Common-Sense Thinking and Programming Skills

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: Why do some individuals excel at programming while others, despite high intelligence or formal training, struggle? This paper argues that the key differentiator lies not in traditional academic metrics but in a general-purpose cognitive capacity termed *common-sense thinking* (CST). Defined as adaptive, intuitive, and context-sensitive reasoning, CST enables individuals to apply sound judgement and flexible problem-solving strategies in real-world situations. Drawing from psychology, cognitive science, and programming literature, the paper develops a theoretical model connecting CST with programming proficiency. It contrasts CST with computational thinking, the structured reasoning taught in formal computer science education, and highlights the necessity of both for effective software development. Empirical and anecdotal evidence, ranging from research on programming novices, expert debugging behaviours, and performance disparities, to industry trends favouring self-taught developers, supports the hypothesis that CST is a primary enabler of programming talent. The paper critiques the limitations of IQ, mathematical training, and formal CS curricula in predicting programming success, showing that many high-IQ individuals lack the practical reasoning needed to handle the ambiguity and decision-making inherent in real coding tasks. Conversely, individuals with strong CST, regardless of academic background, often thrive by relying on experiential heuristics and adaptive judgement. By suggesting CST as foundational to programming expertise, this article complements prevailing assumptions, emphasising cognitive flexibility, practical intelligence, and real-world engagement over purely analytical measures. The findings have clear implications for computer science education.

Keywords: Common-sense thinking, Programming skills, Cognitive flexibility, Computational thinking, Problem solving, Practical intelligence, Coding aptitude, Computer science education

Introduction

Why are some individuals great software developers while others with similar interests, IQs and training struggle to write effective code? This question has long puzzled educators in the computing field. Academic institutions, as well as what can be considered conventional wisdom, often emphasise analytical intelligence and formal education in computer science as the key ingredients for programming success. Yet, striking counterexamples abound: mathematically brilliant minds often struggle with practical coding tasks, and numerous self-taught programmers without advanced degrees have achieved outstanding feats in software development. These observations suggest that another cognitive factor, often colloquially termed *common sense*, plays a pivotal role. We suggest that individuals who become highly skilled programmers typically exhibit high levels of common-sense thinking (CST), defined as a general-purpose capacity for adaptive, situation-sensitive, non-formal reasoning and sound judgement. In other words, beyond raw analytical prowess, it is one's practical reasoning ability and intuitive grasp of real-world complexities that may predict programming excellence.

This article develops a comprehensive theoretical model to explain why CST should be considered a core enabler of programming expertise. We draw on psychological literature on reasoning, practical intelligence, and cognitive flexibility, and on computer science literature about software craftsmanship, coding practice, and programmer cognition. In the first half, we delineate the concept of CST and its components, situating it within established frameworks such as dual-process theories of thought, Sternberg's triarchic intelligence model, and the notion of computational thinking. We argue that programming as a problem-solving activity inherently demands the kind of adaptable, context-aware reasoning that CST provides. In the second half, we examine empirical, conceptual, and circumstantial evidence for the hypothesis. We explore why high-IQ individuals (including those with strong mathematical backgrounds) may underperform in programming relative to high-CST individuals, why extended formal education in computer science does not guarantee programming skills (although it does not hurt), and why self-taught programmers sometimes excel. We integrate insights from research findings, historical examples, and industry observations. From the success of software craftsmanship approaches to studies of novice programmers, to substantiate the claim that CST is a decisive factor in programming talent. By using perspectives from both software development and psychology, we aim to show that common-sense thinking is a foundational cognitive ability that underlies much of effective coding practice.

Common-Sense Thinking

At its core, common-sense thinking (CST) refers to the dynamic, situation-responsive application of ordinary reasoning and intuitive judgement to solve problems in real-life contexts. It goes beyond the mere possession of common knowledge. Instead, CST is about how one flexibly uses experience and intuition. In simple terms, CST entails using sound, prudent judgement based on a direct perception of a situation, often informed by experience rather than formal rules. For example, knowing not to touch a hot stove or to save money for a rainy day does not require advanced education; it stems from basic experiential learning. However, CST is more than just these static common-sense facts; it involves actively thinking through everyday problems in an adaptive manner.

It is important to distinguish CST from raw academic intelligence. Someone may have an extensive store of factual knowledge or excel at abstract logic, yet lack common sense in applying that knowledge wisely. High IQ or subject-matter expertise does not guarantee practical judgement. Common-sense reasoning tends to be contextual, experience-guided, and pragmatic, rather than purely abstract or formal. It deals with the nuances of everyday life's requirements, the kind of unstructured problems and situational decisions that textbooks often gloss over. In the programming domain, having theoretical knowledge (for example knowing algorithms or Big-O complexity) is undeniably valuable, but CST governs the ability to apply that knowledge appropriately. For instance, deciding when a simple, well-understood solution is better than an optimised but convoluted one, or intuitively foreseeing how a user might actually interact with a software feature.

Another way to frame CST is as a mix of cognitive skills working in concert. CST draws on executive functions (like planning and cognitive flexibility), on rational thinking (the propensity to think logically when it matters), and on both fluid and crystallised intelligence. We can see CST as a *composite mental ability*: part practical knowledge, part flexible problem-solving ability, part self-regulation of thought. This ability allows individuals to navigate situations that do not come with clear instructions. In essence, CST bridges intuitive insight and deliberate analysis. It thrives on experience in learning from what has worked or failed in the past, yet remains open to novel solutions when faced with new problems.

Thus, CST is a general-purpose cognitive capacity for making sound decisions and solving problems in real-world, variable contexts. It relies on intuitive judgement honed by experience, but remains adaptable and reflective. As we move to connect this concept with programming, an initial intuition emerges: writing good software is an endeavour that often involves ill-defined problems, continual decision-making, and learning from feedback, precisely the sort of activity where CST would prove invaluable. Before developing that connection fully, we go deeper into the psychological foundations of CST to understand why it might be a decisive factor in something as ostensibly logical as programming.

Practical Intelligence and Tacit Knowledge

The idea that common sense is distinct from academic intelligence has strong support in cognitive psychology, notably in Robert Sternberg's work on practical intelligence. Sternberg's triarchic theory of intelligence divides intelligence into three facets: analytical, creative, and practical. While analytical intelligence aligns closely with the abilities measured by IQ tests (logical reasoning and abstract analysis) and creative intelligence involves innovation and imagination, practical intelligence corresponds to the ability to apply knowledge in real-world contexts, essentially to have common sense in achieving one's goals. Sternberg defines practical intelligence as the ability to adapt to, shape, or select environments to meet one's goals, and it involves applying knowledge to real contexts, often measured through tacit knowledge and situational judgement tests. Sternberg and Wagner's studies (1986) have demonstrated that practical intelligence is distinct from analytical IQ. One can excel in academic problem-solving yet falter in practical situations, and vice versa.

Research has shown that practical intelligence (as measured by tacit knowledge, the informal know-how gained from experience) can be a better predictor of real-world success in certain domains than conventional IQ scores. For example, in studies of business managers, responses to scenarios requiring practical judgement (e.g. the best way to handle a problematic employee) predicted managerial success more strongly than did IQ. Likewise, in educational settings, Sternberg found that testing students with practical problems can identify talents that standard analytical tests miss. In effect, practical intelligence emerges as a psychological cousin to common sense. It formalises the notion of common sense into a measurable set of skills for solving real-world problems (wisdom-in-action).

Translating this to programming, such tasks often present as real-world problem-solving scenarios rather than abstract puzzles. A good computer science student might be good at algorithm theory (analytical intelligence) but struggle to write a well-designed piece of software or to design a user-friendly interface if they lack the practical intuition for how real systems fail or how real users behave. In contrast, a practitioner with high practical intelligence, say a self-taught coder, might quickly produce an effective solution when faced with a new coding challenge, drawing on analogous past experiences and a keen sense of what will work in practice. This aligns with Sternberg's assertion that tacit knowledge often underlies success. Knowing *how* and *when* to apply knowledge is as important as the knowledge itself. Indeed, practical intelligence in programming could be seen in one's capacity to troubleshoot problems under constraints, to adapt code from known patterns, or to judiciously choose simple solutions over complex ones based on an intuitive cost-benefit judgement.

