The Craft of Coding

Balancing Creativity and Quality

Mats Danielson[1,2]

[1] Dept. of Computer and Systems Sciences, Stockholm University
PO Box 7003, SE-164 07 Kista, Sweden
mats.danielson@su.se
[2] International Institute for Applied Systems Analysis, IIASA
Schlossplatz 1, AT-2361 Laxenburg, Austria

Abstract: This article examines programming as a craft rather than merely a branch of science or engineering. Tracing the historical tension from early computing, through the institutional push for software engineering discipline, to the resurgence of craftsmanship ideals, it argues that true software quality stems from the skilled application of consistent practices balanced with creative design. The article outlines how low-level standardisation of coding style reduces errors and cognitive load, while high-level software architecture remains an arena for genuine creative judgment. It critiques over-reliance on rigid processes, showing that craftsmanship-oriented cultures yield better software with less bureaucratic overhead. The article discusses the Philips Terminal Systems (PTS) coding guidelines from (Danielson, 2012) as a model for disciplined coding standards, demonstrating their potential to approach near-error-free software development. It highlights the importance of individual responsibility within a supportive community of practice and explains how aesthetic considerations, clarity, simplicity, and coherence, are vital aspects of high-quality code. Integrating influences from computational thinking, agile principles, and craftsmanship philosophy, the article makes the case that good programming must blend strict discipline at the implementation level with freedom and ingenuity at the design level. This dual approach leverages human skill where it matters most and reduces the need for compensatory processes. Ultimately, the craft perspective elevates programming to a form of rational creation: precise, deliberate, yet deeply human in its blend of technical excellence and artistic vision.

Keywords: Software craftsmanship, Coding standards, Software quality, Programming practices, Creative design in software, Error-free development, Software engineering philosophy

1. Introduction

In the early days of computing, programming was often seen through the lens of other disciplines. Sometimes as a mathematical science, other times as an emerging form of engineering. Donald Knuth (1974) declared that computer programming is an art because it applies accumulated knowledge to the world, which requires skill and ingenuity, and especially because it produces objects of beauty (ibid., p. 668). Knuth's use of the term *art* related back to its older meaning of *ars* (skill), likening programming to the art of carpentry or architecture: creative yet grounded in skilled practice. This perspective set the stage for viewing coding as a craft, emphasising individual skill, aesthetic judgement, and the creation of something well-made. At the same time, the academic establishment of computer science sought legitimacy by aligning with science and engineering. By the late 1960s, there was a decades-long conflict over the place of programming in academic computing. The famous 1968 NATO conference on software development introduced the provocative term *software engineering*. A rallying cry to bring rigor and discipline to what some considered a smorgasbord of sloppy practices in programming. This marked a turning point: the field began oscillating between formal, engineering-oriented approaches and the recognition that programming might be an exploratory, craft-like activity not easily regimented by classical engineering methodologies.

By the 1970s, proponents of a scientific and engineering view, such as Dijkstra (1972) and Hoare, advocated for rigorous methods like formal verification and structured programming to tame software complexity. Yet at the 1968–69 NATO conferences, some participants debated whether programming could truly be engineered or whether it was inherently a craft. As projects grew in scale, for example IBM's OS/360 requiring hundreds of person-years of effort, new methodologies, standards, and even academic curricula were introduced to professionalise coding. However, treating programming purely as an engineering problem proved challenging. Unlike traditional engineering artefacts, software is intangible. Its complexity is not constrained by gravity but by complexity itself. The realisation grew that software is a very different kind of creation. It lacks the physicality that allows engineers to test bridges or machines; instead, software's reality is not inherently embedded in space and cannot be fully understood until it executes. Brian Kernighan captured this reality when he observed that controlling complexity is the essence of computer programming (cited in Danielson, 2012). The inherent complexity and invisibility of software make it difficult to manage with upfront formulas or blueprints alone. All these factors led to a growing recognition that programming is as much an empirical, creative craft as it is an applied science or an engineering discipline.

By the 1980s and 1990s, the pendulum swung back toward more practical and human-centric approaches. Techniques like modular design, object-oriented programming, and design patterns gained popularity. Not strictly scientific innovations, but pragmatic ways to manage complexity. Still, many large projects remained bogged down by heavy methodologies, excessive documentation, and rigid processes. In reaction, a movement emphasising agility and craftsmanship emerged in the late 1990s. The Agile Manifesto (Beck et al., 2001) explicitly valued individuals and interactions over processes and tools and working software over comprehensive documentation, a clear pushback against the overly bureaucratic interpretation of software engineering. In essence, the Agile movement reasserted the importance of human adaptability and craft over strict processes. This historical back-and-forth illustrates a fundamental tension in the philosophy of programming: on one hand, the desire for predictable, standardised, assembly-line production of software; on the other, the reality that building software is an exploratory, creative endeavour resistant to too much rigidity. The modern consensus tends toward a hybrid view: certain aspects of software development (version control, testing protocols, and project management) benefit from engineering discipline, whereas at the code face, where programs are actually written, programming remains a creative craft. The best results seem to occur when individual programmers are empowered as craftspeople, taking ownership of their work, within a supportive framework of sound engineering practices. In this light, programming can be understood as a craft in its own right, distinct from but complementary to science and engineering.

2. Software Engineering Practices

The push for software engineering in the 1970s and 1980s introduced important practices to improve software reliability and manage large projects. These included systematic project management techniques, extensive up-front design (often using formal diagrams or specifications), code reviews, testing protocols, and later, process maturity models (such as the SEI CMM) intended to institutionalise quality. Many of these practices were, in a sense, compensatory measures: responses to the high bug rates, cost overruns, and unpredictability of software development often dubbed the software crisis. By imposing process and methodology, organisations hoped to make software construction more like traditional engineering, where following

a well-defined process yields predictable outcomes. Indeed, early definitions of software engineering stressed moving from a *craft* to an *industrial process*, replacing personal idiosyncrasies with repeatable routines.

There is no doubt that engineering practices add valuable rigor. Techniques such as design reviews and formal inspections catch errors that individual programmers might overlook. Test-driven development and quality assurance teams introduce systematic bug detection. Architectural modelling can expose design flaws early. However, these practices also have inherent limitations when the underlying codebase is poorly written or overly complex. Brooks (1975), reflecting on decades of experience, noted that no single development in either technology or management will ever solve all software problems, thus no *silver bullet* exists (Brooks, 1987). In his later work, Brooks emphasised that great designs come from great designers, not from great design processes (Brooks, 2010). In other words, the effectiveness of methodologies is bounded by the skill and insight of the people involved. If programmers are churning out convoluted or inconsistent code, even the best process will struggle to prevent defects or delays. As Brooks observed, our education and training often are not conducive to the development of great designers. The implication is that too heavy a reliance on process can distract from cultivating the very thing that matters most: the individual craftsmanship and judgement that experienced programmers bring to a project.

