLS heartbeat message. This one mistake in C code allowed attackers to read out sensitive data from servers' memory, affecting an estimated two-thirds of websites and forcing emergency patching worldwide. While the direct costs of Heartbleed in terms of responding to the incident were substantial, the bigger cost was arguably the hit to public confidence in internet security. Here again, the root cause was simple: a length field provided by the user was not validated against the actual buffer size, a mistake any diligent code review should have caught. The fallout required days of work by thousands of sysadmins globally (another labour cost) and a reputational black eye for open-source security software. Cases like Heartbleed show that small coding mistakes can have a global impact, undermining fundamental trust in systems (in this case, the trust that your encrypted communication cannot be spied upon). In safety-critical software, the equivalent is a loss of public trust in technology, for instance the 737 MAX crashes, led to a worldwide grounding and a crisis of confidence in Boeing that the company is still trying to regain. The indirect economic impacts (lost sales, legal liability, etc.) stemming from such trust issues often dwarf the direct cost of fixing the bug.

The Takeaway

Whether measured in direct financial losses, wasted developer hours, or intangible reputation damage, the bulk of the cost of software failures arises from *preventable errors* introduced during implementation and insufficiently caught in testing. These errors are usually not deep mysteries of computer science; they are the kind of issues that every programmer is taught to avoid, yet which still slip through under real-world pressures. As one industry report noted, *most forms of testing are less than 50% efficient in finding bugs*, whereas techniques like code inspections can catch 85% or more (Jones, 2011). The implication is that if we rely only on standard testing and if our coding processes are not meticulous, at least half of the bugs (many of them trivial in cause) will escape into the wild. Those escaped bugs then incur massive clean-up costs. The next section turns to the critical question: if small mistakes cause such outsized problems, what can be done to prevent them? The answer

lies in treating software development with the same kind of uncompromising discipline seen in mature engineering fields: addressing the hygiene factors that, while not glamorous, have proven effective in catching or preventing the "small" errors.

Preventing the Preventable

If the devil is in the details when it comes to software, then the defence against failure lies in an almost obsessive attention to those details. In practical terms, this means instituting "software hygiene." A set of best practices and cultural norms in development teams that ensure mistakes are caught (or avoided) as early as possible. Just as hand-washing and sterilization revolutionised safety in medicine (turning surgery from a high-risk gamble to a routine procedure), certain fundamental practices can dramatically reduce the incidence of software bugs. What are these practices? Many of them have been known for decades, yet they are not uniformly followed, which is why we continue to see the same types of errors. In this section, we outline key hygiene factors and explain how each directly addresses the kinds of trivial mistakes discussed earlier. The ideas here stem from the author's early years as a programmer at Philips Terminal Systems, the world's largest manufacturer of banking computers, the PTS 6000 series (Danielson, 2023). These banking computers were sold as systems sales to banks, including hundreds and often thousands of workstations and encompassing both hardware and software. The software for the customer projects was most often developed by Philips, not by the customer. This centralisation and the coding standards upheld resulted in the software delivered almost invariably being shipped error-free (Sandén, 2011, p.246). The overarching theme is *discipline*: getting developers to consistently do the "boring" things that prevent bugs, even when shortcuts attract. The PTS 6000 series was programmed in assembler language and in a COBOL-inspired interpreted proprietary language called CREDIT. Philips then moved on to C in the mid-1980s, and many of the author's other projects were also C-based. Since C has survived as a major language until this day, it is the language used in the examples in the book. But the principles are easy to transfer to any imperative language, object-oriented or not.

Coding Standards and Consistent Style.

One of the simplest yet most effective hygiene practices is to adopt a strict coding standard. A set of stylistic and programming rules that everyone on the team follows. Superficially, coding standards (naming conventions, brace styles, etc.) might seem unrelated to bug prevention, some even view them merely as aesthetic guidelines. However, well-designed coding guidelines explicitly aim to reduce the likelihood of mistakes. The idea is that by enforcing consistency and disallowing known dangerous practices, the code becomes more readable and less error-prone. A trivial example: requiring that every `if` or `while` block use braces even if it is a single line. This prevents the classical bug of an unintended dangling else or a misplaced statement that is not guarded by the condition (a source of many logic errors in C/C++). Another common rule is to ban the use of magic numbers or hard-coded constants in favour of named constants or enumerations, which reduces the chance of using the wrong value or misinterpreting units. Standards often outlaw certain language features known to be problematic, For example, in C functions like `gets()` that do not check bounds, or implicit conversions that could truncate data. By removing these pitfalls, the standard guides programmers toward safer patterns. The evidence of effectiveness is strong: organisations that rigorously apply such standards have observed reductions in bug density and debugging time. One reason is that a common style makes it easier for developers to review each other's code. Inconsistencies or odd constructs stand out like sore thumbs, prompting closer inspection of where a hidden bug might lurk. Additionally, many coding standards include *"robustness" rules*, such as always checking the return value of system calls and library functions. This directly catches errors that would otherwise propagate (for instance, if a file open fails and the code does not check, subsequent operations might behave incorrectly). By mandating these checks everywhere, the standard instils a habit of defensive programming. We can see the benefit in cases like the Mars Polar Lander: had the software been developed under rules requiring redundant confirmation of critical sensor signals, the false touchdown detection might have been averted. In sum, adopting a rigorous coding standard is akin to a checklist for coding. It ensures that each line of code adheres to known best practices and avoids patterns with high bug potential. It may not eliminate all bugs (no standard can guarantee that), but it

can certainly eliminate a sizable fraction of the trivial ones to become essentially error-free, the promise in the title of the book that never came to fruition. It sets a baseline of discipline: no matter how rushed or tired a developer is, the standard is a safety net reminding them of the details not to forget.

Code Reviews and Inspections

Human fallibility is what it is, even the best programmers miss their own mistakes. That is why a cornerstone of software quality is independent reviews: having peers inspect code changes. Formal code inspections, as practiced in some organisations, involve systematically reading through the code in team meetings and using checklists to find errors. Numerous studies have shown that such inspections are extremely effective at catching bugs early. According to one analysis, formal design and code inspections are more than 65% efficient in finding defects and often top 85% efficiency (Jones, 2011), whereas unit testing might only catch 25-50% of defects on average. The reason inspections work well is because they force a slow, methodical examination of the "boring details" that automated tests might not execute. A reviewer might notice, for example, that a certain variable is not initialised, or that a loop's boundary condition seems off by one. Things that might not immediately cause a failure in a limited test, but are indeed bugs. Inspections also leverage the fact that different people have different strengths; one reviewer might be good at catching arithmetic precision issues, another at spotting potential null-pointer dereferences. By pooling their attention, the team's overall attention to detail is amplified. A study by IBM on its cleanroom development process (which emphasise prevention over testing) found that with extensive inspections and careful adherence to process, they could achieve near-zero defect rates in delivered code. In practice, not every project can afford extremely formal inspections, but even lightweight reviews (such as pair programming or GitHub pull request reviews) have proven their value. For instance, the aerospace industry often requires that every line of code for flight software is reviewed by multiple people. This is how the Space Shuttle on-board software achieved only 17 errors in 420,000 lines across many years. It was not magic but method: every change was scrutinised, tested, and verified exhaustively. For less critical software, the stakes may not be life-or-death, but the mindset should

still apply: treat every abnormality in code as potentially serious until proven otherwise. Modern development tools also aid reviews. Static code analysers (discussed shortly) highlight suspicious code constructs, effectively acting as an automated reviewer that never tires of nit-picking. But ultimately, having a human in the loop who says "*I do not quite understand why you did X here; could that cause a problem if Y*?" often exposes hidden bugs. The key is creating a culture where such critique is welcomed as a positive, quality-improving step, not as a personal affront. When done right, code reviews catch those "little" mistakes (typos, logic slips, omissions) before the code ever runs. This approach maps well to the earlier stories: a thorough review of the climate orbiter navigation code would likely have questioned the units at an interface, or a review of the 737 MAX changes would have asked, "What if the sensor feeding MCAS is wrong?" Many post-mortems of failures reveal that someone in the organisation *did* raise a concern but it was not heeded or formally followed up. A disciplined review process institutionalises the addressing of such concerns. To sum up, code reviews enforce collective attention to detail, making it far less probable that trivial bugs survive in the wild.

Automated Static Analysis and Tools

In addition to human reviewers, automated analysis tools serve as an ever-vigilant eye for certain classes of mistakes. Static analysis examines source code (or compiled binaries) without executing them, looking for patterns that are likely errors: e.g., possible null pointer dereferences, variables used before initialization, buffer overflows, inconsistent use of values, unreachable code, etc. Many trivial bugs have signatures that static analysers can detect. For example, a static tool can easily catch that a function has two return paths, one of which fails to assign a value to the output variable, a mistake a human might easily gloss over. The advantage of static analysis is that it can check every single path and combination up to a certain complexity, something human testers cannot do exhaustively. Mature static analysis tools have been shown to significantly reduce defect density when applied regularly. In mission-critical systems, use multiple static analysis tools alongside coding standards and inspections in order to achieve cumulative defect removal rates above 95%. The reason multiple tools are mentioned is that different tools have different strengths.

One might be better at spotting concurrency issues and another at spotting arithmetic edge cases. By incorporating these into the build process (e.g., as part of continuous integration, code must pass static analysis checks with zero warnings), teams enforce a level of detail-checking that goes beyond what any individual could do. It is similar to having a tireless proofreader for code. Another form of static checking is the use of more stringent compilers or compiler settings. Many languages allow optional warnings (for instance, the `-Wall -Wextra` flags in C/C++ compilers turn on a host of useful warnings). Ensuring the code compiles with all warnings enabled and treating warnings as errors can prevent a lot of silly mistakes (like using = when == was intended in C, which is a classic bug that compilers can flag as a suspicious assignment in a conditional). For managed languages like Java or C#, linters and code quality analysers play a similar role. These tools embody the collective wisdom of many experienced programmers, flagging constructs that have led to bugs before. Using them is an obvious hygiene step, yet in the rush of development, many teams skip them or ignore their output. Do not ignore a single warning lightly. If the static analyser says there's a possible null pointer, investigate it. Either prove it is a false positive or fix it, but always attend to it. Taking tools seriously is part of cultivating an attention-to-detail mindset. When static analysis is combined with adherence to a safe subset of the language, it can even approach formal verification levels for certain properties. For example, the absence of any out-of-bounds array access can be practically guaranteed if one follows certain rules and uses static analysis to enforce them. In safety-critical circles, it is often said that a compiler warning fixed is a failure avoided. The bottom line is that in modern software engineering, there is little excuse for letting common mistakes slip by, given the quality of tools available. It is more a matter of will and policy, requiring that these tools be used and their output acted upon. This too is an element of discipline: it can be tedious to fix all "potential null dereference" warnings in a large codebase, but doing so systematically will likely eliminate some real bugs and certainly improve the clarity of the code (if only by making the programmer think about whether that null case can happen). It is similar to cleaning up every corner of a workshop to ensure nothing dangerous is left lying around. It takes time, but it prevents accidents.

Thorough Testing at Multiple Levels

Testing may seem too obvious to mention. Of course everyone tests their software. But the key is *how* testing is done and at what stages. A disciplined approach involves multiple layers: unit tests for individual functions or modules, integration tests for components working together, system tests under realistic scenarios, and so on. The concept of improved testing as a way to eliminate bugs was strongly endorsed in the NIST study which said one-third of the bug cost could be saved by improved testing. The idea is that better testing is essentially better attention to detail during verification. However, simply having tests is not enough; one needs to consider test coverage and test design. It is easy to write cursory tests that check the expected cases and call it a day, and thus miss the edge case that will later cause a problem. Best practices encourage writing tests not just for correct inputs but also for erroneous and extreme inputs, to ensure the software handles them gracefully. This is especially crucial for catching off-by-one errors, overflow, and other boundary issues that are frequent culprits of failures. For example, had the Ariane 5 software been tested with simulated input values exceeding the 16-bit limit (values that Ariane 4 never produced but Ariane 5 did), the overflow would have been discovered before the flight. Similarly, testing the Mars lander software with spurious sensor signals might have revealed the premature engine cut-off logic flaw. Good test practices also involve regression testing. Every time a change is made, re-running the full suite to ensure nothing else broke. This is a guard against the scenario where a "fix" for one bug introduces another bug (a phenomenon not uncommon when the root cause is not fully understood or the fix is rushed). Automated testing frameworks make it feasible to run hundreds or thousands of tests on each code commit. However, writing a comprehensive test suite is itself a matter of discipline and attention to detail. It requires thinking about what could go wrong in every corner of the code. One effective approach is test-driven development (TDD), where developers write tests for the functionality *before* implementing it, clarifying the expected behaviour and edge cases upfront. TDD forces you to consider the little details (like "What should happen if this input is zero or negative?") early on, often revealing potential issues in the design or assumptions. While TDD is not a panacea, teams

practicing it often report fewer defects, precisely because it bakes in a habit of checking details. In safety-critical industries, testing extends to simulation and even formal methods. For instance, aerospace software goes through hardware-in-the-loop simulations where every sensor and actuator is modelled to test the software in conditions as close to real as possible. They perform stress tests and fault injections (e.g., simulate a sensor giving nonsense data) to ensure the software responds safely. The general software industry can learn from this by at least adopting chaos testing or fault injection in staging environments to see how robust systems are to weird inputs or partial failures. Such tests often uncover the kinds of small oversights (like an uninitialised variable that only matters when a certain flag is set under high load) that would be hard to catch otherwise. Ultimately, a key principle is that testing should be aimed at breaking the software. Testers (or automated test generators) should actively seek out the weak spots rather than just demonstrating that the software works on expected input. This adversarial mindset in testing complements the constructive mindset in coding. Where a developer might subconsciously avoid thinking of extreme cases because they want their program to work, a good tester does the opposite: "How can I make this fail?" Adopting that perspective within the development team itself, early on, is extremely beneficial. In fact, some organisations rotate developers into testing roles or have them review each other's modules with the intent to find bugs, not to validate designs. Doing so reinforces attention to detail by making everyone more aware of the kinds of mistakes that can happen.

