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 However, many people's intuitions about human learning and memory are incorrect, such as memorising facts is obsolete with access to the Internet.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.

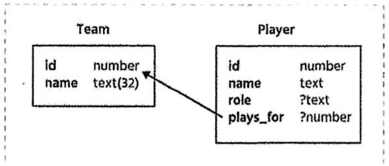
Another curious feature of human memory is "spreading activation" [4]. Our memories are stored in interconnected neural pathways. When we try to remember something, we activate a pathway of neurons to access the targeted information. However, activation is not contained within one pathway. Some of the 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



compared to

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.