Ten things software developers should learn about learning

Neil C. C. Brown
neil.c.c.brown@kcl.ac.uk
King's College London
London, UK

Felienne Hermans
f.f.j.hermans@vu.nl
Vrije Universiteit
Amsterdam, The Netherlands

Lauren E. Margulieux
lmargulieux@gsu.edu
Georgia State University
Atlanta, Georgia, USA

ABSTRACT

Learning is a core part of being a software developer: new technologies and paradigms are forever being created. It is necessary to constantly learn just to keep up. However, many people's intuitions about human learning and memory are incorrect, such as memorising facts is obsolete with access to the Internet. Further, many of the strategies that we may have relied upon in school, such as cramming for an exam, are ineffective. In this article, we present ten research-derived findings about learning that will enable software developers to learn, teach, and recruit more effectively.

INTRODUCTION

Learning is necessary for software developers. Change is perpetual: new technologies are frequently invented, and old technologies are repeatedly updated. Thus, developers do not learn to program just once – over the course of their career they will learn many new programming languages and frameworks.

Just because we learn does not mean we understand how we learn. One survey in the USA found that the majority of beliefs about memory were contrary to those of scientific consensus [70]: people do not intuitively understand how memory and learning work.

As an example, consider learning styles. Advocates of learning styles claim that effective instruction matches learners' preferred styles – visual learners look, auditory learners listen, and kinesthetic learners do. A 2020 review found that 89% of people believe that learners' preferred styles should dictate instruction, though researchers have known for several decades that this is inaccurate [50]. While learners have preferred styles, effective instruction matches the content, not learning styles. A science class should use graphs to present data rather than verbal descriptions, regardless of visual or auditory learning styles, just like cooking class should use hands-on practice rather than reading, whether learners prefer a kinesthetic style or not [54].

Decades of research into cognitive psychology, education, and programming education provide strong insights into how we learn. In the next ten sections, we will give research-backed findings about learning that apply to software developers and discuss their practical implications. This information can help with learning by yourself, teaching junior staff, and recruiting staff.

1 HUMAN MEMORY IS NOT MADE OF BITS

Human memory is central to learning: as Kirschner and Hendrick [37] put it, "learning means that there has been a change made in one's long-term memory." Software developers are familiar with the incredible power of computer memory, where we can store a series of bits and later retrieve that exact series of bits. While human memory is similar, it is neither as precise nor as reliable.

Due to the biological complexity of human memory, reliability is a complicated matter. With computer memory we use two fundamental operations: read and write. Reading computer memory does not modify it, and it does not matter how much time passes between writes and reads. Human long-term memory is not as sterile: human memory seems to have a "read-and-update" operation, wherein fetching a memory can both strengthen it and modify it – a process known as reconsolidation [3, 12]. This modification is more likely on recently formed memories [72]. Because of this potential for modification, a fact is not in a binary state of either definitively learned or unknown: it can exist in intermediate states. We can forget things we previously knew, and knowledge can be unreliable, especially when recently learned.

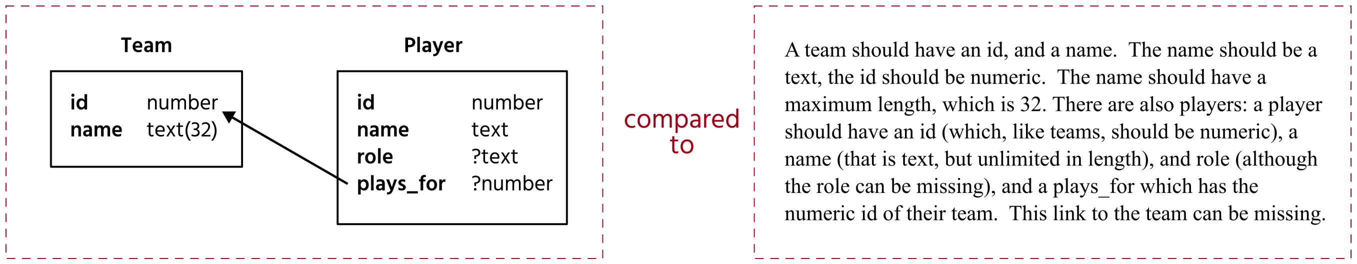
Another curious feature of human memory is "spreading activation" [4]. Our memories are stored in interconnected neural pathways. When we try to remember something, we activate a pathway of neurons to access the targeted information. However, activation is not contained within one pathway. Some of the activation energy spreads to other connected pathways, like heat radiating from a hot water pipe. This spreading activation leaves related pathways primed for activation for hours [6].

Spreading activation has a negative implication for memory [4, 61] and a positive implication for problem-solving [58]. Spreading activation means that related, but imprecise, information can become conflated with the target information, meaning our recall of information can be unreliable. However, spreading activation is also associated with insight-based problem-solving, or "ah-ha moments". Because pathways stay primed for hours, sometimes stepping away from a problem to work on a different problem with its own spreading activation causes two unrelated areas to connect in the middle. When two previously unrelated areas connect, creative and unique solutions to problems can arise [71]. This is why walks, showers or otherwise spending time away from the problem can help you get unstuck in problem solving.

In summary, human memory does not work by simply storing and retrieving from a specific location like computer memory. Human memory is more fragile and more unreliable, but it can also have great benefits in problem-solving and deep understanding by connecting knowledge together. We will elaborate further on this in later sections, especially on retrieving items from memory (section 2) and strengthening memories (section 5).

2 HUMAN MEMORY IS COMPOSED OF ONE LIMITED AND ONE UNLIMITED SYSTEM

Human memory has two main components that are relevant to learning: *long-term memory* and *working memory*. Long-term memory is where information is permanently stored and is functionally limitless [6]; in that sense it functions somewhat like a computer's disk storage. Working memory, in contrast, is used to consciously



A team should have an id, and a name. The name should be a text, the id should be numeric. The name should have a maximum length, which is 32. There are also players: a player should have an id (which, like teams, should be numeric), a name (that is text, but unlimited in length), and role (although the role can be missing), and a plays_for which has the numeric id of their team. This link to the team can be missing.