Risk Assessment and Worst-Case Thinking

A more mindset-oriented practice, but crucial: developers should be trained (and encouraged) to think about worst-case scenarios and to assume things will go wrong. Much of the time, small bugs slip in because developers assume a particular condition will never occur (e.g., "the sensor will never send crazy data", "the user will never input a 10000-character string", or "this array will never go out of bounds"). As we have seen, those assumptions often do not hold. Adopting a *defensive programming* stance means always coding as if the worst can happen. One practical way to enforce this is to add explicit checks for conditions that "should not happen,"

paired with safe handling or at least clear logging if they do. For example, if a function is only supposed to be called with a non-null pointer, an assert or explicit null check with an error return can catch any violation of that contract. It is a simple addition, but it turns a potentially catastrophic bug (null pointer leading to crash or corruption) into a controlled failure or a logged anomaly that can be fixed. In high-integrity systems, this is taken further with mechanisms like built-in tests (BIT) and sanity monitoring. Essentially code that continuously verifies that internal assumptions hold, and if not, transitions the system to a safe state. The broader notion is sometimes referred to as *design by contract*, where functions explicitly state preconditions and postconditions, and the code checks them. These checks act like tripwires for bugs: if a bug causes an unexpected condition, the program halts or alerts at the point of detection, making diagnosis easier and preventing compounding damage. It is far better to fail fast than to propagate erroneous data and later fail mysteriously. In less formal settings, even code comments that flag assumptions can help future maintainers not violate them. The overall culture should be that every time someone writes "this should never happen" in a comment, they either write code to handle it just in case or at least an assertion to catch if it does happen. Another aspect of worst-case thinking is considering performance and resource limits. Many failures in the real world occur when load or input exceeds tested ranges (e.g., a web service times out or crashes when requests spike because of a subtle race condition that only manifests under high concurrency). By doing stress tests (as part of the testing regimen) and planning for capacity, teams can often uncover issues that result from those "we never expected so many users/files/etc." conditions. Again, these issues often trace to small mistakes: maybe a data structure that becomes inefficient at scale, or an algorithm that fails when a list is empty (which suddenly happens under heavy load due to timing). Paying attention to detail means also *attention to rare events*. A disciplined team might, for example, have a checklist item: "Have we handled all error returns from this library call? What if the network is down? What if the disk is full?" It can feel tedious to ponder all these unlikely problems, but such foresight distinguishes robust software from brittle software. The cost of adding code to handle edge cases is usually low (often just a few lines, as Greg Travis's eight lines for MCAS illustrate), but the cost of not adding them can be immense.

Continuous Integration and Fast Feedback Loops

Effective error prevention also relies on getting quick feedback to developers when something goes wrong. The longer a bug lives in the code, the more context is lost and the harder it is to root-cause-track and fix. Modern best practices use continuous integration (CI) that automatically builds and tests the software on every commit. This means that if a developer introduces a bug that breaks some tests, it is caught within minutes and they are immediately notified. Rapid feedback tightens the attention loop and developers learn of their mistakes while the code is fresh in their mind, and can correct it before moving on. It helps avoid the scenario of a small bug lurking for months and causing a failure later in production when no one remembers that piece of code well. CI can also incorporate static analysis and style checks so that any deviation from the agreed standard or any new warning is flagged instantly. Essentially, automation enforces attention to detail. If a developer forgets to run tests, the CI will run them. If he skimmed over a warning, the CI will remind him by failing the build if any warnings are present (if configured to be strict). This removes reliance on memory or personal diligence alone, embedding diligence into the process. Many open-source projects have adopted a policy of "no regressions allowed." If a commit increases the number of warnings or fails any test, it cannot be merged. Such policies, when followed, maintain a very high baseline quality. Another practice is code hygiene days or regular refactoring, essentially scheduling time to pay off technical debt and tidy up the code. During these times, teams might focus on cleaning up compiler warnings, improving error messages, increasing test coverage, etc. While this might seem tangential to features, it directly addresses those lurking small issues that otherwise might be forgotten. Managers who understand software quality often allocate, say, 10–20% of iteration capacity to these tasks, treating them as first-class work. This kind of proactive maintenance is crucial; otherwise, as projects mature, the accumulation of little mistakes and kludges eventually slows development to a crawl or results in a major failure.

Learning Culture and Retrospectives

Even with all of the above, bugs will still occur. The difference in a detail-oriented culture is how these bugs are treated when found. Rather than just patching and

moving on, high-maturity teams conduct *post-mortems* or retrospectives even on small incidents or escaped defects. The goal is to identify the root cause and, importantly, improve the process to prevent similar bugs in the future. This might lead to adding a new test case, expanding the coding standard, or educating the team about a particular gotcha. For example, after the Mars Climate Orbiter loss, NASA convened multiple review boards and disseminated the lesson about unit consistency to all projects. The result was process changes to ensure no similar lapse in unit conversion would recur. In a corporate software setting, if a production outage was caused by, say, a misconfigured setting that was not caught, the post-mortem might result in improved deployment scripts or monitoring checks. This continuous improvement mindset closes the loop in attention to detail: it is not enough to be detail-oriented in writing code; one must also be detail-oriented in examining failures. A blameless retrospective asks "What went wrong in our process that allowed this bug through, and how can we tighten the sieve?" This way, even mistakes become fuel for preventing future mistakes, a virtuous cycle of quality improvement. Over time, such teams build a deep collective knowledge of pitfalls to avoid. Notably, some of the best engineering organisations (like those behind space missions or critical infrastructure) institutionalise this knowledge in formal guidelines and training. The lessons-learned databases at NASA or the checklists airlines use before flights (spawned from past incidents) are examples. Adapting that approach, software teams could maintain an internal wiki of patterns to avoid or an internal "bug of the month" discussion where they dissect a recent bug. All of this contributes to raising everyone's attentiveness.

Towards Software Craftsmanship

The evidence and examples discussed lead to a clear conclusion: most software disasters and much everyday software grief could be avoided by a more disciplined, craftsmanship-like approach to development. It is not that we lack the knowledge of how to write correct software; it is that we often fail to apply this knowledge consistently. In effect, the industry's challenge is more cultural than technical. We must elevate *attention to detail* from a personal trait of only some programmers to a core value embraced by every developer, team, and organisation. This calls for something

like rules for software hygiene, a set of principles that developers commit to, emphasizing that quality is not negotiable and that small things matter tremendously. The final part of this chapter articulates such a perspective, synthesising the lessons learned into a set of guiding principles.

Principle 1: *Bugs Are Not Inevitable*

Aim for Zero, Do not Tolerate Good Enough.

Too often, a fatalistic attitude prevails that all software has bugs or that certain amounts of failure are just part of the trade-off for rapid innovation. While it is true that complex software will never be absolutely perfect, the mindset of inevitability is dangerous. It leads to complacency and corners cut under the assumption that some bugs are okay. A more productive stance is to treat every bug as preventable until proven otherwise. Historically, people once accepted that infections in surgery were inevitable. It took medical pioneers like Semmelweis to show that simple hygiene (hand-washing) could nearly eliminate them. Likewise, software teams should operate under the assumption that with enough care, most defects can be prevented or caught early. This does not mean features take forever. It means building quality activities into the normal workflow. The psychological shift is key: developers taking pride in writing code that just does not fail. This can be fostered by setting quality goals explicitly (e.g., targeting a certain low defect rate or mean time between failures) and celebrating achieving them just as one would celebrate feature delivery. The principle "first, do no harm" from medicine can translate to "do not introduce new bugs" in software. For instance, teams can adopt a zero-tolerance policy for known critical bugs in the backlog at any time; they must be fixed immediately. By not accepting bugs as a norm, teams will naturally push themselves to refine their processes and skills to approach zero-defect software. Notably, some software organisations have achieved extremely low defect rates (on the order of 0.1 defects per kLoC) by adhering to such high standards. Their example disproves the myth that you must choose between speed and quality. In fact, quality-focused teams often end up faster in the long run because they do not waste time on endless fixes.

Principle 2: *Prevention Over Cure*

Catch Mistakes Early, Before They Manifest as Failures.

This principle echoes a recurring theme: it is far cheaper and easier to prevent a bug than to fix it later. Practically, this means investing effort in activities that happen before the software is running in production. Code reviews, static analysis, and comprehensive test suites may seem like overhead, but they are the front-line mechanisms for bug prevention. If something is suspected to be wrong during development, fix it or investigate it immediately. Do not defer it with the idea of testing it later or patching it in maintenance. The earlier a bug is removed, the less impact it has. The prevention-over- cure mindset also encourages developers to think carefully when writing each piece of code: *How can I write this in a way that inherently avoids errors?* This might mean using higher-level abstractions (for example, using safe libraries rather than writing low-level pointer manipulations), or leveraging language features (such as strong typing or immutability) that eliminate classes of bugs by design. Modern programming languages and frameworks often provide constructs to help with this, but they must be intentionally used. Teams should prefer coding practices that make incorrect code harder to write. One illustration is using enumerations or distinct types instead of generic integers for values that represent different domains (like using a `Distance` type vs. a raw number, so one cannot accidentally mix meters and feet). This builds unit safety into the code, a direct lesson from the Mars orbiter loss. If one had a `Force` class with a unit attribute, it would be impossible to mix up Newton- and Pound-force without an explicit conversion, which would have made the error obvious. This prioritization justifies spending time on design, using static type checkers, formal specifications for critical algorithms, etc., because those are prevention tools. It is striking how in other engineering fields, enormous effort is spent in design verification (e.g., multiple prototypes, stress-testing materials) to ensure things do not break later. Software should be no different. As an industry, we should shift resources earlier in the lifecycle. A point made in the classic Boehm curve, which showed that errors become exponentially more costly to fix later. The good news is that with today's automation, a lot of prevention can be done quickly (e.g., running thousands of static checks in seconds).

Principle 3: *Hygiene Factors Matter*

Embrace the Boring Best Practices Diligently.

There is a set of development practices often considered pedantic or tedious: writing documentation, commenting code, using version control properly, continuously refactoring to keep code clean, sticking to conventions, etc. These might not be as exciting as developing a new feature, but they are exactly analogous to washing hands or cleaning tools in a workshop, i.e. essential for preventing mistakes. For managers and stakeholders, it is important to allocate time for these tasks explicitly. If a team is pressured to deliver features at breakneck speed with no time for cleanup, the debt incurred will likely surface as customer-visible bugs. A critical practice under hygiene is configuration management: ensuring that environments are consistent, deployments are automated, and rollback procedures exist. Many outages occur not because of bad code but because a good code change was deployed incorrectly or to the wrong environment. Strict procedures and tooling around build and release (e.g., infrastructure as code, one-click deploys with validation) are the hygiene that prevent such fiascos. Another one is backups and monitoring. Those are not directly coding practices, but operational hygiene. Having good monitoring can catch a small anomaly before it becomes a big incident (e.g., a slow memory leak can be fixed during routine maintenance if noticed, rather than causing a crash in peak business hours). The overarching message is that nothing is beneath attention. In a culture of true craftsmanship, even the smallest component or process is given due care. A telling example is from the Apollo program: the on-board guidance computer's software famously had an implicit "do not blow up" rule. They put enormous effort into ensuring the software would reset gracefully and not send spurious commands if it encountered an error. That required diligently handling even the unlikely edge cases. It was not glamorous, but it was necessary. Modern developers should channel that same thoroughness. The end goal of software hygiene is that when someone asks, "Did you consider X? Did you handle Y?" the answer is always "Yes." Perhaps one of the reasons the tech industry has often moved fast and broken things (to borrow a Facebook motto) is the youthful culture and lack of formal professional standards compared to, say, civil engineering. But as systems become more safety-critical and the costs of failure mount, the industry is maturing. There are

calls to treat software engineering more like a profession with agreed standards of conduct. This is not contrary to the view of programming as an art. The art is in the design and overall programming of the systems, not in the good or bad habits of detailed coding. Any artist working with physical artefacts must still make sure their artwork holds together to stand the test of time. This is no different for programs. It is telling that in engineering or medicine, negligence in basic practice can cost one's license. In software, anyone who can code can potentially work on critical software without certification. While formal licensing is a debated topic, at the very least teams can self-impose high standards. We can view things like thorough testing and code reviews as our professional oath to do right by the users who depend on our code.

Principle 4: *Learn from Failure*

Every Incident or Near-Miss Is a Lesson to Internalise.

High-reliability organisations have a characteristic: they are preoccupied with failure, even the small ones. They investigate, disseminate findings, and adjust accordingly. Software teams should do the same. Instead of seeing bugs as an embarrassment to be quickly hidden, they should be seen as valuable data. Adopting blameless post-mortems (focusing on what in the system or process allowed the bug, not who made the mistake) encourages openness. For example, after a critical outage, the team can publish a brief report: "Here's what happened, why, and what we're doing to prevent it going forward." This not only helps the team not repeat mistakes but contributes to industry knowledge when shared publicly. The aviation industry's safety record improved dramatically through transparent investigation of crashes and the sharing of recommendations. In software, companies often operate in silos regarding failures, but there is a trend of more transparency (some companies openly blog about outages and fixes). Embracing this is part of the error-eradicating attitude: it is okay to err, but not okay to fail to learn. When developers see their mistakes leading to constructive changes (rather than just reprimand), they are incentivised to bring forward issues early. It also sets a tone that quality is everyone's responsibility. If a bug slips through, it is not just "the tester's fault" or "that developer's fault," it is a symptom that the team's process can improve. This communal

approach reduces ego and defensiveness that sometimes hinder acknowledging bugs. In essence, the mindset respects the complexity of software by acknowledging that avoiding all mistakes is hard but then doubles down on systematically driving down the mistake rate. Over time, an organisation with this culture builds a reputation for reliability. Customers and users notice when a service or product just works consistently. Conversely, they also notice when each update introduces new problems. Ultimately, fostering trust through quality can be a competitive advantage. It aligns with the point that many software-related losses (time, money, goodwill) are preventable. So preventing them is not just good engineering, it is also good business.

Principle 5: *True Professionalism*

Recognise the Real-world Consequences of Software Errors.

Those of us who create software must be cognizant of the impact our mistakes can have on others. In safety-critical fields, this is obvious, but even a "simple" app can cause harm if it fails at a bad time. Consider an electronic health record system glitch that mixes up patient data or a navigation app error that misroutes drivers into danger. These are software mistakes with potentially grave outcomes. Thus, attention to detail is not just a technical nicety; it is a responsibility. The "Software Engineering Code of Ethics" (adopted by professional societies like ACM/IEEE) emphasises that engineers shall ensure their products are as safe and error-free as possible and shall be honest about limitations. Taking that to heart means pushing back on unrealistic deadlines that force cutting corners, and ensuring management understands the risks of not investing in quality. It might mean spending extra personal effort to test that one more scenario because it could save user frustration. This ethical perspective fuels the critical tone of our discussion. It is, after all, calling for change. We must, as a profession, move away from celebrating only speed and features, and give equal billing to reliability and care. There is room for optimism: industries like automotive and aerospace have steadily raised the bar on software quality through standardization and regulation (for example, ISO 26262 for automotive software safety). While nobody wants stifling regulation in all software domains, the writing is on the wall that if we do not improve self-discipline, external forces will impose

it after enough failures. Better that we voluntarily improve. In practical terms, this principle encourages mentoring and training. New developers should be taught not just how to code, but how to *code well*. Universities often underemphasise secure and robust coding in favour of quick hacks to get assignments working. Industry can fill that gap by inculcating best practices from day one. Senior engineers should exemplify calm, detail-oriented work, rather than "cowboy coding." When juniors see that careful consideration is valued more than clever one-liners, they will emulate that.

Summary

In conclusion, the argument that "error-free software is largely dependent on attention to detail" is supported by abundant evidence, and it points towards a necessary evolution in our approach to software development. The small mistakes that plague software projects are not an intractable mystery; they are solvable problems with known techniques if we choose to apply them. The challenge is mustering the will to do so consistently. This is why a diligent stance can be beneficial: it frames software quality not as a bureaucratic burden, but as a cause worth championing. It asserts that quality is an integral part of the definition of done, not a desirable afterthought. It appeals to professional pride: hopefully, nobody wants to be associated with sloppy work that fails dramatically. And it appeals to rational self-interest too since high-quality software is cooler in the end and enhances reputation.

