DOI:10.1145/3584859

Understanding how human memory and learning works, the differences between beginners and experts, and practical steps developers can take to improve their learning, training, and recruitment.

BY NEIL C.C. BROWN, FELIENNE F.J. HERMANS, AND LAUREN E. MARGULIEUX

10 Things Software Developers Should Learn about Learning

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 careers 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 U.S. found that the majority of beliefs about memory were contrary to those of scientific consensus: People do not intuitively understand how memory and learning work.[37]

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.[28] 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 classes should use hands-on practices rather than reading, whether learners prefer a kinesthetic style or not.

Decades of research into cognitive psychology, education, and programming education provide strong insights into how we learn. The next 10 sections of this article provide research-backed findings about learning that apply to software developers and discuss their practical implications. This information can help with learning for 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 put it, "Learning means that there has been a change made in one's long-term memory."[20] Software developers are familiar with the incredible power of computer memory, where we can

» key insights

- **Learning is vital for programmers, but the human mind works quite differently than a computer.**
- **Understanding how humans learn can help you learn more effectively.**
- **The Internet and LLMs have not made learning obsolete; learning is essential and takes time.**
- **Expertise changes how you think, letting you solve problems more easily but also potentially hindering your ability to teach.**

ILLUSTRATION BY LISA SHEEHAN


LEARNING
DEVELOPERS
SHOULD
LEARN
about
SOFTWARE

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 and modify it—a process known as *reconsolidation*. This modification is more likely on recently formed memories. Because of this potential for modification, a fact does not exist in a binary state of either definitively known 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."[1] 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.[1]

Spreading activation has a negative implication for memory[1] and a positive implication for problem-solving.[3,2] 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 "aha moments." Because pathways stay primed for hours, sometimes stepping away from a problem to work on a different one 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. This is why walks, showers, or otherwise spending time away from a 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 offer 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 and strengthening memories.

2. Human Memory Is Composed of One Limited and One Unlimited System

Human memory comprises 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;[1] in that sense, it functions somewhat like a computer's disk storage. Working memory, however, is used to consciously reason about information to solve problems;[2] it functions like a CPU's registers, storing a limited amount of information in real time to allow access and manipulation.

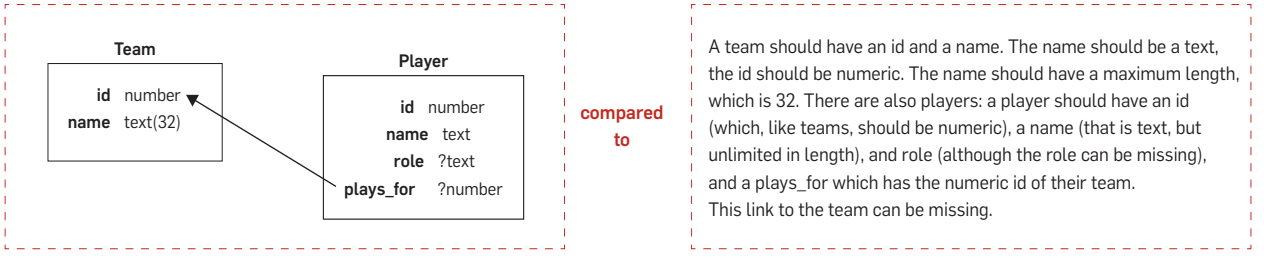
Working memory is limited, and its capacity is roughly fixed at birth.[2] While higher working-memory capacity is related to higher general intelligence, working-memory capacity is not the be-all and end-all for performance.[22] Higher capacity enables faster learning, but our unlimited long-term memory removes limitations on how much we could ultimately learn in total.[1] 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*.[a] Chunking allows the multiple pieces of information to act as one piece of information in working memory. For example, when learning an email address, a familiar domain, such as 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.[38] Using

a This is not an informal description: the technical term is actually "chunks."

Figure 1. Two ways of presenting the same database schema description with differing extraneous cognitive load.