Tedre's (2014) analysis of computing's history reinforces this point. He describes how the early push for software engineering created an easily identifiable sticking point in debates about the nature of computing. Many practitioners felt that what they do is closer to engineering than to mathematics or natural sciences. But others cautioned that slapping a fancy name on what was still a collection of ad-hoc practices could be counterproductive. In practice, the first decades of software engineering saw many proposed cures, from structured programming to formal specification languages, that were only partially adopted because they were often cumbersome. For example, formal verification of programs (proving correctness mathematically) is powerful in theory but has proved impractical on a broad scale. As a result, it has remained limited to specialised critical systems. Heavyweight documentation and bureaucratic change control made sense for life-critical aerospace software, but in other domains, they slowed progress without obviously improving quality. Over time, even staunch advocates of methodology recognised that human factors dominate software outcomes. A widely cited study by Sackman et al. (1968) had earlier revealed order-of-magnitude differences in productivity and error rates between programmers, even using the same tools. This constitutes evidence that individual talent and discipline far outweigh process in determining project success. The Peopleware studies by DeMarco and Lister (1987) similarly showed that the best programmers outperform average ones dramatically, suggesting that selecting and nurturing skilled individuals is more effective than imposing ever more rules.

The risk of *over-reliance on process* is what might be called the bureaucratic paradox: teams can become so consumed with following procedures (writing extensive design documents, convening frequent status meetings, adhering to rigid phase gates) that they lose sight of the actual software quality. If the code itself is poor, no amount of process will rescue the project. In fact, process can create a false sense of security. Teams might think that *"We followed the checklist, so the software must be good,"* even as the codebase deteriorates under the weight of fulfilled processes. As Tedre (2014) notes, the early notion of software engineering was just a controversial name with little content and its real success was in getting people to acknowledge the problems, not necessarily in providing solutions. Over time, many organisations quietly backpedalled from the strictest interpretations of engineering discipline, especially as the Agile

movement demonstrated that lightweight, people-centred approaches could deliver quality software faster.

Agile methodologies themselves can be seen as an attempt to find a middle ground. Practices like daily stand-ups, user stories, and iterative development offer *just enough* structure to keep projects on track without stifling developers. Yet, even Agile can be interpreted shallowly. Merely doing Scrum ceremonies will not yield any clean code, which is why the Agile Manifesto's signatories later embraced the idea of *software craftsmanship* as a guiding value (Martin et al., 2008). The Manifesto for Software Craftsmanship explicitly extends Agile by valuing well-crafted software over mere working software and a community of professionals over just individuals and interactions. This was a recognition that the quality of implementation matters greatly. Agile's emphasis on working code could, if misapplied, encourage cutting corners ("if it works, ship it"). The craftsmanship movement countered that by insisting that the working code should also be clean, elegant, and maintainable, the signs of a crafted artefact.

In summary, decades of experience have shown the limits of a purely process-driven mentality. Software engineering practices are necessary but not sufficient. They provide scaffolding but inside that framework, the *code* must still be written by humans exercising skill and judgement. If those humans are not well-equipped in the craft of coding, the resulting software will require even more compensatory process (extensive testing, endless bug fixes, refactoring cycles, etc.) to reach acceptable quality. As one commentator put it, *great software is made by great programmers, not by managers, sales departments, processes or tools alone*. Recognising the centrality of craftsmanship does not mean abandoning all process. Rather, it means ensuring that process serves to amplify developer skill, not substitute for it.

3. Software Craftsmanship

By the turn of the 21st century, the notion of programming as a craft experienced a renaissance. Two influential books, *The Pragmatic Programmer* by Hunt and Thomas (1999) and *Software Craftsmanship* by McBreen (2001), explicitly positioned software development as the heir to medieval guild traditions. They drew metaphors from apprenticeship, journeymanship, and master-level artisanship. The software craftsmanship movement they inspired argued that developers should see themselves not as assembly-line workers nor as research scientists, but as craftspeople who take pride in their work and continuously hone their skills. This movement gained further momentum with the publication of Martin's *Clean Code: A Handbook of Agile Software Craftsmanship* (2008), and culminated in the aforementioned Software Craftsmanship Manifesto (2009). In these works and forums, one finds a strong rejection of the view that programmers are interchangeable cogs in a process. Instead, each programmer is an artisan responsible for the quality of the code they produce. The Craftsmanship Manifesto's values not only working software but also well-crafted software, and incrementally adding value rather than just reacting to change. That reflects a commitment to excellence and professionalism in coding.

One of the earliest and clearest articulations of the craftsmanship viewpoint appears in *The Pragmatic Programmer*. Hunt and Thomas (1999) open enthusiastically by stating that programming is a craft and that as a programmer, you work small miracles every day. They encourage developers to embrace the role of problem-solving artisan. Part architect, part builder, and part diagnostician. Hunt and Thomas emphasise pragmatism over dogma. There is no one best way to program, only context-sensitive judgement. They advise against being wedded to any particular tool or language, recommending a broad toolkit and continuous learning. This pragmatic mindset resonates with craftsmanship, it is about cultivating *skill and adaptability*.

This means paying attention to details, continuously reflecting on one's work, and striving for excellence even when the schedule is tight. The authors explicitly challenge the notion that large teams preclude individuality. In a section comparing software to medieval cathedral-building, they state that the construction of software should be an engineering discipline, which does not preclude individual craftsmanship. They describe how the great cathedrals took decades or more and involved thousands of workers. Yet the quality and character of the result owed much to the personal skills of carpenters, stonecutters, and glassworkers. These are all craftspeople interpreting engineering requirements to produce a whole that transcended the purely mechanical. The builders' mindset was captured in the so-called quarry-worker's creed: *"We who cut mere stones must always be envisioning cathedrals."* This analogy is used to argue that even within a disciplined engineering project, individual creativity and pride in workmanship are paramount. The lesson for software: a development methodology can provide the blueprint and requirements, but it is the developers' craft that determines whether the final system is elegant and robust or merely adequate.