Figure 1: Two ways of presenting the same database schema description with differing extraneous cognitive load. The left-hand dashed red box contains exactly the same information as the awkward textual description in the right-hand dashed red box. But if a developer only received one of the two to create an SQL database, they are likely to find the diagram easier than the text. We say that the text here has a higher extraneous cognitive load.

reason about information to solve problems [7]; it functions like a CPU's registers, storing a limited amount of information in real time to allow access and manipulation.

Working memory is limited, and its capacity is roughly fixed at birth [7]. While higher working memory capacity is related to higher general intelligence, working memory capacity is not the be-all and end-all for performance [40]. Higher capacity enables faster learning, but our unlimited long-term memory removes limitations on how much we could ultimately learn in total [6]. Expert programmers may have low or high *working* memory capacity but it is the contents of their *long-term* memory that make them experts.

As people learn more about a topic, they relate information together into *chunks*[1]. Chunking allows the multiple pieces of information to act as one piece of information in working memory [41]. For example, when learning an email address, a familiar domain, like gmail.com, is treated as one piece of information instead of a random string of characters, like xvjki.wmt. Thus, the more information that is chunked, the larger working memory is functionally [74]. Using our computer analogy, our working memory/CPU registers may only let us store five pointers to chunks in long-term memory/disk, but there is no limit on the size of the chunks, so the optimal strategy is to increase the size of the chunks.

When learning new tools or skills, it is important to understand the *cognitive load*, or amount of working memory capacity, demanded by the task. Cognitive load has two parts [73]: intrinsic load and extraneous load. Intrinsic load is how many pieces of information or chunks are inherently necessary to achieve the task, and it cannot be changed except by changing the task. In contrast, extraneous cognitive load is unnecessary information that, nevertheless, is part of performing the task. Presentation format is an example of how extraneous cognitive load can vary. If you are implementing a database schema, it is easier to use a diagram with tables and attributes than a plain English description – the latter has higher extraneous load because you must mentally transform the description into a schema, whereas the diagram can be mapped directly (see Figure 1 for an example). Extraneous load is generally higher for beginners because they cannot distinguish between intrinsic and extraneous information easily.

When faced with a task that seems beyond a person's abilities, it is important to recognize that this can be changed by reorganising the task. Decomposing the problem into smaller pieces that can be processed and chunked will ultimately allow the person to solve complex problems. This principle should be applied to your own practice when facing problems at the edge of or beyond your current skills, but it is especially relevant when working with junior developers and recruits.

3 EXPERTS RECOGNISE, BEGINNERS REASON

One key difference between beginners and experts is that experts have seen it all before. Research into chess experts has shown that the primary advantage of experts is that they *remember and recognise* the state of the board. This allows them to decide how to respond more quickly and with less effort [29]. Kahneman [35][2] describes cognition as being split into "system 1" and "system 2" (thus proving that it's not only developers who struggle with naming things). System 1 is fast and driven by recognition, relying upon pattern recognition in long-term memory, while system 2 is slower and focused on reasoning, requiring more processing in working memory. This is part of a general idea known as dual-process theories Robins [60].

Expert developers can reason at a higher-level by having memorised (usually implicitly, from experience) common patterns in program code, which frees up their cognition [11]. One such instance of this is "design patterns" in programming, similar to chunks from section 2. An expert may immediately recognise that a particular piece of code is carrying out a sorting algorithm, while a beginner might read line-by-line to try to understand the workings of the code without recognising the bigger picture.

A corollary to this is that beginners can become experts by reading and understanding a lot of code. Experts build up a mental library of patterns that let them read and write code more easily in future. Seeing purely-imperative C code may only partially apply to functional Haskell code, so seeing a variety of programming paradigms will help further. Overall, this pattern matching is the reason that reading and working with more code, and more types of code, will increase proficiency at programming.

[1]This is not an informal description: the technical term is actually "chunks".

[2]Parts of Kahneman's book were undermined by psychology's "replication crisis", which affected some of its findings, but not the idea of system 1 and 2.

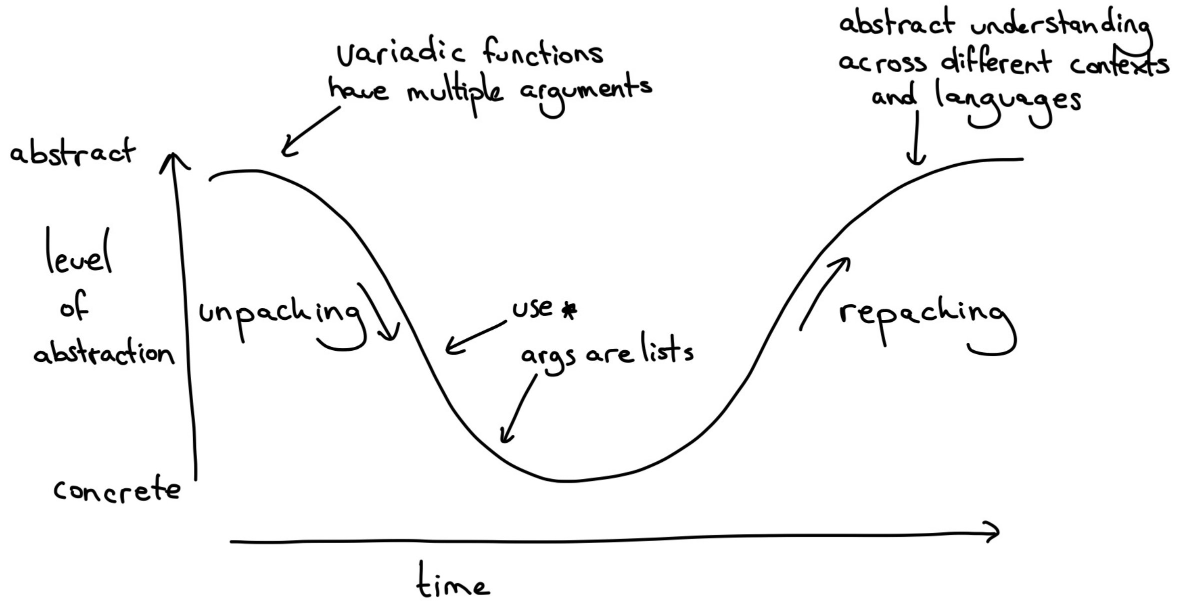

Variadic functions have multiple arguments
abstract understanding across different contexts and languages
abstract
level of abstraction
concrete
unpacking
use * args are lists
repacking
time

Figure 2: The semantic wave for variadic functions

4 UNDERSTANDING A CONCEPT GOES FROM ABSTRACT TO CONCRETE AND BACK

Research shows that experts deal with concepts in different ways than beginners. Experts use generic and abstract terms that do not focus on details and look for underlying concepts, whereas beginners focus on surface details and have trouble connecting these details to the bigger picture [20]. These differences affect how experts reason (as discussed in section 3) but also how they learn.