To return to the examples that opened the chapter. We should live in a world where no space probe is lost due to a unit conversion error because every space software team double-checks interfaces by habit. A world where aircraft software does not surprise pilots because every new line of code is scrutinised under multiple failure scenarios. A world where the only outages in banking systems are from unavoidable external events, not from a developer forgetting to synchronise a config file. Such a world is achievable; the knowledge exists. The cost of achieving it is mainly discipline, and that is a cost that yields returns many times over. As the old adage goes, *"The devil is in the details"*, but with diligent angels (the developers) attending to those details, the devils of chaos and error can be exorcised from our

software. This is the path to truly engineering software, to making it a rigorous profession on par with others, and to drastically reduce the unnecessary losses that today we far too commonly chalk up to "computer glitches." The time has come for software engineering to grow up and recognise that its biggest challenges are not unsolvable algorithms, but the mundane and vital work of getting every little thing right.

The cause of most software-related failures and losses can be traced to small preventable mistakes made during implementation and insufficiently caught during testing. By treating those small mistakes as unacceptable and attacking them with disciplined practices, we can avoid most of the dire consequences they would otherwise produce. This often calls for a culture change, a recommitment to programming fundamentals. The benefits of such a change would be enormous: more reliable systems, less downtime, fewer safety incidents, and huge savings of time and money. The case studies and data presented serve as both warning and motivation. Small bugs might be trivial in isolation, but aggregated they shape the fate of projects and companies. The onus is on us, as software professionals, to heed the warnings and take up the meticulous, unglamorous, yet ultimately satisfying work of making our software as close to error-free as humanly possible. This is our collective hygiene challenge, our call to ensure that the next Mars orbiter, the next aircraft, and the next financial platform all benefit from the hard lessons etched in the history of software failures. The PTS way of programming can help with this.

4. Guidelines for Error-Free Code

In software development, the significance of general coding rules cannot be overstated. These Philips PTS guidelines serve as a foundation upon which other organisation-specific rulesets should rest, shaping how code is structured, interpreted, maintained, and validated. Not least in systems where reliability is a non-negotiable constraint and debugging is often difficult due to, for example, complex unforeseeable user interactions, it is important to establish a uniform style and methodology that can be trusted across project teams, tool-chains, and architectural targets.

Conscious restriction of language extensions and preprocessor manipulation is important. It might be tempting for a developer to redefine keywords or insert macros to simulate language-level constructs, such as `#define forever for(;;)` or `#define begin {`. These alterations, while sometimes clever, are hazardous. They mislead developers, break expectations for those accustomed to standard C, and make it nearly impossible to port the code across compilers or tools. The preprocessor should be used for inclusion guards, conditional compilation, and header abstractions, not as a mechanism to rewrite the language itself. Compiler-specific constructs such as `__attribute__`, `#pragma`, and inline assembly should be isolated to interface modules and documented rigorously. If they leak into application logic, they undermine portability and increase the mental overhead of reading and debugging the software.

Code line length is another general rule that supports not only readability but also collaboration. Source code lines should not exceed 80 characters. This guideline has persisted through decades of computing history because it reflects the limitations of terminal widths and printing formats. More importantly, it allows tools like `diff` (`fc` if you are Windows-bound) and code review platforms to present changes side-by-side without truncation. Excessively long lines make code harder to scan and maintain, and they are often a sign of overly complex logic that should be broken down into simpler components. A good heuristic is: If a single line of code contains more logic than a developer can comprehend in one glance, it is too complex. The same goes for functions and procedures (methods for OOPers). In the old days of 24x80 screens, a single procedure should fit onto the screen to be comprehendible.

With today's screens having at least 80 lines, a screenful can be a stretch, but 50 lines is a good maximum size for a procedure (OOPers will object; see Chapter 5).

The structure of control flow blocks such as `if`, `else`, `while`, `for`, and `switch`, should be explicit and uniform. Each such block should be enclosed in braces, regardless of whether the body contains a single statement. This rule is a direct response to some of the most catastrophic software bugs in history, including the "goto fail" bug in Apple's TLS implementation (from the same year as the TLS Heartbleed bug discussed in Chapter 3, but is something completely different). The absence of braces leads to ambiguity. It invites logical errors when code is modified, and it relies on indentation to imply meaning, which is not enforced by the compiler. Always placing opening braces on their own lines, aligned with the controlling keyword, creates a visual symmetry that enhances readability and debugging.

Another important rule involves the use of parentheses in expressions. Most languages' operator precedence rules are complex and sometimes counterintuitive. Programmers make frequent errors by assuming a particular evaluation order, especially when combining logical operators (`&&`, `||`) with relational or arithmetic operators. To avoid ambiguity and ensure intent is clear, wrap each logical clause in its own set of parentheses. For example:

```
if ((temperature > THRESHOLD) && (pressure < LIMIT)) {
    // Safe to proceed
}
```

Even if redundant, these parentheses prevent accidental misinterpretation and serve as documentation of intended precedence. They are particularly useful in conditions involving multiple layers of logic, where a missing set of parentheses can change the meaning of the entire condition.

Mixing logical operators (`&&`, `||`) with bitwise operators (`&`, `|`) might sometimes yield the same results under certain circumstances. But code containing such mistakes runs a large risk of failing sometime in the future, perhaps after maintenance.

Comments are not decorations; they are integral to code quality. All comments should be written in full sentences, using correct grammar and spelling. Each comment should provide insight into the purpose or rationale behind the code. Avoid

trivial comments like `// increment x` next to `x++;`. Instead, explain why `x` is being incremented, under what conditions, and what role it plays in the system's operation. When disabling code, never use block comments (`/* ... */`) to comment out large segments. Instead, use conditional compilation with `#if 0 ... #endif` to make it clear to both the compiler and the reader that the code is intentionally excluded. Always add a reason, such as

```
#if 0 // Temporarily disabled due to thread race condition
```

Annotations like `TODO`, `FIXME`, `NOTE`, and `WARNING` should be used consistently and in uppercase, preferably at the beginning of a comment line. These markers stand out visually and can be easily searched or flagged by tools. For example:

```
// TODO: Refactor this loop to handle overflow correctly
```

Whitespace conventions are another essential aspect of general coding rules. Code should use four spaces per indentation level and spaces, not tabs, should be used. This is easily configured in any IDE. Tabs render inconsistently across editors and platforms, leading to misaligned code and readability issues. There should be a space on both sides of binary operators, after commas in parameter lists, and before braces. Proper whitespace usage not only makes code easier to read, it also reduces the likelihood of introducing syntax errors during editing.

Files should terminate with a newline character and a comment like `// end of file`. This seemingly minor practice serves practical purposes. The final newline ensures compatibility with version control tools and POSIX file handling. The end-of-file comment is a visible signal that the file is complete and not accidentally truncated. This can be especially useful in headers, where an incomplete macro definition or missing semicolon can lead to hours of frustrating debugging.

Lastly, projects should adopt consistent and automatable enforcement of these rules. Use linters, formatters, and code review checklists to ensure compliance. Even the best guidelines lose value when inconsistently applied. Automated tools ensure that rules are followed uniformly, freeing developers to focus on logic rather than style policing.

These general rules form the scaffolding of disciplined C programming, and similar rules can be set up for any language and environment. They are the smallest unit

of structure and yet influence every subsequent layer of abstraction and modularity. Ignoring these rules results in codebases that are brittle, difficult to read, and error-prone. Three "qualities" that have no place in systems where correctness and resilience are targets.

Naming Conventions

Naming conventions form the verbal architecture of a codebase. In important systems, where software often lives for decades and must be interpreted by engineers far removed from its original authors, consistency in naming becomes a lifeline. Clear and predictable names reduce on-boarding time, facilitate cross-team collaboration, and simplify auditing and debugging processes. They are especially valuable in large codebases where navigation relies as much on lexical patterns as it does on structure.

File names should be lowercase and consist only of letters, digits, and underscores (some languages enforce that, others do not). They should reflect the module's purpose or domain clearly, avoiding abbreviations unless absolutely standard within the specific project. For instance, `reconcile.c` is clear for an end-of-day module, whereas `rec.c` is cryptic. Underscores improve readability over camel case in file names, which are often manipulated through command-line tools that favour underscore delimiters. File names should avoid collisions with standard library headers such as `math.h` or `time.h`, and every `.c` file should include its own header first to validate internal consistency.

Variable names should be long enough to convey meaning but short enough to remain readable. For local variables, one-letter names such as `i` or `j` are acceptable only in the smallest of scopes, such as within a `for` loop. In all other cases, names should describe the content or purpose of the variable. For example, `buffer_index`, or `timeout_counter` are preferable to `bi`, or `tc`. Boolean variables should be named to reflect their binary nature using affirmative phrasing, such as `is_done` or `has_data`.

Prefixing is another key strategy. Global variables should be prefixed with `g_`, pointers with `p_`, and booleans with `b_`. This prefixing system enhances scanning

and reduces bugs from misuse. For example, accidentally dereferencing a non-pointer becomes less likely when all pointers are explicitly marked. Similarly, recognising that `b_configured` is a boolean, not an integer, allows a reader to immediately interpret expressions where the variable appears.

Function names should reflect the actions they perform. They should begin with a verb and be written either in lowercase with underscores separating words (snake case) or with a cap letter separating words (camel case). A strict naming convention increases clarity and ensures consistency. Examples include `read_user_data()`, `init_display_module()`, and `handle_error_condition()` for snake case, which we use in this chapter. Public functions should be prefixed with the module name to avoid symbol conflicts and to signify ownership. For example, in the `led` module, public functions might be named `led_init()`, `led_on()`, and `led_toggle()`.

Naming conventions also extend to type declarations. All `typedefs` should use lowercase and terminate with `_t`, such as `timer_config_t` or `user_data_t`. Public types should also be prefixed with the module name, just as public functions are. This ensures a one-to-one mapping between symbol and module, which becomes essential when debugging symbol tables or navigating complex project hierarchies. Structures, enums, and unions should always be declared with a `typedef` to simplify syntax and enforce consistency.

Macros and constants also benefit from naming rules. Macro names should as a convention always be written in all uppercase with underscores: `MAX_BUFFER_SIZE`, `DEFAULT_CURRENCY`, and `ENABLE_LOGGING`. This convention distinguishes them from variables and functions at a glance. Constants defined via `#define` or `const` variables should follow the same convention, as they often play a similar role in logic and configuration.

Abbreviations should be used with care. Developers often assume their own shorthand is universally understood, which leads to opaque identifiers. Use only widely accepted abbreviations. Project-specific abbreviations must be centrally documented in a version-controlled glossary file. This file should be reviewed and maintained like source code, as it ensures that the naming conventions evolve deliberately rather than arbitrarily.

Misleading names can cause severe misunderstandings. For instance, naming a number read function `read()` in a system where some input numbers are integers while others are floats is ambiguous. Better names would be `read_float()` and `read_int()`, which convey intent and precision. Similarly, avoid names that imply units where none exist, or vice versa. If a variable is named `usd`, readers will expect it to represent US dollars. If it stores any currency, then `money` is far more accurate.

Consistency across naming conventions is more valuable than the particular style chosen. Teams should follow a shared convention, even one that is slightly flawed, than for each module to follow its own ad-hoc rules. Inconsistent naming disrupts cognitive flow, increases search times, and introduces uncertainty. Consistency allows developers to "guess" names and be correct more often than not, increasing velocity and confidence during navigation and debugging.

Notation where types are embedded in variable names (e.g. `szBuffer`, `ulCount`), should be avoided. It adds clutter, is prone to becoming stale, and often communicates less information than well-chosen prefixes and descriptive identifiers. Moreover, modern tools and IDEs already display type information when needed. The goal of naming is not to encode type, but intent.

In code reviews, naming should be a first-class topic. A good reviewer should ask: Is the purpose of each identifier immediately clear? Does the name reflect the scope and lifespan of the variable? Are similar concepts named similarly across modules? Are abbreviations known and documented? These questions enforce a high standard of communication and encourage developers to write code that is readable even when the context is stripped away. Ultimately, naming conventions serve as a shared vocabulary. They encode domain knowledge, design intent, and architectural boundaries into the very fabric of the code. When applied consistently, they reduce ambiguity, streamline maintenance, and provide an on-boarding path for new developers who can “read the code” in the same way they would read a well-written document.

Data Types and Constants

Especially in programming where memory and processing constraints are strong, the careful selection and use of data types and constants becomes a strategic engineering decision rather than a stylistic choice. This is not merely a matter of coding hygiene but one of correctness, performance, and cross-platform predictability. Errors in type usage can lead to overflow, sign extension issues, misinterpretation of sensor data, protocol mismatch, and system instability. Particularly damaging in environments where determinism and precision are essential.

Knowing your available data types is essential to achieve error-free software. Since all software run on digital hardware, the truly basic data types can be categorised into four categories: *pointers* (to the other three categories), *integers, floating point numbers*, and the rest. The rest might be called *strings*, but can also be called characters or even general memory space. In addition to these, we have machine instructions as equally first-class citizens in the computer's memory but for our purposes, we do not count them as data. Some languages do, but those languages usually belong to the hacker domain rather than the professional one.

Note that an integer is really a misnomer in computing. The `int` data type resembles mathematical integers but has properties very different from those. The most important differences are 1) that real integers reach all the way to infinity (both on the positive and the negative side) while `int` has a very small range depending on the number of bits used to represent the `int` in computer memory, and 2) that an undefined result cannot be adequately represented as an `int` (or at all). And this is only the beginning of the arithmetic blues. Surprisingly, the four basic arithmetic operations (addition, subtraction, multiplication and division) are a frequent source of bugs, even in critical software. One might assume they are trivial to handle since they are mastered early in school. But in computing, the rules shift. Architectures impose strict limits. Numeric representation distorts expectations. Precision is no longer infinite. What appears obvious on paper can fail silently in code.

One of the most important recommendations is the use of fixed-width integer types defined in the `stdint.h` standard header. These types (`uint8_t`, `int16_t`, `uint32_t`, etc.) explicitly declare the number of bits used to represent the variable,

removing any ambiguity tied to compiler- or architecture-specific definitions of `int`, `long`, or `short`. For example, if `int` is 16 bits on one target architecture and 32 bits on another, it can lead to subtle bugs, especially in arithmetic operations, data alignment, and external data interfaces. By enforcing the use of fixed-width types, code becomes robust, portable, and easier to audit.

Avoiding ambiguous native types is particularly important when dealing with data shared across system boundaries. If a structure containing mixed-width or ambiguous types is shared between packages of different origins, the lack of type clarity can result in data misinterpretation. Even a seemingly irrelevant difference in assumed type size or alignment, such as small integers fitting any `int` size, can still cause bitwise logical mismatches or checksum failures. The key is as always that you never know how your code is being used after the next system revision.