McBreen (2001) similarly concludes that software developers need not see themselves as part of the engineering establishment; instead, they can adopt a craft model. He argues that the software engineering paradigm, with its emphasis on process and predictability, often fails because it undervalues the human, creative aspect of writing software. As an alternative, he proposes the craft model, focused on people and their skills. This model encourages apprenticeship (experts mentoring less experienced developers), an emphasis on personal responsibility for code quality, and a culture of sharing tacit knowledge. One concrete example he gives is the idea of masterpieces: just as apprentice carpenters or blacksmiths would create a masterpiece to demonstrate their skill, programmers can refine their craft by working on pet projects or open-source contributions to showcase clean, well-crafted code. The underlying philosophy is that quality software emerges from skilled individuals practicing their craft with devotion, rather than from rigorous conformance to process alone.

The craftsmanship viewpoint also brought a renewed appreciation for aesthetics in code. This is not about superficial beauty, but about internal qualities of code that experienced programmers recognise and value: clarity, simplicity, consistency, and adaptability. A common refrain in craftsmanship literature is that code should read well, i.e. it should be easily understood by other humans, not just executable by a machine. Such code has an aesthetic of proportion and coherence: it does no more or less than what is needed, and every part fits together meaningfully. This idea of *code aesthetics* ties directly into the concept of craft. Just as a well-crafted chair is not only sturdy but also graceful in its proportions, well-crafted software is not only correct but also exhibits a kind of elegant simplicity. The experienced programmer develops a sense of taste for what good code looks like. These judgements are often subtle and cannot be fully captured by formal rules; they are perceived and refined through practice and critique. This is why practices like code review and pair programming are emphasised in the craftsmanship community: they serve not only to catch bugs but to spread knowledge of good style and to cultivate a collective sense of quality.

It is important to note that the craftsmanship approach does not reject the importance of correctness and rigorous thinking. On the contrary, it insists that quality and correctness are dual goals. As Danielson (2012) observes, code can appear clean while hiding subtle bugs, so style is no substitute for soundness. The craftsperson's responsibility is to write code that *both* reads well to humans *and* runs correctly on machines. Achieving this demands a high level of discipline and self-awareness, far beyond just making the code compile. The programmer must play the role of both author and editor, both dramatist and critic. That kind of mindful practice

is what raises programming to the level of a craft, as opposed to just the clerical task of writing instructions.

In summary, the software craftsmanship viewpoint reframes programming as an endeavour where continuous learning, personal responsibility, and pride in workmanship are paramount. It borrows culture from the tradition of apprentices, journeymen, and masters to highlight mentorship and community in skill-building. It also reintroduced ethical and professional dimensions: if you think of yourself as a craftsperson, you are more likely to care about the impact and longevity of your code (just as a master builder cares that a bridge stands for generations). The movement thus shifted focus away from following rigid processes toward cultivating excellence in the act of coding itself. However, this raises an important question: if every developer is an individual craftsperson, does that imply complete freedom to code as one pleases? And if so, how do we avoid the chaos of a project where each contributor follows their own style or conventions? The next section examines this tension between individual creativity and the need for consistency in collaborative software projects.

4. Individual Style vs. Team Consistency

A common critique of the coding-as-craft mentality is that it might encourage a free-for-all approach to style and technique. If each programmer is a unique artisan, will a team of programmers produce a patchwork quilt of differing styles, idioms, and solutions? In practice, many projects have indeed suffered from what we might call the hotchpotch problem: a lack of consistency in coding style and architecture that leads to a confusing, hard-to-maintain codebase. Newcomers to such a project often experience a kind of whiplash as they move from module to module. It feels like each was written in a different era or by a different culture. Inconsistency can manifest in many ways: varying naming conventions, different approaches to error handling, incompatible formatting or indentation, divergent patterns for similar tasks, and so on. The result is increased cognitive load in understanding the system and a higher likelihood of mistakes, since assumptions made in one part of the code may not hold in another.

Ousterhout (2018) underscores the importance of consistency as a means to combat complexity. He argues that consistency is a powerful tool for reducing the complexity of a system and making its behaviour more obvious. When similar things are done in similar ways throughout a codebase, developers can transfer their knowledge from one context to another, leveraging cognitive leverage. Conversely, if each part of the system uses a different pattern, developers must learn about each situation separately, greatly increasing the time and mental effort needed to work with the code. Moreover, inconsistency breeds bugs. A developer may see a familiar-looking piece of code and assume it follows the same rules as elsewhere, only to be tripped up if it does not. Ousterhout points out that a consistent system lets developers make safe assumptions, whereas an inconsistent one is full of pitfalls. In short, consistency allows developers to work more quickly and with fewer mistakes.

Interestingly, the very authors who championed software craftsmanship also acknowledged the need for uniformity in a team. Hunt and Thomas (1999) discuss the idea that large projects require shared standards. They note that some people worry that there is no room for individuality on large teams and quote the objection that software construction is an engineering discipline that breaks down if individual team members make decisions for themselves. Their response is the cathedral metaphor, but it is important to consider their point. They are not advocating an anarchic approach. They accept that software construction should partly be an engineering discipline and that lessons learned should be passed down (which is essentially what a

coding standard does). Their emphasis is that within an overall discipline, individual craftsmanship can flourish. They illustrate this by how medieval builders all followed architectural plans (a form of standard), yet each craftsman's contribution added value beyond the plan. They also explicitly encourage developers to document and share their conventions. In fact, they suggest writing down your team's style conventions and using tools to enforce them. They further advise that if you do not have a clear idea of how you'll comment or format code, it is easy to end up doing it inconsistently or not at all. This sentiment aligns well with Ousterhout's (2018) advice is to create a document that lists the most important overall conventions, such as coding style guidelines, and encourage the programmers to review it periodically. The best way to enforce conventions is to use tools (e.g. linters and formatters) that enforce them. In essence, some of the craftsmanship literature does not oppose having standards; rather, it opposes mindless adherence to standards that make no sense. A well-crafted project can definitely have a consistent style.

Nonetheless, in practice, many teams that adopt a craftsmanship viewpoint initially fail to establish strong shared guidelines. This can happen because each developer feels a sense of ownership and may bristle at being told how to code. There is a psychological aspect: telling a proud craftsperson to adhere to a mundane style guide might seem to diminish their creative input. Additionally, the early craftsmanship movement, in stressing individual responsibility, sometimes ignored the value of uniformity. The result has been seen in some agile teams: lots of personal experimentation, which is good, but insufficient coordination, which is bad. For instance, one developer might favour a more functional programming style, another object-oriented patterns. One likes very descriptive variable names, another uses terse abbreviations. One thinks exceptions should be used for all error handling, another insists on error codes for minor errors. If not reconciled, these differences turn a codebase into a jarring mosaic. The maintenance cost of such inconsistency can be enormous. Imagine a new developer joining. To fix a bug, he or she might have to learn multiple idiomatic sub-languages within the same codebase. In worse cases, inconsistencies lead to bugs when code written with one assumption interacts with code written under a different assumption.