The dashed box on the left contains exactly the same information as the awkward textual description in the dashed box on the right. 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.



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 by practicing using information and solving problems.

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: *intrinsic load* and *extraneous load*. Intrinsic load is how many pieces of information or chunks are inherently necessary to achieve the task; 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). 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 reorganizing 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 Recognize, Beginners Reason

One key difference between beginners and experts is that experts have seen it all before. Research into chess experts has shown that their primary advantage is their ability to *remember and recognize* the state of the board. This allows them to decide how to respond more quickly and with less effort.[15] Kahneman[19,b] describes cognition as being split into "system 1" and "system 2" (thus proving that not only developers 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.[34]

Expert developers can reason at a higher level by having memorized common patterns in program code, which frees up their cognition.

Expert developers can reason at a higher level by having memorized (usually implicitly, from experience) common patterns in program code, which frees up their cognition.[4] One such instance of this is "design patterns" in programming, similar to previously discussed chunks. An expert may immediately recognize 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 recognizing 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 the 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.

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 look for underlying concepts and do not focus on details, whereas beginners focus on surface details and have trouble connecting these details to the bigger picture. These differences affect how experts

b 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.

reason 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 while 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,[25] 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,[23] such as seeing a mutable variable labeled 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 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, for example, the `abs` function that returns the absolute value of a number.[17] 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. 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.

Problem-solving is (incorrectly) conceived as a generic skill. However, this is not how problem-solving works in the brain.

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.[10] 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.

The reason spacing works is due to the relationship between long-term and working memory previously described in this article. 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.[5] 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 unspaced practice is that people can solve problems, but only when they are told which concept to use.[5] While interleaving different types of problems, such as loop and conditional problems, can help, pathways take time to return to baseline, making spacing necessary to get the most out of practice time.[10] In addition, the brain needs rest to consolidate the new information that has been processed so that it can be applied to new problems.

Going against this time-tested principle, intensive coding bootcamps require learners to cram their problem-solving practice into unspaced 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.[10] For people whose only vi-

able 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.[21] The neurochemical balance in the brain makes concentration difficult after this point.[21] After each learning bout, take at least 20 minutes to rest.[21] 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.

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.[5] Be forewarned, though, that randomizing the order improves learning outcomes but requires more effort.[6] 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.[18]

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 such as 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.[1] 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.[1] 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.[1]

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).[11] 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.[14] It seems that searching may rob the brain of the benefits of the memory-strengthening effect of recalling information.

There is also the issue of cognitive load discussed earlier. 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 memorized, 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 memorize information, despite it being available on the Internet.

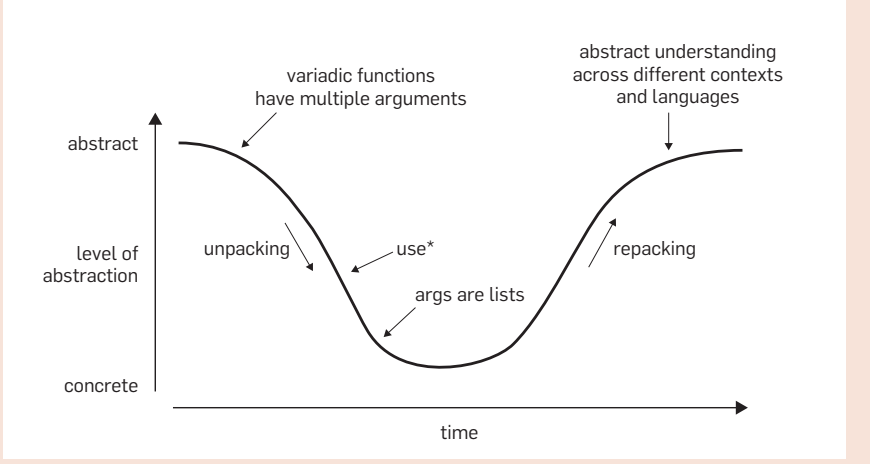
7. Problem-Solving Is Not a Generic Skill