Signedness should be explicit. Prefer `uint32_t` over `unsigned int`, not just for the width clarity but also for consistency. Use suffixes like `u`, `U`, `L`, and `UL` when defining numeric constants in macros or code to match the intended type precisely. For example, `1000U` is safer than `1000` when used with unsigned comparisons or when passed to functions expecting unsigned values. This eliminates implicit conversions and warnings (or worse, silent bugs) related to sign mismatches.

Mixing signed and unsigned types in expressions is a known pitfall in C programming. This practice triggers implicit conversions and type promotions that can lead to logic errors. A common example is comparing a `signed int` loop counter with an `unsigned int` size value, which can cause premature loop termination or infinite loops. As a rule, if two variables are to be compared or combined arithmetically, ensure they have the same signedness and ideally the same width. Static analysis tools should be configured to flag such inconsistencies.

Bitwise operations should only be performed on unsigned types. The results of bitwise operations on signed integers are implementation-defined in C, especially when dealing with right shifts or bit masking. To avoid undefined or unexpected behaviour, always cast signed variables to their unsigned counterparts before applying bitwise manipulation. For example:

```
uint32_t mask = 0xFF;
uint32_t result = ((uint32_t)my_value) & mask;
```

Floating-point types should be used with caution. On some platforms, floating-point arithmetic is not hardware-accelerated and incurs a significant performance penalty. Even where hardware support exists, using floating-point operations in real-time control loops or interrupt routines can introduce delays. Moreover, equality comparison with floating-point values is a known source of error due to the inability to represent some decimal fractions exactly. Instead of writing:

```
if (total_sum == 2.0f) { ... }
```

it is mandated to use a small tolerance:

```
if (fabs(total_sum - 2.0f) < 0.001f) { ... }
```

When floating-point values must be checked, functions like `isfinite()`, `isnan()`, and `isinf()` from `<math.h>` should be used to guard against invalid values propagating through control logic. A silent `NaN` or `Inf` can propagate through algorithms and cause erratic or dangerous behaviour. The fact that it is silent means that the execution flow seems correct, but it carries nonsense, not data.

One important way to safeguard against arithmetic arthritis is to replace all critical math operations with macros (or inline counterparts) during development and testing, and then replace them when the software ships. That way, nearly all of the arithmetic risks are cleaned out in the production stage. For example, a floating point number that was supposed to be zero at some point, but due to rounding errors ended up being a small negative number, brings havoc to the `sqrt()` function. An example of such a safeguarding macro set is the following, calling `trace_ifperr()` on error:

```
#define  idiv(x,y) ((y)==0?trace_ifperr(0,(x)==0?1:0):(x)/(y))       // x/0 takes prece
#define _idiv(x,y) ((y)==0?(x)==0?1:trace_ifperr(0,0):(x)/(y))       // x/x takes prece
#define  irem(x,y) ((y)==0?trace_ifperr(1,0):(x)%(y))                // x irem 0 is 0 (
#define  iabs(x)   ((x)==INT_MIN?trace_ifperr(2,INT_MAX):((x)<0?-(x):(x)))   // iabs(-i
#define _iabs(x)   ((x)==INT_MIN?trace_ifperr(2,0):((x)<0?-(x):(x))) // iabs NaN handli
#define  isgn(x)   ((x)<0?-1:((x)?1:0))                              // isgn(x) is 0 fo
#define  fdiv(x,y) ((y)==0.0?(real)trace_ifperr(0,(x)==0.0?1:0):(x)/(y))     // x/0 tak
#define _fdiv(x,y) ((y)==0.0?((x)==0.0?1.0:(real)trace_ifperr(0,0)):(x)/(y)) // x/x tak
#define _frem(x,y) ((y)==0.0?(double)trace_ifperr(1,0):fmod(x,y))    // C fmod is not f
#define _fabs(x)   (real_NaN(x)?(real)trace_ifperr(2,0):fabs(x))     // detect NaN (INF
#define _fsgn(x)   (real_NaN(x)?(real)trace_ifperr(3,0):((x)<0.0?-1.0:((x)>0.0?1.0:0.0)
#define _sqrt(x)   (_fsgn(x)<0.0?((x)>-DMC_EPS?0.0:(real)trace_ifperr(4,0)):sqrt(x)) //
```

Also make good use of the numeric error flags provided by almost all CPUs today. Especially a lot of floating-point errors can be detected this way.

Fixed-point arithmetic is often a better alternative in systems without floating-point hardware.[6] By scaling integer values to represent decimal precision, developers can maintain high performance and determinism while preserving sufficient resolution. The use of fixed-point libraries or macros that wrap the logic and hide the scaling factors is highly recommended.

Constant values, whether magic numbers or configuration values, should never be embedded directly in code. They should be defined in headers or configuration files using descriptive names and documented units. For example, instead of writing:

```
if (total_sum > 100.0f) { ... }
```

use:

```
#define TOTAL_XYZ_THRESHOLD 100.0f
if (total_sum > TOTAL_XYZ_THRESHOLD) { ... }
```

This change improves readability, supports maintainability, and ensures that constants are easy to audit and update across builds. Always include the unit in the name of the constant if there is one to decrease ambiguity for readers.

Enums should be used for symbolic constants, not numeric tuning parameters. They should be defined via `typedef`, use all-uppercase enumerators, and be given explicit starting values where alignment with external systems (e.g. protocol states) is necessary. Unnamed enums or those used as cheap integer aliases should be avoided in favour of properly scoped and documented ones.

The use of `const` is powerful and underused. Declaring constants with `const` instead of `#define` allows type checking, debugger visibility, and scope control. For example:

```
static const uint16_t MAX_RETRIES = 5;
```

This declaration is preferable to `#define MAX_RETRIES 5` because it respects scope boundaries, avoids macro side effects, and integrates better with IDE tooling and type-aware debuggers. Constants that are module-private should always be `static const` to prevent leakage into the global namespace.

[6] Not common in 2023, but definitely still around in 2001 when this was written.

All type usage should be justified and documented. Type choices for every structure, buffer, field, and function parameter should be made deliberately and never by default. The mental discipline of explaining type selections improves design and provides clarity for reviewers. By adopting rigorous practices around data types and constants, developers protect themselves from a class of subtle and dangerous bugs. These choices form the arithmetic and logical backbone of well-designed software. Precision, predictability, and transparency should be the guiding principles.

Data from one type can be made into another type by means of typecasting. Always make sure any intended type casting is expressed explicitly in the code. A common source of error is that implicit typecasting takes place without the programmer realising it. The remedy is twofold: 1) explicit type casting when it is intended, accompanied by a comment on why it is being done. This allows others to follow the reasoning. 2) the compiler and lint program can warn about the rest, and these warnings should be considered errors until vindicated by either explicit type casting or corrected if they were true errors. For example, in C the built-in `sizeof` operator returns the number of bytes that an item occupies. Assume that a snippet of code wants to explore the area around a particular string. The following code fails because of implicit typecasting:

```
for (i = -sizeof(strg); i <= sizeof(strg); i++)
     // do something
```

The intention of the loop was to let the `int i` run over the interval [`-sizeof(strg)`, `sizeof(strg)`] but the result was something quite different because of implicit type casting.

Variables and Scope

Managing variables effectively is a fundamental discipline in programming. While variables are the basic units of data storage, their declaration, scope, lifetime, and mutability have direct implications on software safety, maintainability, and performance. A well-structured approach to variable usage supports error-freenesss.

To begin, all variables should be declared as close as possible to their first use. This practice ensures clarity of intent, confines visibility, and helps the reader correlate variable purpose with its operational context. When declarations are scattered at the top of functions or files, particularly those spanning dozens or hundreds of lines, readers are forced to scroll or context-switch, increasing cognitive load and the potential for misuse.

Every variable should be initialised before use. Uninitialised variables are one of the most dangerous and common classes of bugs. They lead to unpredictable behaviour that can vary between builds, compilers, or memory configurations. The risk is magnified in systems lacking active memory protection, where a misused pointer can silently corrupt critical state. Further, there is often a deceptive kindness when compilers in development mode initialises variables in the background, something that becomes an unpleasant surprise when optimised away.

Global variables are sometimes necessary, but they must be used sparingly and always prefixed with `g_` to clearly indicate their broader visibility. Globals introduce tight coupling between modules, make testing and mocking more difficult, and can easily become unintended side channels for state sharing. Every global should be accompanied by a comment justifying its scope, whether for performance, synchronisation, or persistent state sharing.

Static variables are preferable when a persistent state is needed across function invocations but should not be visible outside the compilation unit. Declaring such variables as `static` ensures internal linkage, supporting encapsulation. For example, in a driver file, a `static uint8_t rx_buffer[BUFFER_SIZE];` serves as a well-contained, reusable resource that cannot be accessed or modified by unrelated modules.

Local variables should always be preferred unless there is a compelling architectural reason for a broader scope. Use the smallest possible scope for any variable. For example, loop counters should not leak into the function body, and error flags used in specific branches should be declared within that branch if the language version and style allow it.

The `const` keyword is essential for safeguarding variables against unintended mutation. Whenever a variable should remain immutable after initialisation, mark it as `const`. This applies to function parameters as well: if a pointer parameter is intended only for input, declaring it as `const` both informs the reader and enforces the contract at compile time. For example:

```
void process_data(const uint8_t *data, size_t length);
```

Mutability must be an explicit design choice, not a default condition. By applying `const` consistently, you gain both documentation and enforcement, reducing the risk of accidental side effects.

The `volatile` keyword is also important. It prevents the compiler from applying optimisations that assume variables do not change unexpectedly. Variables shared with other threads must be marked as `volatile` to ensure the compiler does not cache or reorder accesses. For example:

```
volatile uint8_t system_flags;
```

Failure to use `volatile` correctly can lead to elusive bugs that appear only under specific timing or load conditions. These are often among the hardest to diagnose, as stepping through code with a debugger alters timing and can mask the problem.

Pointer variables require special care. Every pointer should be initialised, and if it does not point to valid memory at declaration, it should be explicitly set to `NULL`. This allows safer conditional checks and avoids dereferencing wild or dangling pointers. Where possible, use restrict qualifiers or clearly document ownership semantics to avoid aliasing and double-free errors.

Variable naming should reinforce semantic intent. The use of prefixes such as `p_` for pointers, `b_` for booleans, `h_` for handles, and `g_` for globals creates a visual map of variable roles. These conventions allow developers to scan function headers or struct declarations and immediately infer variable purposes without needing to cross-reference definitions. For instance:

```
static const uint8_t *p_config_data;
static bool b_ready_to_send;
```

All variable names must avoid leading underscores, which are reserved for system and library use. Names should be no longer than necessary, typically capped at

31 characters for legacy tool compatibility, but long enough to be descriptive. Acronyms within variable names should be capitalised consistently or avoided if project naming policies dictate.

Avoid declaring multiple variables on the same line. While syntactically valid, such declarations reduce clarity and complicate initialisation tracking. Prefer:

```
int status_code;
int retry_count;
```

over:

```
int status_code, retry_count;
```

Variables involved in inter-task communication or shared memory regions must be accompanied by synchronisation logic or documented access policies. Whether through semaphores, atomic operations, or disable-interrupt blocks, the data flow must be predictable and race conditions eliminated.

Unused variables should never be left in the code. They should be removed entirely unless they are reserved for future implementation, in which case a comment such as `// reserved for driver extension` should justify their presence. Compiler warnings should be treated as errors, and the build system should prevent warnings from being ignored.

By applying thoughtful, consistent, and defensively structured variable management, developers can prevent a wide range of defects before they reach runtime. Variables, properly handled, are allies of system clarity and correctness; mishandled, they are liabilities that increase entropy and degrade system behaviour over time.

Functions and Macros

Functions are the building blocks of structured software. They are more than just logical units of behaviour, they are fundamental to maintaining clarity, testability, and determinism. Poor function design contributes to tangled control flows, hard-to-track bugs, and brittleness under change. Macros, on the other hand, are a powerful but dangerous tool in the C language. Used wisely, they simplify code; used poorly, they become sources of hidden behaviour, type errors, and maintenance nightmares.

Each function should serve a single, clearly defined purpose. The function name should reflect this purpose as an action or verb-noun phrase. For example, `init_scratchpad()` clearly conveys that the function performs an initialisation routine specific to a scratchpad module. Avoid general names like `handler()` or `do_work()` which obscure what the function is actually doing. Clarity in naming is the first line of documentation and supports both code navigation and automated tool analysis.

Functions should ideally be no longer than 100 lines. This is not a hard limit, but a strong heuristic. Functions that exceed this length often do too much, hide latent bugs, and resist unit testing. Break long functions into smaller, more focused subroutines, even if these are declared `static` and used only internally. Doing so supports readability, testability, and the reuse of logic. For example, a complex startup routine might be divided into `init_gpio()`, `init_peripherals()`, and `init_communication_stack()`, each encapsulating a subset of responsibilities.

Every function should have a single exit point at the end. This rule is particularly valuable in environments where resource cleanup, critical section exits, or logging must occur before the function returns. While early returns might be acceptable in some coding standards when they improve clarity, single-exit strategies reduce the chance of unfreed resources, dangling locks, or inconsistent states. A good rule of thumb is to only allow early returns up until the first external (to the procedure) resource is used, and that point should be marked with a comment stating “point of no early return”.

Function parameters should be as minimal and explicit as possible. Prefer passing parameters by value when the data size is small (e.g., integers or enums), and by pointer or reference when larger structures or buffers are involved. Use `const` qualifiers on pointer parameters whenever the function does not modify the data. This enforces intent and allows compilers to apply further optimisation and static analysis. For example:

```
void transmit_packet(const uint8_t *p_data, size_t length);
```

Functions returning success or failure should use an explicitly defined return type, such as an enum with values like `RESULT_OK` and `RESULT_FAIL`. Avoid using raw

integers or mixing magic return codes. Clearly define what each return code means and document it in the function's header comment.

Every function should be documented using a consistent format, preferably one that supports automated extraction such as Doxygen. A proper comment block should include a description, parameter list, return value, side effects, and any usage restrictions. This is especially important in public APIs, but private functions benefit as well.

Avoid using functions with implicit side effects unless clearly justified. For example, `read_sensor()` should not also reset error flags or clear a buffer unless this behaviour is documented and expected. Functions with observable side effects should be carefully named to suggest their impact. Predictable behaviour is more valuable than cleverness in function implementation.

Macros should be used with great restraint. They do not perform type checking, are evaluated through textual substitution, and can introduce subtle bugs when improperly defined. If a macro is necessary, always wrap parameters and bodies in parentheses to avoid precedence issues:

```
#define SQUARE(x) ((x) * (x))
```

Never write a macro with a side-effect-prone argument. For example, `SQUARE(x++)` expands to `((x++) * (x++))`, which increments `x` twice with undefined order. These bugs are hard to diagnose and often only appear under specific timing or compiler configurations.