Academic research and industry experience alike confirm that consistency correlates with quality. A study by Buse and Weimer (2010) found that codebases with consistent naming and styling have fewer errors per line of code, even when controlling for other factors. Large tech companies like Google and Microsoft enforce strict style guides and attribute part of their success in managing gigantic code repositories to these conventions. Style guides make code easier to read and can reduce some kinds of errors. Inconsistent code, on the other hand, stands out during reviews, which ironically can be a mechanism to catch issues if done properly.

Thus, the key challenge is how to reconcile individual creativity with project-wide consistency. The philosophy emerging in recent years, and the one advocated by Danielson (2012), is that different levels of abstraction warrant different degrees of freedom. At the low level of code (syntax, basic structures, or common patterns), strict consistency and even standardisation yield big benefits. At higher levels (design, architecture, and algorithms), creativity and individual judgement should be encouraged. This approach tries to get the best of both worlds: consistency where it matters for understanding and integration, and freedom where it matters for innovation and optimal solutions. In the next section, we delve into an exemplar of this philosophy, the PTS coding guidelines, which aim to strictly standardise the lower-level coding practices while leaving room for true craft in higher-level composition.

5. The PTS Guidelines for Error-Free Software

Danielson (2012) presents a compelling case for rigorous standardisation of coding practices at the lower levels of software implementation. Central to his argument are the so-called Philips Terminal Systems (PTS) guidelines, a set of coding standards originally developed at Philips in the 1970s and 1980s, aimed at achieving ultra-reliable software for high-profile banking systems. Sandén (2011) recounts that software projects at Philips produced mostly error-free software, a level of quality so high that other divisions and companies struggled to replicate it. The PTS guidelines thus serve as a concrete example that error rates can be dramatically reduced by enforcing a disciplined, uniform coding style.

What exactly do these PTS (or similar) guidelines entail? They are essentially a collection of best practices and prohibitions that eliminate known sources of bugs and ambiguities in code. Many of the rules might seem pedantic or overly restrictive to a casual programmer, yet each has a rationale grounded in reliability. For example, a classic PTS rule is to always enclose control flow blocks in braces, even if they contain only a single statement. This prevents the well-known bug where a dangling `else` or a mis-indented line can alter program logic. Such a bug famously caused Apple's "`goto` fail" security flaw and other catastrophic failures in the past. Another rule: limit line length to 80 characters. This is not about cosmetic preferences; it is about ensuring that code fits within a developer's field of view, making it easier to grasp and review. If a single line of code contains more logic than a developer can comprehend in one glance, it is too complex. Similarly, keeping functions short forces clear abstraction and reduces complexity. Other guidelines include avoiding dangerous language features and extensions, such as C macros that redefine syntax (e.g. `#define begin {`), which can confuse and mislead. In essence, these guidelines force a uniform structure and clarity in every part of the code.

Importantly, Danielson argues that such low-level uniformity *frees developers to focus on higher-level design*. By using a consistent grammar for code, mental energy is not wasted on deciphering each other's idiosyncrasies. The code becomes more transparent. By enforcing consistency and disallowing known dangerous practices, the code becomes more readable and less error-prone. A trivial example is banning magic numbers. Every constant should be named, which prevents misunderstandings of what a literal value means and ensures that changing it in one place updates it everywhere. Another is outlawing certain C library functions known to be unsafe (like `gets()` which can overflow buffers). Each rule addresses a class of common bugs or maintenance headaches. By removing these pitfalls, the standard guides programmers towards safer patterns. Over time, adherence to these rules inculcates a habit of defensive programming. For instance, always checking the return value of system calls, and always handling the error cases, so that omissions that frequently cause errors are systematically avoided.

The effectiveness of such strict guidelines is borne out by experience. Danielson (2012) notes that organisations that rigorously apply such standards have observed reductions in bug density and debugging time. Consistency also makes code reviews far more efficient: if everyone writes code the same way, then any deviation stands out and can be examined to determine if it is a bug or a necessary special case. The codebase becomes self-documenting to some extent because there is a conventional way to do everything. This is akin to having a coding language dialect that everyone on the team is fluent in. When you switch contexts or read someone else's module, you encounter no surprises in *how* the code is structured, letting you focus on *what* it is doing. One might think that imposing so many rules would slow developers down or suppress creativity. In practice, after a short learning curve, it tends to speed up development,

because fewer bugs are introduced that later need fixing, and developers spend less time reformatting or refactoring each other's code for clarity. It also reduces arguments over style in code reviews. The standard settles those debates upfront.

Danielson (2012) goes so far as to tie this into the promise of *error-free software*. While no non-trivial software can be literally error-free, he argues that by eliminating the *trivial bugs* (the ones caused by typos, oversights, and well-known pitfalls), developers and testers can concentrate on the more complex logic or integration issues. Indeed, he suggests that a sufficiently rigorous standard can eliminate a sizable fraction of bugs, bringing the software closer to zero-defect quality. Notably, this philosophy mirrors approaches in safety-critical domains. For instance, Holzmann's Power of Ten rules for NASA (Holzmann, 2006) likewise mandate simple control flows (no `goto`, limited recursion), fixed loop bounds, limited pointer use, etc., to enable static analysis and prevent hard-to-detect errors. Several of Holzmann's rules are directly analogous to PTS. For example, no function should be longer than what can fit on one printed page (about 60 lines in this case). The fact that independent groups arrived at similar rules underscores their validity. In both cases (PTS and NASA rules), the underlying theme is that discipline in low-level coding saves you from many mistakes that otherwise require heavy downstream processes to catch. It is telling that NASA found these rules to lessen the burden on developers and lead to better code clarity and analysability. In other words, if you write code in a simple, standardised way, you need less complex tooling and heroic late-night debugging efforts to ensure correctness.

Danielson's perspective is that adopting strict guidelines is not antithetical to craft, but rather enables true craft at a higher level. 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. Just as an expert cabinetmaker still follows fundamental rules of joinery and measurement, however creative their overall design is, a master programmer should follow fundamental rules of coding style and structure. By doing so, the programmer's mind is freed to focus on design, algorithmic innovation, and solving the problems at hand, rather than constantly dealing with avoidable errors. The lower-level standardisation thus elevates the playing field for everyone. Junior developers learn good habits from the start and senior developers can operate with the confidence that basic pitfalls are handled effectively.