Another related construct is adaptive expertise, which complements this discussion. While not from Sternberg per se, the concept of adaptive expertise in educational psychology distinguishes between routine experts (who excel in efficiency at familiar tasks) and adaptive experts (who can handle novel problems by flexibly learning and innovating). High CST individuals are akin to adaptive experts. They possess the *cognitive flexibility* to adjust strategies when facing a new situation. Good programmers often exhibit adaptive expertise: when a familiar approach fails, they nimbly try alternative methods or reframe the problem, rather than being stuck. This flexibility is grounded in practical reasoning. A programmer with adaptive, common-sense thinking might say: If I cannot solve this bug directly, perhaps I can break the problem down or replicate it in a simpler scenario, echoing Polya's heuristic that if you cannot solve a problem, then there is an easier problem you can solve instead. In fact, Pólya's principles (1945) for problem-solving (originally for mathematics) are compatible with guidelines for practical intelligence and CST at work. Pólya advocated understanding the problem deeply, devising a plan, checking the result, and importantly, if stuck, finding a similar, simpler problem to get insight. These heuristics, to draw analogies to past problems, restate the problem and solve a part of it first, are strategies any seasoned programmer will recognise. They are not formulaic, but rather *common-sense approaches* to overcoming hurdles. A coder with high CST will naturally employ such strategies, whereas a more rigid thinker might insist on more formal approaches even when those are not yielding any progress.

Sternberg's work also highlights the role of tacit knowledge, the unspoken, experience-based know-how, in practical intelligence. In programming, tacit knowledge might include an intuitive feel for code readability, a sense of which algorithmic approach usually works best for a type of task, or knowing the typical pitfalls in a certain programming language. Such knowledge is rarely taught explicitly in school; it is gained through hands-on practice and exposure to many examples. It is, in effect, the common sense of the programming world. Software craftsmanship literature often speaks of the craft knowledge that expert programmers accumulate, which is often not fully captured in textbooks or formal methods. For instance, experienced developers learn idiomatic patterns that prevent errors and promote clarity, a form of tacit knowledge that guides their day-to-day decisions in coding. This practical know-how can distinguish a skilled programmer from a novice as much as, if not more than, their understanding of computer science theory.

The distinction between analytical intelligence and practical intelligence maps onto the distinction between formal problem-solving ability and common-sense reasoning. Programming is a domain that demands both, but evidence suggests that the practical side (CST) is important for actually getting things done in real programming tasks. An individual high in CST will effectively leverage tacit knowledge, adapt to new challenges, and apply wisdom gleaned from experience, all of which greatly facilitate programming work. Meanwhile, someone may have high analytical prowess but, if low in CST, might overthink simple tasks, misjudge what a problem requires in practice, or fail to learn from past mistakes. This idea will be further reinforced when we examine evidence of high-IQ individuals struggling with programming. First, we turn to another facet of CST: rational thinking and the avoidance of cognitive biases, which further differentiates common sense from raw intellect.

Rationality and Heuristics

Common-sense thinking is closely tied to what Stanovich calls *rational thinking* (2009). Stanovich has extensively argued that standard intelligence (which he equates with the brain's algorithmic processing power) is not the same as rationality, the ability and disposition to think logically and make decisions that align with one's goals in real-world contexts. In Stanovich's view, many smart people suffer from what he terms *dysrationalia*: the failure to behave rationally despite having adequate intelligence. Dysrationalia is essentially a lack of common sense, a disconnect between one's cognitive capacity and one's practical reasoning in everyday situations. For example, an individual might excel at abstract algebra yet consistently make poor financial decisions or fall for bogus schemes, indicating a deficit in rational judgement. Stanovich's research has shown that many people with high IQs nonetheless perform poorly on tests of reflective reasoning, such as the Cognitive Reflection Test, which assesses whether a person can override an intuitive but wrong answer with deliberation. Intriguingly, these tests reveal that intelligent people often *miss obvious considerations in a rush to intuitive answers*. In other words, they can lack the common-sense checkpoint that flags if a result seems too obvious and needs double-checking. To quantify this, Stanovich proposed a Rationality Quotient (RQ) to complement IQ, effectively measuring the quality of one's common-sense reasoning and decision-making.

How does this apply to programming expertise? Programming frequently requires rational thinking in Stanovich's sense. One must avoid cognitive biases and illogical leaps when designing software. A classic example is the need for a developer to test their assumptions. A common cognitive pitfall is the confirmation bias (seeing only evidence that supports your belief). A programmer with high CST (and thus high rationality) is more likely to catch themselves and thoroughly test a scenario, rather than assume their code is correct without proof. On the other hand, an intelligent but theoretically overloaded programmer might go ahead with a complex design without checking feasibility, a clever-silly approach that backfires when real-world constraints emerge. The term *clever sillies* was coined by Charlton (2009) to describe high-IQ individuals who are surprisingly deficient in common sense. Charlton observed that extremely analytical people can fall into the trap of overusing abstract reasoning even when a simple, common-sense solution is more appropriate. He suggested that an increasing relative level of IQ brings with it a tendency to override instinctive and spontaneous forms of evolved behaviour, which could be termed common sense. In effect, very brainy programmers may trust their elaborate reasoning so much that they dismiss basic intuitive judgement, sometimes leading to silly program functions that an average person would have avoided. Charlton gives the example of social situations: highly intelligent people might consider theoretical knowledge for social problems and propose solutions that are actually maladaptive, whereas an average person's evolved common sense would have guided them to a more practical solution. Translating this to software: a mathematically educated programmer might engineer an overly complex, theoretically optimal solution for a simple problem, a solution that impresses in abstraction but fails in practicality, perhaps because it is too brittle or too hard for others to understand. Meanwhile, a programmer with better common sense might choose a simpler, well-trodden design that gets the job done reliably. The clever solution can turn out to be the foolish one in context, if it ignores domain-specific adaptive behaviours that common sense would dictate.

Empirical research reinforces the point that rational common sense is an independent cognitive domain that matters greatly. Stanovich's studies found that rational thinking skills (like avoiding framing effects, correctly applying probabilistic reasoning, etc.) do not strongly correlate with IQ. Thus, it is quite possible to have top-notch formal intelligence yet consistently make poor decisions or reasoning errors in everyday or applied settings. Programming, as both a common-sense and a creative endeavour, poses many decisions that are not straightforwardly answered by formal logic or calculation. One must decide how to allocate effort (is it worth optimising this code, or is it good enough?), how to interpret ambiguous requirements, or how to prioritise which bug to fix first. All these require judgement calls under uncertainty. High CST programmers excel at these judgement calls; they deploy a combination of learned heuristics and reflective thought to steer projects effectively, whereas low-CST programmers might either make impulsive decisions or overanalyse without action, missing the common sense.

It is also worth discussing the role of heuristics in common-sense reasoning. Gigerenzer and colleagues (1999) have argued that simple cognitive heuristics can be surprisingly effective, characterising them as fast-and-frugal methods that often outperform complex deliberation in real-world conditions. For instance, a heuristic like if a solution feels too complicated, try a simpler approach first is not logically guaranteed to succeed, but experienced problem-solvers know that it *often* yields good results in practice. These heuristics are essentially distilled common sense. In programming, expert developers rely on many such heuristics, for example suspect the simplest cause of the bug before imagining exotic ones. Such rules of thumb encapsulate practical wisdom. Gigerenzer's research showed that heuristics can exploit the structure of environments to make decision-making more robust and efficient than an impractically exhaustive analysis. A high-CST programmer has a rich repertoire of these *experientially validated heuristics*, and importantly, the judgement to know when each applies. By contrast, a less grounded programmer might either rely on a single method for every problem or attempt a full formal analysis of every decision, either extreme being suboptimal. As common sense aligns with this kind of rough-and-ready reasoning that is not rigorously optimal but works well in typical situations, it complements analytical thinking by guiding it within the bounds of practicality.

In sum, rational common sense is about doing the cognitively right thing in a given context, not just the smart thing in the abstract. It means noticing obvious cues, avoiding needlessly contrived solutions, and being aware of one's cognitive biases. A programmer with strong CST will avoid the pitfall of being clever-silly. They will not let a predisposition for complexity or an elegant theory override the empirical feedback their code is giving them. They will use heuristics to navigate complex design spaces effectively and will know when to switch from intuitive mode to a more rigorous mode of thought. All these abilities directly enhance programming performance by preventing errors in judgement that could lead to project failures or interminable debugging sessions. Next, we examine another aspect of cognition that underlies CST: cognitive flexibility and the capacity to adapt one's thinking, an important skill for a domain as dynamic as software development.

Cognitive Flexibility and Adaptive Reasoning

A key element of common-sense thinking is cognitive flexibility, the ability to shift thinking strategies, consider multiple perspectives, and adapt to changing conditions. In psychology, cognitive flexibility is often discussed as part of *executive functions* (which also include working memory and inhibitory control). It enables someone to think outside the box or simply to recognise when a habitual approach is not working and to try something different. In everyday terms, this is the aspect of common sense that stops a person from, say, stubbornly continuing down an obviously failing path. Instead, they notice the signs and change tactics. In programming, cognitive flexibility is indispensable: one might plan an implementation one way, then realise partway through that a different approach is needed. The ability to pivot, to refactor code or even redesign the solution when requirements change or new insights emerge, is a hallmark of seasoned programmers and is closely tied to CST.

