On the Nature of Programmer Expertise

Chris Parnin
NC State University
Raleigh, NC, USA
cjparnin@ncsu.edu

Janet Siegmund
University of Passau
Passau, Germany
siegmunj@fim.uni-passau.de

Norman Peitek
Leibniz Institute for Neurobiology
Magdeburg, Germany
npeite@lin-magdeburg.de

Abstract

Many experts in fields such as mathematics, medicine, and chess display intellectual marvels undiminished with age. However, software engineers, much like athletes, seem to have a limited lifetime for applying their expertise. Compared to other areas of expertise, the elements of which programming expertise is built upon is unstable, short-lived, and often non-transferable. In this position paper, we derive insights from psychology, cognitive neuroscience, and decades of software engineering research on expertise. Using these insights, we strive to understand what representations, strategies, and cognitive processes and mechanisms experts use when performing exceptional programming feats. In particular, we want to understand how expertise shapes an expert's mind, and understand the intricate patterns and strategies that expert programmers hone over the years. To answer these questions, we propose to use several brain-imaging techniques to study expert software engineers. Finally, based on these results, we wish to derive guidelines in order to help companies and teachers in identifying and training programmers to quickly adapt to changes in terms of languages, projects, teams, and techniques.

1. Introduction

Some of the most amazing feats are performed by human experts, who demonstrate a mastery of skills obtained through many years of deliberate practice (Ericsson & Lehmann, 1996). As an expert, programmer's eyes glide across source code of a program, and in just seconds, they can extract a deep understanding from abstract symbols and text arranged in program files, which may take a novice programmer a good part of a day to truly understand. Experts' eyes dance around source code, finding points of interest, such as method signatures, and follow relationships such as data-flow relationships between program elements. They can spot typical errors like an off-by-one error incredibly fast, demonstrating that they have abstracted the comprehension process to be more efficient. Expertise in programming, as in any other areas, requires years and years of continuous and deliberate practice. But even after extensive experience in programming, there are still tremendous differences in programmer expertise: So called 10xers can be 10 or more times as productive as other programmers who have spent an equal amount of time with programming (McConnell, 2010; Sackman, Erikson, & Grant, 1968). Likewise, there are novice programmers who can grasp the concepts much faster than their peers, but until today, we can only hardly explain why such differences occur.

Ever since the emergence of software engineering as a discipline, having programmer expertise on board of a project has tremendous importance. However, still today, it is difficult to evaluate whether a programmer will be a good addition to a software project, especially when different software technologies, architectures, and products may be involved. Despite decades of research on expertise and program comprehension, there is a general lack of understanding of what programmer expertise actually is. Solving this problem is of fundamental importance. We believe that an understanding of the fundamental human limitations associated with gaining expertise in programming skills will enable developers and companies to more effectively invest in technology choices and training strategies. For example, the ten-thousand rule[1] is a common rule of thumb for estimating how long it takes to gain the highest level of expertise. If an expert Java server-side developer needs to transition into a new role developing front-end web technologies, does the developer also need to reinvest ten years of effort before regaining proficiency or is it just six months?

[1] The ten-thousand rule states that it takes 10 000 hours of practice of a skill to become an expert.

Why is it so difficult to understand programmer expertise? In this position paper, we will bridge together several insights into expertise and also define several research experiments that will help us find a way out of this dilemma. To this end, we will first look at expertise from different angles:

- Cognitive Psychology: Cognitive psychology describes how humans create information from all the data surrounding them. It matured as a discipline over many decades, so it offers lots of relevant insights, for example, how learning strategies differ and how experts are created. Expertise in many areas, including chess and tetris, is a well-studied phenomenon. Thus, this mature research will help us to better understand the nature of programmer expertise.
- Neuroscience: Neuroscience uses neuro-imaging techniques, such as fMRI, EEG, CT, to relate brain activation to cognitive processes. This area is relatively new, but it extends from cognitive psychology, in that it can interpret the brain activation in relation to the well-studied cognitive processes. For example, results show that experts use their brain more efficiently than novices, such that they use a specialized brain area. Expert golfers need just one small area to perform a golf swing, but novices use a large network of brain areas (Milton, Solodkin, Hlustík, & Small, 2007).
- Software Engineering: Also in software engineering, expertise has been studied, but rather superficially in comparison to cognitive psychology and neuroscience. Specifically, researchers have been studying how tasks, tools, and complexity affect programming productivity, but the relationship to programmer expertise has not been well-studied. Thus, re-evaluating this research in its potential to shed more light on programmer expertise will give us valuable insights, also into why there was and is so little progress on programmer expertise.