It is worth addressing a potential misconception: does rigid standardisation turn programming into a rote activity, devoid of creativity? No, the artistic freedom lies at the conceptual level, not the code level. This distinction is important. The conceptual level includes the system's architecture, the choice of algorithms, and the overall approach to meeting requirements. In these realms, creativity and judgement are not only allowed but encouraged. By contrast, the code level (such as how a loop or conditional is written, how variables are named, and how files are organised) should hew to consistency because it is in these minute details that personal quirks can unnecessarily introduce errors. Far from stifling creativity, this separation of concerns *enhances* it. Developers channel their ingenuity into designing better modules and systems, rather than inventing yet another way to denote a loop or a string buffer. Programming is a creative and at the same time scholarly pursuit that blends precision and ingenuity, and it combines beauty with utility and creativity with discipline (Danielson, 2012). True craft is about reconciling these apparent opposites of being creative and disciplined. The PTS guidelines exemplify the discipline part, while a well-designed software architecture exemplifies the creative part. It should be noted, though, that a software architect often means the one selecting the third-party vendor includes rather than devising the overall structure of a complex program.

To illustrate, consider this scenario: a team is building a complex banking application. At a high level, they need to invent algorithms for fraud detection. This is complex, creative work requiring deep thought, experimentation, and insight. Now, when it comes to implementing these algorithms, say, in C++, the team might each do it their own way. But if one developer writes very terse, clever but unorthodox code, and another writes verbose, differently styled code, debugging integration could be very painful. Instead, if both follow a common set of ground rules (for example how to handle error cases, how to format the code, and which language features to avoid), then they can read each other's code with ease and spot any logical issues faster. The creativity lies in devising the fraud detection method; there is no loss of creativity in writing its code clearly according to agreed conventions. In fact, one might argue that imposing some constraints at the coding level can spur creativity at the design level. By limiting oneself to simple constructs and clear patterns, one is forced to solve problems in a more straightforward, thoughtful way instead of falling back on obscurities or shortcuts that get it working at the cost of clarity. With no comparisons about occupations, *the form of a poem does not kill the poet*.

By standardising the lower levels of coding strictly, we create a solid foundation upon which skilled developers can exercise their craft in the higher levels of abstraction. Much like how jazz musicians practice scales and chord progressions (rigid patterns) so that they can improvise beautifully at the structural level of music. The next section will delve deeper into this idea of directing craft to the higher-level composition and architecture of software, and explore how that interacts with and benefits from lower-level uniformity.

6. Craft at Higher Levels

If following strict coding standards handles the hygiene of software, preventing common errors and ensuring consistency, where does the craftsperson truly shine? It is in the higher-level composition and architecture of code that *real* skill and creativity emerge. This is where programmers make design decisions that affect the modularity, extensibility, and performance of the system. It is where they choose how to break a problem into components, how those components communicate, and how to manage state and complexity over the life of the program. These tasks are less about obeying predefined rules and more about applying experience, intuition, and cleverness to create a well-structured solution. Within the structure of a software project, there is ample room for individual skill and judgement. The difference is that this individual judgement is exercised within a culture of shared lower-level discipline.

A useful analogy is architecture in building construction. All builders follow basic building codes (for safety, etc., but also for the environment), but not all buildings are equal, far from it. The architect's vision and the master builder's execution can produce a cathedral or a hut. In software, this translates to high-level structure: how responsibilities are divided among modules or services, what design patterns are employed, where data lives and how it flows. These decisions require creativity and skill because there are many possible ways to design a system, each with its trade-offs. For example, deciding whether to use a microservice architecture versus a monolithic design is a high-level choice that impacts everything from team organisation to failure handling. There is no one standard for such decisions. Rather, the software craftsperson must weigh principles such as low coupling and high cohesion against practical needs like development speed and performance in each given context. And there is also the issue of maintainability, the expected future effort that will go into modifications of the code.

By handling low-level consistency, we reduce the *noise* and can better see the *signal* at the design level. When code is uniformly written, architectural flaws become more apparent because they are not obscured by tangles of inconsistency. For instance, if every module follows the same style, one can notice more easily if a module is doing too much and thus violates single-responsibility principles. The craft at the high level might involve refactoring a system to have deeper modules (à la Ousterhout's recommendation that modules should be *deep*, meaning a simple interface with powerful functionality behind it) rather than shallow ones. It could involve designing clear and minimal interfaces so that the coupling between modules is reduced. These are creative choices that require knowledge and foresight.

Ousterhout (2018), while advocating consistency at the code style level, focuses on high-level design principles: tactical vs. strategic programming, modularity, deep modules, information hiding, etc. He advises developers to adopt a strategic programming mindset where working code is not enough. One must produce a good design that will stand the test of time. This means investing effort now (such as simplifying an interface or refactoring duplicated code into a reusable module) to reap benefits later. Ousterhout suggests the primary goal is producing this good artefact, which aligns exactly with where the craft lies: in the design. That sentiment echoes the craftsperson's viewpoint: care about the quality of the solution, not just that it passes the tests today. A pragmatic craftsman does not code just to get the immediate job done but to create a maintainable piece of software that will continue to serve its purpose well.

Let us briefly consider an example of a high-level craft: an error-handling strategy. At the coding level, one rule might be to check every function's return value for errors. That's a good low-level rule. But at a higher level, a designer might decide whether to use a centralised error handling module with error codes or use exceptions across the architecture. Craft is in designing that strategy coherently across the system. Another example: designing for concurrency. The craft might be in choosing a message-passing architecture vs. shared memory locks, or deciding whether to use an actor model. These are decisions that drastically affect complexity and robustness, and a misjudgement can be very costly down the line. Coding standards cannot dictate those choices. They come from understanding the problem domain and the implications of different approaches. Skilled programmers draw on concepts that worked previously to inform these decisions. This is where we see the blend of science, engineering, and craft: knowledge of computer science (e.g. which algorithms are optimal, or which concurrency models exist) and engineering heuristics (e.g. do not introduce more threads than the hardware can handle, or ensure fault isolation) combine with a creative mindset to architect a solution.

It is at this level that programming truly becomes a craft of composition. As Naur argued already in (1985): what a programmer really develops is a *theory* of how the program works, an understanding in their head that connects requirements to code. A team needs to share enough theory to work together effectively. High-level design discussions and documents (when done right) serve to build this shared theory. The craftsperson uses whatever tools will help, from informal diagrams to prototypes, to ensure the team has a coherent vision. Notably, if low-level code is standardised, the team does not have to waste time theorising about *how* someone's code does something (since it is written in the standard way); they can focus on *why* the code is organised that way and whether it is the right approach.