Imagine a scenario where a developer is using a particular library to achieve a task but keeps encountering bugs or limitations. A rigid thinker might persist, assuming that with enough brute force or intellectual effort, they can make it work. A more flexibly minded (high-CST) developer would more readily ask, Is there a simpler way or a different tool that avoids these issues?, and perhaps switch to an alternative solution, saving time and frustration. This decision requires not just analytical evaluation but a kind of practical meta-reasoning: recognising the point of diminishing returns and adapting strategy. Such judgements come from experience, and one learns warning signs and evolves heuristics such as if a bug defies all logic, consider that the assumption about the source might be wrong. These are manifestations of cognitive flexibility supported by common sense.

Polya's problem-solving heuristics (1945) mentioned earlier are again illustrative. He explicitly encourages the solver to try alternate formulations of a problem or solve a related simpler problem, if stuck. This is essentially advice to be cognitively flexible: do not get fixated; be willing to change your approach. Empirical studies of expert problem-solvers (including programmers) find that experts indeed spend more time reframing problems and considering different strategies upfront, whereas novices often rush in one direction and get stuck. The expert's advantage is not just greater knowledge but better self-regulation of the problem-solving process, an ability strongly tied to CST. It is the *practical intelligence* of knowing when to abandon a line of attack and try another angle.

Another concept relevant here is learning from failure, which requires a blend of humility, reflection, and flexibility, essentially a common-sense attitude. When a piece of code fails, a high-CST individual treats it as feedback and adjusts their understanding. They debug not just by systematically tracing execution (analytical skills) but by intuitively suspecting likely sources (practical heuristic) and, if wrong, quickly moving to test the next hypothesis. In contrast, someone without this adaptive mindset might either fail to generate new hypotheses (tunnel vision on one bug cause) or flail randomly. Common sense in debugging often tells us, for example, to check recent changes first when something breaks (because experience says new changes often introduce bugs), a simple strategy, but one that requires noticing patterns and updating one's approach accordingly. Cognitive flexibility also extends to stepping away from a problem when needed, the sense to take a break, and then return with fresh eyes if not making progress. Far from being a purely soft notion, this is a proven method to break mental fixation and often leads to sudden insight after the incubation period.

In terms of psychological research, cognitive flexibility is known to correlate with creativity and innovation. A programmer exercising flexibility might come up with a clever workaround to a limitation or repurpose an existing piece of code in an unexpected way to solve a problem, essentially creative leaps that require breaking from standard scripts. Importantly, these leaps are often guided by intuition about what might work, and then verified logically. That interplay of generating ideas (flexibly, even whimsically) and then checking them (analytically) is at the heart of effective problem-solving. It echoes the dual-process interplay we discussed: an intuitive conjecture followed by a verification step. Baron (2008) has described good thinking as actively open-minded, being willing to reconsider and explore alternatives, which is a fair description of cognitive flexibility in action, and an attribute of rational common sense.

In the programming literature, cognitive flexibility can be linked to the concept of abstraction skills, the ability to move up and down levels of abstraction. A strong programmer can reason about high-level architecture (broad system design) and low-level details (like off-by-one errors in a loop) and importantly switch between these levels fluidly as needed. This too is a form of flexible thinking: zooming out for perspective, zooming in for specifics. It is common sense to know that sometimes one must stop debugging line-by-line and consider the bigger picture of why the module's approach might be flawed, or conversely, to realise that a grand design issue might actually manifest as a small local bug. The *judgement of which level of detail* *to focus on* at a given time is guided by CST. Beginners often get lost either in too much detail or too much big-picture dreaming; experts toggle focus deftly.

To summarise, cognitive flexibility underpins the adaptive, situation-sensitive nature of CST. It enables programmers to adjust their approach to the unique demands of each programming task, rather than rigidly applying one method. This flexibility is evident in effective debugging, in creative problem-solving, and in the capacity to learn and improve over time. It shows why someone with moderate raw ability but high adaptability can outshine someone with higher raw ability who is rigid, a theme that will recur when we discuss formal education versus self-taught paths. The cognitively flexible, common-sense approach is essentially *learning-oriented*: it treats each challenge as information about what strategy works. Over years of programming, such a person amasses a wealth of practical strategies and the wisdom of when to use them. This is the essence of CST contributing to expertise.

COT vs. CST

Before directly arguing how CST governs programming prowess, it is useful to contrast CST with the concept of computational thinking (COT), a term popularised in computer science education to describe the mental skills needed to formulate problems in a way that computers can solve. Wing (2006) defines computational thinking as the process of abstraction and algorithmic formulation of problems, involving methods like decomposition, pattern recognition, abstraction, and algorithm design. In essence, COT is about thinking *like a computer scientist*: taking a messy problem and expressing it in clear, computable terms (data structures, logical steps). For instance, breaking down a task into subtasks or recognising that a new problem has the same structure as a known problem (and thus can reuse the solution), are classic computational thinking skills. COT has been rightly championed as a fundamental skill for programmers, indeed, it underlies the formal education in algorithms and programming methodology.

However, our hypothesis highlights that common-sense thinking is an indispensable complement to computational thinking in real programming tasks. While COT provides the formal tools and structured methods, CST provides the contextual judgement and adaptability. One way to frame their relationship is: computational thinking helps ensure a solution is logically correct and efficient *in theory*, whereas common-sense thinking helps ensure the solution is appropriate *in practice*. A person might be able to devise a very clever algorithm (COT at work), but deciding whether the problem at hand truly needs that clever algorithm or if a simpler approach suffices is a CST question. Likewise, COT can guide one in systematically debugging code (e.g. binary search through possible fault locations), but CST might tell the programmer which part of the system is most likely to be at fault based on practical experience (It is probably the new module we added last night, let's check there first).

It is instructive to recall that computational thinking, as Wing described, involves *abstraction* and *automation*. Essentially, it teaches us to remove extraneous detail and focus on the computational essence of a problem. Yet, when engaging with real-world software, an over-emphasis on abstraction can sometimes lead to a disconnect from reality. This is where common sense steps in: it reminds us of the real-world considerations that might not be captured in an abstract model. For example, COT might lead a programmer to design a perfectly normalised database schema (a very logical design), but common sense might suggest denormalising certain parts for performance because the real usage patterns demand it. Or COT might produce an algorithm that is optimal in runtime complexity, but common sense raises a flag that the constant factors or memory usage might actually be impractical given the environment (something a purely formal analysis might overlook if it fixates only on Big-O). In both cases, the combination of COT and CST yields the best outcome: the solution remains computationally sound but also pragmatically sensible.

Another contrast: computational thinking is often domain-general but context-insensitive, it teaches one to apply the same approach to any problem (formulate, abstract, solve). Common-sense thinking is inherently context-sensitive, it asks *what is unique about this situation?* A high-CST programmer will pay attention to the context: Who is going to use this program? What constraints are truly important (e.g. is it more important that the code runs blindingly fast, or that it is easy to maintain)? They use that context to guide their technical decisions. This is aligned with the software engineering saying that context is key: good programmers choose methods and tools appropriate to the context rather than a one-size-fits-all. It is not that computational thinking lacks awareness of context (indeed, part of COT is choosing appropriate levels of abstraction), but common sense explicitly brings in real-world awareness that might lie outside the formal problem definition. A computer will do exactly what it is told, but a programmer needs the common sense to tell it the *right* thing, including handling the things the specification did not mention.

In educational discussions, some have warned against focusing on computational thinking to the exclusion of broader thinking skills. Our argument reinforces that: a purely COT-trained individual might excel at coding competitions or textbook exercises (which are neatly defined), yet struggle with an open-ended software project or ambiguous customer requirements. Those latter situations demand what could be called computational common sense, knowing how to deal with incomplete information, how to iteratively refine an understanding of the problem, and how to integrate feedback. It is telling that many experienced developers say that understanding the *problem* is often harder than coding the solution. Common sense is important in understanding problems: asking the right questions, clarifying assumptions, and keeping in mind practical goals.

To be clear, computational thinking is a powerful skill set and certainly part of what makes a great programmer. Mastery of algorithmic thinking, pattern generalisation, recursion, etc., provides one with mental frameworks to tackle coding problems methodically. We claim that *in addition* to those, one needs the less formal, more experience-driven reasoning ability, CST, to excel in practice. Empirical support for this combined requirement can be seen in the hiring practices of tech companies: they often test for CS fundamentals (reflecting COT) but also pose open-ended design questions or scenario-based questions to gauge practical judgement. The latter is essentially testing a candidate's CST, for instance, asking how they'd design a system under certain constraints or how they'd troubleshoot a vague issue. Pure theoretical knowledge might not carry one through those questions without common-sense reasoning.