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, and so on). 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 do not 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. Research into chess found little or no effect of learning it on other academic and cognitive skills, and the same is true for music instruction and cognitive training.[36] This inability to transfer problem-solving skills is why "brain training" is ineffective for developing general intelligence.[29]

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 mental-

Figure 2. The semantic wave for variadic functions.



ly 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.[24] Yet, spatial training improves performance on a range of non-verbal skills regardless of initial ability, age, or type of training task.[40] Recent work has even demonstrated that spatial training can improve efficiency for professional software developers, likely because they are still learning new concepts.[30] 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 brain-teaser puzzles, such as how to weigh a jumbo jet. As Google worked out by 2013, this is a waste of time[7]—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.

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 increased cognitive load) 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: 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 careers. 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. 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 the transfer of knowledge.[39]

Experts often help to train beginners, but experts without experience in training others often do not realize 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: difficulty in seeing things through the eyes of a beginner once you have become an expert. 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.[1] 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, 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. They can help the beginner develop new knowledge and the expert to rediscover 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. 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. 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.[35] There was an industry of aptitude tests for programming that began in the 1960s, but as Robins[33] summarizes, 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 a higher reputation.[27] 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,[31] 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 and working memory capacity.[4] 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. A sub-measure of these two factors, *spatial reasoning*, is a stronger predictor of success in programming, though still quite moderate.[30] Spatial reasoning also predicts success in other science and math fields,[24] so this is not programming-specific. Further, these weak-to-moderate correlations largely disappear with increased experience for various rea-

sons. 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.

Attempts to produce a predictive test for programming ability have generally come to naught.

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. One of the more compelling theories is the idea of learning edge momentum,[33] 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 are born with it or not), which has little empirical evidence.[26] As discussed in the previous section, we have recently come to understand differences in programming ability as differences in prior experience.[16] 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 highly technical field that is optionally taught before university (for example, CS, physics, and engineering).[9]

The binary split view, and its effects on teaching and learning, have been studied across academic disciplines in research about fixed versus growth mindsets.[12] 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 Cheryan et al.,[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. 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.[12]

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.[8] In particular, there are two common misconceptions about how to promote a growth mindset that prove ineffective. The first misconception is to reward effort rather than performance because a 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.[13] 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 toward 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.[13]

A related concept to fixed and growth mindsets is goal orientation. This is split into two categories: approach and avoidance. The "approach" goal orientation involves wanting to do well, and this engenders positive and effective learning behaviors: working hard, seeking help, and trying new and challenging topics. In contrast, the "avoidance"

goal orientation involves avoiding failure. This leads to negative and ineffective behaviors: disorganized 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 toward "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 and accept that they will make mistakes without chastising them. 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 toward a fixed mindset in the face of failure, but it can also be redeveloped and made 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.

Summary

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 with 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.

Intelligent people will not always make good programmers, and good programmers need not be high in general intelligence.

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. To emphasize a specific point: Do not test candidates with brain-teaser puzzles.
- At least among young developers, years of experience may not be a very reliable measure of ability.
- A related recommendation from Behroozi et al.[3] 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 someone become a more efficient programmer.
- Experts are not always the best at training beginners.
- Learning takes time, including time *between* learning sessions. Intense cramming is not effective, but spaced repetition is.
- Similarly, spending time away from a problem can help to solve it.
- Just because you can find it through an Internet search or generative AI tool does not mean learning has become obsolete.
- Use examples to go between abstract concepts and concrete learnable facts.
- Seeking to succeed (rather than avoid failure) and believing that ability is changeable are important factors in resilience and learning.

Further reading. Many books on learning center 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 Daniel T. Willingham provides a short

more online

A list of full references and supplementary information is available at https://bit.ly/3G2NPNl.

and readable explanation of many of the principles of memory and how the brain works.

