The Myth of Programming Aptitude

Mats Danielson

[1] Dept. of Computer and Systems Sciences, Stockholm University
PO Box 7003, SE-164 07 Kista, Sweden
mats.danielson@su.se

Abstract: This article challenges the prevailing notion that programming ability is a specialised, domain-specific aptitude and instead suggests that a lot of what is regarded as programming talent stems from general-purpose common-sense thinking (CST). CST is defined as adaptive, context-sensitive reasoning grounded in intuitive judgement, experiential heuristics, and cognitive flexibility. Drawing on research from cognitive psychology, computer science education, and software engineering, the article develops a theoretical framework linking CST to programming expertise. The analysis distinguishes CST from traditional measures of intelligence such as IQ, highlighting how high analytical intelligence can coexist with poor practical reasoning. Empirical studies, including comparisons of expert and novice programmers, investigations into debugging practices, and assessments of programming education, support the hypothesis that CST is a key trait for successful programming. The article argues that programmers with strong CST often mistakenly attribute their success to a unique coding aptitude, leading to self-identification as natural programmers and missed opportunities to apply their cognitive strengths in other fields, perhaps more to their liking if only they knew. The article critiques the validity of programmer aptitude tests and formal CS curricula as predictors of real-world software competence, noting the prevalence of self-taught professionals and interdisciplinary career entrants in the field. Finally, it explores how high-CST individuals might flourish in leadership, entrepreneurship, design, or other domains demanding practical intelligence. By reframing programming also to be an application of CST rather than only an isolated skillset, the article advocates for a broader understanding of cognitive talent and encourages more deliberate career exploration among programming-inclined individuals.

Keywords: Common Sense Thinking, Programming Aptitude, Cognitive Flexibility, Practical Intelligence, Software Development, Career Misattribution

Introduction

In the discourse on software development skills, the spotlight often falls on programming aptitude, an assumed innate knack for code that some individuals possess. Many highly skilled programmers attribute their success to this programming-specific talent, viewing themselves as naturally predisposed to excel in writing software. However, this article advances a different thesis: that a major enabler of success in programming is not a mysterious domain-specific gift, but rather a high capacity for *common sense thinking* (CST) and general practical reasoning. By CST, we refer to the broad ability to apply sound judgement, adaptive problem-solving, and context-aware reasoning in everyday situations. We argue that many accomplished programmers thrive due to strong CST abilities (e.g. intuitive reasoning, cognitive flexibility, pragmatic decision-making), which they mistakenly credit to a unique coding brain. As a result, these individuals often pursue programming as a primary career under the belief that it uniquely matches their talents, when in fact their underlying cognitive skills could have enabled success in numerous other fields. This misattribution can lead to a narrowed professional identity (see below) and missed opportunities for career exploration.

To support this argument, the article is organised into two main parts. The first part develops a theoretical and cognitive model of common-sense thinking and examines its relevance to programming. We draw on literature from cognitive psychology and computer science to define CST, distinguishing it from conventional measures of intelligence, and discuss cognitive flexibility and programming as a craft to show conceptual links between CST and the demands of coding. The second part builds the case that CST underlies much of programming expertise,

using research from software psychology and education. We review studies of expert vs. novice programmers, attempts to measure programming aptitude, and evidence from career trajectories to demonstrate that what is often labelled as programming talent is better explained as domain-general reasoning ability. Finally, we reflect briefly on the implications: if high-CST individuals recognised their skills' transferability, they might broaden their career horizons to fields where practical thinking is equally or even more valued. Throughout, we cite academic literature (cognitive science, software engineering, and education research) to ground the analysis in established scholarship. By reframing programming prowess as an instance of applied common sense, we aim to challenge prevailing assumptions about talent in computing and encourage a more expansive view of one's potential career pathways.

Defining Common Sense Thinking

Common sense has long been understood as the intuitive ability to make sound judgements in everyday situations, transcending any one domain of knowledge. In psychological terms, *common-sense thinking* (CST) can be defined as the capacity for adaptive, context-sensitive reasoning that draws on informal knowledge and prudent judgement. It goes beyond rote intelligence or formal logic, encompassing the practical know-how that enables individuals to navigate real-world problems. CST involves a blend of rapid, experience-based intuition and slower, deliberate analysis, aligning with what decision theorists often call *rational thinking* in everyday life. Notably, CST is not measured well by standard IQ tests. As Stanovich (2009) observes, IQ exams effectively assess abstract problem-solving and memory, but they fail when it comes to measuring those abilities important to making good judgements in real-life situations, such as the capacity to evaluate information and override cognitive biases. In other words, traditional intelligence tests capture only a subset of good thinking skills, missing the practical reasoning and judgement component that defines common sense. High IQ individuals can therefore still act foolishly in everyday matters, a phenomenon sometimes termed the clever-fool or even nutty-professor effect (high analytic intelligence paired with surprisingly poor everyday judgement).