For example, when explaining a variadic function in Python to someone new to the concept, experts might say that it is a function that can take a varying number of arguments. A beginner may focus on details such as the exact syntax for declaring and calling the function, and may think that passing one argument is a special case. An expert may more easily understand or predict the details from having the concept explained to them.

When you are learning a new concept, you will benefit from both forms of explanation: abstract features and concrete details with examples. More specifically, you will benefit from following the *semantic wave*, a concept defined by Australian scientist Karl Maton [45, 77], as illustrated by Figure 2.

Following the semantic wave, you continuously switch between the abstract definition, and several diverse examples of the concept. The more diverse the examples are, the better. Even wrong examples are beneficial when compared to correct examples to understand why they are wrong [42], such as seeing a mutable variable labelled as non-constant when trying to learn what a constant is. This process is called unpacking.

With these diverse examples, you can then (re)visit the abstract definition and construct a deeper understanding of the concept. Deeper understanding stems from recognizing how multiple details from the examples connect to the one abstract concept in the definition, a process which is called *repacking*.

Programming frequently involves learning about abstract concepts. Faced with an abstract concept to learn, such as functions, people often reach for concrete instantiations of the concept to examine [31]. For example, the `abs` function that returns the absolute value of a number. One challenge is that as concepts get more abstract (from values to variables/objects to functions/classes to higher-order functions/metaclasses and eventually category theory), the distance to a concrete example increases. The saving grace is that as we learn abstract concepts, they become more concrete to us [66, 67]. Initially a function is an abstract concept, but after much practice, a function becomes a concrete item (or chunk) to us and we can learn the next level of abstraction.

5 SPACING AND REPETITION MATTER

How often have you heard that you should not cram for an exam? Unless, of course, you want to forget everything by the next day. This advice is based on one of the most predictable and persistent effects in cognitive psychology: the spacing effect [21]. According to the spacing effect, humans learn problem-solving concepts best by spacing out their practice across multiple sessions, multiple days, and ideally, multiple weeks [17, 33].

The reason spacing works is due to the relationship between long-term and working memory described in section 2. When learners practice solving problems, they practice two skills. First, matching the information in the problem to a concept that can solve it (such as a filtering loop), and second, applying the concept to solve the problem (such as writing the loop). The first skill requires activating the correct neural pathway to the concept in long-term memory [13]. If learners repeatedly solve the same kind of problem, such as for-each loop problems, then that pathway to long-term memory stays active, and they miss practicing the first skill. A common result of un-spaced practice is that people can solve problems, but only when they are told which concept to use [13]. While interleaving different types of problems, like loop and conditional problems, can help, pathways take time to return to baseline, making spacing necessary to get the most out of practice time [21]. In addition, the brain needs rest to consolidate the new information that has been processed so that it can be applied to new problems [22].

Going against this time-tested principle, intensive coding bootcamps require learners to cram their problem-solving practice into un-spaced sessions. While this is not ideal, researchers of the spacing effect have known from the beginning that most learners still prefer to cram their practice into as little time as possible [21]. For people whose only viable option to learn programming is intensive bootcamps, we can apply the spacing research to maximize their outcomes.

To structure a day of learning, learners should limit learning bouts to 90 minutes or less [26, 38]. The neurochemical balance in the brain makes concentration difficult after this point [38]. After each learning bout, take at least 20 minutes to rest[38]. Really rest by going for a walk or sitting quietly – without working on other tasks, idly browsing the internet, or chatting with others. Rest speeds up the consolidation process, which also happens during sleep [39, 69].

Within a learning bout, there are a couple of strategies to maximize efficiency. First, randomize the order of the type of problem being solved so that different concepts are being activated in long-term memory [13]. Be forewarned, though, that randomizing the order improves learning outcomes but requires more effort [14]. The second strategy is to take short breaks at random intervals to enhance memory consolidation. A 10-second break every 2-5 minutes is recommended [34].

6 THE INTERNET HAS NOT MADE LEARNING OBSOLETE

The availability of programming knowledge changed with the advent of the Internet. Knowledge about syntax or APIs went from being buried in reference books, to being a few keystrokes away. Most recently, AI-powered tools like ChatGPT, Codex, and GitHub Copilot will even fill in these details (mostly accurately) for you. This raises an obvious question: why is it worth learning details – or anything at all – if the knowledge is available from the Internet within seconds?

We learn by storing pieces of knowledge in our long-term memory and forming connections between them [6]. If the knowledge is not present in the brain, because you have not yet learned it well, the brain cannot form any connections between it, so higher-levels of understanding and abstraction are not possible [5]. If every time you need a piece of code to do a database join you search online for it, insert it, and move on, you will be unlikely to learn much about joins. The wisdom of relying on the Internet or AI differs between beginners and experts: there is a key distinction between a beginner who has never learned the details and thus lacks the memory connections, and an expert who has learned the deeper structure but searches for the forgotten fine details [5].

There is even some evidence to suggest that searching the Internet is less efficient for remembering information. One study found that information was remembered less well if it was found via the Internet (compared to a physical book) [23]. Another found that immediately searching the Internet led to worse recall of the same information later, compared to first trying to think of the answer before resorting to searching [28]. It seems that searching may rob the brain of the benefits of the memory-strengthening effect of recalling information (discussed in section 1).

There is also the issue of cognitive load from section 2. An internet search requires a form of context switching for the brain; its limited attention and working memory must be switched from the task at hand (programming) to a new cognitive task (searching the Internet and selecting a result or evaluating an AI-generated result). If the required knowledge is instead memorised then not only is access much faster (like using a cache versus fetching from a hard disk) but it also avoids the cognitive drain of context switching and filtering out extraneous information from the search. So there are multiple reasons to memorise information, despite it being available on the Internet.