In bridging CST with COT, we might view programming as a bilingual mental activity: one language is that of formal logic and computation, the other is the vernacular of real-life reasoning. The best programmers are fluent in both, translating back and forth. Donald Knuth, in his 1974 Turing Award lecture titled *Computer Programming as an Art* highlighted that programming requires skills and ingenuity and produces artefacts of beauty, likening it to creative arts like carpentry or architecture. Knuth's use of *art* (from Latin *ars*, meaning skill) was deliberate. He was pointing out that programming is not just applied mathematics or engineering. It also involves craft, aesthetics, and human judgement. In other words, beyond the scientific method, there is an artful, intuitive side, which we call CST. He noted that a programmer blends precision with ingenuity and combines beauty with utility and creativity with discipline. This beautifully encapsulates the union of computational thinking (discipline, precision) with common-sense thinking (ingenuity, sense of utility).

Thus, computational thinking and common-sense thinking should not be seen as competitors but as partners in programming. COT provides the formal structures and habits of mind to deal with complexity through abstraction; CST keeps those abstractions tethered to reality and ensures adaptability. A high-CST individual will make the most of computational thinking skills by applying them in the right measure and context. Conversely, without CST, computational thinking might be applied inappropriately or rigidly. Theoretical knowledge must be put into service of common-sense goals and values to be meaningful. All the computational thinking in the world is only useful if guided by the common-sense goal of delivering a working, useful program.

Programming as a Craft

The nature of programming itself offers clues that common-sense thinking is essential to mastery. There has long been a debate: is programming a science, an engineering discipline, or a craft? Increasingly, voices in the field recognise that programming, especially at the level of writing and maintaining code, resembles a craft practice where experience and judgement matter as much as theory. The software craftsmanship movement explicitly frames software development as a craft apprenticeship model, valuing practical wisdom, code aesthetics, and continuous learning over rigid formal processes. McBreen (2001) argues that developers need not see themselves purely as engineers implementing specifications, but rather as craftspeople who take ownership of their work and rely on skills and judgement honed through practice. This perspective inherently elevates the status of CST in programming: it is the craftsman's common sense, their intuitive feel for good code and prudent design, that guides their hand, much as a woodworker's tacit knowledge guides them beyond what any blueprint may specify.

Historically, the push for software engineering in the late 1960s and 1970s sought to introduce more formalism and process into software development (a reaction to many failed software projects). While this introduced valuable discipline, it also became clear that no amount of process can compensate for a lack of individual skills or judgement. Brooks (1986) points out that the essence of software design, i.e. building complex conceptual structures, is an inherently difficult task not easily automated or systematised. He noted that among the essential difficulties of software are complexity, conformity, changeability, and invisibility, and that great designers (individuals) make a huge difference. He attributed this partly to individual talent. We can infer that the talent in question is not just raw IQ (plenty of very bright teams were still failing) but a combination of analytical ability with design judgement and insight, essentially those with a propensity for managing complexity by wise simplification and intuition. According to Brooks, conceptual integrity is the most important consideration in system design, and it is best maintained by a unified vision from one mind or a small team. Conceptual integrity in practice means keeping a system's design straightforward and coherent, a goal which is served by common-sense judgement, i.e. knowing which features to cut, which design is too convoluted, etc.

Modern agile methodologies, which have largely replaced heavy upfront brute force, implicitly trust developers to use their judgement iteratively: test-driven development, refactoring, and close customer collaboration all rely on developers continuously making decisions about what is the simplest thing that can possibly work and how to respond to feedback. Agile principles de-emphasise exhaustive documentation in favour of working software, which again puts the focus on developers' tacit understanding and sensible communication. This trend in industry underscores that after trying formalised heavy processes, the pendulum has swung toward valuing adaptability and practical wisdom. Danielson (2012) provides a contemporary analysis of programming as a craft and reinforces why CST is vital. He traces how the institutional push for strict software engineering was tempered by a resurgence of craftsmanship ideals, concluding that true software quality arises from skilled application of practices *balanced with creative design*. He argues that low-level standardisation (like consistent coding style) can reduce errors and cognitive load, but high-level architecture still needs genuine creative judgement. In other words, you can enforce certain rules to handle routine matters, but there remains an irreducible need for personal judgement in higher-level decisions. This judgement is exactly the realm of CST. Danielson shows that in successful software teams, there's an equilibrium: discipline at the micro-level and freedom at the macro-level. Achieving this requires programmers to know when to be rigid and when to be flexible, itself a common-sense discernment. He gives the example that enforcing a consistent style in writing code (braces, naming, error handling conventions) actually *frees* the mind to focus on the truly hard problems (the creative design). This perspective resonates with the idea that common sense might tell us where to apply formal rules and where to allow intuition. Less experienced or less sensible developers often err either by having chaos at the low level or by stifling creativity at the high level; a craftsman finds the sweet spot, guided by an intuitive sense of what matters for quality and what does not. Danielson states that good programming must blend strict discipline at the implementation level with freedom and ingenuity at the design level, leveraging human skills where it matters most. Those human skills being leveraged are largely common-sense skills, the ability to inject good judgement and creativity appropriately.

An interesting point Danielson (2012) makes is about coding guidelines aiming for near-error-free software. These guidelines are cited as a model of disciplined practice that achieved very low defect rates, showing the value of rigorous consistency. Yet, even in advocating such discipline, the emphasis is that these rules handle the mundane so that humans can focus on the conceptual. This again is instructive: by routinising what can be routinised, the remaining work *amplifies* the role of human judgement. It implies that as tools and processes eliminate some sources of error (for instance, modern languages taking care of memory management), the relative importance of the programmer's common sense in the remaining tasks (like concurrency issues, user experience considerations, etc.) becomes even greater, those are not fully solved by tools and demand savvy reasoning.

In sum, viewing programming as a craft reveals why formal training alone is insufficient. Like any craft (carpentry, metalworking), apprenticeship and practice underlie mastery; one learns the feel of the material, the tricks of the trade, the balance between following rules and bending them when appropriate. Common sense is essentially the craftsman's internal compass. It guides decisions that have no clear algorithm. A craftsman programmer knows when to adhere strictly to specifications and when to negotiate changes because the spec might be flawed; they know when performance optimisations are premature and when they are important; they know how to simplify a design for clarity's sake. These judgements come from experience and practical reasoning.

Even the day-to-day act of reading and modifying code (which is what developers spend a lot of time on) benefits from CST. Code can be seen as a form of communication. A developer with high common sense will write code that others can understand (because they empathise with the reader, a social aspect of common sense) and will interpret others' code charitably, looking for the intent behind it. They will also use practical reasoning when navigating a large codebase: rather than exhaustively read everything, they might make a reasonable assumption about where to look, much like one uses common sense to find information in a library by first looking at signs and catalogues rather than inspecting every book. These are trivial examples, but they demonstrate that at every level, programming involves choices where rules do not dictate one correct move, you must rely on judgement.

Having built a theoretical understanding that CST, comprising practical intelligence, rational thinking, cognitive flexibility, and craft judgement, is deeply interwoven with what it means to program effectively, we now turn to evidence supporting the hypothesis. Do real-world observations and empirical studies back the claim that high-CST individuals make better programmers (and that lacking CST can hinder programming performance)? We will examine studies of programmer performance, contrasts between individuals of different backgrounds, and anecdotal patterns from the history and practice of software development to validate the linkage between common-sense thinking and programming expertise.

High IQ and Mathematical Aptitude

One provocative line of evidence for our hypothesis comes from cases where high general intelligence or mathematical talent does *not* translate into corresponding programming success. Historically, many assumed that top programmers would be those with the strongest analytical and mathematical backgrounds, and indeed early computing was often done by mathematicians. Yet, over time, it became apparent that the correlation is not so straightforward. Some brilliant mathematicians or high-IQ individuals have struggled to deliver software on time or to debug effectively, whereas individuals with a more ordinary academic profile have thrived as programmers. This apparent paradox is explained when we consider CST: the academically brilliant may lack common-sense approaches or the practical mindset needed in programming, whereas others compensate for a moderate IQ with excellent CST.