▸ *The Programmer's Brain* by Felienne Hermans et al.[c] 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 Paul A. Kirschner and Carl Hendrick[20] 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 websites; you may wish to try searching the Web 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. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

c Full disclosure: This is written by one of the authors although the other authors recommend it as well.

References

1. Anderson, J.R. *Cognitive Psychology and its Implications*. Macmillan (2005).
2. Baddeley, A. Working memory. *Science 255*, 5044 (1992), 556–559.
3. Behroozi, M., Shirolkar, S., Barik, T., and Parnin, C. Does stress impact technical interview performance? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conf.* and *Symp. on the Foundations of Software Engineering* (2020), 481–492; https://bit.ly/3Sj3AHn
4. Bergersen, G.R. and Gustafsson, J.-E. Programming skill, knowledge, and working memory among professional software developers from an investment theory perspective. *J. of Individual Differences 32*, 4 (2011), 201.
5. Bjork, R.A. and Allen, T.W. The spacing effect: Consolidation or differential encoding? *J. of Verbal Learning and Verbal Behavior 9*, 5 (1970), 567–572.
6. Bjork, R.A. and Bjork, E.L. Desirable difficulties in theory and practice. *J. of Applied Research in Memory and Cognition 9*, 4 (2020), 475.
7. Bryant, A. In head-hunting, big data may not be such a big deal. *The New York Times* (June 2013); https://nyti.ms/3Msdraa.
8. Burgoyne, A.P., Hambrick, D.Z., and Macnamara, B.N. How firm are the foundations of mind-set theory? The claims appear stronger than the evidence. *Psychological Science 31*, 3 (2020), 258–267; https://bit.ly/3MsdPp4
9. Cheryan, S., Ziegler, S.A., Montoya, A.K., and Jiang, L. Why are some STEM fields more gender balanced than others? *Psychological Bulletin 143*, 1 (2017), 1.
10. Dempster, F.N. The spacing effect: A case study in the failure to apply the results of psychological research. *American Psychologist 43*, 8 (1988), 627.
11. Dong, G. and Potenza, M.N. Behavioural and brain responses related to Internet search and memory. *European J. of Neuroscience 42*, 8 (2015), 2546–2554; https://bit.ly/49loGL6
12. Dweck, C.S. *Mindset: The New Psychology of Success*. Random House (2006).
13. Dweck, C.S. and Yeager, D.S. Mindsets: A view from two eras. *Perspectives on Psychological Science 14*, 3 (2019), 481–496.
14. Giebl, S. et al. Thinking first versus Googling first: Preferences and consequences. *J. of Applied Research in Memory and Cognition* (2022); https://bit.ly/3QlXun5
15. Gobet, F. and Simon, H.A. 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 (1996), 52–55; https://bit.ly/40oT4jY
16. Grabarczyk, P., Nicolajsen, S.M., and Brabrand, C. On the effect of onboarding computing students without programming-confidence or experience. In *Proceedings of the 22nd Koli Calling Intern. Conf. on Computing Education Research* (2022), 1–8.
17. Hazzan, O. Reducing abstraction level when learning abstract algebra concepts. *Educational Studies in Mathematics 40*, 1 (Sept. 1999), 71–90; https://bit.ly/49lZd4f
18. Huberman, A. Teach & learn better with a "neuroplasticity super protocol". *Neural Network* (Oct. 2021).
19. Kahneman, D. *Thinking, Fast and Slow*. Macmillan, 2011.
20. Kirschner, P.A. and Hendrick, C. *How Learning Happens: Seminal Works in Educational Psychology and What They Mean in Practice*. Routledge (2020).
21. Kleitman, N. Basic rest-activity cycle—22 years later. *Sleep 5*, 4 (1982), 311–317.
22. Kyllonen, P.C. and Christal, R.E. Reasoning ability is (little more than) working-memory capacity?! *Intelligence 14*, 4 (1990), 389–433.