7 PROBLEM-SOLVING IS NOT A GENERIC SKILL

Problem-solving is a large part of programming. One common (but incorrect) idea in software development is to directly teach problem-solving as a specific skill, which can then be applied to different aspects of development (design, debugging, etc.). Thus, problem-solving is (incorrectly) conceived as a generic skill. However, this is not how problem-solving works in the brain.

While humans do have some generic problem-solving skills, they are much less efficient than domain-specific problem solving skills, such as being able to debug programs. While we can learn to reason, we don't learn how to solve problems in general. Instead, we learn how to solve programming problems, or how to plan the best chess move, or how to create a knitting pattern. Each of these skills is separate and does not influence the others [9]. Research into chess found little or no effect of learning it on other academic and cognitive skills [63], and the same is true for music instruction and cognitive training [64]. This inability to transfer problem-solving skills is why "brain training" is ineffective for developing general intelligence [51].

The one exception to this rule appears to be spatial skills. Spatial skills allow us to visualize objects in our mind, like a Tetris shape, and mentally manipulate those objects, like rotating a Tetris shape. Training these generic skills can improve learning in other disciplines. This phenomenon is so unusual that it has caused much consternation in cognitive and learning sciences [43]. Yet, spatial training improves performance on a range of non-verbal skills regardless of initial ability, age, or type of training task [76]. Recent work has even demonstrated that spatial training can improve efficiency for professional software developers, likely because they are still learning new concepts [53]. Even with this strange exception, the best way to learn how to solve programming problems is still to practice solving programming problems rather than looking for performance benefits from learning chess or other cognitive training.

There is a secondary implication here for recruitment. One popular idea for screening programming candidates was to give brainteaser puzzles, such as how to weigh a jumbo jet [56]. As Google worked out by 2013, this is a waste of time [15] – there is no reliable correspondence between problem-solving in the world of brain teasers and problem-solving in the world of programming. If you want to judge programming ability, assess programming ability.

8 EXPERTISE CAN BE PROBLEMATIC IN SOME SITUATIONS

We have discussed many ways in which expertise benefits learning and performance. However, being an expert can also lead to problems, as we will detail in this section.

Programmers use tools and aids to be more effective, such as version control systems or IDEs. Such tools can have different effects on beginners and experts. Beginners may get overwhelmed by the amount of options available in professional tools (due to the increased cognitive load, explained in section 2), and may benefit from beginner-friendly hints on how to use the tool. However, experts find the same hints more distracting than useful because they already know what to do. This is known as the expertise-reversal effect [36]: hints and guides that help beginners can get in the way of experts and make them less productive.

Programmers usually learn multiple programming languages throughout their career. Knowing multiple languages can be beneficial once they have been mastered, but sometimes transferring knowledge from one programming language to another can lead to faulty knowledge [68]. For example, a programmer may learn about inheritance in Java, where one method overrides a parent method as long as the signatures match, and transfer this knowledge to C++, where overriding a parent method additionally requires that the parent method is declared virtual. These kinds of differences – where features are similar in syntax but different in semantics between languages – specifically hinder transfer of knowledge [75].

Experts often help to train beginners, but experts without experience in training others often do not realise that beginners think differently. Thus, they fail to tailor their explanations for someone with a different mental model. This is known as the expert blind-spot problem: a difficulty in seeing things through the eyes of a beginner once you have become an expert [49]. It can be overcome by listening carefully to beginners explain their current understanding and tailoring explanations accordingly.

Sometimes, however, knowledge becomes so automated that it is difficult for experts to verbalize it [5]. This automated knowledge is why experts have intuitions about how to solve problems or explain their process as, "I just know." In these cases of tacit knowledge, beginners might better learn from instructional materials designed to support beginners, often called scaffolded instruction [44], or from a peer rather than an expert. A more knowledgeable (but still relatively novice) peer is a highly valuable resource to bridge the gap between beginners and experts [1]. They can help the beginner develop new knowledge and the expert to re-discover automated knowledge.

9 THE PREDICTORS OF PROGRAMMING ABILITY ARE UNCLEAR

The success of learning programming, like most activities, is built on a mix of inherent aptitude and practice. Some people believe it is purely about aptitude – the "you're born with it" view – and some believe it is almost entirely about practice – the "10,000 hours" idea that only sufficient practice is required for expertise [27]. Both extreme views are wrong, and in this section, we will explore the evidence for the differing effects of aptitude and practice.

There has been much research to try to predict programming aptitude but few reliable results. Attempts to produce a predictive test for programming ability have generally come to naught [18]. Research has found that all of the following fail to predict programming ability: gender, age, academic major, race, prior performance in math, prior experience with another programming language, perceptions of CS, and preference for humanities or sciences [62]. There was an industry of aptitude tests for programming that began in the 1960s but as Robins [59] summarises, the predictive accuracy was poor and the tests fell out of use.

There is mixed evidence for the importance of years of experience, which relates to practice. There is a correlation between the reputation of programmers on Stack Overflow and their age: older people have higher reputation [48]. However, a recent study found only a weak link between years of experience and success on a programming task among programmers who were relatively early in their careers [55], suggesting that aptitude may have a stronger effect than experience, at least early in programmers' careers.

As in most domains, two factors that weakly predict success in early programming are general intelligence [46] (see section 3) and working memory capacity [11] (see section 2). These factors roughly represent reasoning skills and how much information a learner can process at once. As such, they predict the *rate of learning* rather than absolute ability [57]. A sub-measure of these two factors, spatial reasoning, is a stronger predictor of success in programming, though still quite moderate [8, 52, 53]. Spatial reasoning also predicts success in other science and math fields [43], so this is not programming-specific. Further, these weak to moderate correlations largely disappear with increased experience for reasons discussed in section 7 and section 2. Thus, intelligent people will not always make good programmers, and good programmers need not be high in general intelligence.