Function-like macros should almost always be replaced with `static inline` functions. These provide the same inlining benefit while also offering full type safety and scoping. For example:

```
static inline uint16_t square(uint16_t x) {
    return x * x;
}
```

Use macros primarily for compile-time constants and conditional compilation, not for behaviour. Constants should be defined with `#define` or better, as `const` variables when appropriate. For example:

```
#define MAX_BUFFER_SIZE 256
```

```
static const uint8_t retry_limit = 3;
```

In conditional compilation, clearly comment the purpose of each condition. Avoid nesting `#ifdef` blocks deeply, and always provide a fallback or default case. For example:

```
#ifdef USE_EXTERNAL_CLOCK
configure_external_clock();
#else
configure_internal_clock();
#endif // USE_EXTERNAL_CLOCK
```

Macros that redefine control flow (e.g., `#define forever for(;;)`) should be avoided entirely. They mislead readers, hide true syntax, and create barriers for new developers. Code that appears clever at first glance often becomes unmaintainable. Variadic macros (those accepting `...`) should only be used when interfacing with logging or diagnostic systems and never for business logic. These macros are difficult to parse, cannot be easily validated by static analysis, and tend to introduce side-channel complexity.

Functions and macros form the interface and behaviour contract of your software. Their design should favour clarity, predictability, and decomposability. The careful use of `inline` functions, well-documented interfaces, and type-safe constructs allows programmers to code with speed and confidence. In contrast, lazy macro usage and unclear control flow create fertile ground for technical debt and latent bugs.

Above all, good function and macro design is not about saving keystrokes. It is about encoding and preserving intent so that the system's behaviour is stable, discoverable, and testable not just today, but for the lifetime of the product.

Control Flow Structures

Control flow is the skeleton of software behaviour. In systems where timing and predictability matter more than elegance, control structures must be both deliberate and auditable. Ambiguity, inconsistency, or excessive cleverness in control flow leads to software that is fragile, difficult to test, and dangerous to modify. Accordingly, best practices in control flow design prioritise simplicity, transparency, and robustness.

The `if-else if-else` construct remains the most fundamental and readable path-selecting construct. Its structure should always be reinforced with explicit braces for each clause, regardless of line count. This removes ambiguity, prevents bugs introduced by unintentional dangling clauses, and supports better formatting and viewing in diff tools (fc in MS Windows) and code reviews. Braces also signal code boundaries clearly, especially when editing or extending logic blocks. The style:

```
if (condition) {
    // action
} else if (other_condition) {
    // other action
} else {
    // fallback
}
```

should be used consistently. Ending a chain with a final `else` block is required, even if the default path only logs an error or calls an assertion. This practice future-proofs the code against undefined branches and highlights developer intent.

Avoid nesting more than three levels of `if` statements. Deep nesting makes code harder to follow and obscures logical relationships. When nesting is unavoidable, consider breaking the logic into helper functions or restructuring the flow to short-circuit early. For instance:

```
if (!valid_input(data)) {
    return ERROR_INVALID;
}
if (!resource_ready()) {
    return ERROR_BUSY;
}
return process_data(data);
```

This flattened structure improves readability, testability, and logical traceability.

`Switch` statements should always include a `default` case. Omitting `default` invites silent failure when inputs fall outside expected ranges, particularly when dealing with enums or protocol message types. Even if the `default` case does nothing or only logs an error, its presence documents completeness. For example:

```
switch (command) {
    case CMD_START:
        start();
        break;
    case CMD_STOP:
        stop();
        break;
    default:
        log_warning("Unknown command: %d", command);
        break;
}
```

Case labels should be aligned vertically and terminated with `break` unless fall-through is intentional and clearly marked with a comment such as `// fall-through`. Accidental fall-through is a notorious bug source, and compilers can be configured to warn on missing `break` statements if annotated properly.

Loop constructs such as `for`, `while`, and `do-while` should be used with care and consistency. Loop counters should be declared within the loop where possible and have clearly bounded ranges. Magic numbers must not be used for limits; instead, define named constants or use the size of relevant arrays or buffers. For example:

```
for (size_t i = 0; i < NUM_SENSORS; i++) {
    read_sensor(i);
}
```

Avoid modifying loop control variables within the loop body. This practice leads to unpredictable flow and makes formal verification and static analysis more difficult. If the loop termination logic is complex, document it with a comment or use clearer constructs such as `break` and `continue` only when justified and documented.

Infinite loops should be written as `for (;;)` and never as `while (1)`. The latter can be mistaken for a conditional check or a typo. Using `for (;;)` is a clear signal to both the compiler and the reader that the loop is intentionally unbounded. When infinite loops are used in firmware tasks or main routines, ensure there is a documented mechanism for escape or reset under fault conditions.

Avoid `goto` under almost all circumstances. While C permits its use, and it can simplify certain cleanup patterns, it also undermines structured flow and introduces hard-to-follow code paths. If `goto` is used for error unwinding in deeply nested logic, all labels should be clearly named, such as `cleanup_resources:` or `exit_with_error:`, and not placed arbitrarily. An example:

```
if (!init_subsystem()) {
    goto exit_with_error;
}
// ...
exit_with_error:
    shutdown_all();
    return ERROR_INIT_FAIL;
```

This limited use is acceptable only if accompanied by detailed commentary and limited to forward jumps within the same function.

Avoid using `continue` unless absolutely necessary. Its use often breaks the linear reading flow of loops and skips over valuable instrumentation such as logging or counters. Where used, it should be accompanied by a comment explaining the early skip condition.

Boolean expressions in control statements should be self-explanatory. Avoid assignments in conditions:

```
if ((status = read_sensor()) == SENSOR_OK) // wrong
```

This may be legal C, but it is confusing. Prefer separating assignment from evaluation:

```
status = read_sensor();
if (status == SENSOR_OK)
```

Place constants on the left side of equality tests to prevent assignment mistakes from compiling silently:

```
if (MAX_RETRIES == retry_count) { ... }
```

If = is accidentally typed instead of ==, the compiler will flag an error, reducing one of the most persistent logic bugs in C programming.

When using `while` loops, ensure they cannot silently degenerate into infinite loops unless that is the explicit goal. Where appropriate, insert timeouts, state checks, or watchdog resets. For example:

```
while (!data_ready() && retry-- > 0) {
    delay_ms(10);
}
```

Such patterns make loop behaviour deterministic and traceable, essential qualities in real-time or safety-critical firmware.

In conclusion, control flow should be structured to be not only correct but also readable, predictable, and maintainable. These structures underpin how the system behaves, reacts to inputs, and recovers from faults. The more deliberate and transparent they are, the more confidence we can place in the system's operation under every conceivable condition.

Interrupts and Multitasking

This section addresses risks particular to real-time programming. Interrupt Service Routines (ISRs) and real-time task management should be engineered with great precision. Improper design or careless implementation in these domains can result in unpredictable system behaviour, data corruption, or unresponsive applications. This section outlines detailed guidelines for handling interrupts and multitasking in a manner that promotes safety, determinism, and maintainability.

All ISRs should be written with minimal latency and determinism in mind. An ISR should do as little work as possible, ideally limited to acknowledging the interrupt source, capturing time-critical state, and setting flags or buffering data for later processing in a lower-priority context. The rationale is twofold: long ISRs block other interrupts from executing (depending on the interrupt controller's design), and

lengthy ISR logic increases worst-case response times for the entire system.

ISRs should be declared `static` to restrict their visibility to the local translation unit. They should also be explicitly marked with compiler attributes that designate their purpose, such as `__attribute__((interrupt))` for GCC-based systems. This informs both the compiler and the reader that the function has special linkage and calling conventions. For example:

```
volatile uint8_t data_ready_flag;
```

Accesses to shared variables should be performed atomically. This means reading or writing data in units that match the processor's word size or using atomic operations if available. Multi-byte data structures should be protected with critical sections such as blocks where interrupts are temporarily disabled or synchronisation primitives like mutexes in a real-time environment.

For multitasking systems, threads should be carefully prioritised and scheduled with predictable timing in mind. Task priorities must reflect system-level criticality. For instance, a watchdog refresh or motor control task must run at higher priority than a user interface updater. Improper priority assignments lead to starvation, jitter, or missed deadlines.

Each task should have a clearly defined role, lifecycle, and termination condition. Threads should not run forever unless they are true background tasks with well-scoped responsibilities. Thread stacks must be sized conservatively and checked during testing to ensure that no overflow occurs, almost regardless of what a user does. Many debuggers provide stack watermarking or usage statistics. Use them to confirm runtime safety margins.

Avoid dynamic memory allocation inside ISRs or real-time tasks. Memory allocation functions (`malloc`, `calloc`, `free`) are typically non-reentrant, unbounded in execution time, and prone to fragmentation. Use statically allocated memory pools or ring buffers instead. Any dynamic structures should be preallocated at system startup and reused via managed pools.

Synchronisation between tasks and ISRs must be deliberate. Real-time operating systems often provide mechanisms like semaphores, event flags, and message queues. These primitives allow ISRs to signal work to tasks without busy-waiting

or polling. For example, an ISR might use `xSemaphoreGiveFromISR()` to notify a task that new data has arrived. The task, blocked on `xSemaphoreTake()`, then wakes up, processes the data, and returns to sleep.

Watchdog timers should be integrated with the task model. Every long-running thread should periodically notify the watchdog timer subsystem that it is alive. This allows the system to reset automatically if a task hangs, a resource deadlocks, or an infinite loop occurs. Ensure the watchdog kick is performed only after all safety conditions are met, not merely on entry to the task.

To defend against stack overflows, make use of memory protection units (MPUs) if available. Configure separate stacks for ISRs and tasks, and instrument the start-up code to detect stack violations. Avoid recursion unless depth can be bounded and justified. Recursive calls consume stack unpredictably and are difficult to test exhaustively.

Interrupt nesting should be handled carefully. Some microcontrollers allow higher-priority interrupts to preempt lower ones. While this increases responsiveness, it also complicates system timing and resource usage. Always document which interrupts are enabled at which priority levels, and avoid deep nesting unless latency constraints require it.

Unused interrupts should not be left unhandled. Instead, install a default ISR that logs the event, disables the offending source, or performs a safe system reset. This practice prevents hangs or faults from unassigned interrupt vectors, which can occur due to misconfiguration or peripheral start-up glitches, something rather frequent in Philips PTS systems.

Finally, all interrupt and multitasking behaviour should be documented in both code and architecture diagrams. Specify which resources are accessed in each ISR, which tasks respond to which signals, and what the expected timing is for each event type. This documentation becomes essential during certification, regression testing, and field servicing.

The challenge of asynchronous behaviour is central to complex system design. Interrupts and multitasking are powerful and necessary tools, but risk-free only

when used with precision and a good understanding of the system's temporal constraints. Mastery of these elements distinguishes stable systems from those that are flaky, intermittent, or dangerously unpredictable.

Modularity and File Structure

Modular design is one of the hallmarks of robust software architectures. Without clear boundaries between components, even well-written code becomes fragile and difficult to evolve. File structure, naming conventions, and dependency control all contribute to a system's long-term maintainability, scalability, and testability.

At its core, a module should encapsulate a single concept or abstraction. Each module should reside in its own pair of files: a header (.h) and a source (.c). These files should share a base name. This simple convention helps tools, IDEs, and humans navigate large codebases with ease.

Header files define the module's public interface. They should contain only what is necessary for other modules to interact with it: typedefs, enums, macro definitions, function declarations, and documentation comments. The implementation details, such as private helper functions, static variables, and internal logic, must remain in the .c file. This distinction ensures encapsulation, a cornerstone of software engineering. By exposing only the interface and hiding the implementation, changes to internal logic do not ripple through dependent modules.

All header files should begin with guards using the `#ifndef`/`#define`/`#endif` pattern or the more modern `#pragma once` where supported. The macro name in traditional guards should be unique and consistent, often constructed from the file path or module name in all caps. These guards prevent multiple inclusion and avoid mysterious compile errors caused by double declarations or recursive header dependencies.

The first include in every .c file should be the module's own header. This practice enforces interface validation. If the header and source become desynchronised, say if a function is declared in the header but renamed in the source, the compiler will catch the mismatch immediately. After the self-include, standard library headers

come next, followed by third-party or project-wide includes, all grouped and separated by blank lines.

Each source file should follow a consistent internal structure. A recommended layout includes:

1. File-level comment block describing the module's purpose
2. `#include` directives
3. Macro definitions
4. `typedef`s and enums
5. Static (private) global variables
6. Forward declarations of static functions
7. Public function implementations
8. Static function implementations

This disciplined order helps readers scan the file efficiently and maintain predictable structures across modules. When every file follows the same format, teams can delegate, audit, and refactor with greater confidence. Avoid defining variables in header files. The use of `extern` declarations in headers is permissible, but only when the symbol is truly shared across multiple modules. And even then, use them with restraint. Define the variable in a single .c file, and include a detailed comment explaining why a global scope is justified. Modules should depend only on what they need. Avoid gratuitous inclusions or circular dependencies between headers. If a module depends on another, include only the header file and never access internal implementation symbols. This supports clean dependency graphs and enables modular compilation, testing, and reuse.

For shared interfaces, such as buffer routines used by multiple sub-systems, introduce interface abstractions. These may take the form of function pointer structures, `typedef`ed callbacks, or abstracted init/config/data-transfer routines. This decoupling supports unit testing and enables mocks to replace real user interaction during simulation.

Public type names, constants, and function names should always be prefixed with the module name. This avoids name collisions in large projects and reinforces ownership. Modularity also applies to configuration. Avoid scattering `#define` constants throughout source files. Instead, centralise configuration into a dedicated

header that groups all tuneable parameters, timeouts, sizes, and hardware-specific mappings. These constants should also be named clearly and grouped by function.

Keep module files focused and atomic. If a single module becomes even slightly bloated, it should be decomposed into sub-modules. The top-level module can aggregate these, offering a unified API while preserving a separation of concerns underneath. Each module should be independently testable. This is easier when modules avoid tight coupling, global state, or architecture dependencies. Use dependency injection, interfaces, or layered design to isolate logic from architecture. Doing so enables the use of host-based unit testing frameworks and simulators.

Document each module clearly at the file level. The header file should include a comment block describing the purpose, usage, assumptions, and any dependencies. Public functions should be documented inline using a consistent format to allow automated documentation generation and serve future maintainers.

In larger projects, define module ownership and code stewardship policies. Who reviews changes to this module? Where are the test plans? What are the architectural dependencies? For example, how many browsers should the JavaScript code run in? These meta-rules ensure that the knowledge embedded in your file structure is preserved and transferred as teams evolve.

In summary, a strong modular structure is the backbone of embedded software quality. It enables abstraction, isolates risk, clarifies design, and supports maintainable growth. While naming, structure, and encapsulation may seem mundane compared to algorithms or driver logic, they provide the navigational and cognitive scaffolding that every programmer depends on, whether they realise it or not. This concludes the guidelines as they were once transformed from proprietary languages at Philips Data Systems to a generic C formulation, which in turn is rather easily mapped onto any imperative language today. The hope is that these guidelines will help foster more 0.1X programmers (see Chapter 2) that the world so desperately needs today. And while explicitly directed towards programmers, there are many hackers that clearly need these PTS guidelines as well. In their circles, the 10X coder is unfortunately much more of a hero than the 0.1X one.