Another philosophical position comes from Tedre's (2014) reflection on computing as a discipline. He notes that computing borrows tools from both science and engineering, yet transcends both, making it a craft in a true sense while requiring experience and yielding outcomes judged by elegance and maintainability as well as correctness. Elegance and maintainability are

largely architectural qualities. They emerge from choices like: did we avoid global state? Is the dependency graph of modules acyclic and clean? Is each part of the system understandable on its own? These are design-time questions, answered by the architecture of the software. A well-crafted architecture often exhibits what practitioners call conceptual integrity, the sense that the system's parts fit into an overarching, sensible design. Brooks (1975) discussed conceptual integrity, asserting that it is one of the most important considerations in system design. He argued that this integrity comes from a unified vision, often best achieved by a small number of architects or a strong design culture, not by committee. This again highlights the role of individual or small-group craft at the architectural level. It is not a contradiction to have conceptual integrity and use strict standards; in fact, standards support integrity by eliminating gratuitous differences.

One might ask: can high-level design itself be standardised or made algorithmic? To a limited extent, design patterns (Gamma et al., 1994) attempt that. They catalogue common solutions to recurring design problems, like how to structure an observable system or how to decouple through interfaces. These patterns are taught to developers so that they recognise when to apply them. But even design patterns are just guides; applying them is an art. Two systems using the same patterns can have vastly different quality depending on how appropriately the patterns were employed and how well the transitions between patterns are handled. It is here that experience plays a huge role. A master programmer, through years of seeing systems evolve and break, develops an intuition for which design will be resilient. They might foresee, for example, that a quick fix now will lead to technical debt later, and instead implement a more flexible abstraction that is not strictly needed yet but will pay off. In contrast, a less experienced developer might not realise the long-term consequences of certain dependencies or data layouts. This is why Brooks insisted that we must cultivate great designers, essentially architects with deep expertise, if we want great software.

Great software is made by great programmers, not by processes or tools alone. This also has a cultural aspect. By recognising the craft element, we encourage continuous learning and a community of practice where knowledge is shared.. This means that high-level craft is not individual genius at work; it thrives in a community where lessons (the successes and failures of past projects) are communicated. This is similar to how building architects learn by studying famous building designs, analysing why some structures fail and others endure. In software, post-mortems of project failures or write-ups of architecture successes serve a similar function. The craft mindset values these learnings more than rigid adherence to a methodology.

In practice, how does a team implement the idea that high-level design is the locus of craft? One approach is to hold architecture reviews that are more free-form and creative, while keeping code reviews more checklist-oriented (to enforce standards). In an architecture review or pair design session, people brainstorm different models and debate pros and cons. There may be multiple valid solutions. This is where developers get to code as they please, in a sense, by proposing novel structures or unconventional approaches if they have merit. However, once a direction is chosen, the coding of it should adhere to standards so that the idea is realised cleanly. Another approach is to designate certain experienced developers as design mentors who roam between teams to provide advice, ensuring conceptual integrity across the system. This mirrors the master craftsman's oversight in an atelier, who might not carve every statue himself but guides the journeymen in making sure each part fits the overall artistic vision.

To sum up, true coding craftsmanship is most evident in the high-level composition and architecture of software. This is where generic engineering guidance ends and personal expertise takes over, deciding how best to map a complex set of requirements onto a code structure that is robust and adaptable. It is a creative act, constrained by practical realities but not reducible to a formula. By standardising the lower-level coding practices (as discussed earlier), a team actually amplifies the impact of high-level craftsmanship, because the brilliant architectural ideas will not get lost in translation due to inconsistent or error-ridden implementation. Instead, the architecture will manifest clearly in the code. In a well-crafted codebase, one can often infer the architecture just by reading the code, because the intent is not obscured. Conversely, in a poorly crafted one, the architecture might be indiscernible even if it exists, because the code is chaotic. Thus, the philosophy that true skill and craft lie in the higher-level composition and architecture of code resonates strongly with both historical and contemporary views. It positions programmers as makers whose creativity is directed at design, analogous to how an architect or a composer operates, while also being engineers who apply standard solutions for well-understood low-level problems. The next section will consider how this approach can make many traditional software engineering compensatory practices less critical, as the need for extensive rework, debugging, and overhead diminishes when software is crafted right from the start.

7. Reducing the Reliance on Process

A significant implication of treating coding as a craft and enforcing strong low-level standards is that many of the heavyweight software engineering practices can be scaled back without compromising quality. When code is clean, consistent, and developed with foresight, there is simply less need for some of the compensatory processes that teams traditionally rely on to catch mistakes or enforce quality after the fact. It is important to clarify: this is not to suggest eliminating testing, reviews, or documentation. Rather, it is about right-sizing these activities in proportion to need and allowing teams to be more agile and streamlined because the code itself is more self-sufficient.

One area where we see this is in software testing and debugging. In typical projects, a large chunk of time is spent debugging. Some estimates say debugging can consume 50% or more of development time. A lot of debugging effort goes into chasing down trivial bugs: off-by-one errors, null pointer dereferences, mis-used APIs, incorrect assumptions about state, etc. Many of these arise from inconsistent practices or oversight. If a team rigorously applies defensive programming and standard checks (as per coding standards), a whole class of bugs is caught either at compile time or immediately at runtime with clear assertions. For example, one of NASA's ten rules (Holzmann, 2006) states that each function must check the validity of its parameters and each calling function must check return values. Code written under this rule will rarely have unhandled error conditions, which translates into less time spent diagnosing mysterious failures that propagate from non-validated inputs. If something like the Mars Polar Lander's software had been developed under such rules (specifically, requiring redundant confirmation of critical sensor signals), the mishap that caused its demise might have been avoided. That was a mission-critical bug that a simple standard could have prevented at coding time. If developers systematically avoid dangerous practices and include consistency checks, debugging becomes easier because fewer errors slip through silently. Additionally, consistent code means that when a bug is found, understanding the surrounding code is easier so diagnosing the root cause is faster.

Next, consider quality assurance (QA) and testing teams. In many organisations, there is a huge separate effort expended in writing extensive test suites, performing manual testing, and so on, to find bugs. While testing is indispensable, the nature of that testing changes if the code is crafted well. Instead of QA finding lots of silly mistakes, they can focus on edge cases and integration scenarios. In an ideally crafted system, the unit tests written by developers (perhaps as part of their practice) already catch the straightforward bugs, leaving very few surprises for independent testers. As a result, the overall lifecycle can involve fewer test-debug-fix cycles. This was observed in some projects that adopted formal coding standards: they achieved *first-pass yield* (ratio of software that passes major tests in the first round) far higher than average. Such teams can iterate faster because less time is lost in the stabilising phase.