In short, it is very hard to predict who will be able to program, especially in the long term. Programmers could come from any background or demographic, and links to any other factors (such as intelligence) are generally fleeting in the face of experience. Therefore, in recruiting new programmers there are no shortcuts to identifying programming ability, nor are there any reliable "candidate profiles" to screen candidates for programming ability.

10 YOUR MINDSET MATTERS

There is a long-standing idea of a binary split in programming ability: you either can program or you cannot. There have been many competing theories behind this [2]. One of the more compelling theories is the idea of learning edge momentum [59], that each topic is dependent on previous topics so once you fall behind you will struggle to catch up. A less compelling theory is the idea of a "geek gene" (you're born with it, or not), which has little empirical evidence [47]. Most recently, we have come to understand differences in programming ability as differences in prior experience [30]. Learners who might seem similar (for example, in the same class, with the same degree, completing the same bootcamp) can have vastly different knowledge and skills, putting them ahead or behind in terms of learning edge momentum or, within a snapshot of time, making them seem "born with it" or not. A similar effect is found in any STEM field that is optionally taught before university (for example, CS, physics, and engineering) [19].

The binary split view, and its effects on teaching and learning, have been studied across academic disciplines in research about fixed versus growth mindsets [24]. A fixed mindset aligns with an aptitude view – that people's abilities are innate and unchanging. Applied to learning, this mindset says that if someone struggles with a new task, then they are not cut out for it. Alternatively, a growth mindset aligns with a practice view – that people's abilities are malleable. Applied to learning, this mindset says that if someone struggles with a new task, they can master it with enough practice.

As described in 9, neither extreme view is true. For example, practically everyone can learn some physics, even if they are not initially good at it. However, practically no one can earn the Nobel Prize in Physics, no matter how much they practice. In between these extremes, we are often trying to figure out the boundaries of our abilities. When teachers and learners approach new tasks with a growth mindset, they tend to persist through difficulties and overcome failure more consistently [24].

While the evidence for this effect is strong and intuitive, research suggests it can be difficult to change someone's mindset to be more growth-oriented [16]. In particular, there are two common misconceptions about how to promote growth mindset that prove ineffective. The first misconception is to reward effort rather than performance because growth mindset favors practice over aptitude. But learners are not stupid; they can tell when they are not progressing, and teachers praising unproductive effort is not helpful. Instead, effort should be rewarded only when the learner is using effective strategies and on the path to success [25]. The second misconception is that when someone approaches a task with a growth mindset, they will maintain that mindset throughout the task. In reality, as we face setbacks and experience failure, people skew towards a fixed mindset because we are not sure where the boundaries of our abilities lie. Thus, we must practice overcoming setbacks and failures to maintain a growth mindset approach [25].

A related concept to fixed and growth mindsets is goal orientation. This is split into two categories: approach and avoidance [65]. The "approach" goal orientation involves wanting to do well, and this engenders positive and effective learning behaviours: working hard, seeking help, and trying new challenging topics. In contrast, the "avoidance" goal orientation involves avoiding failure. This leads to negative and ineffective behaviours: disorganised study, not seeking help, anxiety over performance, and avoiding challenge. It is important that learners can make mistakes without severe penalties if they are to be directed towards "approach" rather than "avoidance".

When learning a new skill or training someone in a new skill, remember that approaching tasks with a growth mindset is effective but also a skill to be developed. Unfortunately, we cannot simply tell people to have a growth mindset and reap the benefits. Instead, nurture this skill by seeking or providing honest feedback about the process of learning and the efficacy of strategies. For mentors, praise areas where a mentee is making progress. For learners, reflect on how skills have improved in the past weeks or months when you are doubtful about your progress. Further, expect that a growth mindset will shift towards a fixed mindset in the face of failure, but it can also be re-developed and become stronger with practice. Feeling discouraged is normal, but it does not mean that you will always feel discouraged. If you feel like quitting, take a break, take a walk, consider your strategies, and then try again.

CONCLUSION

Software developers must continually learn in order to keep up with the fast-paced changes in the field. Learning anything, programming included, involves committing items to memory. Human memory is fascinatingly complex. While it shares some similarities to computer architecture, there are key differences that make it work quite differently. In this article we have explained the current scientific understanding of how human memory works, how learning works, the differences between beginners and experts, and related it all to practical steps that software developers can take to improve their learning, training, and recruitment.

Recommendations

We have split up our recommendations into those for recruiting and those for training and learning.
For recruiting, we make the following recommendations:

- There are no good proxies for programming ability. Stereotypes based on gender, race, or other factors are not supported by evidence. If you want to know how well candidates program, look at their previous work or test them on authentic programming tasks. (See section 9, section 7.)
  - To emphasise a specific point: do not test candidates with brain-teaser puzzles. (See section 7.)
- At least among young developers, years of experience may not be a very reliable measure of ability. (See section 9.)
- A related recommendation from Behroozi et al. [10] is to get candidates to solve interview problems in a room on their own before presenting the solution, as the added pressure from an interviewer observing or requiring talking while solving it adds to cognitive load and stress in a way that impairs performance.

For learning and training, we make the following recommendations:

- Reading a lot of code will help to become a more efficient programmer. (See section 2, section 3.)
- Experts are not always the best at training beginners. (See section 8.)

- Learning takes time, including time *between* learning sessions. Intense cramming is not effective, but spaced repetition is. (See section 5.)
- Similarly, spending time away from a problem can help to solve it. (See section 1.)
- Just because you can find it through an Internet search doesn't mean learning has become obsolete. (See section 6.)
- Use examples to go between abstract concepts and concrete learnable facts. (See section 4).
- Seeking to succeed (rather than avoid failure) and believing that ability is changeable are important factors in resilience and learning. (See section 10.)

Further Reading

Many books on learning are centred around formal education: they are aimed at school teachers and university lecturers. However, the principles are applicable everywhere, including professional development. We recommend three books:

- "Why don't students like school?" by Willingham [78] provides a short and readable explanation of many of the principles of memory and how the brain works.
- "The programmer's brain" by Hermans [32] [3] relates these concepts to programming and describes how techniques for learning and revision that are used at school can still apply to professional development.
- "How learning happens: Seminal works in educational psychology and what they mean in practice" by Kirschner and Hendrick [37] provides a tour through influential papers, explaining them in plain language and the implications and linkages between them.