5. The Elephant in the Room

Object-oriented programming (OOP) began as a brilliant and intuitive model for designing software by mirroring real-world entities. The earliest OOP languages, such as Simula in the 1960s, were explicitly created for simulation programs that needed to model complex real-world processes with interacting objects (Dahl and Nygaard, 1966). The idea was that software objects could behave like physical or conceptual entities. They would encapsulate state and expose behaviours, allowing programmers to reason about systems in terms of "actors" that send messages to each other (Kay, 1993). This was a profound shift from the earlier *packaging* paradigm of procedural imperative programming, which, while effective for many tasks, did not inherently lend itself to modelling rich interactions between multiple independent components. It was still imperative as opposed to other coding paradigms like functional and logic programming. If you ever find a claim that OOP is another code paradigm, stay sceptical. It is another way of packaging imperative code; no more, no less.

In the decades that followed, OOP was presented as a way to manage complexity by bundling data with the code that operates on it (encapsulation) and by allowing classification hierarchies through inheritance, so that more specific concepts could be built upon general ones (Gamma et al., 1994). Especially in domains like graphical user interfaces (GUIs) and simulations of real-world systems, this approach felt natural and powerful. A window on the screen could be represented as a window object, containing buttons as objects, all with properties and methods corresponding to their real-world or conceptual counterparts. Early proponents of OOP described it as a way to make code more reusable and modular, enabling a new level of abstraction that could tame large software projects. By the 1990s, OOP had moved from a niche concept to the *de facto* standard in software engineering orthodoxy, often touted as the silver bullet for the software crisis of complexity (Brooks, 1987) and the only, right way of writing programs. Indoctrination started early, students were taught to think in terms of classes and objects as the right way to program, and major languages like C++ and Java made object-oriented design their core organising principle. In documented cases, not following the doctrine made students fail

their exams, even in project courses with no mandated programming language.

However, as with many grand ideas in software, the success of OOP also bred an orthodoxy. A dogmatic mindset that OOP is the only proper way to design software, regardless of context. What began as a set of useful concepts hardened into an almost religious stance: everything in a system should be an object, all code should reside in classes, and alternative approaches were often dismissed as inferior, misguided or outdated. This book argues that OOP's rise to dominance, while historically justified by its early benefits, has led to an overreach. The object-oriented packaging became so deeply ingrained that it started to ignore or even suppress other approaches that in many situations are more appropriate or straightforward. Instead of being one tool among many, OOP was treated by many practitioners and educators as an end in itself, the fundamental pillar of "good" software design. This orthodoxy can be seen in how programming guidelines and textbooks from the late 1990s through the 2010s relentlessly emphasised OOP design patterns, class hierarchies, and "pure" object-oriented design principles (Gamma et al., 1994; Martin, 2008). While these principles have their merits, the dogmatic application of OOP to every problem domain has revealed significant drawbacks. Ironically, some of the very goals OOP set out to achieve, such as reducing complexity and improving maintainability, have been undermined by overzealous object-oriented designs that disregard the strengths of the human mind and better-suited packaging paradigms. Forcing a carpenter to use screwdrivers for both nails and screws does not enhance speed or quality. It slows the process and weakens the result. Allowing the occasional use of a hammer when the task demands it leads to better workmanship with less strain and also to a superior end product.

One of the clearest manifestations of this OOP overreach is the extreme fragmentation of code that often results from rigid adherence to object-oriented "best practices." Under OOP orthodoxy, it became common to decompose systems into dozens or hundreds of tiny classes, each with dozens of trivial methods, in an effort to achieve maximal encapsulation and single-responsibility components. Influential texts like Clean Code admonish programmers to write "small methods" and keep each method focused on a single task, often just a few lines of code (Martin, 2008). The intention behind such guidance is at first sight understandable. Smaller methods

should be easier to verify and reuse, and a method that does only one thing is less likely to have unforeseen side effects. However, taken to extremes, this style leads to codebases where the logic of even simple operations is fragmented across a labyrinthine call graph of many tiny functions. The inherent complexity of software systems is pushed from the code itself to a combination of code lines and a network of method calls, making it harder to follow for a human. The natural flow of a program, the sequence of steps that accomplish a task, is no longer easily visible in one place but is instead spread across numerous class files and method definitions. Each step or subcomponent of an algorithm might reside in its own method, often in a different class, requiring the reader to jump back and forth through multiple layers to understand what is happening. The cognitive burden of such fragmentation can dwarf the benefits of neat subdivisions. Unfortunately, many OOP systems consist of shallow modules. They present a complex surface of many interconnected pieces (interfaces), but each piece does very little, so the actual functionality is severely distributed and hard to grasp in aggregate. In practice, this leads to code that is formally modular (in the sense of many modules), thus passing code reviews, but not honestly modular in the sense of localisation of concern. Instead of one coherent module handling a significant piece of functionality, as in classical procedural imperative programming, you get many shallow modules that collectively handle the functionality, but require understanding all of them to see the whole picture.

To illustrate the problem, consider a common scenario influenced by strict OOP design advice: Suppose we need to implement a routine to process a user's order in an e-commerce system: checking the order items, charging the customer's payment, and then arranging shipment. In a straightforward procedural style, one might write a function `processOrder(order)` that performs these steps in sequence: verify items in stock, calculate totals, charge payment, then schedule a shipment. The logic could be written in a clear, top-down manner, essentially narrating the steps as they occur and as they would be described to anyone asking what the process as a whole does, i.e. how a human best comprehends it. If each sub-task is complex, it might be factored into a helper function (e.g., a function `chargePayment(details)`), but crucially, the structure of the operation is visible in one contiguous block of code. A programmer reading this code can follow the flow much like reading a story: first,

this happens, then that, and so on.

Now contrast this with an object-oriented approach. One might have an `OrderProcessor` class, which uses a `StockVerifier` object, a `PricingCalculator` object, a `PaymentService` class hierarchy, and a `ShipmentScheduler` object. Each of these might have a method like `execute()` or, worse, a generic name like `process()` that is called in sequence. The `OrderProcessor.process()` method might do nothing but call methods on these collaborators: e.g., `verifier.verify(order)`, `calculator.calculateTotal(order)`, `paymentService.charge(order)`, `shipper.schedule(order)`, etc. In isolation, each of those calls appears simple (just one line), and each corresponding class contains a small bit of logic. But to truly understand the end-to-end process, one must open the `StockVerifier` class to see what it does, then the `PricingCalculator`, and so forth. If each of those in turn calls further methods (perhaps `StockVerifier` calls `Inventory.check()` and `AlertService.notifyIfOutOfStock()`), the reader must mentally juggle multiple classes and methods at once. The simple narrative of verify-calculate-charge-ship is lost amid indirections. The natural reading flow of the code is disrupted. Instead of reading one coherent procedure, the poor reader is bounced around the codebase like a pinball, chasing the thread of execution through numerous narrow methods. This is code fragmentation, the breaking of what could be a clear narrative into a dozen pieces scattered across the codebase. Not a problem for the compiler, but for humans.

Such fragmentation runs counter to the cognitive and perceptual strengths of human programmers. Humans, by nature, often find it easier to understand a sequence of events when those events are presented in order, without unnecessary interruptions. Psychological research into program comprehension has shown that programmers construct mental models of code that often include a narrative flow or program model, especially when reading procedural code (Pennington, 1987; Wiedenbeck et al., 1999). This should not come as a surprise to anyone who learned reading as a young kid, where text comprehension crucially depended on some kind of narrative or line of reasoning being built in the reader's mind. A study by Pennington (1987)

found that experienced programmers reading a piece of code tend to form two mental representations. One of the program's control flow (the "program model") and another of the domain semantics or data flow (the "situational model"). Critically, for imperative code, forming the program model (what happens first, then next, and so on) is straightforward because the code's structure directly reflects execution order. In object-oriented code, especially event-driven or highly decoupled OOP code, this is harder. The control flow is often implicit, following pointers from object to object, rather than explicitly laid out. Empirical evidence suggests that novices, in particular, struggle with understanding OOP control flow. Wiedenbeck et al. (1999) compared novice comprehension of object-oriented vs. procedural programs and found that the procedural style led to stronger mental representations of the program's functionality for beginners, whereas the object-oriented style could be more confusing without proper conceptual scaffolding. The novices reading OOP had difficulty following the flow because it was distributed among interacting objects (Wiedenbeck et al., 1999). This indicates that the human mind does not automatically track scattered pieces of a narrative as easily as a single-threaded story.

From a cognitive-load perspective, every time a reader of code has to jump to another function or class, there is a context switch. The reader must recall what was happening in the caller, load the callee's details into mind, and then perhaps jump back, all the while maintaining the high-level goal of the code. Each such jump incurs what cognitive psychologists call extraneous cognitive load, which is mental effort expended not on the inherent problem, but on the structure used to represent the problem (Sweller, 1988). In well-written code, we strive to minimise extraneous cognitive load so that a programmer's mental energy goes into understanding the problem being solved, rather than deciphering the code's tangled organisation. Overly fragmented OOP code, with numerous tiny methods and classes, adds a heavy extraneous load. It forces the programmer to keep track of many more names, more indirections, and more incomplete fragments of logic than would a simpler, more consolidated representation. As an analogy, consider reading a novel where each sentence of a paragraph is on a different page. The reader would have to flip pages constantly to get through a single paragraph's worth of meaning. That cognitive thrashing is similar to what happens when a developer must click in and out of

dozens of definitions to follow a single execution path. The human working memory is limited. As already Miller (1956) observed, we can only hold about 7±2 chunks of information at once. If a piece of code requires you to hold the context of Class A, B, C, method D, E, and F simultaneously to understand it, you are quickly overloading those limits. In contrast, a more linear piece of code, even if longer, can often be digested in a piecemeal fashion: you understand the first part, then move on to the next, and so on, without having to remember too many disconnected pieces at once.

It is important to recognise that this critique of excessive fragmentation is not an indictment of *all* uses of small functions or abstraction. It is about balance and context. The OOP orthodoxy often promotes small methods as an inherent good, citing benefits like reuse and testability. And indeed, small, well-named methods can be very useful for factorising repeated logic or isolating conceptually distinct sub-tasks, but so can procedures as well. The problem arises when the drive for small methods is taken to a dogmatic extreme, divorced from an understanding of human cognitive needs. There comes a point where breaking a procedure into ten pieces does not reduce complexity but instead increases it for the reader, because the cost of assimilating the overall logic outweighs the benefit of each piece's simplicity. This is witnessed by practitioners who have seen "clean" codebases that are theoretically spotless in adherence to principles, yet practically nearly impossible to read or modify due to their sheer decoupling and abstraction. Many OOP designs expose a large set of numerous classes that collectively provide only a very limited amount of functionality. Such designs force the programmer to learn many class interfaces to achieve something that might have been done in a single module otherwise. In other words, the fragmentation taxes the programmer's ability to quickly understand and make changes to the code, energy that could have been used in better ways.

The orthodoxy of "everything is an object" also led to sidelining many traditional programming concepts that were not only perfectly valid but often more suited to certain tasks. One example is the concept of the finite state machine (FSM). FSMs are a time-honoured way to model entities that have a finite number of states and transitions (such as protocols, device controllers, game logic, etc.). In a procedural packaging, implementing an FSM might involve an explicit state variable and a

`switch` statement or lookup table for transitions. This approach centralises the logic of state transitions in one place, which can be very clear for anyone reading the code: you see all possible states and how you can move between them, often in a few dozen lines of straightforward code. However, within a strict OOP mindset, an FSM is often implemented via the state design pattern. One might create separate classes for each state, each class implementing a common interface (for the actions or transitions), and use polymorphism to switch behaviour when the state changes (Gamma et al., 1994). While this is an object-oriented way to model state, it also splinters the FSM's logic into multiple pieces (one for each state class). If you want to understand the whole state machine, you have to open each class and piece together the transitions. The clarity of the single state-transition table is lost. In some cases, the state pattern is elegant, especially if states have significantly different behaviours or complex internal logic, but in many simple cases, it is overkill. Again, using OOP is not black or white. The point is that OOP's insistence on treating the FSM as a set of objects, rather than allowing a simple tabular or procedural representation, will in some cases make the code harder to follow. The OOP-indoctrinated programmer does not have the understanding or experience to make the choice. It is a situation where the "everything is an object" mentality has arguably made things less clear, not more. This is not just a hypothetical scenario. Many developers have encountered code where a simple problem was solved with an unnecessarily elaborate object structure when a few loops or conditionals would have sufficed with greater transparency. In many cases, it is not their own fault. The object design is often mandated from above, by ill-informed managers and misguided organisational policies.

Imperative control flow constructs like loops, conditionals, and straightforward function calls have also been sidelined in favour of an OOP style of indirection and callbacks. In traditional imperative programming, if you want to perform a series of operations, you just write them out in order. If you want to repeat something, you use a loop. If you need to handle two kinds of cases, you use an `if/else` or `switch`. These constructs map directly to how we might describe algorithms in plain language: "Do X for every item; if Y is true, do Z; otherwise, do W." That these three constructs plus procedure calls suffice for any conceivable program was shown by

Böhm and Jacopini (1966) in what is perhaps the most influential paper of all time in computer science. Object-oriented design sometimes encourages replacing these straight-line flows with polymorphic dispatch. For example, rather than a single function that handles multiple cases via conditionals, one might design a class hierarchy where each subclass implements a method in a different way. Thus, choosing the subclass becomes the way to choose the code path (a classic replacement for a switch case is polymorphism). This of course works, but it also means the logic that was once clearly visible as branching in one place is now spread out over multiple class definitions. A reader must know the class hierarchy to know what happens in each case. Similarly, rather than a simple loop, some OOP frameworks encourage an "iterator object" pattern, where an object maintains the iteration state and the client just calls a method like `iterator.next()` repeatedly, again adding a layer of abstraction between the human reader and the actual control flow (which is conceptually just a loop). Inversion of control frameworks (common in UI toolkits and enterprise applications) pushes this to an extreme: the overall flow of the program is managed by a framework (often object-oriented), and the developer provides many small object implementations (handlers, listeners, etc.) that the framework will call at appropriate times. This event-driven and object-heavy approach can make it difficult to trace the sequence of actions because there is no single place in the code where the sequence can be seen. It is orchestrated by the framework at runtime via numerous callbacks. It can be speculated that this architectural style reflects a broader lack of confidence within the field of computer science and that a remedy should be to make the field more "advanced" by unnecessarily complicating matters and obscuring the obvious. The tendency to fragment logic and obscure execution paths could be interpreted as an attempt to appear more sophisticated by introducing unnecessary complexity rather than favouring transparency and understanding, perhaps with a touch of guild thinking. While event-driven design is necessary for interactive programs, a pure OOP implementation can increase indirection. The key observation is that OOP tends to turn explicit control flow into implicit control flow, hidden behind object interactions. This implicitness often hampers program comprehension. Many studies in software engineering have underscored the importance of explicitness for maintainability (Ko et al., 2006).