Researchers have conceptualised CST in various ways. Sternberg, for instance, distinguishes *practical intelligence* from analytical intelligence (Sternberg, 1985). In his triarchic theory, practical intelligence, often equated with common sense, is the ability to apply knowledge to real-world contexts, solve ill-defined problems, and adapt to one's environment, in contrast to analytic intelligence measured by academic tests (Sternberg and Wagner, 1986). Practical intelligence relies heavily on *tacit knowledge*, the informal know-how one accumulates through experience (Polanyi, 1966; Sternberg and Wagner, 1986). A person high in practical intelligence has an extensive store of unspoken rules of thumb for what to do in various situations, and an intuitive grasp of which strategies are likely to work. Importantly, studies find that practical intelligence measures (e.g. situational judgement tests) correlate weakly with traditional IQ scores, reinforcing that one can be academically smart yet lack common sense, and vice versa. Stanovich (2009) similarly argues that rational thinking, the kind needed for good daily decisions, is a separable cognitive domain from general intelligence. He notes that intelligence tests do not assess whether someone can avoid reasoning traps or frame problems appropriately in real life. Indeed, Stanovich and others have documented individuals with superior analytical skills who nonetheless commit logical mistakes or poor decisions in personal and financial matters because they lack reflection and context-awareness (Stanovich and West, 2000). These findings underscore that *common-sense reasoning* is a distinct facet of cognition, involving

skills like inference from experience, awareness of one's cognitive biases, and context-dependent judgement.

Charlton (2009) offered a description of the disconnect that can occur between high intelligence and common sense. He coined the term clever sillies for high-IQ people who make foolish decisions in everyday affairs. Charlton's hypothesis is that very intelligent individuals have a tendency to overapply abstract, analytical thinking even when a simple heuristic or instinct would yield a better result. In his words, an excessively analytic thinker may over-ride those instinctive and spontaneous forms of evolved behaviour which could be termed common sense. In social situations, for example, basic common sense (informed by social intuition and norms) usually produces the right response, whereas a purely analytical solution may be tone-deaf. Charlton suggests that the most intelligent people are more likely than those of average intelligence to have novel but silly ideas in domains like social interaction, precisely because their reasoning detaches from the practical realities that typical people consider. While aspects of Charlton's conjecture are controversial, it highlights the key point that common sense involves more than raw analytic power. It requires knowing when and where to apply one's analytical abilities or when to rely on gut-level judgement. In sum, CST can be viewed as a *master skill* of knowing how to think and act appropriately across a range of everyday contexts. It synthesizes perceptive observation, accumulated experience, intuitive heuristics, and self-important reasoning. This broad cognitive flexibility is what enables someone to, say, troubleshoot a household problem, handle interpersonal situations, or make decisions under uncertainty with reasonable success, even without training in each specific domain.

It is important to clarify that common-sense thinking is a *process*, not just a static set of beliefs. Everyone has a body of *common-sense knowledge* (basic facts about how the world works that every person knows), but effectively using that knowledge in novel or complex situations is the hallmark of CST. Some theorists distinguish between a quick, reflexive type of common sense and a more deliberative type. For example, an analogy can be drawn to Kahneman's Systems 1 and 2 (2011): we might say *System A* common sense is the fast, intuitive aspect, the instantaneous recognition of obvious consequences (e.g. knowing not to touch a hot stove or understanding a shivering person is cold), whereas *System B* common sense is the slower, reflective reasoning through less obvious scenarios. Most people exhibit decent System A common sense in everyday routine situations. But truly skilled common-sense thinkers excel at System B: they can project forward, imagine consequences, and adapt their thinking to unusual or complex real-life problems in a way that goes beyond automatic intuitions. Being good at System B CST is rarer, even among those with high education or IQ. This framework again resonates with Sternberg's practical vs. analytical intelligence distinction, one needs more than just accumulated facts; one needs the *thinking strategies* to use knowledge judiciously.

Cognitive Flexibility

A core component of CST is *cognitive flexibility*, the ability to shift one's thinking strategy or perspective in response to the demands of the context. Flexible thinkers can move between different modes of reasoning (intuitive vs. analytical, top-down vs. bottom-up) and adjust when new information arrives. Cognitive flexibility is considered a key executive function underlying adaptive problem-solving (Spiro and Jehng, 1990; Diamond, 2013). In practical terms, it means not getting stuck in one approach: if a straightforward method is not working, a high-CST individual will nimbly try alternative angles, whereas a more rigid thinker might perseverate on the same failing strategy. This flexibility is closely tied to the use of heuristics, simple rules or shortcuts that often yield good results efficiently. Far from being a mark of laziness, judicious

use of heuristics is a sign of practical intelligence. Gigerenzer et al. (1999) have shown that fast and frugal heuristics can solve many real-world decisions remarkably well, sometimes outperforming complex analytical models. For example, when making everyday judgements under uncertainty, people rely on heuristic cues (like the credibility of a source or the recognisability of an option), which usually (if not always) lead to sensible outcomes. CST involves knowing which heuristic to apply and when to override it. A common-sense thinker might use a rule of thumb (if a deal sounds too good to be true, it probably is) for expedience, but also remain aware of exceptions and be ready to analyse further if something feels off.