The papers cited can also serve as further reading. If you are a software developer you may not have access to all of them. ACM members with the digital library option will have access to the ACM papers, although many of our references are from other disciplines. For more recent papers, many authors supply free PDFs on their website; you may wish to try web-searching for the exact title to find such PDFs. Many authors are also happy to supply you with a copy if you contact them directly.

Acknowledgments

We are grateful to Greg Wilson, who was instrumental in initiating and proofreading this paper, and to our other proofreaders: Philip Brown, Kristina Dietz, Jack Parkinson, Anthony Robins, and Justin Thurman. This work is funded in part by the National Science Foundation under grant #1941642.

REFERENCES

[1] Y. Abtahi. The 'more knowledgeable other': A necessity in the zone of proximal development?. *For the Learning of Mathematics*, 37(1):35–39, 2017.

[2] A. Ahadi and R. Lister. Geek genes, prior knowledge, stumbling points and learning edge momentum: Parts of the one elephant? In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, page 123–128, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322430. doi: 10.1145/2493394.2493416. URL https://doi.org/10.1145/2493394.2493416.

[3] Full disclosure: this is written by one of the authors although the other authors recommend it as well.

[3] C. M. Alberini and J. E. LeDoux. Memory reconsolidation. *Current Biology*, 23(17): R746–R750, 2013. ISSN 0960-9822. doi: https://doi.org/10.1016/j.cub.2013.06.046. URL https://www.sciencedirect.com/science/article/pii/S0960982213007719.

[4] J. R. Anderson. A spreading activation theory of memory. *Journal of Verbal Learning and Verbal Behavior*, 22(3):261–295, 1983. ISSN 0022-5371. doi: https://doi.org/10.1016/S0022-5371(83)90201-3. URL https://www.sciencedirect.com/science/article/pii/S0022537183902013.

[5] J. R. Anderson. Act: A simple theory of complex cognition. *American psychologist*, 51(4):355, 1996.

[6] J. R. Anderson. *Cognitive psychology and its implications.* Macmillan, 2005.

[7] A. Baddeley. Working memory. *Science*, 255(5044):556–559, 1992.

[8] A. D. Baddeley and G. J. Hitch. Developments in the concept of working memory. *Neuropsychology*, 8(4):485, 1994.

[9] S. M. Barnett and S. J. Ceci. When and where do we apply what we learn?: A taxonomy for far transfer. *Psychological bulletin*, 128(4):612, 2002.

[10] M. Behroozi, S. Shirolkar, T. Barik, and C. Parnin. Does stress impact technical interview performance? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 481–492, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409712. URL https://doi.org/10.1145/3368089.3409712.

[11] G. R. Bergersen and J.-E. Gustafsson. Programming skill, knowledge, and working memory among professional software developers from an investment theory perspective. *Journal of individual Differences*, 32(4):201, 2011.

[12] R. A. Bjork. Retrieval as a memory modifier: An interpretation of negative recency and related phenomena. In R. L. Solso, editor, *Information Processing and Cognition: The Loyola Symposium*, pages 123–144. Lawrence Erlbaum, 1975.

[13] R. A. Bjork and T. W. Allen. The spacing effect: Consolidation or differential encoding? *Journal of Verbal Learning and Verbal Behavior*, 9(5):567–572, 1970.

[14] R. A. Bjork and E. L. Bjork. Desirable difficulties in theory and practice. *Journal of Applied Research in Memory and Cognition*, 9(4):475, 2020.

[15] A. Bryant. In head-hunting, big data may not be such a big deal. *NY Times*, June 2013. URL https://www.nytimes.com/2013/06/20/business/in-head-hunting-big-data-may-not-be-such-a-big-deal.html.

[16] A. P. Burgoyne, D. Z. Hambrick, and B. N. Macnamara. How firm are the foundations of mind-set theory? The claims appear stronger than the evidence. *Psychological Science*, 31(3):258–267, 2020. doi: 10.1177/0956797619897588. PMID: 32011215.

[17] S. K. Carpenter, N. J. Cepeda, D. Rohrer, S. H. K. Kang, and H. Pashler. Using spacing to enhance diverse forms of learning: Review of recent research and implications for instruction. *Educational Psychology Review*, 24(3):369–378, Sep 2012. ISSN 1573-336X. doi: 10.1007/s10648-012-9205-z. URL https://doi.org/10.1007/s10648-012-9205-z.

[18] M. E. Caspersen, K. D. Larsen, and J. Bennedsen. Mental models and programming aptitude. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '07, page 206–210, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936103. doi: 10.1145/1268784.1268845. URL https://doi.org/10.1145/1268784.1268845.

[19] S. Cheryan, S. A. Ziegler, A. K. Montoya, and L. Jiang. Why are some STEM fields more gender balanced than others? *Psychological bulletin*, 143(1):1, 2017.

[20] M. T. Chi, P. J. Feltovich, and R. Glaser. Categorization and representation of physics problems by experts and novices. *Cognitive Science*, 5(2):121–152, 1981. ISSN 0364-0213. URL https://www.sciencedirect.com/science/article/pii/S0364021381800298.

[21] F. N. Dempster. The spacing effect: A case study in the failure to apply the results of psychological research. *American Psychologist*, 43(8):627, 1988.

[22] F. N. Dempster and R. Farris. The spacing effect: research and practice. *Journal of Research & Development in Education*, 23:97–101, 1990. ISSN 0022-426X(Print).

[23] G. Dong and M. N. Potenza. Behavioural and brain responses related to Internet search and memory. *European Journal of Neuroscience*, 42(8):2546–2554, 2015. doi: https://doi.org/10.1111/ejn.13039. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/ejn.13039.

[24] C. S. Dweck. *Mindset: The new psychology of success.* Random House, 2006.

[25] C. S. Dweck and D. S. Yeager. Mindsets: A view from two eras. *Perspectives on Psychological science*, 14(3):481–496, 2019.

[26] K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer. The role of deliberate practice in the acquisition of expert performance. *Psychological review*, 100(3):363, 1993.

[27] K. A. Ericsson et al. The influence of experience and deliberate practice on the development of superior expert performance. *The Cambridge handbook of expertise and expert performance*, 38(685-705):2–2, 2006.