Eventually, we expect that the research that we outline in this paper may eventually enable better guidelines for training and measuring expert programmers. For example, with a set of validated fMRI biomarkers of effective programming expertise, we can improve educational interventions by identifying deviations from desired brain activation when reasoning about a particular programming problem or concept such as recursion.

2. Expertise

In this section, we highlight the insights from cognitive psychology, neuroscience, and software engineering. Afterward, we describe how these different viewpoints can help us answer about expertise, and establish a roadmap toward understanding programmer expertise.

2.1. Insights from Cognitive Psychology

2.1.1. Deliberate Practice

Research in cognitive psychology revealed that experts demonstrate a mastery of skills obtained through many years of deliberate practice (Ericsson & Lehmann, 1996). In understanding what makes an expert an expert, scientists have studied the training strategies, cognitive representations, and problem-solving strategies used when performing tasks with exceptional skill. Initial research focused on the training strategies and researchers found that there is a consistent difference between experience (in terms of years spent) and expertise (performance levels) (Camerer & Johnson, 1991; Shanteau & Stewart, 1992)—the primary difference in performance arises from *how* experts were trained and not necessarily how *long*. For example, when comparing chess experts who have spent equal time in gaining experience, the consistently best performers are the ones who repetitively studied specific chess positions and scenarios, as opposed to lower performers who just practiced in tournaments (Charness, Krampe, & Mayr, 1996).

2.1.2. Problem Representation

Cognitive expertise involves *chunking* of information, or organizing a stream of perceptual cues into a more meaningful pattern (De Groot, 1978). Experts use more effective problem representations and generate better "next steps or moves" (in chess) (Simon, 1990) or select the best diagnostic option (in medicine) (Elstein, Shulman, & Sprafka, 1990). Experts differ from novices in how they process infor-

mation and arrive at an answer, such that they look a bit deeper and process next steps faster (Holding, 1992), resulting in improved qualities of answers (Elstein et al., 1990).

2.2. Insights from Neuroscience

2.2.1. Neural Efficiency and Cortical Differences

Studies of brain activity find that experts demonstrate more efficient neuronal firing patterns than novices with the same tasks (Neubauer & Fink, 2009). Expert brains work differently than non-expert brains. When novices are compared with experts performing the same kinds of tasks, the differences can be remarkable. When novice golf players try to perform a golf swing, their brains are alight with activity throughout many areas of the brain as they clumsily try to coordinate the swing in their mind, whereas experts have conceptualized the movements of a golf swing into a simple, focused, and energy-efficient action in the brain (Milton et al., 2007). Not only does an expert's brain act more efficiently, experts sometimes also have a larger brain mass in these areas. A larger right posterior parietal cortex is seen in expert video game players (Tanaka et al., 2013). Experienced London taxi drivers have larger parahippocampal regions with size correlated with years of experience (Maguire, Woollett, & Spiers, 2006).

2.2.2. Specialization

Although the brain appears to have specialized areas for specialized tasks, when humans develop new skills, there is often no specific area of the brain that supports that skill. Instead, learning processes often recruit existing information-processing networks of the brain in support of the new skill. For example, the fusiform face area is strongly associated with face perception, but it also gets recruited when identifying a specific object, such as for bird experts who can distinguish between a vast variety of bird species or for car experts who can identify distinct differences between many models and makers of cars. Interestingly enough, bird experts who are not car experts do not use the fusiform face area when observing cars, and vice versa (Gauthier, Skudlarski, Gore, & Anderson, 2000). Other studies also show the involvement of the fusiform face area in experts of tasks that require visual perception, including the categorization of chest radiographs (Harley et al., 2009) or understanding chess positions (Bilalić, Langner, Ulrich, & Grodd, 2011).