23. Margulieux, L. et al. When wrong is right: The instructional power of multiple conceptions. In *Proceedings of the 17th ACM Conf. on Intern. Computing Education Research* (2021), 184–197.
24. Margulieux, L.E. Spatial encoding strategy theory: The relationship between spatial skill and STEM achievement. In *Proceedings of the 2019 ACM Conf. on Intern. Computing Education Research*, 81–90; 10.1145/3291279.3339414.
25. Maton, K. Making semantic waves: A key to cumulative knowledge-building. *Linguistics and Education 24*, 1 (2013), 8–22; https://bit.ly/3tXhvZt.
26. McCartney, R. et al. Folk pedagogy and the geek gene: geekiness quotient. In *Proceedings of the 2017 ACM SIGCSE Technical Symp. on Computer Science Education* (2017), 405–410.
27. Morrison, P. and Murphy-Hill, E. Is programming knowledge related to age? An exploration of stack overflow. In *2013 10th Working Conf. on Mining Software Repositories (MSR)*, 69–72; https://bit.ly/3Sp5Mgv.
28. Newton, P.M. and Salvi, A. How common is belief in the learning styles neuromyth, and does it matter? A pragmatic systematic review. *Frontiers in Education* (2020), 5; https://bit.ly/47h0Jmo.
29. Owen, A.M. et al. Putting brain training to the test. *Nature 465*, 7299 (2010), 775–778.
30. Parkinson, J. and Cutts, Q. Relationships between an early-stage spatial skills test and final CS degree outcomes. In *Proceedings of the 53rd ACM Technical Symp. on Computer Science Education 1*, (2022), 293–299; 10.1145/3478431.3499332.
31. Peitek, N. et al. Correlates of programmer efficacy and their link to experience: A combined EEG and eye-tracking study. In *Proceedings of the 30th ACM Joint European Software Engineering Conf.* and *Symp. on the Foundations of Software Engineering* (Nov. 2022), 120–131.
32. Raufaste, E., Eyrolle, H., and Mariné, C. Pertinence generation in radiological diagnosis: Spreading activation and the nature of expertise. *Cognitive Science 22*, 4 (1998), 517–546; https://bit.ly/3sidUoq
33. Robins, A. Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education 20*, 1 (2010), 37–71.
34. Robins, A.V. Dual process theories: Computing cognition in context. *ACM Trans. Comput. Education 22*, 4 (Sept. 2022); 10.1145/3487055.
35. Rountree, N., Rountree, J., Robins, A., and Hannah, R. Interacting factors that predict success and failure in a CS1 course. *ACM SIGCSE Bulletin 36*, 4 (2004), 101–104.
36. Sala, G. and Gobet, F. Does far transfer exist? Negative evidence from chess, music, and working memory training. *Current Directions in Psychological Science 26*, 6 (2017), 515–520; 10.1177/0963721417712760.
37. Simons, D.J. and Chabris, C.F. What people believe about how memory works: A representative survey of the U.S. population. *PLOS ONE 6*, 8 (Aug. 2011), 1–7; 10.1371/journal.pone.0022757.
38. Thalmann, M., Souza, A.S., and Oberauer, K. How does chunking help working memory? *J. of Experimental Psychology: Learning, Memory, and Cognition 45*, 1 (2019), 37.
39. Tshukudu, E. and Cutts, Q. Understanding conceptual transfer for students learning new programming languages. In *Proceedings of the 2020 ACM Conf. on Intern. Computing Education Research*, 227–237; 10.1145/3372782.3406270.
40. Uttal, D.H. et al. The malleability of spatial skills: A meta-analysis of training studies. *Psychological Bulletin 139*, 2 (2013), 352.

Neil C.C. Brown is a senior research fellow at King's College London, U.K.

Felienne F.J. Hermans is a professor at Vrije Universiteit Amsterdam, The Netherlands.

Lauren E. Margulieux (lmargulieux@gsu.edu) is an associate professor at Georgia State University, Learning Sciences, Atlanta, GA, USA.

Watch the authors discuss this work in the exclusive *Communications* video. https://cacm.acm.org/videos/ten-things-software-developers