Charlton's concept of clever sillies (2009), discussed earlier, is directly relevant. Charlton noted that high-IQ people can fall prey to reasoning that is too abstract and detached, leading them to override plain common sense. In programming, this might manifest as over-engineering, creating designs of theoretical elegance but impractical complexity. Famous anecdotes in software folklore tell how some PhD-holding developers produced an overly complex solution where a simpler hack would have sufficed, and products failed or were delayed because the simple things never got done right. Charlton's explanation is that some high-IQ individuals have a cognitive style that leans toward novelty and complexity (correlated with the personality trait *Openness to Experience*), so they may devise novel solutions that seem clever but are actually *maladaptive* in the real context. Without the moderating influence of common sense, high analytic ability can generate what are essentially brilliant wrong answers. The real problem arises if the person is high in analytic intelligence but deficient in another facet that correlates with common sense or presumably *practical intelligence*. An intellect untempered by practical sense can go astray. There are many accounts from tech companies of prodigious hires who struggled with actual coding assignments due to perfectionism, inability to prioritise, or difficulty adapting to the messiness of real projects. These accounts, while not rigorous data, paint a consistent picture that raw brainpower alone does not guarantee programming success if the person cannot apply reasoning pragmatically.

Empirical support for this comes from Stanovich's dysrationalia findings (2009): many high-IQ individuals make silly errors on simple reasoning problems that require common sense to check intuitive answers. An intelligent programmer might implement a complex algorithm correctly but then miss the simple fact that an input can be null (leading to a runtime error), something a more practically-minded programmer would guard against. It is a mundane example, but such errors are common: forgetting basic cases, not considering user input errors, etc. These are failures of common sense, not of logic per se. Notably, experienced programmers often develop what is jokingly called a paranoia about things going wrong. They add checks and logs, they anticipate misuses, essentially applying common sense to think about what a user or environment could reasonably do that would break the code. Those without that intuition might write code that works only in ideal (read: often school) conditions, as assumed in their analytical model, but crashes or misbehaves in reality.

On the other hand, consider individuals with high CST but perhaps not top-tier IQ or no formal math training. Many self-taught programmers fall in this category: they may not score in the 90th percentile on standardised tests, but they excel in practical problem-solving and have an intuitive sense for how to make technology serve a purpose. These folks often outperform colleagues with fancier credentials. They might not derive an algorithm from scratch as elegantly, but they know how to find a solution (perhaps by smartly adapting existing solutions or trial-and-error) and, importantly, how to integrate it into a working whole. A high-CST programmer is likely to produce a piece of software that, above all, works reliably and meets users' needs, even if internally it uses some brute-force or ad-hoc fixes. In contrast, a low-CST but high-IQ programmer might produce an ingenious piece of code that fails on corner cases or is incomprehensible to others, undermining its value.

Scientific literature on programmer characteristics provides some backing to these observations. Early studies by Sackman et al. (1968) found enormous variability in programmer performance, with some developers more than 10 times as productive as others when performing the same tasks. Importantly, these differences were not fully explained by differences in experience or academic background available at the time, hinting that an inherent problem-solving approach and reasoning style played a role. In smaller teams, productivity is mostly dependent on individual aptitudes and abilities. While aptitude historically was often conflated with IQ or math ability, modern understanding inclines that it includes less tangible cognitive skills like the ones we call CST. The 10 X programmer notion (heavily exaggerated in folklore) underscores that certain individuals tend to produce working and effective solutions fast. That often looks like an ability to cut through complexity with simple, common-sense designs and to troubleshoot issues quickly. It is not that they write 10 times more code, but rather they make 10 times better decisions and fewer mistakes to avoid waste and dead ends. Danielson (2012) argues that a more correct label is 0.1 X programmers, making at most 10% of the errors an ordinary programmer does.

There is also the phenomenon of some competitive programming or math prodigies struggling in software industry environments, which has been informally observed. Competitive programming contests focus heavily on algorithmic puzzles under time pressure, pure computational thinking skills. Some champions of these contests have later remarked that real software development required a quite different skill set (team collaboration, dealing with vague requirements, debugging large systems) for which their contest prowess did not fully prepare them. Those additional skills are largely in the CST realm: communication, practical debugging strategies, time management, etc. Conversely, many highly effective programmers were not math stars but have street smarts about coding gained from practical projects.

To avoid misinterpretation, we are not arguing that high IQ or mathematical ability are negative for programming, certainly not. On the contrary, they are often helpful. But without CST, they can result in a skewed skill set. When an extremely analytical individual intentionally works on their practical reasoning and gains experience, they can become formidable programmers, combining the best of both worlds. The issue arises when high analytical ability comes with low inclination towards, say, testing assumptions or learning from concrete feedback. A well-documented example of smart people making poor programmers comes from Dijkstra's observation (1989). He noted that some students who had learned mathematically strict approaches sometimes struggled with the messiness of real programming, whereas those with a more heuristic approach managed to get programs working. Dijkstra himself advocated formal methods, so this remains a debated point, but the key takeaway is that different cognitive approaches yield different outcomes. Our hypothesis sides with the need for a strong practical component.

One might ask: Are there specific studies comparing, say, mathematicians vs. others in programming tasks? Direct academic studies are few, but there is Charlton (2009) who posited that high-IQ individuals such as top academics may lack common sense in practical affairs. Charlton's thesis, while outside computer science, gives a conceptual frame. He even suggests an evolutionary reason: that common sense evolved for typical scenarios, whereas extremely high analytic reasoning is an evolutionarily novel trait that can sometimes misfire in everyday contexts. If we apply that notion to programming, the coding itself is a novel, artificial activity, but it simulates problem-solving that engages both formal and informal reasoning. Perhaps a highly analytical person treats programming purely like math (strict, symbolic, requiring complete precision from the get-go), whereas a high-CST person treats it more like an exploratory activity (write something, test it, refine it, use intuition). Interestingly, the exploratory, trial-and-error strategy was long considered inferior to the planning strategy in programming, but studies have shown that many successful programmers do a healthy amount of tinkering and iterative development (especially in the context of modern interactive environments). This is akin to *scientific pragmatism*: try something and see what happens, a common-sense experimental approach, rather than deducing everything in one's head first. High-IQ individuals may resist such an approach if they've been trained that everything must be deduced and optimised mentally. Yet in practice, playing around can be a very effective way to solve programming problems.

In conclusion on this point, numerous examples and arguments indicate that higher analytical intelligence alone is neither necessary nor sufficient for programming excellence. It must be paired with strong CST to result in a top programmer. A balanced individual with a moderately high IQ and very high common sense might outdo someone with a high IQ but poor common sense in many programming scenarios. This sets the stage for examining the influence of formal education in computer science, which traditionally emphasises analytical and theoretical training, versus (partially) self-taught routes, which often cultivate practical skills earlier.

The Value of Experience

Formal computer science education, i.e. a university degree, undoubtedly provides a strong foundation in computational thinking, exposing students to algorithms, data structures, formal languages, etc. However, it has been observed that some graduates of CS programs still lack the ability to build software effectively in real-world conditions. This gap highlights that formal knowledge does not automatically translate to practical programming competence. A striking study by McCracken et al. (2001) assessed introductory programming students across multiple institutions and found disappointing results: many students at the end of an introductory course could not effectively program a required task, with an average score around only 20% on a programming assessment. In other words, many students *who had been taught programming* still did not know how to program in practice. A later multinational study by Lister et al. (2004) has similar concerns. A significant portion of students, even after a couple of courses, struggle with basic code reading and writing tasks. Failure rates in introductory programming courses can range from 30% to 60% as reported in various institutions. While part of this is due to the inherent difficulties of learning to program, it also points to a subset of students for whom mastering the formal content does not translate into skills. Some educators have noted that certain students just get it and others do not, and the difference often is not predicted solely by their performance in math or other subjects. This mysterious "it" could well be CST at work, the intuition and judgement about how to approach writing a program that is not directly taught.

Moreover, even among graduates who easily pass their courses, employers often comment that new graduates need significant initial practical training. It is common in industry to find that academic high performers might still need mentorship to write clean, robust code. On the other hand, students who maybe were middle-of-the-form academically but built things on their own (like contributing to open-source, building hobby projects) often turn out to be very competent programmers upon hiring. What is the difference? The latter have exercised their common-sense thinking in real programming contexts, they have debugged without a prescribed recipe, dealt with users or requirements changes, experienced the pitfalls of poor decisions and learned. Essentially, they have cultivated their CST through practice. The academically focused student may have solved idealised problems but might not have faced the same breadth of messy problems or been forced to rely on intuition.

A telling indicator is the rise of technical interview questions that focus on practical scenarios. Many tech companies will present candidates with a coding problem that is not a tricky algorithm but a realistic task or a debugging exercise. They want to see if the person approaches it methodically, tests their solution and considers edge cases, all markers of common sense. Some candidates with stellar GPAs sometimes underperform in these interviews if they have not internalised those habits. This has led some companies to even drop the requirement of a CS degree for hiring, realising that formal education correlates only imperfectly with job performance.