Another area where OOP's dominance starved out traditional approaches is in concurrency models. Classic concurrent programming relies on constructs like threads, locks, semaphores, and message passing between processes (Dijkstra, 1968; Hoare, 1974). These are lower-level primitives that any programmer dealing with concurrent systems must understand, regardless of the packaging paradigm. Yet, there was a period when OOP was so ascendant that curricula and developers tended to introduce concurrency through an OOP lens. Java, for instance, integrated threads as objects (`Thread` class) and provided synchronised methods, essentially an object-oriented veneer over the concept of a mutex lock. While this in itself is not problematic, many in the OOP camp treated these concurrency primitives as just library features, not fundamental concepts, leading to developers who could use a `synchronised` block but perhaps never learned the broader theory of semaphores or the pitfalls of shared mutable state. OOP does not inherently solve concurrency issues; if anything, shared-object concurrency can be more error-prone due to unexpected aliasing and side effects. Models that move away from "shared state" concurrency, such as the Actor model (Hewitt, 1973) used in Erlang or the communicating sequential processes (CSP) model (Hoare, 1978) that influenced languages like Go, deliberately eschew shared object state in favour of message-passing or isolated processes. Yet OOP orthodoxy, with its emphasis on objects everywhere, often meant that these models were not taught or utilised as widely as they could have been. Instead, large object-oriented systems often attempted to handle concurrency by layering object frameworks on top of threads (e.g., Java's Enterprise Beans had complex object lifecycles partly to deal with threading and distribution, rather than encouraging simpler concurrency models). The result was frequently enormous complexity. Reentrant code had to be carefully written in objects, with numerous design patterns (like double-checked locking, thread-safe singletons, etc.) to cope. If one steps back, many of these problems were the result of trying to stay within an object-oriented worldview while dealing with inherently non-object-oriented problems (like synchronizing access to a shared resource). In simpler terms, a developer thoroughly indoctrinated in OOP might think of first solving a concurrency problem by creating an "object manager" or adding more classes, rather than considering time-honoured, simpler solutions like a semaphore or a producer-consumer queue. Thus, important concepts like semaphores, monitors, or even just the idea of an explicit

state flag, were sometimes dismissed as "low-level" or archaic, when in fact they were the proper tools for certain situations.

This dogma of everything-as-object reached a peak where even data that had no behaviour was wrapped in objects, and actions that did not naturally belong to any object were forced into class methods. Yegge (2006) ridiculed this in *Execution in the Kingdom of Nouns.* In his satirical essay, he describes the Kingdom of Javaland where no one is allowed to use verbs; instead, all actions must be encapsulated inside noun objects (Yegge, 2006). If a task like "taking out the garbage" were to be programmed in such a kingdom, one could not simply write a procedure `takeOutGarbage()`. Instead, one would have to create a `GarbageDisposalManager` class, instantiate a `GarbageBag` object, then call a method like `garbageDisposalManager.execute()`, which internally calls various other noun-objects like `GarbageTransporter` or `DestinationLocator`steve-yegge.blogspot.comsteve-yegge.blogspot.com. The result is an absurdly roundabout way of doing something conceptually simple, all because of a bias that favours nouns (objects) and disfavours verbs (functions). Yegge's critique highlights a real cognitive point: humans think about actions and processes all the time, not just about static things. By overly objectifying every aspect of a program, we risk losing the clarity of describing what the program *does*. It is as if one were required to describe a recipe not as a sequence of steps (verbs), but as a collection of kitchen object interactions (nouns). The cognitive load on the person trying to understand the recipe would skyrocket, and the elegance of a straightforward sequence would vanish. In programming terms, sometimes a function is just the most direct and clear way to represent a piece of logic. There is no benefit in turning it into an object with a single `execute()` method. As Yegge notes, other programming paradigms and languages freely mix nouns and verbs. They let you have first-class functions, for instance, which are just actions that can be passed around. But OOP frowns on free functions or procedural sequences as if they were design failures, rather than simply another tool in the toolbox.

Historically, the everything-is-an-object mantra can be traced back to the influence of languages like Smalltalk, where indeed every value is an object and even control structures are messages sent to objects, and to certain interpretations of what

Kay (the grandfather of OOP) intended. It is ironic, though, that Kay himself lamented the narrow interpretation of OOP in mainstream languages. He said (1993) that he did not have in mind the static, class-heavy OOP of C++/Java when he coined the term object-oriented programming. His vision was more about message-passing, dynamic objects that simulate independent agents. In practice, however, the popularity of Java and C++ in the 1990s cemented a view of OOP that revolved around class hierarchies, rigid type systems, and deep taxonomies of object types for everything. This had the beneficial effect of standardising certain good practices (like encapsulating data), but it also had a stifling effect on the use of plain data structures and procedures where they make sense. In some circles, writing code in a non-OOP style came to be seen as a kind of heresy or at least a mark of poor design and lack of skills. The author has lost consulting assignments when suggesting that an object-oriented approach was not the best for a certain task. Even when other paradigms are clearly a better fit, OOP designs are contrived to solve the problem. We can see this in the way some developers would cast purely functional problems (like mathematical computations or data transformations) into an object form. For instance, creating classes to represent every function or using objects to simulate higher-order functions, instead of simply using actual functions or more appropriate constructs. This dogmatic approach has led to absurd outcomes, where straightforward tasks became bloated with boilerplate code. A striking example often cited in discussions is the enterprise Java era, where something simple like reading a configuration might involve creating a ConfigReaderFactory, a ConfigReader interface, multiple implementation classes, etc. when a five-line code snippet would have done the job. The complexity was justified in the name of flexibility and extensibility. These are important principles but applied dogmatically and aimlessly when no such flexibility was even needed.

Orthodoxy

The human cost of this orthodoxy is evident in large codebases that become opaque to those who must work with them. It is not uncommon to hear experienced programmers complain that in some enterprise environments, to make sense of the code you have to navigate an object jungle, where you are handed not just the banana you

asked for, but the gorilla holding the banana and the entire jungle with it, a vivid metaphor given by Armstrong, the creator of Erlang at Ericsson Telecom (Siebel, 2009, p.220). This captures a key issue: OOP's fondness for interrelated objects can make it very hard to isolate a piece of code. To use or understand one class, you often need to understand its collaborators, and those collaborators' collaborators, and so on. Soon, you have the whole jungle in view. In contrast, he advocated the use of referentially transparent code (as in functional programming), where each piece is self-contained. You give it inputs, it gives you outputs, and there is no hidden state or context (Siebel, 2009). Such code is indeed easier to reuse and reason about, because you do not need the gorilla or the jungle, just the banana. The perspective comes from the world of Erlang, a functional, actor-model language designed for concurrency and reliability, which took almost the opposite approach to complexity. Keep functions small and side-effect-free and processes isolated. It succeeded in domains such as telecom where complexity had to be managed rigorously. It is telling that Armstrong, coming from outside the mainstream OOP camp, observed what many within the OOP camp had come to accept as normal: the heavy coupling and context in OOP systems that impede understanding and reuse, thus being contra-productive.

Not all prominent software thinkers fell into the orthodoxy. There have always been voices in industry and academia cautioning against over-reliance on any one paradigm. Brooks warned that there is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement in productivity (Brooks, 1987, p.10). OOP was often touted as a silver bullet in the late 20th century, yet by Brooks' reasoning it would not be. Indeed, while OOP improved how we model certain problems, it did not eliminate the essential complexity of software. By the mid-2000s, we began to see a more pragmatic shift: the rise of multi-paradigm programming. Languages like Python, for example, treat OOP as optional. One can write procedural scripts or one can define classes, or mix both as needed. C++ (which was originally described by its creator as a multi-paradigm language) allows object-oriented programming but also supports generic programming, template metaprogramming, and plain C-style procedural program-

ming (Stroustrup, 1997). The emergence of functional programming in the mainstream, with the incorporation of functional features in Java and C#, provided alternatives and reminded developers that we can combine paradigms. We do not have to be purists. Purism is the enemy of productivity when it blinds us to the merits of other approaches. Unfortunately, in many corporate and educational settings, OOP had so much mindshare that it effectively became a purist dogma. But gradually, experience showed that a more eclectic approach yields better results.

Cognitively, a more flexible paradigm that lets the problem dictate the solution style is inherently more "human-oriented." Human programmers vary in how they best conceptualise a problem. For some problems, thinking in terms of objects with data and associated behaviours is very natural (especially if the problem domain itself is about entities interacting, like a simulation or a GUI). For other problems, it is easier to think in terms of procedures or transformations (for instance, parsing and processing a data file might be easiest as a pipeline of functions). A rigid insistence on OOP in the latter case would be forcing a round peg into a square hole. The result will likely be code that is both inefficient (because it does not use the most direct solution) and harder for humans to understand (because the form does not match the way we naturally think about that problem). The call for a more human-oriented programming paradigm is essentially a call for open-mindedness and pluralism in software design. It is revolutionary in spirit because it urges a revolt against the one-size-fits-all mentality.

What might this flexible, human-centric approach look like? First, it involves using OOP where it fits naturally but not elsewhere. If we are designing a user interface toolkit, it makes perfect sense to have objects like Window, Button, TextBox, each encapsulating state (properties like size, colour, text content) and behaviour (methods like `draw()`, `onClick()` handlers). The interactions of these objects mirror what happens on screen. OOP shines here by making it easy to handle many similar entities and polymorphically treat them. For example, a framework can hold a list of UIComponent objects and call `draw()` on each without caring if it is a Button or a Checkbox. This is polymorphism used in its sweet spot. Similarly, if modelling an employee management system, having Employee objects, Manager as a subclass,

etc., might be reasonable. Although one should always watch out for over-complicating simple data records with trivial methods. OOP is not at all a bad idea, it is just not a Swiss army pocket knife.

Second, a human-oriented approach would resurrect and integrate those "sidelined" traditional concepts alongside OOP. Finite state machines, for instance, could be elevated as a first-class design tool: one might incorporate a small domain-specific language or use a state table within an otherwise object-oriented program when it makes the behaviour clearer. There is no shame in a good old `switch` statement driving a piece of logic if it is actually the simplest representation. In fact, often a well-structured `switch` (or pattern matching in modern languages) can be far more legible than an intricate object hierarchy, precisely because it is explicit. The same goes for concurrency: instead of burying concurrency control inside objects, a human-centric approach might expose it clearly, e.g. using well-named locks or higher-level concurrency libraries that make the flow obvious (like a `Parallel.ForEach` in C# which let you see the pipeline of data). The everything-is-an-object school might frown at these as being impure, but a pragmatic approach cares that the resulting code is safer, clearer, and more maintainable by humans. We should use plain threads and semaphores when appropriate, or actor frameworks when those are a better abstraction, rather than trying to stay within a single unsuitable paradigm.

Third, a human-oriented paradigm encourages readability and writeability as top concerns, aligning with what Knuth advocated as literate programming (Knuth, 1992). He suggested that code should be written as if it is an essay intended to be read by humans, explaining the logic in a flowing manner. This philosophy is not inherently tied to any one paradigm, but it pushes back on techniques that make code harder to read linearly. If we adhere to the literate programming's spirit, we might choose to write a section of code in an imperative style because it reads like a story, and that is more important than rigidly applying an object abstraction. If not allowed to for policy reasons, we might at least include comments and documentation that highlight the overall flow that OOP obscured. In a sense, a literate or human-oriented program might mix paradigms in the same source file. It could start with some imperative overview, then delegate to an object-oriented module where that makes

sense, and perhaps even use a functional style for a specific calculation, all commented and explained so that a reader is never lost. This stands in contrast to a dogmatic OOP stance where every piece must be a class, and anyone reading has to navigate through the class graph.

Furthermore, education and community norms need to evolve. New programmers should be taught multiple ways to approach a problem, not just object-first-or-not-at-all. There is evidence that starting with a functional or simple procedural approach can actually build a better understanding of algorithmic thinking for novices, before adding the complexity of objects (Wiedenbeck et al., 1999). If object-oriented design is introduced as one paradigm among several, with clear explanations of when it is useful and when it might be overkill, future developers will be less likely to become susceptible to OOP dogma. They will more naturally reach for a simple loop instead of an iterator object when the loop is clearer, or write a straightforward function instead of an entire class when one will do. They will see that sometimes less abstraction is more clarity, a principle that can be as important as any design pattern in achieving maintainable code.

Language designers also have a role. The rigidity of OOP in practice was partly enabled by languages that enforced it. Java, for example, requires every piece of code to be inside a class, even the `main` method. This makes writing a quick script or simple procedural program awkward in Java. Everything has to be wrapped in the ceremony of a class. Python, on the other hand, has always allowed to write code in a script without any class if classes are not needed.[7] Such s shift removes some of the pressure to over-OO-ify a solution. Additionally, multi-paradigm languages like C++ allow programmers to choose different approaches in different parts of the program. Even purely object-oriented languages like Ruby have culturally embraced a more flexible stance: Rubyists often write scripting code without classes, or they use mix-ins and dynamic typing to achieve things that in Java would require a hierarchy. The takeaway for language design is that supporting multiple paradigms and not forcing the programmer's hand can lead to more human-friendly code, at least if combined with sensible organisational policies. It empowers the programmer to use

[7] However, it is not typed in the usual sense, creating other problems that are not the topic of this text.

the right level of abstraction for each task. Being a craft, the programmer should be allowed to use the tools that fit the task at hand.

From a theoretical perspective, one might worry that mixing paradigms leads to chaos. But in practice, many big software systems are already multi-paradigm. The Unix operating system, for instance, is often lauded for its design philosophy (Raymond, 2001): "Write programs that do one thing and do it well; write programs to work together." This is not object-oriented thinking; it is a pipeline (imperative) and composition approach. Yet, Unix is incredibly successful and forms the backbone of modern computing. On the other hand, some software is beautifully designed with objects. Consider GUI frameworks or well-crafted game engines that use OOP patterns appropriately for entities and game objects. The best programmers borrow techniques from everywhere. A database engine might use FSMs and locks internally (procedural and low-level), but present an object-oriented API to user applications. A web application might use a functional style for request handling but object-oriented modelling for the domain entities. The orthodoxy that one paradigm must rule all others is simply not born out of the best practices seen in successful systems. Instead, successful systems sometimes quietly bypass OOP dogma when it is beneficial, by intent or by happenstance. By openly acknowledging this and encouraging it, we can do better at a conscious level rather than as an underground practice.

Let us also address efficiency and performance: OOP orthodoxy not only affects readability, but it can also impact performance. Objects and dynamic dispatch have some overhead, and while usually minor, the insistence on many layers can add up (Sweeney, 2006). Many developers indirectly favour a more data-oriented design, where data is laid out in arrays or structures for efficiency and processed with tight loops (which is a procedural/functional style) rather than as scattered objects. The cognitive justification we have focused on is mirrored by a machine-level justification: sometimes it really is more efficient to not use an object for everything. When you have thousands or millions of entities (think particles in a physics simulation), an array of `struct`s will be much simpler and faster than a hierarchy of tiny objects. OOP orthodoxy tended to dismiss such concerns as something that would be solved by faster hardware or just premature optimisation to worry about. But in truth, good architecture is about balancing all these factors: clarity for humans, and efficiency

for machines. A flexible paradigm choice helps because you might use a simple array for those particles (imperative style) even if the rest of your engine is nicely object-oriented for higher-level entities. There is no betrayal in that; it is using the right tool for the job.