A less inspiring but essential area is meetings and documentation. Often, projects create voluminous documentation to clarify things that arguably should have been clear in the code. For instance, a design document might explain the expected inputs/outputs of functions, or the meaning of flags, etc. Well-crafted code with good naming, clear structure, and perhaps brief comments where needed can serve as its own documentation to a large extent. This is an idea that emerged in agile and craftsmanship circles: self-documenting code. If code is readable enough, one can reduce reliance on external documents that can become outdated or inconsistent with the code. This again does not mean no documentation, but the documentation can focus on the big picture (architecture, intent, requirements) rather than duplicating what the code should convey. Meetings, such as lengthy design reviews or status meetings to triage quality problems, also become less frequent when the software is under control. A team that produces reliable builds consistently does not need daily bug triage meetings or war rooms to save a project nearing its deadline.

The role of managers and formal oversight might also shift. In a craftsmanship culture, developers are trusted more, and in return, they accept more responsibility for quality. The need for, say, a separate inspection team that polices code might be less if every developer internalises the standards. This aligns with the Agile Manifesto's emphasis on trusting motivated individuals and giving them the environment to succeed. If code quality is consistently high, managers can focus on higher-level concerns (are we building the right features; is the user satisfied?) instead of micromanaging how code is written. In essence, process overhead is a substitute for trust and skill. The less you trust the skill, the more process you add. If you increase skill and consistency, you can safely dial back process.

We should highlight, however, that some processes remain valuable not as compensations but as complements. For example, code review is still useful in a highly skilled team, but its character changes. Instead of reviews that debate coding style or point out obvious mistakes, reviews can focus on deeper issues: is there a better way to implement this? Is the approach in this module optimal? The review becomes more of a peer dialogue about craft than a filter to catch sloppiness. This is a far more satisfying use of senior engineers' time. It is akin to master craftsmen critiquing each other's work, not because it is full of flaws but in the pursuit of an even higher quality or elegance. Similarly, refactoring (the process of improving code structure after it is written) becomes more proactive than reactive. In a poorly crafted codebase, refactoring is often a desperate attempt to impose structure after things have become unmaintainable. In a well-crafted codebase, refactoring is a continuous micro-activity: since functions are short and modules are well-defined, small improvements can be made piecewise without massive upheaval. The system evolves more naturally, and one rarely faces the scenario of "we need to rewrite everything".

An important practice that good craftsmanship lessens the dependence on is the use of heavy design patterns or frameworks to compensate for messy code. Sometimes teams bring in complex frameworks because the underlying code is so brittle that only by imposing an external structure can they manage it. But if the code were simple and well-structured to begin with, a lightweight approach or native language features might suffice. Over-engineering is a known pitfall: adding layers upon layers (an example of accidental complexity) in an attempt to control chaos, which ironically often adds more chaos. Sufficiently skilled developers often choose simpler solutions, for example using straightforward composition instead of an elaborate inheritance hierarchy, because they can see that the simpler solution will be easier to maintain. In doing so, they preempt the need to later rescue the design with complicated retrofits.

One might observe that the craftsmanship approach aligns with the Lean principle of building quality in rather than inspecting quality afterwards. It is much cheaper and more effective to prevent mistakes than to detect and fix them later. This is well-understood in manufacturing and was a core idea of Deming's (1986) quality philosophy. In software, building quality in translates to writing code correctly the first time (to the extent possible) and structuring it well from the start. This requires skilled developers, which brings us back to education and training. If programming is taught as an art of thinking and design, not just as coding to pass tests, graduates will bring that mindset into industry. Computational thinking involves much more than coding tricks; it is about understanding how to model problems and reason about solutions systematically. When young programmers learn to value clarity and to anticipate how code will be read by others, they are effectively being inculcated with the craft perspective from the beginning. This could reduce the need for remedial training or the trial-and-error phase many go through.

Another benefit of high craftsmanship is in the maintenance and evolution of software. Software engineering practices like extensive documentation, elaborate change control boards, and so on often exist because maintaining software over the years is hard when the original authors are gone or the code is obscure. But if the code is clean and self-explanatory, and if the project culture encourages knowledge sharing (through that community of practice), then new engineers can on-board more quickly by reading code and design summaries, rather than sifting through outdated documents or requiring long briefing meetings. The code itself serves as a truthful source of knowledge. Modern version control and code review systems (such as GitHub/GitLab) facilitate this. One can use *blame* (a somewhat aggressively named function) to see why a line was added, or read discussions on a merge request to understand the rationale of a change, rather than relying on a separate document.

Finally, we consider whether an emphasis on craft can coexist with formal methods in critical systems. It is not an either/or. In safety-critical domains (medical devices, avionics), formal verification and rigorous testing (even proofs) are needed. But even there, having code crafted to high standards makes those methods more tractable. For example, software that follows simple control flow and limited scope (as per coding standards) is easier to model check or prove properties about, because it avoids the state-space explosion from `goto`s or recursion. So craftsmanship can actually enhance the effectiveness of formal software engineering techniques, not just replace them. It sets the stage so that advanced tools (static analysers, model checkers) work on a solid substrate.

To put it succinctly, better coding craftsmanship shifts effort from back-end correction to front-end creation. The time saved on debugging, crisis management, miscommunication, and workarounds can be reinvested in adding new features or doing more thorough upfront design.

As a result, projects can achieve higher velocity *and* higher quality, a combination that standard wisdom in software engineering often treats as a trade-off. The craft perspective argues it is not a zero-sum game: quality improves productivity when done right. Empirical support for this comes from studies of high-maturity teams (such as those at CMMI Level 5 or practicing extreme programming diligently) which show significantly better outcomes. A disciplined, craftsman-like approach in XP (with practices such as continuous refactoring, pair programming, and collective code ownership) led to some teams achieving near-zero defect rates in production, reducing the need for long stabilisation phases (Beck, 2000). Those practices embody the craft mindset: fix problems when you see them (not later), keep the codebase clean, and share knowledge.

In closing this section, it is worth acknowledging that not every team or context will achieve this ideal. There are legacy systems, time pressures, and skill gaps that might necessitate more process and less freedom. But the argument stands that to the extent you can improve craftsmanship, you can correspondingly cut down on the weight of process needed. With good and consistent coding guidelines in place, the human in the loop is admittedly still the same old human and subject to mistakes, but that human can be far more effective. The guidelines and craft-oriented culture act as multipliers for human ability, letting developers achieve reliability that otherwise would require massive external checks. This aligns with what Freeman Dyson said (as cited in McBreen, 2002): that software remains largely a craft industry and that the craft of writing software will not become obsolete. The reason is that human creativity and judgement are irreplaceable at the higher levels of design, and human care and diligence make all the difference at the lower levels of implementation.