Another piece of evidence comes from industry surveys on the backgrounds of working developers. A 2016 global Stack Overflow survey of over 50,000 developers found that approximately 69% reported being at least partially self-taught, with 13% saying they were *entirely* self-taught programmers. In contrast, 43% had a bachelor's degree in computer science. Moreover, 31% had no formal college degree at all, some came via coding boot camps or just self-learning (McFarland, 2016). It was actually that survey that triggered the writing of this article. These numbers are eye-opening: a majority of practicing developers did not learn everything in a classroom setting. The success of so many self-taught programmers suggests that hands-on experience and self-directed problem-solving, which exercise CST, can effectively substitute or surpass formal training in producing programming ability. One might argue that those self-taught individuals are self-selected enthusiasts (which is true), likely possessing strong intrinsic problem-solving skills (again, likely CST). But it also shows that the formal curriculum is not the only or even primary route into programming skills for many people.

Formal Education

Why might formal education struggle to impart CST? By nature, an academic course has to simplify and structure problems for students to handle in a limited time. This sometimes means students practice more of the computational thinking (structured problems with known solutions) and less of the open-ended problem solving. Real projects involve dealing with ambiguity, integrating many pieces, and making judgement calls, things that are hard to simulate in a classroom, except perhaps via capstone projects. Additionally, academic grading often rewards a very specific kind of correctness and efficiency, which may inadvertently de-prioritise creativity or risk-taking. A student might get full marks for a textbook implementation that in a real scenario might be overkill or not robust; meanwhile, a scrappy but effective solution might be discouraged in class but could be exactly what is needed in a hackathon or start-up environment.

There is also evidence that prior programming experience (often gained informally) is a predictor of success in a CS education. Early practice builds an intuition, likely connected to CST, that formal instruction alone does not immediately grant. The "two humps" hypothesis by Dehnadi and Bornat (2006), though later honourably retracted (Bornat, 2014), originally claimed they found a test to predict who would succeed in learning programming, dividing students into those who conceptually get assignments and those who do not. While their specific test did not hold up, the underlying sound idea was that some cognitive schema (not measured by standard IQ) determined initial programming aptitude, and that idea still has credit. One interpretation is that a certain mode of reasoning, the ability to reason with unknowns and apply consistent rules (some aspect of logical common sense), is necessary to grasp the concept of a variable in programming. Many students who fail to learn programming never internalise the idea of a variable as a storage that can change. They instead think in ad-hoc case-specific ways. This inability to generalise could be seen as a lack of a certain form of reasoning that better programmers have early on.

Another facet is the common lament in the industry that universities teach theory, but we have to teach graduates how to *actually code* on the job. Companies often run training programs for new grads focusing on practical skills: using version control, writing tests, understanding software lifecycle, etc. These skills, while technical, also involve a lot of common sense (e.g., commit logical chunks of code so others can follow, write tests for edge cases that a sensible person would check). The necessity of these trainings underscores that some very practical aspects are not coming naturally from a pure CS education for many. It is no surprise then that some of the best preparation for programming careers comes from project-based learning, internships, or personal projects, which allow students to engage their CST in a realistic setting.

Formal education also tends to treat problems as individual endeavours, yet big programming projects are team efforts. Here, social common sense enters: communication, understanding the perspective of others, and writing code that someone else can maintain. A person might be a whiz at coding alone, but if they lack the social reasoning aspect of CST, they may produce unmaintainable spaghetti code or not collaborate well. The best developers often have strong empathy (a component of common sense in social contexts). They think about how users will experience the software and how fellow programmers will read their code. These soft aspects are increasingly recognised as important in software engineering, hence the emphasis on things like clean code (code written for humans to read, not just for machines to execute) in professional practice. Clean code guidelines essentially encode common-sense principles: use meaningful names, keep functions simple, handle errors gracefully, etc., many of which stem from putting oneself in others' shoes or considering future consequences beyond the immediate algorithm.

In summary, while formal CS training provides necessary knowledge, it does not guarantee programming skills because it cannot fully instil the common-sense thinking habits that arise from practical engagement. Many graduates need to cultivate CST on the job through experience. By contrast, those who have engaged in lots of practice (even without formal education) have already built a reservoir of CST relevant to coding. This leads to the next point: the success of self-taught programmers and how their learning path might inherently foster CST.

Unconventional Pathways

The tech industry has numerous prominent examples of successful programmers (and tech entrepreneurs) without formal education in computer science. Bill Gates and Mark Zuckerberg dropped out of college to start companies (though they did have some college and prior programming experience), Steve Jobs was not formally trained in CS, and self-taught developers like John Carmack (legendary game programmer) or Margaret Hamilton (lead Apollo software engineer, who did have a math background but largely learned computing on the job) became pioneers through experiential learning. While fame is anecdotal, broader data (such as McFarland, 2016) confirms that a substantial proportion of programmers are either partially or wholly self-educated in programming. This directly suggests that the abilities needed to program can be acquired outside formal curricula, and often those who do so demonstrate exceptional capabilities, precisely because they had to rely on their own reasoning and resourcefulness.

When someone is self-taught in programming, what does their learning look like? Typically, it involves a lot of trial and error: writing code, seeing what works, debugging when it does not, and gradually taking on bigger challenges. This process heavily exercises common-sense thinking. The learner must diagnose errors (often with no teacher to ask, so they use search, forums, logs, essentially detective work), they must make decisions about which tools or languages to learn, and they often pursue projects of personal interest (which ensures engagement and iterative improvement). Self-taught developers accumulate a wealth of practical problem-solving episodes. By the time they are building something significant, they've likely internalised many lessons learned, essentially an arsenal of heuristics and mental models (I've seen something like this before; last time it was due to X, so check that first). This repository of experience is exactly what constitutes CST in action for a programmer. It is no surprise that when such a person faces a new bug, they often solve it faster than someone who has mostly seen curated textbook problems. They have a head start from analogous experiences.

One might wonder if self-taught programmers excel only in small-scale hacking and might falter in larger system design without a theoretical background. Sometimes that can happen, theory does matter, but many self-taught folks eventually pick up theory when they need it (often self-teaching that as well). And importantly, they might approach design with a more empirical mindset: build a prototype, see if it scales, and refactor if needed. This iterative, common-sense approach can succeed where a top-down and theoretically optimal design might fail due to unforeseen requirements. The rise of agile and iterative development models actually plays to the strengths of practically minded developers.

There is also evidence that enthusiasm and interest (often characteristics of self-driven learners) correlate with success. Those factors may be tied to what psychologists call *grit* or a growth mindset, but they also tie into CST: someone deeply interested in making something work will try many approaches and learn from mistakes (practical intelligence), whereas someone without that drive might do only what is formally required. Passion projects are in fact an excellent training ground for CST because constraints (like time and resources) force creative solutions, and personal investment drives careful reasoning and learning from errors. Many self-taught coders have portfolios of small programs, websites, etc. Each of those is a story of encountering a problem, solving it, and internalising that solution for the future.

The anecdote of my best developers do not have degrees is heard often enough from start-up circles. While degrees certainly provide a lot of value, the statement often underscores that the degree by itself was not the predictor, the individual's problem-solving ability was. Reputable companies like Google and Apple have, in recent years, relaxed the requirement for a college degree in their job listings for programmers, acknowledging that capable talent can come through different pipelines. Hiring managers often look instead for evidence of projects completed, contributions to open-source, etc., which reflect real-world programming experience and, implicitly, demonstration of common-sense thinking in solving them. In a trend long present in art, a portfolio is becoming more and more important in programming as well, reinforcing the view of programming as an art. Accountants, as an example, seldom have portfolios but rather standard CVs instead.

Even in academic research, there are natural experiments where individuals from outside CS become great programmers because necessity demanded it. For example, scientists or engineers from other fields often write software for their research. Some struggle, but some excel and even shift careers to software without formal CS training. Those who excel likely had high CST. They approached coding pragmatically, learned incrementally, and perhaps had domain common sense that transferred (e.g., a physicist's intuition about systems helps in structuring simulation software).

We should also mention programming competitions like hackathons, where people from varied backgrounds collaborate to create a working prototype quickly. Success in a hackathon often hinges on practical ingenuity and adaptability, classic CST traits, rather than deep schoolbook knowledge. Teams with balanced skills (a lot of coding knowledge plus some common sense about scope and user needs) often outperform teams of straight-A students who might attempt overly ambitious projects or ignore practicality. This again constitutes circumstantial evidence: time and again, in contexts where delivering something that works is the goal, those with better practical judgement shine.

All of this is not to downplay the value of formal education or intelligence. Rather, it highlights that *in the absence of a formal path, people who have the right thinking approach can still achieve mastery*. In fact, being self-taught might even reinforce certain common-sense habits: since there is no professor to debug your code, you become self-reliant and develop diagnostic strategies (like careful reading of error messages, simplifying the problem, rubber-duck debugging, etc.). These are teachable, but often learned best by necessity.