In contrast, less practical thinkers might either blindly trust a heuristic when it is inappropriate, or reject heuristic thinking altogether in favour of laborious analysis even for trivial problems. Both extremes can lead to suboptimal decisions. Effective common sense lies in balancing intuitive and analytical approaches. Cognitive scientists note that humans are satisficers. We aim for a good-enough solution given time and information constraints (Simon, 1956). CST is essentially the intelligent satisficing that comes from experience: the person has seen which solutions tend to work, and can quickly home in on a plausible approach without needing an exhaustive search. As new information emerges, they readily update their approach. This flexible, heuristic-driven problem-solving is evident in many expert behaviours across domains. For instance, experienced physicians use pattern recognition (a fast, schema-based process) to diagnose typical cases, yet they remain vigilant and switch to a deeper analytic mode if symptoms do not fit the usual pattern, a cognitive flexibility that novice medical students must learn. Similarly, an expert auto mechanic might initially guess a car issue from a quick observation (based on years of tacit knowledge), but will test that hypothesis and switch to systematic troubleshooting if the first guess is wrong. Such adaptability is a hallmark of CST.

Programming as a Craft

How does all this relate to computer programming? At first glance, writing code might seem to be a purely analytical task, one that demands logical reasoning, mathematical thinking, and formal knowledge of syntax and algorithms. Indeed, computer science as an academic field grew out of mathematics and engineering, emphasising abstract reasoning and theoretical rigor. Classic computer programming involves designing correct procedures and data structures, which certainly calls for logical precision. However, decades of software practice and research have revealed that programming is not a purely formal activity; it is equally an art or craft that relies on experience, intuition, and informal reasoning (Knuth, 1974). In the 1970s, Knuth described computer programming as an art as well as a science, highlighting that elegance and creativity in coding go beyond a mechanical application of rules. More recently, the software craftsmanship movement (McBreen, 2002) has argued that good programming resembles traditional craftsmanship: excellence comes from apprenticeship, practice, and cultivated judgement, not merely from academic knowledge.

When we view programming as a craft, the role of CST becomes apparent. Craftwork, whether carpentry or coding, involves making many decisions that have no single correct answer but require *practical judgement*. A master carpenter knows how to improvise when a material is flawed or a design needs adjustment. This know-how is not fully captured in engineering blueprints, just as a great programmer's decision about how to structure a complex module draws on intuition and past experience, not only on textbook algorithms. True software quality stems from skilled application of consistent practices balanced with creative design, rather than rigid adherence to process. In his view, low-level standardisation (like consistent coding style) can reduce cognitive load and errors, but high-level architecture remains an arena for creative

judgement. This implies that developers must use common sense to decide when to follow established patterns versus when to invent a new approach for a unique problem. Over-reliance on formal methodology can backfire in software development. Brooks (1987) argued that there is no single technique or technology that can magically simplify software engineering because much of the difficulty is essential complexity, related to understanding the real-world problem and requirements, which requires human insight and judgement. In practice, experienced programmers often talk about *smelling code* (using intuition to sense a problem in design), *refactoring for clarity* (improving a program by applying aesthetic and practical judgement), or *using a simple, common-sense solution instead of a convoluted clever hack*. These are all reflections of CST in programming (Danielson, 2012).

Even the process of writing a correct program can involve common-sense reasoning. For example, consider the approach to solving a simple programming task: a novice might dive straight into coding, whereas an expert pauses to consider edge cases and what the problem *really* requires (which is a common-sense step of clarifying goals). The expert's planning step often reflects asking practical questions, what are reasonable inputs, what could go wrong, what's the simplest way to handle this?, which go beyond merely applying a known algorithm. If the task at hand is slightly ambiguous or underspecified (as real-world problems often are), the programmer must use judgement to interpret the requirements. This involves understanding the user's likely intent, a form of social common sense. In large software projects, programmers also operate in teams, meaning communication and understanding others' code is as important as writing one's own. Social and collaborative aspects of programming engage what might be called social common sense: awareness of teammates' perspectives, willingness to ask for help or review (vs. overconfidence), and prudent decision-making in group contexts (Danielson, 2012). These soft skills differentiate successful programmers and are fundamentally grounded in CST (good judgement, self-awareness, and adaptability in social situations). In recognition of this, industry hiring practices have evolved to evaluate not just technical knowledge but also problem-solving approach and teamwork. Many tech companies now include pair programming interviews or take-home projects in hiring to see how candidates *think and adapt* in realistic scenarios, rather than relying only on puzzle quizzes. This shift implicitly acknowledges that a coder's effectiveness lies in their practical thinking process, not just their memorised knowledge.

In summary, programming is a rich cognitive activity that engages analytical intelligence *and* common-sense thinking. A programmer certainly needs logical ability and domain knowledge (computer science concepts), but to use those effectively they also need CST: the ability to understand what problem needs solving in context, to make judgement calls among alternative solutions, to anticipate pitfalls (e.g. Will this approach be error-prone or hard to maintain?, a question of practical foresight), and to learn from trial and error. A cognitively inflexible programmer may write code that is technically complex but not robust in the real world, or they may struggle when a problem is stated in an unfamiliar way. By contrast, a programmer with high common sense might more readily simplify a complex problem, draw analogies to past tasks, and apply iterative trial-and-error until a working solution emerges. Such traits have been observed in expert programmers, as we explore next. First, however, we consider how the industry and educational field have attempted to identify programming aptitude and why these attempts point toward general thinking skills rather than domain-specific magic.