8. Conclusions

The discussion above has traversed the landscape of programming philosophy, from seeing programming as an art or science to enforcing engineering discipline, further to re-embracing the notion of craft, and finally to reconciling these views. We have made the case that coding is most effective when treated as a genuine craft: an endeavour requiring specialised skills, creativity, and judgement, exercised within a framework of disciplined practices. This perspective does not reject the contributions of computer science (theory) or software engineering (process). Rather, it situates programming alongside them as an equal but different form of rational engagement with the world. Science gives us truths about computation and complexity, engineering gives us structured processes and tools, but craft gives us the bridge between the two. The day-to-day practice of writing code that is correct, clear, and fit for purpose. By comparing and contrasting various viewpoints, we can isolate a few key principles for the craft of coding:

Craftsmanship and Consistency: The craft view is not a license for anarchic personal style; rather, true craftsmen value consistency as part of their pride in workmanship. Consistency in naming, formatting, and basic structure creates cognitive leverage and reduces errors. As Ousterhout (2018) and others insist, a consistent codebase is easier to grasp and maintain. Great programmers establish and follow standards conscientiously. Not out of pedantry, but because it elevates the whole team's ability to collaborate and innovate on top of the code.

Discipline Enables Creativity: The argument that low-level coding should be strictly standardised is persuasive. When every developer follows guidelines that prevent common mistakes (e.g., no unchecked errors, no ambiguous constructs), the software achieves a baseline reliability and clarity. This, in turn, frees developers to exercise creativity where it matters more: in

architecting the system and solving novel problems. We know that the PTS guidelines (Danielson, 2012) and similar standards act as a safety net, catching trivial bugs and ensuring uniformity. Thus, programmers are liberated to focus on the design and behaviour of the system, rather than on fighting fires in the code. The artistic freedom in programming is at the conceptual level, not in how one indents or braces an `if` statement.

Programming is a Design Activity: Hunt and Thomas's quarry worker's creed "envisioning cathedrals while cutting stones" illustrates that each line of code contributes to a greater whole. The craft-aware programmer never writes a loop or a function in isolation, but always considers how it fits the architecture. This aligns with Reeves's (1992) argument that *the code is the real design*. If we accept that, then writing code is not just implementation, it is an act of design. Approaching it as a craft means we aim for conceptual integrity and a clear expression of intent in that design. We strive for code that not only works but whose structure *makes sense*. A codebase where one can infer the requirements and design principles by reading it.

Individuals within a Community: The balance between individuals and teams is delicate. The Software Craftsmanship Manifesto (2001) explicitly valued a community of professionals, highlighting that craftsmanship is not lone wolf heroics but a community endeavour. Great programmers do make a great difference. So, while we empower individuals to make decisions and take ownership (an Agile tenet), we also embed them in a community of practice. This reduces the need for top-down management controls because the community self-regulates quality through culture and shared pride. In such an environment, methodologies become supportive rather than prescriptive. As seen, Agile methods dovetail with craftsmanship when interpreted in this light: they provide minimal process to enable collaboration, while the developers bring engineering excellence.

The Limits of Formalism: Tedre's (2014) historical analysis and our discussion show that attempts to reduce programming to pure engineering formulae or scientific theory have limits. There will always be a need for human insight and creativity. No methodology can anticipate every twist of a project; no static analysis can catch every logical flaw that a thoughtful human might. Brooks (1975) called this the essence of software's difficulty, i.e. the conceptual complexity that resides in the problem itself. Craftsmanship accepts this reality and says that given the unpredictability, better to have empowered, skilled humans at the helm than to overly rely on process to magically solve it. That said, craftsmanship does not reject formal methods when applicable. It simply does not put all faith in them. It treats them as tools in the craftsperson's broad toolkit, to be used judiciously.

In conclusion, the perspective that emerges is a holistic one: Software development is a craft informed by science and engineering. It is a craft because writing software is ultimately a human creative act, requiring ingenuity and yielding something with aesthetic and practical value. It is informed by science (computation theory, algorithms) and by engineering (process control, metrics, tools), but it transcends them in the act of creation. Recognising programming as a craft has profound implications. It means we invest in educating programmers not just in syntax or frameworks, but in judgement, design thinking, and responsibility for quality. It means organisations evaluate developers not only by how many features they churn out but by the quality and longevity of the code they produce (rewarding the reduction of technical debt, for example). It means teams strive for a shared code culture, wherein writing clean code is everyone's job, not an afterthought.

By adopting strict standards at the code level and fostering creative skill at the design level, software teams can achieve a harmony that neither pure free-form hacking nor heavy-handed

engineering can provide. We move away from seeing those approaches as opposites and towards seeing them as *layers*: the foundation of uniform craftsmanship supporting the edifice of innovative architecture. As Knuth (1974) noted, programming can truly create artefacts/objects of beauty. In the end, the craft of coding is about taking responsibility for every line of code, while constantly keeping in mind the greater system design.

References

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

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

Brooks, F. P. (1987). *No Silver Bullet – Essence and Accident in Software Engineering*. IEEE Computer, 20(4), 10–19.

Brooks, F. P. (2010). *The Design of Design: Essays from a Computer Scientist*. Addison-Wesley.

Buse, R. P. L., & Weimer, W. (2010). Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4), 546–558.

Danielson, M. (2012). *A Little Book on Error-Free Software*. Sine Metu.

DeMarco, T., & Lister, T. (1987). *Peopleware: Productive projects and teams*. Dorset House.

Deming, W. E. (1986). *Out of the crisis*. MIT Press.

Dijkstra, E. W. (1972). *The Humble Programmer*. Communications of the ACM, 15(10), 859–866.

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

Holzmann, G. J. (2006). *The Power of Ten: Rules for Developing Safety-Critical Code*. IEEE Computer, 39(6), 95–97.

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

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

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

Naur, P., & Randell, B. (Eds.). (1969). *Software engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*. NATO Scientific Affairs Division.

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

McBreen, P. (2001). *Software Craftsmanship: The New Imperative*. Addison-Wesley.

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

Ousterhout, J. (2018). *A Philosophy of Software Design*. Yaknyam Press.

Reeves, J. W. (1992). *What Is Software Design?* C++ Journal, 5(2).

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.

Tedre, M. (2014). *The Science of Computing: Shaping a Discipline*. CRC Press.