One might question: could it be that those who self-teach are just as analytically smart as those in formal programs (just minus the degree)? Sometimes yes, but the diversity of self-taught developers suggests many are very smart, but their talents lie in being autodidactic and practically oriented. There are also cases of individuals who did not thrive in structured academic environments (perhaps due to learning differences or simply a lack of interest) but later found their stride in the more free-form context of programming on their own terms. Such individuals might have had perfectly fine logical ability but needed a different mode to cultivate it, one that engaged their common sense and creativity more than formal schooling did.

As a final note in this section, consider the role of mentors or master-apprentice relationships, which are central in craft fields. In programming, mentorship (whether via open-source communities or in a workplace) is often where tacit knowledge is transferred. A mentor will often give tips that amount to common-sense advice: e.g., *check for off-by-one errors first when the output looks almost right*, or *when designing, think about how you will test this*. These are not found in textbooks in bold letters, but they are gold nuggets of practical wisdom if filtered through a CST filter. Self-taught programmers often seek out communities to ask for help. The answers they get frequently instil common-sense reasoning: perhaps someone on a forum does not just give the solution but explains how they arrived at it, teaching the heuristic. In essence, the community becomes the teacher of CST.

Given the significant representation and success of self-taught programmers and those from non-traditional paths, we have a strong empirical basis to say that *practical common-sense reasoning and experience can compensate for, and sometimes outperform, formal training*. It also suggests that CST is an underlying enabler. Those who had it (or cultivated it) figure out programming one way or another.

Case Studies in Programmer Performance

The hypothesis that CST underlies programming expertise can also be examined by looking at observed differentials in programmer performance and specific case studies or experiments. We have touched on the 10x productivity studies. Let's delve a bit deeper: In the classic Sackman et al. (1968) study, not only did coding time vary by over a factor of 10 between best and worst performers, but so did debugging time and error rates. The fastest, most accurate programmers wrote code that ran sooner and with fewer defects. This strongly indicates qualitative differences in approach, not just speed of typing or knowledge of syntax. High performers likely employed better mental models and sanity checks (i.e. common-sense verifications) as they coded, preventing errors in the first place, whereas low performers probably got stuck in cycles of trial and error without systematic strategies (a sign of poor CST in problem-solving). Nearly all attempts to debunk the 10 X myth concede that there are differences, even if they disagree on the exact factor.

Another interesting observation is how expert programmers approach code comprehension compared to novices. R. Brooks (1983) proposed a model that experts form hypotheses about code function and structure in a top-down manner, guided by domain knowledge and expectations. This means an expert reading unfamiliar code tries to map it to familiar patterns (Ah, this looks like a standard sorting routine, maybe using quicksort) and then verify details, whereas a novice might get lost in line-by-line details without context. The expert's approach is a common-sense strategy: use prior experience to generate a likely interpretation, then check. It is essentially applying an analogy and confirming, which is both faster and cognitively less taxing than a blind read-through. This aligns with Letovsky's (1987) description of programmers as opportunistic processors who easily change strategy based on cues. Flexibility and hypothesis-driven reading are signs of CST at work: the expert uses intuition to guess a meaning, then analysis to confirm. If wrong, they adjust. Studies have found that expert programmers spend more time at the outset reading and thinking, building a mental model, rather than diving directly into editing code. This upfront investment is a common-sense allocation of effort, as the saying goes, measure twice, cut once.

Consider also the differences in error debugging: A common finding is that experts debug in a highly goal-directed way; they often can zero in on a bug by reasoning about what *must* be wrong given the symptoms (sometimes even before running the program, they suspect certain areas). Novices often resort to wandering through the code or making many random prints. A controlled study by Vessey (1985) on debugging showed that more skilled debuggers formed hypotheses and systematically eliminated them, whereas less skilled ones tended to change many things without a clear rationale, sometimes introducing new errors. The skilled behaviour again reflects rational problem-solving and good judgement, i.e., CST, in action.

From the history of software, we can identify episodes that underline CST. The infamous project failures during the first software crisis (in the late 1960s) often came when teams tried to build large systems with insufficient feedback or practicality. Essentially, they tried to engineer everything perfectly on paper (a very formal approach), and reality broke those designs. The move towards iterative development was a triumph of common sense: acknowledge that humans cannot foresee everything, so build in small increments, test and adapt. It is a formalisation of common-sense practice into a process. But even the best processes will fail if individuals do not exercise judgement within them.

One circumstantial but illuminating piece of evidence is how programming competency is evaluated in practice. In addition to technical quizzes, many employers rely on references or past work as indicators. A glowing reference often speaks to how the person solved problems under pressure, how reliable their code was, how well they learned new things, essentially describing common-sense qualities. Meanwhile, there are countless stories of hiring someone with a stellar academic CV who then underperforms on practical tasks. These outcomes have forced companies to refine their hiring tests to include pair programming sessions or take-home assignments that simulate actual work, because that is where CST becomes visible.

We should highlight the importance of domain knowledge as well, which might be considered part of crystallised intelligence or tacit knowledge. A programmer with extensive domain experience (say in finance or operating systems) will often have much more common sense about typical pitfalls in that domain, which gives them an edge. This is why companies value experienced hires: not just for what languages they know, but for the judgement that comes from having seen many projects. It is telling that a study in Sommerville's textbook (2016) indicates that domain knowledge is a major factor in individual productivity. Such knowledge could be seen as a component of CST specific to a field. It is essentially knowing what usually works or fails in a specific context.

Another perspective is the role of rationality in code quality. A recent area of research examines how cognitive biases affect software engineers. For example, confirmation bias can lead a developer to stick to an initial diagnosis of a bug even as evidence mounts against it. Good developers are often mindful of this. They will not jump to conclusions and instead try to get a fresh set of eyes on things. Those are rationality-preserving behaviours, essentially common-sense checks on one's own fallibility. Teams that practice code reviews implicitly inject more common sense (since two minds may catch each other's blind spots). If programming were purely an application of taught rules, such practices would not be needed. But because human reasoning quirks can lead to issues, the social process compensates. In instances where projects skipped such sanity checks, huge blunders have occurred (e.g., NASA's Mars Climate Orbiter failure in 1999 because one team used imperial units and another metric…a failure of common-sense communication and verification rather than a lack of calculus skills).

Finally, consider talent identification efforts. IBM's Programmer Aptitude Test (PAT) from the 1960s tested things like pattern recognition, flowchart following, etc., to predict who could be a programmer. Tukiainen and Mönkkönen (2002) sees the PAT as a trendsetting tool in assessing programming aptitude. The study evaluates the predictive validity of programming aptitude tests, including the IBM PAT, in forecasting students' success in learning programming concepts. It also discusses the components of PAT and its historical usage in the industry. However, it was deemed not highly predictive. Why? Likely because it did not capture the essence of what makes a good programmer. It tested basic logic, but not the full suite of CST. Modern equivalent aptitude tests have similarly struggled. The best indicator might be to give someone a problem and see how they solve it, essentially observing their CST in action rather than proxying it with puzzles and other IQ-test-style games. One well-known informal test is FizzBuzz (actually a children's game: here it is to write a program to print the numbers 1–100, replacing multiples of 3 with Fizz and multiples of 5 with Buzz). It is an extremely simple programming problem, yet shockingly many CS graduates have difficulty coding it. The test has become well-known because it exposes that some students can discuss complexity theory, yet not easily translate a simple set of rules into working code. Those who fail FizzBuzz often overcomplicate or freeze, signs of a lack of basic problem decomposition skills (which is partly CST, but also being grounded enough to see an easy approach rather than conjuring a complex one). On the other hand, someone with common sense will straightforwardly implement a loop and conditionals and be done. The fact that FizzBuzz needed to exist as a filter implies that a subset of applicants lacked a certain baseline of practical coding sense despite their credentials.

In aggregate, all these pieces, productivity studies, debugging behaviours, hiring practices, and project outcomes converge on the idea that a particular kind of thinking differentiates the effective programmer from the ineffective one. That thinking is not fully captured by academic achievement or IQ, but it does align with CST: the ability to apply reasoning in context, to adapt, to foresee issues, and to make judicious decisions.