The Myth of Programming Aptitude

The notion that programming success comes from a unique innate aptitude has a long history in computer science education. As early as the 1960s, companies sought tests to predict who would make a good programmer. The most famous was IBM's Revised Programmer Aptitude Test (PAT), which consisted of puzzles in number series, figure analogies, and basic calculations, essentially an IQ-test-like battery targeting logical pattern recognition and grade 7–9 arithmetics (McNamara and Hughes, 1961). It is telling that what poses as a programming aptitude test is really an IQ test in disguise, quite far from testing anything CST-like (Danielson, 2014). While widely used in hiring, the IBM PAT's ability to predict actual job performance was quite limited over time, even though the initial IBM-sponsored study showed good correlations (McNamara and Hughes, 1961). Already Mayer and Stalnaker (1968) noted that such tests did not capture many qualities that affect real programming outcomes. Through the 1970s and 1980s, other aptitude tests were developed (Wolfe, 1971, 1971b; Evans, 1976), adding more sections (following instructions, symbol manipulation, etc.). Wolfe was an open critic of the IBM PAT (Esmenger, 2010) and removed the arithmetic part, replacing it with multi-step problems, but the fundamental problem remained. To complicate matters, aptitude, which is the potential to learn a task, is not necessarily a good indicator of performance on the task per se. In other words, even if one could measure raw ability to learn programming concepts, it would not straightforwardly translate into how well someone actually writes or maintains programs in practice. Many other factors (motivation, problem-solving approach, perseverance, etc.) mediate performance. McNamara and Hughes (1961) discuss this for the IBM PAT. And, above all, high IQ and high CST do not correlate highly, so the tests miss their aim twofold. A historical overview of programming aptitude tests is given in (Esmenger, 2010, Ch.3).

Empirical studies in CS education support a more nuanced reality. Multi-institution research has shown that introductory programming courses see a wide distribution of achievement, with a significant minority of students failing to grasp basic concepts (Bennedsen and Caspersen, 2007; Lister et al., 2004). These studies also highlight that teaching methods and prior exposure play roles. For example, a multinational study by Lister et al. (2004) found that many students struggle with reading and tracing code, a fundamental skill, by the end of CS1. This suggests the issue may lie partly in what is taught (and how), rather than solely in the student's inherent ability. Still, educators often anecdotally observe that some students pick up programming concepts almost intuitively. Our thesis is that these natural programmers are likely leveraging strong general reasoning skills. Essentially, they apply common-sense strategies to a new domain, rather than exhibiting a domain-specific genetic gift. Supporting this, consider that a large proportion of professional developers do not have formal CS degrees and are self-taught to a significant extent. Many successful coders originally trained in other fields (physics, music, even history) and transitioned into programming. Those who excel often had what could be called *domain-general talent*: they were good problem solvers or logical thinkers in general, which enabled them to learn programming outside a traditional classroom. In essence, they had high CST, they approached coding pragmatically, experimented and learned from mistakes, and drew on analogies from other domains (e.g. a physicist-turned-programmer might use physical intuition about systems when structuring simulation code). This is not to deny that certain cognitive abilities (e.g. spatial visualisation or working memory) can help in coding; rather, those abilities are not exclusive to programming but are beneficial in many technical/problem-solving arenas.

It is telling that when companies hire programmers, they increasingly focus on seeing *how* candidates solve problems instead of querying specific knowledge. Modern hiring interviews commonly include a live coding exercise or a take-home project where the candidate must devise a solution. Interviewers often pay attention to whether the candidate thinks aloud sensibly, breaks the problem down, anticipates edge cases, and reacts well to hints or new requirements, essentially assessing their CST in action. A person who instantly starts writing code without clarifying requirements might be viewed as less promising, even if they eventually get a solution, compared to someone who asks prudent questions and plans an approach. Many seasoned engineers have witnessed colleagues with stellar academic credentials struggle in real projects because they lack common sense in software development (for example, they might over-engineer a solution that fails to meet the actual user needs). These anecdotes align with research on the limited predictive power of academic performance for software job performance. Academic excellence correlates with general intelligence and conscientiousness, but in a complex job like software development, those with moderate academic profiles can outperform brainiacs if they have superior practical problem-solving skills and judgement. As one commentary analogised, a high IQ (or by extension, strong formal skills) is like height in basketball, it is an advantage up to a point, but there is a lot more to being a good player than being tall, and there is a lot more to being a good thinker than having a high IQ. The lot more refers to exactly those CST traits we have discussed.