[28] S. Giebl, S. Mena, R. Sandberg, E. L. Bjork, and R. A. Bjork. Thinking first versus googling first: Preferences and consequences. *Journal of Applied Research in Memory and Cognition*, 2022. doi: 10.1037/mac0000072. URL https://doi.org/10.1037/mac0000072.

[29] F. Gobet and H. A. Simon. The roles of recognition processes and look-ahead search in time-constrained expert problem solving: Evidence from grand-master-level chess. *Psychological Science*, 7(1):52–55, 1996. doi: 10.1111/j.1467-9280.1996.tb00666.x. URL https://doi.org/10.1111/j.1467-9280.1996.tb00666.x.

[30] P. Grabarczyk, S. M. Nicolajsen, and C. Brabrand. On the effect of onboarding computing students without programming-confidence or-experience. In *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research*, pages 1–8, 2022.

[31] O. Hazzan. Reducing abstraction level when learning abstract algebra concepts. *Educational Studies in Mathematics*, 40(1):71–90, Sep 1999. ISSN 1573-0816. doi: 10.1023/A:1003780613628. URL https://doi.org/10.1023/A:1003780613628.

[32] F. Hermans. *The Programmer's Brain: What every programmer needs to know about cognition.* Manning, 2021. ISBN 978-1617298677.

[33] D. L. Hintzman. *Theoretical implications of the spacing effect.*, pages 77–100. Theories in cognitive psychology: The Loyola Symposium. Lawrence Erlbaum, Oxford, England, 1974. ISBN 0470812281.

[34] A. Huberman. Teach & learn better with a "neuroplasticity super protocol". *Neural Network*, October 2021.

[35] D. Kahneman. *Thinking, fast and slow.* Macmillan, 2011.

[36] S. Kalyuga. Expertise reversal effect and its implications for learner-tailored instruction. *Educational psychology review*, 19(4):509–539, 2007.

[37] P. A. Kirschner and C. Hendrick. *How learning happens: Seminal works in educational psychology and what they mean in practice.* Routledge, 2020.

[38] N. Kleitman. Basic rest-activity cycle—22 years later. *Sleep*, 5(4):311–317, 1982.

[39] J. G. Klinzing, N. Niethard, and J. Born. Mechanisms of systems memory consolidation during sleep. *Nature Neuroscience*, 22(10):1598–1610, Oct 2019. ISSN 1546-1726. doi: 10.1038/s41593-019-0467-3. URL https://doi.org/10.1038/s41593-019-0467-3.

[40] P. C. Kyllonen and R. E. Christal. Reasoning ability is (little more than) working-memory capacity?! *Intelligence*, 14(4):389–433, 1990.

[41] J. E. Laird, P. S. Rosenbloom, and A. Newell. Towards chunking as a general learning mechanism. In *AAAI*, pages 188–192, 1984.

[42] L. Margulieux, P. Denny, K. Cunningham, M. Deutsch, and B. R. Shapiro. When wrong is right: The instructional power of multiple conceptions. In *Proceedings of the 17th ACM Conference on International Computing Education Research*, pages 184–197, 2021.

[43] L. E. Margulieux. Spatial encoding strategy theory: The relationship between spatial skill and STEM achievement. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, page 81–90, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450361859. doi: 10.1145/3291279.3339414. URL https://doi.org/10.1145/3291279.3339414.

[44] L. E. Margulieux, R. Catrambone, and M. Guzdial. Employing subgoals in computer programming education. *Computer Science Education*, 26(1):44–67, 1 2016. doi: 10.1080/08993408.2016.1144429. URL https://doi.org/10.1080/08993408.2016.1144429.

[45] K. Maton. Making semantic waves: A key to cumulative knowledge-building. *Linguistics and Education*, 24(1):8–22, 2013. ISSN 0898-5898. doi: https://doi.org/10.1016/j.linged.2012.11.005. URL https://www.sciencedirect.com/science/article/pii/S0898589812000678. Cumulative knowledge-building in secondary schooling.

[46] R. E. Mayer, J. L. Dyck, and W. Vilberg. Learning to program and learning to think: What's the connection? *Commun. ACM*, 29(7):605–610, jul 1986. ISSN 0001-0782. doi: 10.1145/6138.6142. URL https://doi.org/10.1145/6138.6142.

[47] R. McCartney, J. Boustedt, A. Eckerdal, K. Sanders, and C. Zander. Folk pedagogy and the geek gene: geekiness quotient. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 405–410, 2017.

[48] P. Morrison and E. Murphy-Hill. Is programming knowledge related to age? an exploration of stack overflow. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 69–72, 2013. doi: 10.1109/MSR.2013.6624008.

[49] M. J. Nathan, K. R. Koedinger, M. W. Alibali, et al. Expert blind spot: When content knowledge eclipses pedagogical content knowledge. In *Proceedings of the third international conference on cognitive science*, volume 644648, 2001.

[50] P. M. Newton and A. Salvi. How common is belief in the learning styles neuromyth, and does it matter? a pragmatic systematic review. *Frontiers in Education*, 5, 2020. ISSN 2504-284X. doi: 10.3389/feduc.2020.602451. URL https://www.frontiersin.org/articles/10.3389/feduc.2020.602451.

[51] A. M. Owen, A. Hampshire, J. A. Grahn, R. Stenton, S. Dajani, A. S. Burns, R. J. Howard, and C. G. Ballard. Putting brain training to the test. *Nature*, 465(7299): 775–778, 2010.

[52] M. C. Parker, A. Solomon, B. Pritchett, D. A. Illingworth, L. E. Marguilieux, and M. Guzdial. Socioeconomic status and computer science achievement: Spatial ability as a mediating variable in a novel model of understanding. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, page 97–105, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356282. doi: 10.1145/3230977.3230987. URL https://doi.org/10.1145/3230977.3230987.

[53] J. Parkinson and Q. Cutts. Relationships between an early-stage spatial skills test and final CS degree outcomes. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2022, page 293–299, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450390705. doi: 10.1145/3478431.3499332. URL https://doi.org/10.1145/3478431.3499332.