Towards a Unified View

Bringing together the theoretical and evidentiary threads, we can formulate a unified understanding of why CST is important in programming expertise. High programming skills seem to emerge from a synergy of abilities: formal analytical thinking (for algorithmic logic), creative thinking (for innovation), and practical reasoning (for judgement and adaptability). CST is essentially the glue and governor between these abilities, ensuring they are used effectively. One might say CST provides the *executive control* in the programmer's cognition, guiding when to unleash analytical rigor and when to rely on intuition or experience. It also fills the gaps where formal knowledge is silent: when facing uncertainty, when requirements change, when a novel bug appears, when working with others. In those moments, common sense is the navigator. Our hypothesis posits that individuals high in CST learn programming more readily and achieve higher proficiency because they:

- Learn more from experience: They notice patterns and principles from each bug fix or project (reflective learning), which builds their tacit knowledge base faster. Each experience reinforces heuristics for next time.
- Adapt better to new problems: Instead of being thrown off by an unfamiliar task, they draw analogies to things they do know (practical intelligence at work) and tackle it. This means they are rarely stuck for long, an attribute of top coders.
- Make fewer catastrophic mistakes: Through a combination of rational doubting (checking assumptions), they test their code, thinking about what could go wrong, which catches errors early.

- Communicate and simplify: High CST often correlates with being able to explain things clearly or simplify complexity, because the person is focused on the essence (what matters in practice). Skilled programmers often talk about finding the simple design hiding in a complex problem, essentially using common sense to strip away over-engineering.
- Possess self-regulation and metacognition: They think about their own thinking. For instance, realising they are approaching something the wrong way or they need help with that. This self-awareness is part of rational thinking and practical intelligence. It prevents time sinks and encourages collaboration when needed.
- Are more motivated by problem-solving than by proving themselves right: This somewhat touches attitude, but common sense usually entails a certain humility, caring that the job gets done, not that one's method was used. In coding, this means a willingness to refactor or throw away one's code if it is not working, without ego attachment. That often distinguishes good programmers in team settings.

If we attempt a graphical metaphor, imagine programming skills as a growing plant. Academic knowledge and IQ are like the seeds and initial conditions, but common-sense thinking is like the soil quality, sunlight, and water that allow the plant to grow. With poor soil (low CST), even a genetically strong seed (high IQ) will grow stunted. Conversely, rich soil (high CST) can help a modest seed grow strong. And of course, the best case is a strong seed in rich soil, those are the superstar programmers who are both brilliant and pragmatic.

Another way to frame it is that programming can be seen as problem-solving under constraints. IQ helps remove the constraints of limited cognitive processing (to think through logic). But CST helps navigate the constraints of reality: time, ambiguity, and human factors. A high-CST person manages trade-offs exceptionally well: they know when to use a quick-and-dirty solution and when to invest time in a robust one; when to stick to a plan and when to pivot. Essentially, they optimise not just the code, but their whole approach to the task. This is why someone with slightly less raw ability might still finish a project faster, they chose a wiser path.

Common-sense thinking is also self-reinforcing in programming. Success in delivering working software feeds back positively: the individual gains confidence in their common-sense approach, which encourages them to trust their judgement and perhaps explore bolder ideas (creative thinking). Meanwhile, a person without much CST might face repeated failures or setbacks (code not working as expected, projects ballooning in complexity). Without introspecting that their approach might be the issue, they may conclude that programming is just inherently impossible to manage, leading to frustration or rote reliance on others' solutions. This perhaps explains why some people give up on programming, not for lack of intellect, but because they did not adopt effective strategies and mindsets early on, and the endeavour became too frustrating. Our argument provides explanatory power for several phenomena observed in software development:

- It explains why adding more formal processing does not always fix projects, because if the people lack common-sense judgement, they will rigidly follow processes to the ground or find ways to game them, whereas skilled teams bend processes wisely.
- It explains why some genius coders produce unmaintainable systems. They maximised complexity beyond what is sensible or ignored user needs, so their output, though technically impressive, fails in practical terms (a lack of pragmatic balance).
- It sheds light on why diverse teams (cognitively and experientially) often outperform homogeneously brilliant teams. A mix of perspectives means more chances that someone's

common sense will catch an issue or propose a simpler approach. A room of only theoreticians might collectively miss practical blind spots. Diversity often increases the CST quotient of a team.

- It underscores the importance of teaching strategies in CS education. Recently, curricula have started incorporating more project-based and heuristic learning, effectively trying to nurture CST in students (not just COT). Techniques like pair programming in education, debugging assignments, and code reading exercises are aimed at bridging the gap between theory and practice by forcing students to engage their common sense and reflection. This author advocated pair programming in education but not in industry due to its time inefficiency and common mismatch in CST between the pair.

In conclusion, individuals who excel in programming do so not merely because they learned the right algorithms or scored high on IQ tests, but also because they approach programming with a robust common-sense mindset. They treat programming problems as multi-faceted challenges requiring flexible thinking, draw on practical experience and intuition to guide their formal skills, and continuously refine their judgement through feedback. Meanwhile, those who rely solely on intellectual ability or formal training without developing CST are at risk of plateauing or underperforming when faced with the full demands of real-world software development.

By recognising the central role of CST, we can better train students (by including practical reasoning and reflection) and better structure teams and processes (valuing intangible skills like adaptability and foresight). In a rapidly evolving software landscape, the problems and tools change, but a programmer with strong common-sense thinking will adapt and thrive no matter the language or paradigm du jour. Thus, common-sense thinking stands out as an enduring cornerstone of effective programming, linking human cognitive strengths to the craft of creating reliable, resilient and useful software. Software design is applied CST more than it is applied science.

References

Baron, J. (2008). Thinking and deciding (4th ed.). Cambridge University Press.

Bornat, R. (2014). Camels and humps: a retraction. Retrieved from https://www.eis.mdx.ac.uk/staffpages/r_bornat/papers/camel_hump_retraction.pdf

Brooks, F. P. (1986). *No silver bullet: Essence and accidents of software engineering*. *Computer, 20*(4), 10–19. https://doi.org/10.1109/MC.1987.1663532

Brooks, R. E. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies, 18*(6), 543–554. https://doi.org/10.1016/S0020-7373(83)80031-5

Charlton, B. G. (2009). Clever sillies: Why high IQ people tend to be deficient in common sense. *Medical Hypotheses, 73*(6), 867–870. https://doi.org/10.1016/j.mehy.2009.05.022

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

Dehnadi, S., & Bornat, R. (2006). The camel has two humps. Retrieved from http://www.eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf [heavily web-circulated and influential in the software community, but never formally published]

Dijkstra, E. W. (1989). *On the cruelty of really teaching computer science*. Communications of the ACM, 32(12), 1398–1404. https://doi.org/10.1145/66926.66928

Gigerenzer, G., Todd, P. M., & the ABC Research Group. (1999). *Simple heuristics that make us smart*. Oxford University Press.

Knuth, D. E. (1974). Computer programming as an art. *Communications of the ACM, 17*(12), 667–673. https://doi.org/10.1145/361604.361612

Letovsky, S. (1987). Cognitive processes in program comprehension. *Journal of Systems and Software, 7*(4), 325–339. https://doi.org/10.1016/0164-1212(87)90027-1

Lister, R., et al. (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin, 36*(4), 119–150. https://doi.org/10.1145/1041624.1041673

McBreen, P. (2002). *Software craftsmanship: The new imperative*. Addison-Wesley.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., ... & Laxer, C. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin, 33*(4), 125–140. https://doi.org/10.1145/572139.572181

McFarland, M. (2016, March 30). *Lots of coders are self-taught, according to developer survey*. The Washington Post. https://www.washingtonpost.com/news/the-switch/wp/2016/03/30/lots-of-coders-are-self-taught-according-to-developer-survey/

Pólya, G. (1945). *How to solve it: A new aspect of mathematical method*. Princeton University Press.

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. https://doi.org/10.1145/362851.362858

Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson Education.

Stanovich, K. E. (2009). *What intelligence tests miss: The psychology of rational thought*. Yale University Press.

Stanovich, K. E., & West, R. F. (2000). Individual differences in reasoning: Implications for the rationality debate? *Behavioral and Brain Sciences, 23*(5), 645–665. https://doi.org/10.1017/S0140525X00003435

Sternberg, R. J. (1985). *Beyond IQ: A triarchic theory of human intelligence*. Cambridge University Press.

Sternberg, R. J., & Wagner, R. K. (1986). *Practical intelligence: Nature and origins of competence in the everyday world*. Cambridge University Press.

Tukiainen, M., & Mönkkönen, J. (2002). Programming aptitude testing as a prediction of learning to program. In Proceedings of the 14th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2002) (pp. 45–57). Brunel University.

Vessey, I. (1985). Expertise in debugging computer programs: An analysis of the content of verbal protocols. *IEEE Transactions on Systems, Man, and Cybernetics, SMC-15*(5), 694–707. https://doi.org/10.1109/TSMC.1985.6313394

Wing, J. M. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33–35. https://doi.org/10.1145/1118178.1118215