Expert Programmers

If CST is indeed a key ingredient in programming prowess, we should expect to see differences in how high-performing programmers approach their work compared to novices or less successful programmers. This is exactly what the literature on the psychology of programming finds. Early studies by Brooks (1983) and others in the 1980s looked at how expert programmers comprehend code and solve programming problems. Brooks proposed that experts form *hypotheses* about the code's purpose and structure as they read, using a top-down strategy guided by expectation. For instance, an expert reading an unfamiliar piece of code might quickly guess this looks like a sorting routine and then verify if that hypothesis holds, rather than reading every line with equal attention. This top-down inference relies on the expert's prior knowledge of common patterns (a form of tacit knowledge) and practical reasoning. They know it saves time to *interpret* code in light of likely intent, instead of treating it as an abstract sequence of symbols. A novice, lacking that experience, often goes line-by-line trying to simulate the code mechanically, which is cognitively exhausting and easy to get lost in. Letovsky's 1987 study memorably described programmers (particularly skilled ones) as opportunistic processors, meaning they do not follow a fixed algorithm to understand or write programs, but rather switch between strategies as needed. They might use a bottom-up approach (details first) in some parts of the code and a top-down approach (big picture first) in others, depending on which seems more fruitful. This flexible switching is a direct manifestation of cognitive flexibility. The expert can adjust their mental strategy when they hit a confusing part of code, perhaps pausing to recall domain knowledge or to draw a quick analogy.

Another area where expert vs. novice differences appear is debugging. Debugging a program, finding and fixing errors, is notoriously difficult and arguably where true programming skill is tested. Studies have consistently found qualitative differences in how experts debug. Vessey (1985) showed that expert debuggers approach bugs with a systematic, hypothesis-driven method: they use the symptoms of a problem to formulate possible causes, devise tests to narrow down the cause, and home in on the bug through logical elimination. Less skilled

debuggers, on the other hand, often use an unsystematic approach, for example, making many changes at once, inserting random print statements, or guessing without a clear plan (Soloway and Spohrer, 1989). Vessey found that experts often mentally simulate what the program *should* do and compare it to what it *is* doing to pinpoint where things go wrong, a process requiring both domain knowledge and rational inference. Novices were more likely to make wild edits that sometimes introduced new errors. The expert behaviour here illustrates common sense thinking: rather than thrashing, the expert debuggers applied a form of scientific method, pose a hypothesis ("given this error, something is likely wrong in module A since that handles related functionality"), then test it ("let us add a check or use a debugger to inspect module A's state"). This is goal-directed and efficient. It shows *good judgement* in problem-solving, as opposed to trial-and-error. In everyday terms, it is like the difference between a doctor who diagnoses by considering symptoms and ordering targeted tests versus one who blindly prescribes various drugs to see what sticks. Debugging expertise is thus a window into the broader reasoning skills of the programmer: experts are essentially exercising rational, commonsensical approaches under the hood.

Interestingly, these differences are not simply due to experts knowing more programming language syntax or tricks. They pertain to *how experts think* about problems. A seminal early study by Sackman et al. (1968) found huge productivity differences between programmers (the oft-cited 10 X difference, where the best programmers were an order of magnitude more productive than the worst in completing the same tasks). While some of that variance might be due to experience or intelligence, much of it, as later analyses suggested, comes from the approaches and mental models programmers use (Weinberg, 1971). One could argue that high performers apply common-sense thinking to manage complexity: they find simpler ways to implement a requirement, they reuse known solutions, they avoid rabbit-holes that waste time, and they verify as they go. Less effective programmers might get enamoured with a complex solution (sometimes to demonstrate their cleverness) or fail to test assumptions early, leading to time-consuming backtracking. An example that circulated in the industry is the *FizzBuzz test*: a trivial programming task (print numbers 1–100 with certain substitutions) used to screen candidates. Despite its simplicity, many job applicants with CS degrees struggled to solve it, often by over-complicating the approach. Those who failed tended to either freeze (overthinking a simple loop) or attempt elaborate code for something that needs just basic control structures. A candidate with solid common sense would quickly see a straightforward solution (a loop and conditional checks) and implement it, whereas some high-theory individuals bizarrely over-engineered or could not translate the simple requirements into code. The FizzBuzz test underscores that practical problem decomposition, a CST skill, is not universal even among those formally trained, and its absence is very conspicuous on an easy task.

Team practices in successful software projects also highlight the role of collective common sense. Techniques like code review or pair programming essentially inject an extra dose of judgement into the development process. A second person reviewing code can catch obvious mistakes or design decisions that, in hindsight, are ill-advised, in effect, adding more common sense to avoid silly errors or overly convoluted solutions. If programming were purely an application of objective rules, one might expect formal verification or strict processes to be sufficient. But in reality, even with methodologies in place, the human element of judgement is important. When that human judgement fails, the results can be costly. A classic example is the NASA Mars Climate Orbiter crash (Danielson, 2012) caused by a units mismatch (one team used imperial units, another metric). This was not a complex algorithmic failure, it was a *common-sense* failure in communication and checking assumptions. Any engineer applying basic

CST would think, "Are we all using the same units throughout?" A simple sanity check that was overlooked in a highly sophisticated organization. It demonstrates that brilliance in aerospace engineering did not guarantee the mundane diligence of practical reasoning. Many large software failures have similar post-mortems: the root cause often lies in assuming something obvious that turned out differently, or not double-checking because each team thought the other had done so. Cultivating a habit of *sanity checks* and *asking stupid questions* is essentially promoting common sense in a team environment, and the best teams have a culture that encourages this. No one is ridiculed for pointing out a seemingly obvious concern, because those concerns might save projects from disaster.