Resolution

The overarching argument is that programming should be centred on human cognition and the problem at hand, not on ideological adherence to a particular model of computation. Object-oriented programming was a means to an end, a powerful set of concepts to help manage complexity. We should celebrate its contributions: the idea of encapsulating state and behaviour, the notion of sending messages between independent objects (which lives on in things like microservices and actor systems), and the rich thinking it inspired in software design (from design patterns to UML modelling). But we should also recognise when the OOP paradigm becomes an orthodoxy that blinds us to practicality. When developers begin to contort solutions to fit an object model rather than asking "What is the simplest, clearest way to solve this problem?", it is a sign that the paradigm is dictating the design, rather than the problem dictating it. Such contortions have real costs: hard-to-read code, slower development (because of all the boilerplate and indirection), and even poorer runtime performance. By contrast, a more open approach encourages clarity. If a piece of code can be written in 20 lines of straightforward, well-commented imperative code, a healthy engineering culture will embrace that, rather than insisting it be rewritten as 5 classes and 200 lines to satisfy some textbook notion of decoupling.

In making these arguments, we ground them not only in anecdotal experience but also in research and historical perspective. Studies in program comprehension (e.g., Pennington, 1987; Wiedenbeck et al., 1999) provide evidence that extreme fragmentation can hurt understanding. Cognitive load theory (Sweller, 1988) supports the idea that more bits of context means more extraneous load on the developer's mind. The history of programming languages shows a pendulum swing: from unstructured assembly to structured programming, from structured programming to object-oriented programming (to handle larger, more complex programs through modularity),

and now from object-only thinking to multi-paradigm pragmatism. We can take lessons from each era. Structured programming taught us the value of clarity and directness. Recall how controversial `goto` was declared harmful because it obscured control flow (Dijkstra, 1968)). That lesson is directly applicable in cautioning against obscured control flow in overly objectified code. OOP taught us the value of modularity and abstraction, but we now learn that too much abstraction can be as harmful as too little, a notion that might seem paradoxical but is well-recognised in other fields. In writing books and papers, too many fancy words can obscure meaning, and in architecture as well as product design, over-abstract designs often worsen human usability.

We would be remiss not to acknowledge that part of the reason OOP orthodoxy took hold was in reaction to genuine problems in earlier code. Before OOP, procedural code in large systems could indeed become spaghetti-like, with global variables and long functions that were hard to reuse or reason about. OOP brought discipline in that it forced developers to bundle data and operations, thus avoiding a lot of global state, and it promoted the DRY (Do not Repeat Yourself) principle through inheritance and polymorphism. These gains are real, and any flexible viewpoint should not throw out the baby with the bathwater. The aim is not to eliminate OOP but to reintegrate OOP into a broader toolset. In a sense, it is to demote OOP from the ruling dictator philosophy to an important approach among several. We can still leverage objects, but we choose them consciously, not by default. The resulting code might look less purely object-oriented, and that is absolutely fine. A single codebase could have some essentially object-oriented modules (with classes and polymorphism), some that are procedural (just a library of functions), and some that are functional (using higher-order functions or closures). Whatever provides the clearest expression, best maintainability and overall stability.

The metric of success for code should be maintainability and comprehensibility by humans (along with secondary metrics like performance, which also often benefit from clarity). If an object-oriented abstraction enhances maintainability, use it. If it hinders it, do not be afraid to break a dogmatic OOP rule. For example, one taboo in OOP is the use of `switch` or type-checking on an object's type because it is seen as betraying polymorphism. But sometimes, a `switch` on a type code or an enum is

actually much simpler and thus effective than a whole polymorphic class hierarchy. An indoctrinated rigid OO thinker might reflexively avoid any `instanceof` checks and create a convoluted pattern instead; a pragmatic thinker will weigh the trade-offs and perhaps decide a little type dispatch in one place is fine if it saves a lot of indirection elsewhere. Similarly, the DRY principle, while generally good, can be taken too far under OOP influence, leading to abstract base classes that unify things that perhaps did not need unifying, just to avoid a few lines of duplication. It may be clearer, for instance, to have two separate code paths (a bit of repetition) than to engineer a one-size-fits-both object hierarchy that actually obscures the differences between the cases. What we seek is to spread that sense of judgment broadly: to everyone, not just the seasoned experts who have learned these lessons through hard experience.

In conclusion, it is time to break the spell of the OOP god. We must remember that object-oriented programming is a means to an end, not an end in itself. Its over-use and misapplication have led to systems that are over-engineered, excessively fragmented, and often adversarial to the way human minds comprehend complex processes. By re-embracing other paradigms, procedural, functional and declarative, and using them alongside OOP, we can create software that is more in tune with both the problem domain and the cognitive abilities of developers. We can have methods as well as functions, objects as well as modules, message-passing as well as shared-memory concurrency, finite state machines as well as class hierarchies, really whatever each specific situation calls for. This book does not claim that OOP has no place; on the contrary, it asserts OOP has a rightful place among the pantheon of programming paradigms. But it firmly rejects the absolutism of "everything must be an object." It calls for a renaissance of flexible thinking in software development, one that prioritises clear communication of intent in code and leverages the full palette of programming techniques available from the past 70+ years of computer science. The ultimate goal is software that is not only correct and efficient but also transparent and straightforward to those who read and maintain it. As Dijkstra and others taught us about structured programming, the clarity of our code is an important concern. We should be able to look at a program and *see* the logic without unnecessary mental gymnastics. By doing so, we align our programming practices

with human strengths: our ability to follow coherent stories, our need to minimise cognitive load, and our capacity to creatively combine ideas to solve problems. It is a call to use objects when they help, but to have the wisdom to step outside the object-oriented mindset when it leads to convoluted, fragmented or obfuscated code. In the end, programming is an endeavour by humans for humans (even when the programs serve machines), and our methodologies must serve us. The deeply ingrained habit of forcing object-oriented programming onto problems that are not object-oriented in nature was ultimately what led the author to leave the programming industry. Being compelled to do the work with screwdrivers alone, even when the task clearly called for a hammer, removed the enjoyment from the craft and left frustration in its place.

References

Abelson, H., & Sussman, G. J. (1985). *Structure and Interpretation of Computer Programs*. MIT Press.

Beck, K. (1999). *Extreme programming explained: Embrace change*. Addison-Wesley.

Beck, K., Beedle, M., van Bennekum, A., et al. (2001). *Manifesto for Agile Software Development*.

Böhm, C., & Jacopini, G. (1966). Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM, 9*(5), 366–371.

Brooks, F. P. (1987). No silver bullet – essence and accidents of software engineering. *Computer*, 20(4), 10–19.

Brooks, F. P. (1975). *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley.

Buxton, J. N., & Randell, B. (Eds.). (1970). *Software engineering techniques: Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969*. Brussels: Scientific Affairs Division, NATO.

CISQ (2020). *The Cost of Poor Software Quality in the US: A 2020 Report* (H. Krasner, Author). CISQ/OMG. Retrieved from https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report/ (accessed 2025-04-23).

Dahl, O.-J., & Nygaard, K. (1966). SIMULA: an ALGOL-based simulation language. *Communications of the ACM, 9*(9), 671–678.

Danielson, M. (1997). *Computational Decision Analysis* (Doctoral dissertation). Royal Institute of Technology, Sweden.

Danielson, M. (2023). *The Rise and Fall of Philips Data Systems*. Sine Metu.

DeMillo, R. A., Lipton, R. J., & Perlis, A. J. (1979). Social processes and proofs of theorems and programs. *Communications of the ACM, 22*(5), 271–280.

Dijkstra, E. W. (1968). *Go To statement considered harmful*. *Communications of the ACM, 11*(3), 147–148.

Dijkstra, E. W. (1971). *A short introduction to the art of programming* (Memo EWD316). Technological University of Eindhoven, The Netherlands.

Dijkstra, E. W. (1970). Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, & C. A. R. Hoare (Eds.), *Structured Programming* (pp. 1–82). London: Academic Press.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gotterbarn, D., Miller, K., & Rogerson, S. (1999). Software Engineering Code of Ethics is approved. *Communications of the ACM, 42*(10), 102–107.

Hamblen, M. (2023, March 15). *Eight lines of code could have saved 346 lives in Boeing 737 MAX crashes, expert says*. Fierce Electronics. Retrieved April 23, 2023, from https://www.fierceelectronics.com/embedded/eight-lines-code-could-have-saved-346-lives-boeing-737-max-crashes-expert-says

Heusser, M. (2012, August 14). *Software Testing Lessons Learned From Knight Capital Fiasco.* CIO. https://www.cio.com/article/286790/software-testing-lessons-learned-from-knight-capital-fiasco.html

Hewitt, C. (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 235–245). Stanford University.

Hoare, C. A. R. (1974). Monitors: An operating system structuring concept. *Communications of the ACM, 17*(10), 549–557.

Hoare, C. A. R. (1984). The emperor's old clothes. *Communications of the ACM, 27*(2), 118–121.

Hotz, R. L. (1999, October 1). *Mars Probe Lost Due to Simple Math Error*. Los Angeles Times. Retrieved April 23, 2023, from https://www.latimes.com/archives/la-xpm-1999-oct-01-mn-17288-story.html

Hunt, A., & Thomas, D. (1999). *The Pragmatic Programmer: From Journeyman to Master*. Reading, MA: Addison-Wesley.

Jones, C. (2011). *Software defect-removal efficiency*. Capers Jones & Associates LLC.

Kay, A. (1993). The early history of Smalltalk. In T. J. Bergin & R. G. Gibson (Eds.), *History of Programming Languages—II* (pp. 511–578). ACM Press.

Kernighan, B. W., & Pike, R. (1984). *The Unix programming environment*. Prentice Hall.

Kernighan, B. W., & Plauger, P. J. (1974). *The Elements of Programming Style.* McGraw-Hill.

Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming language*. Prentice Hall.

Knuth, D. E. (1974). Computer programming as an art. *Communications of the ACM, 17*(12), 667–673.

Knuth, D. E. (1992). *Literate programming*. Stanford University: Center for the Study of Language and Information, CSLI.

Knutson, C., & Carmichael, S. (2008). *Safety first: Avoiding software mishaps*. Embedded Systems Design, 21(10). (Reprinted online at Embedded.com). https://www.embedded.com/safety-first-avoiding-software-mishaps/

Ko, A. J., Myers, B. A., Coblenz, M. J., & Aung, H. H. (2006). An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering, 32*(12), 971–987.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE, 68*(9), 1060–1076.

Leveson, N. G., & Turner, C. S. (1993). An investigation of the Therac-25 accidents. *Computer, 26*(7), 18–41.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

McBreen, P. (2002). *Software Craftsmanship: The New Imperative*. Boston, MA: Addison-Wesley.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review, 63*(2), 81–97.

Naur, P. (1985). Programming as theory building. *Microprocessing and Microprogramming, 15*(5), 253–261.

O'Dell, D. H. (2017). The debugging mindset. *ACM Queue, 15*(1). https://queue.acm.org/detail.cfm?id=3068754

Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks* (Doctoral dissertation, University of Illinois at Urbana-Champaign).

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM, 15*(12), 1053–1058.

Pearce, H., Ahmad, W., Tan, Q., & Javid, Z. (2022). Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Proceedings of the 2022 ACM Conference on Computer and Communications Security (CCS)*, 2041–2055.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology, 19*(3), 295–341.

Pingdom SolarWinds (2009). *10 historical software bugs with extreme consequences*.

Royal Pingdom Tech Blog. Retrieved April 23, 2023, from https://www.pingdom.com/blog/10-historical-software-bugs-with-extreme-consequences/

Polanyi, M. (1966). *The Tacit Dimension*. Garden City, NY: Doubleday.

Raymond, E. S. (2001). *The Cathedral and the Bazaar: Musings on Linux and Open Source*. Sebastopol, CA: O'Reilly. (First ed. 1999)

Sackman, H., Erikson, W. J., & Grant, E. E. (1968). Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM, 11*(1), 3–11.

Sandén, B. I. (2011). *Design of multithreaded software: The entity-life modeling approach*. Wiley-IEEE Computer Society Press.

Siebel, P. (2009). *Coders at Work: Reflections on the Craft of Programming*. Apress.

Stallman, R. M. (1979). *EMACS: The extensible, customizable self-documenting display editor* (AI Memo No. 519). Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

Stroustrup, B. (1997). *The C++ Programming Language* (3rd ed.). Addison-Wesley.

Sweeney, T. (2006). The next mainstream programming language: A game developer's perspective. *Proceedings of the ACM SIGGRAPH 2006 Conference Courses*.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science, 12*(2), 257–285.

Turkle, S. (1984). *The second self: Computers and the human spirit*. Simon & Schuster.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers, 11*(3), 255–282.

Wilson, J. Q., & Kelling, G. L. (1982). Broken windows. *Atlantic Monthly, 249*(3), 29–38.

Yegge, S. (2006). *Execution in the Kingdom of Nouns* [Blog post]. Retrieved from http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html

About the Author

Mats Danielson is a Full Professor in Computer and Systems Sciences at Stockholm University and a Senior Advisor to the President. He is a former Dean of the Faculty of Social Sciences as well as a former Vice President of External Relations, Innovation, and ICT at the university. He has a PhD in Computer and Systems Sciences from the KTH Royal Institute of Technology as well as university degrees in Computer Science and Engineering (from KTH) and in Economics and Business Administration (from Stockholm University). He was working in the software industry for more than 20 years, in the beginning primarily with Philips PTS equipment, before joining academia to work with research as well as algorithm and software design and development within decision analysis and support.

The author spent eight years between high school and university mostly in the software business, and the majority of those years working with PTS computers, peripherals, and software. Having subsequently learned the hard way that other software organisations were not like PTS, and thus not able to produce (almost) error-free software, he took to trying to document, for his own use, why it worked at PTS and then tried to use the same or similar principles in other organisations and settings. The degree to which he could observe error-freeness being achieved in various projects and organisations were roughly proportional to the degree to which they adopted these principles. This is not to say that the principles are the only component required for successful software projects, but it goes quite some way when applied. After more than 20 years in the software industry, the author has since shifted to work in academia but the observations still stand and constitute the basis of this book. The recent emergence of AI/LLM coding tools, such as the GitHub Copilot, does not change the validity of the observations, the conclusions, or the need for guidelines in pursuit of error-free software.

Other books from Sine Metu in the same series

Transcending Business Intelligence, Third Edition, 2022
K. Borking, M. Danielson, G. Davies, L. Ekenberg, J. Idefeldt, A. Larsson
ISBN 978-91-978-4505-2

A Decision-Analytic Manifesto, Second Printing, 2023
L. Ekenberg, M. Danielson
ISBN 978-91-527-5306-4

The Rise and Fall of Philips Data Systems, 2023
M. Danielson, A. Läppinen
ISBN 978-91-527-6233-2

Foundations of Computational Decision Analysis, 2023
M. Danielson
ISBN 978-91-531-0457-5

This book is about how to attain error-free software with small means, essentially adhering to a set of common-sense guidelines. These guidelines originate from Philips' error-free coding culture.