[54] H. Pashler, M. McDaniel, D. Rohrer, and R. Bjork. Learning styles: Concepts and evidence. *Psychological science in the public interest*, 9(3):105–119, 2008.

[55] N. Peitek, A. Bergum, M. Rekrut, J. Mucke, M. Nadig, C. Parnin, J. Siegmund, and S. Apel. Correlates of programmer efficacy and their link to experience: A combined EEG and eye-tracking study, 2022. in press.

[56] W. Poundstone. *How Would You Move Mount Fuji?: Microsoft's Cult of the Puzzle-How the World's Smartest Companies Select the Most Creative Thinkers.* Hachette UK, 2003.

[57] R. Primi, M. E. Ferrão, and L. S. Almeida. Fluid intelligence as a predictor of learning: A longitudinal multilevel approach applied to math. *Learning and Individual Differences*, 20(5):446–451, 2010.

[58] E. Raufaste, H. Eyrolle, and C. Mariné. Pertinence generation in radiological diagnosis: Spreading activation and the nature of expertise. *Cognitive Science*, 22(4):517–546, 1998. doi: https://doi.org/10.1207/s15516709cog2204_4. URL https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog2204_4.

[59] A. Robins. Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education*, 20(1):37–71, 2010.

[60] A. V. Robins. Dual process theories: Computing cognition in context. *ACM Trans. Comput. Educ.*, 22(4), sep 2022. doi: 10.1145/3487055. URL https://doi.org/10.1145/3487055.

[61] H. L. Roediger III, D. A. Balota, and J. M. Watson. *Spreading activation and arousal of false memories.*, pages 95–115. Science conference series. American Psychological Association, Washington, DC, US, 2001. ISBN 1-55798-750-5 (Hardcover). doi: 10.1037/10394-006. URL https://doi.org/10.1037/10394-006.

[62] N. Rountree, J. Rountree, A. Robins, and R. Hannah. Interacting factors that predict success and failure in a CS1 course. *ACM SIGCSE Bulletin*, 36(4):101–104, 2004.

[63] G. Sala and F. Gobet. Do the benefits of chess instruction transfer to academic and cognitive skills? a meta-analysis. *Educational Research Review*, 18:46–57, 2016. ISSN 1747-938X. doi: https://doi.org/10.1016/j.edurev.2016.02.002. URL https://www.sciencedirect.com/science/article/pii/S1747938X16300112.

[64] G. Sala and F. Gobet. Does far transfer exist? Negative evidence from chess, music, and working memory training. *Current Directions in Psychological Science*, 26(6):515–520, 2017. doi: 10.1177/0963721417712760. URL https://doi.org/10.1177/0963721417712760. PMID: 29276344.

[65] C. Senko, C. S. Hulleman, and J. M. Harackiewicz. Achievement goal theory at the crossroads: Old controversies, current challenges, and new directions. *Educational Psychologist*, 46(1):26–47, 2011. doi: 10.1080/00461520.2011.538646. URL https://doi.org/10.1080/00461520.2011.538646.

[66] A. Sfard. On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educational Studies in Mathematics*, 22(1):1–36, Feb 1991. ISSN 1573-0816. doi: 10.1007/BF00302715. URL https://doi.org/10.1007/BF00302715.

[67] A. Sfard. Operational origins of mathematical objects and the quandary of reification: The case of function. *The concept of function: Aspects of epistemology and pedagogy*, 25:59–84, 1992.

[68] N. Shrestha, C. Botta, T. Barik, and C. Parnin. Here we go again: Why is it difficult for developers to learn another programming language? *Communications of the ACM*, 65(3):91–99, 3 2022. doi: 10.1145/3511062. URL https://doi.org/10.1145/3511062.

[69] A. L. Simmons. Distributed practice and procedural memory consolidation in musicians' skill learning. *Journal of Research in Music Education*, 59(4):357–368, 2012. doi: 10.1177/0022429411424798. URL https://doi.org/10.1177/0022429411424798.

[70] D. J. Simons and C. F. Chabris. What people believe about how memory works: A representative survey of the U.S. population. *PLOS ONE*, 6(8):1–7, 08 2011. doi: 10.1371/journal.pone.0022757. URL https://doi.org/10.1371/journal.pone.0022757.

[71] U. N. Sio and E. Rudowicz. The role of an incubation period in creative problem solving. *Creativity Research Journal*, 19(2-3):307–318, 2007. doi: 10.1080/10400410701397453. URL https://doi.org/10.1080/10400410701397453.

[72] A. Suzuki, S. A. Josselyn, P. W. Frankland, S. Masushige, A. J. Silva, and S. Kida. Memory reconsolidation and extinction have distinct temporal and biochemical signatures. *Journal of Neuroscience*, 24(20):4787–4795, 2004. ISSN 0270-6474. doi: 10.1523/JNEUROSCI.5491-03.2004. URL https://www.jneurosci.org/content/24/20/4787.

[73] J. Sweller. Cognitive load theory. In *Psychology of learning and motivation*, volume 55, pages 37–76. Elsevier, 2011.

[74] M. Thalmann, A. S. Souza, and K. Oberauer. How does chunking help working memory? *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 45(1):37, 2019.

[75] E. Tshukudu and Q. Cutts. Understanding conceptual transfer for students learning new programming languages. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, ICER '20, page 227–237, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370929. doi: 10.1145/3372782.3406270. URL https://doi.org/10.1145/3372782.3406270.

[76] D. H. Uttal, N. G. Meadow, E. Tipton, L. L. Hand, A. R. Alden, C. Warren, and N. S. Newcombe. The malleability of spatial skills: a meta-analysis of training studies. *Psychological bulletin*, 139(2):352, 2013.

[77] J. Waite, K. Maton, P. Curzon, and L. Tuttiett. Unplugged computing and semantic waves: Analysing crazy characters. In *Proceedings of the 2019 Conference on United Kingdom & Ireland Computing Education Research*, UKICER '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372572. doi: 10.1145/3351287.3351291. URL https://doi.org/10.1145/3351287.3351291.

[78] D. T. Willingham. *Why don't students like school?: A cognitive scientist answers questions about how the mind works and what it means for the classroom.* John Wiley & Sons, 2021. ISBN 978-0470591963.