From these examples, we see a pattern: effective programmers utilise reasoning patterns that are very much like common-sense strategies in everyday problem-solving. They simplify, they analogise to known solutions, they anticipate possible issues (what if inputs are not as expected? what if the user does X instead of Y?), and they balance trade-offs. These skills resemble what good decision-makers do in any field, whether a general solving a battlefield logistics problem or a manager making a business plan. It is not primarily about writing code, it is about solving problems in a context that happens to involve code. This leads to an important realisation: someone who is adept at this kind of reasoning could likely apply it in many domains, not just programming. Why, then, do so many such individuals end up identifying solely as programmers? Part of the reason is the misattribution phenomenon, because their introduction to a challenging task was through programming and they succeeded, they credit *programming-specific* talent rather than general thinking talent. It is a cognitive attribution error bolstered by the surrounding narrative in tech (the myth of the 10 X programmer, i.e. the brilliant coder mystique). The next section will discuss the implications of this misattribution on career choices and self-perception.

Misattribution of Talent

If a high-CST individual happens to excel early in programming, they are likely to infer, logically but perhaps erroneously, that they have a special gift for programming in particular. Psychologically, humans tend to attribute success to personal traits. A student who breezes through an introductory coding class might conclude I'm just naturally good at coding, especially if peers struggled. This could be reinforced by instructors or peers praising their coding talent. However, what if that student also would have excelled at other complex problem-solving tasks like engineering design, scientific research, or even strategic games? Unless they try those, they might never know. There is a risk of *career identity foreclosure*, where an individual settles prematurely on a professional identity without fully exploring alternatives (Erikson, 1968; Marcia, 2002). In the case of programmers, the lucrative and exciting tech industry can further incentivize sticking with coding. Why consider switching when demand and rewards for programmers are so high? Thus, many people with strong general reasoning skills get channelled into programming early (in education or early career) and stay there.

One could argue that this is not a problem per se, after all, programming is a valuable skill and society needs good programmers. The issue is not that high-CST folks become programmers, but that they and their employers sometimes over-ascribe their success to programming-specific abilities, potentially limiting their growth. For instance, a software developer with excellent systems thinking and people skills might actually make a superb project manager or product strategist. Yet if they internally believe their worth is as coders because that's what they are good at, they might be reluctant to pursue management roles (fearing they lack the required special management talent, not realising they already exercise much of it in technical

leadership contexts). Similarly, someone might refrain from transitioning to another field (say starting a business or going into data science for healthcare) because of self-labelling as a tech person and assume the skill set would not translate, when in fact the CST, learning ability, and problem-solving would likely transfer quite well in many cases.

Conversely, the glorification of innate programming talent can discourage people who do not initially self-identify as geniuses from entering or staying in the field, even if they have ample common sense and could become very competent developers. The so-called geek gene myth in computer science education has been criticised for this reason (Lewis, 2012). By re-framing success in terms of learnable general skills (like reasoning strategies) instead of innate genius, we open the door for more diverse talent to thrive in programming and beyond. It is noteworthy that some leading voices in CS education now emphasise *computational thinking* as a universal skill (Wing, 2006). Computational thinking involves formulating problems in a way that a computer (or computational approach) can solve, using concepts like decomposition, abstraction, and algorithmic thinking. Wing's advocacy was essentially to teach everyone the problem-solving mindset that underlies programming, precisely because it is broadly useful outside programming. In a sense, this is the mirror image of our thesis: Wing argued that programming-style thinking benefits all fields, while we argue that a common-sense thinking style benefits programming. Both perspectives dissolve the rigid boundary between coding skills and general cognitive skills, but from different perspectives.

Career development research also suggests that people who identify strongly with a narrow skill may limit their exploration of other roles (Ibarra, 2003). High-CST individuals may feel very competent in the programming domain, which is a rewarding feeling, and thus hesitate to become a novice again in a new domain, even if they can excel there too. This comfort zone effect means a brilliant programmer might remain in pure coding roles well past the point where they are actually growing, instead of, say, branching into interdisciplinary work where they could apply their abilities to, for example, solve complex logistics problems, innovate in education technology, or lead a technical organisation. When the industry reinforces that their highest value is churning out code, they may internalise that perception. In some cases, this can lead to mid-career crises when programmers find themselves wanting new challenges but unsure of their identity outside coding. Realising that their core strength is CST (problem-solving, quick learning, and adaptation) rather than a mystical bond with the computer can be empowering and liberating. It means they can tackle *any* new field with a similar approach to how they tackled programming, by learning the basics and then applying common sense and systematic practice.

Finally, it is worth reflecting on fields or roles where strong CST is equally prized. Leadership is a prime example: effective leaders often have to make decisions with incomplete information, navigate human factors, and adapt plans on the fly, essentially making heavy use of common sense and practical intelligence. It is not uncommon to see former engineers or programmers rise to management and do exceptionally well; their technical background might be less important than their underlying problem-solving and decision-making acumen. Entrepreneurship is another path: entrepreneurs need to improvise solutions, understand customer needs (social intuition), and allocate resources wisely, all areas where CST is vital. In fact, a study in entrepreneurship found that cognitive flexibility is an important predictor of success for entrepreneurs, more so than domain-specific expertise. High-CST individuals might also flourish in fields like product design (which requires empathy and pragmatic creativity), law (applying general principles to specific cases demands practical reasoning), medicine (especially diag-

nostics and patient interaction), or research in sciences (where framing a problem and interpreting data require intuition beyond formulas). Some people indeed make such transitions; for example, a programmer with a passion for public policy might leverage their analytical and pragmatic thinking to become an effective data-driven policymaker. Our point is not that all programmers should leave coding, but that they should recognise the portability of their *thinking skills*. By broadening their self-concept from I am good at programming to I am good at solving complex problems in context, they may unlock confidence to explore interdisciplinary careers or leadership roles when the time is right.

To conclude this section, the evidence and arguments we surveyed converge on a central idea: what differentiates standout programmers from the rest is largely *how they think*, not just what they know. The superior use of common sense thinking, adaptive reasoning, sound judgement, and learning from experience are the silent forces behind their achievements. However, because programming has been the arena in which they exercised these faculties, it is easy for them (and observers) to attribute success solely to programming talent. This attribution, while understandable, overlooks the generality of their skills. It has led to a culture in tech that mystifies programming ability and sometimes narrows individuals' perceptions of their own potential. By demystifying programming prowess and recognising CST as the enabler, we can encourage a healthier perspective: programming is one of many outlets for cognitive talent, and a person who excels in it could likely excel elsewhere too, should they wish to venture. In the next section, we briefly consider the implications of this thesis for education and career development and highlight other domains where CST can be as transformative as it is in software.

Broader Applications of CST Skills

Viewing programming success through the lens of common-sense thinking has several implications. In computer science education, it suggests that curricula should explicitly cultivate practical reasoning skills, not just theoretical knowledge. Encouraging students to reflect on their problem-solving process, to engage in project-based learning, and to work in pairs or teams (practices that force them to externalise and adapt their thinking) can help nurture CST alongside coding knowledge. Indeed, some modern CS pedagogies already emphasise these aspects: for example, integrating debugging assignments that reward systematic reasoning, or using code review exercises to teach students to spot flaws through common-sense critiques. By demystifying the gifted programmer idea, educators can reassure struggling students that improvement is possible by working on strategy and approach, much like one can learn better decision-making, rather than feeling doomed by a lack of an innate trait.

For individuals and career development, the takeaway is to recognise one's *core competencies*. A high-CST programmer should realise that their value lies not only in knowing the latest framework but in their ability to learn and adapt. This realignment of self-perception can make them more resilient in a fast-changing tech landscape: languages and technologies come and go, but someone with strong general problem-solving skills and common sense will quickly pick up new tools. It can also embolden them to step outside of pure coding roles, knowing that their underlying skill set is broadly applicable. We increasingly see careers that blend programming with other fields (bioinformatics, digital humanities, fintech, etc.), and such interdisciplinary paths often reward those who have both domain knowledge and the practical thinking to integrate it with computing. High-CST individuals are prime candidates to lead in these areas because they can connect the dots and make decisions beyond the textbook, essentially acting as common-sense brokers between domains.

Beyond programming, as promised, we lightly reflect on fields where CST is equally or more important. Any field that deals with complex, open-ended problems benefits from common-sense thinking. Management and leadership roles prize good judgement, adaptability, and people-savvy, precisely the strengths of CST. It is no surprise that many tech companies eventually move some of their best engineers into management; those who succeed are not necessarily the top coders but those with broader decision-making ability. Entrepreneurship has been mentioned: starting a company requires handling novel challenges daily, pivoting strategies, and balancing analytical data with gut feel about the market. A former programmer with high CST might excel here by applying their systematic thinking to business problems while staying pragmatic (not every decision can be made with complete data, sometimes one must rely on educated intuition). Research and innovation domains also rely on CST when formal knowledge reaches its limits, deciding which hypotheses to pursue or which engineering approach is worth prototyping often comes down to a researcher's practical wisdom and intuition about what might work. Pólya's 1945 classic *How to Solve It* emphasised teaching heuristic reasoning to budding mathematicians for this reason. Design and architecture (whether software architecture or architectural design) require understanding user needs, constraints, and likely failure modes, again blending creative and practical thought. We can even consider education and mentorship: A teacher with strong common sense will be able to adapt their methods to different students, an ability as important as content expertise. Thus, someone who has honed CST in programming could transition into teaching programming (or another subject) effectively, using their adaptive thinking to convey concepts in accessible ways.

Of course, the ideal scenario is not an either/or. Individuals can apply their CST *within* programming and also cross-pollinate it to other endeavours. Many high-performing technologists eventually find themselves in hybrid roles (e.g. an open-source project leader who does coding, community management, and design decisions). Recognising that their success stems from CST helps them embrace these multifaceted roles instead of feeling they are moving away from their one true talent. It also fosters humility: if we acknowledge that common-sense thinking is a skill developed through experience and reflection, then one can always improve it. The myth of the innately gifted programmer sometimes led to a fatalistic view that one either has it or not. Our reframing encourages a growth mindset. Even those not initially excelling in programming can become excellent by building up their general reasoning approaches. There are documented cases of students who struggled early but became very capable developers through perseverance and strategy training, disproving the idea of fixed unteachability (Robins et al., 2003).

In closing, treating software development as an exemplar of applied common sense rather than as a sui generis ability aligns the field with the broader human toolkit of problem-solving. It demystifies expert programmers as not unicorns with alien brains, but as clever problem-solvers who have leveraged a set of widely relevant cognitive skills. This perspective has a unifying effect: it connects the concerns of computer science with those of psychology (how do we make good decisions?) and education (how do we teach adaptive thinking?). It also reminds high achievers in programming that their aptitude is a general asset, a tree that could bear fruit in many gardens, not just the software industry. By all means, those who love programming should continue doing it; the point is simply that they do so by choice, not by a sense of limitation. After all, as one might say, software design is applied CST more than it is applied science. Recognising this can enrich the self-understanding of those who excel at programming, while opening doors to new opportunities that suit them better and where their abilities can make more of an impact, instead of being trapped in an unfitting career.

References

Bennedsen, J., & Caspersen, M. E. (2007). Assessing process and product: A practical lab exam for an introductory programming course. *Innovations in Teaching and Learning in Information and Computer Sciences, 6*(4), 183–202.

Brooks, F. P. (1987). No silver bullet: Essence and accidents of software engineering. *Computer, 20*(4), 10–19. doi: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. doi: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. doi:10.1016/j.mehy.2009.05.022

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

Danielson, M. (2014). The IBM Revised Programmer Aptitude Test, a comment. Unpublished.

Diamond, A. (2013). Executive functions. Annual Review of Psychology, 64, 135–168. https://doi.org/10.1146/annurev-psych-113011-143750

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

Erikson, E. H. (1968). Identity: Youth and crisis. W. W. Norton & Company.

Ensmenger, N. L. (2010). *The computer boys take over: Computers, programmers, and the politics of technical expertise*. MIT Press.

Evans, M. W. (1976). An experimental evaluation of analytic programming styles. Communications of the ACM, 19(5), 267–273. https://doi.org/10.1145/360051.360057

Gigerenzer, G., Todd, P. M., & the ABC Research Group. (1999). *Simple Heuristics That Make Us Smart*. Oxford University Press.

Ibarra, H. (2003). Working identity: Unconventional strategies for reinventing your career. Harvard Business Press.

Kahneman, D. (2011). *Thinking, fast and slow*. Farrar, Straus and Giroux.

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

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

Lewis, C. M. (2012). The importance of students' attention to program state: A case study of debugging behavior. In Proceedings of the ninth annual international conference on International computing education research (pp. 127–134). ACM. https://doi.org/10.1145/2361276.2361300

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

Marcia, J. E. (2002). Identity and psychosocial development in adulthood. Identity: An International Journal of Theory and Research, 2(1), 7–28. https://doi.org/10.1207/S1532706XID0201_02

Mayer, D. B., & Stalnaker, A. W. (1968). Selection and evaluation of computer personnel—the research history of SIG/CPR. In Proceedings of the 1968 ACM National Conference (pp. 657–670). Association for Computing Machinery.

McCracken, M., et al. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin, 33*(4), 125–140. doi:10.1145/572139.572181

McNamara, W. J., & Hughes, J. L. (1961). A review of research on the selection of computer programmers. *Personnel Psychology, 14*, 39–51.

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

Pólya, G. (1945). *How to Solve It: A New Aspect of Mathematical Method*. Princeton University Press.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. Computer Science Education, 13(2), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

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. doi:10.1145/362851.362858

Simon, H. A. (1956). Rational choice and the structure of the environment. Psychological Review, 63(2), 129–138. https://doi.org/10.1037/h0042769

Soloway, E., & Spohrer, J. C. (Eds.). (1989). Studying the novice programmer. Lawrence Erlbaum Associates.

Spiro, R. J., & Jehng, J. C. (1990). Cognitive flexibility and hypertext: Theory and technology for the non-linear and multidimensional traversal of complex subject matter. In D. Nix & R. Spiro (Eds.), Cognition, education, and multimedia (pp. 163–205). Lawrence Erlbaum Associates.

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. doi: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 *Proc. 14th Workshop of the Psychology of Programming Interest Group (PPIG)* (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, 15*(5), 694–707. doi:10.1109/TSMC.1985.6313394

Weinberg, G. M. (1971). The psychology of computer programming. Van Nostrand Reinhold.

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

Wolfe, J. M. (1971). *Wolfe programming aptitude test (school edition)*. In T. C. Willoughby (Ed.), *Proceedings of the Ninth Annual SIGCPR Conference* (pp. 180–185). ACM Press.

Wolfe, J. M. (1971b). Perspectives on testing for programming aptitude. In *Proceedings of the 1971 ACM Annual Conference* (pp. 268–277). ACM Press. https://doi.org/10.1145/800168.805426