2.3. Insights from Program Comprehension Studies

Over the past few decades, several theories of program comprehension have been proposed and empirical studies attempting to validate them have been performed.

2.3.1. Syntax vs. Semantics

Software engineering researchers have proposed that programmers use knowledge structures that encode semantic (Shneiderman & Mayer, 1979) and domain information (Brooks, 1983) about a program as well as *prime structures* (Linger, Mills, & Witt, 1979), that include elements of syntax, control-flow and data-flow (Pennington, 1987b) of the program. These knowledge structures (Rich, 1981) have been formalized referred to as *programming plans*. Motivation for programming plans was inspired from theoretical constructs in text comprehension, such as *scripts*, which are mental representations of common activities (e.g., eating in a restaurant) and can aid humans in understanding and remembering narrative text (Bower, Black, & Turner, 1979). Programming plans act like schemas that are first instantiated and then its slots are filled with concrete values as a programmer builds an understanding of the code (Soloway, Ehrlich, & Bonar, 1982). Plans may help programmers fill in the "gaps" when trying to understand code. Finally, it was proposed that programs follow basic rules of discourse and that any violation to "accepted conventions of programming" should as a result hamper an expert's ability to use programming plans (Soloway & Ehrlich, 1984).

Evidence that expert programmers have different mental representations from novices has been described in several studies, however not all evidence is consistent with the theory of programming plans. In a series of studies, participants were asked to understand a piece of code and later recall text of the program. Experts recall programs better than novices when the order of presentation is correct (Shneiderman, 1976), but performance difference disappears when programs are presented in random order. Further, when examining the details of what is recalled (Shneiderman & Mayer, 1979),

researchers found that experts could recall semantic information about source code, but incorrectly recalled the exact details. Novices did the opposite: They could more accurately replicate the source code syntax, but often mistook the meaning of the source code. In another study, when categorizing related code snippets, experts and novices differed in their organization (procedural similarity vs. syntax similarity) (Adelson, 1981). Finally, Soloway and Ehrlich (Soloway & Ehrlich, 1984) evaluated the theory of rules of discourse by varying the style of the snippets, such that there were versions that followed typical coding conventions (plan-like) and versions that explicitly violated such conventions (unplan-like). For example, they changed the variable naming, such that in the violated version, the naming did not convey the purpose of the variable, but rather the opposite (`max` was renamed to `min`). The results showed that novice programmers were not affected by the violated coding conventions. However, experts were significantly slower with these version and made significantly more errors—specifically, experts became as slow and as incorrect as novices.

However, another series of studies cast doubt on the nature of programming plans. Gilmore and Green failed to replicate Soloway and Ehrlich's previous results (Soloway & Ehrlich, 1984) when using programming plans from Pascal programs with Basic programmers (Gilmore & Green, 1988). They suggested that programming plans may not generalize across different languages, and that plans cannot represent the underlying deep structure of programs. Bellamy and Gilmore (Bellamy & Gilmore, 1990) examined the protocols generated from experts in different languages as they created programs. Using two different models of programming plans, they found neither model was well supported by protocols; further, different programming language experts generated different types of representations. Finally, Pennington (Pennington, 1987b) theorized that if programmers form plan-based mental representations, then they should recognize lines faster when preceded by lines from the same plan structure. Unfortunately, in the study, stronger priming effects were observed from syntax structure vs. plan structure. Subjects also made fewer errors on control-flow questions, compared with data-flow and functional questions. Pennington concluded that:

> *While plan knowledge may well be implicated in some phases of understanding and answering questions about programs, the relations embodied in the proposed plans do not appear to form the organizing principles for memory structures.*

2.3.2. Strategies

Several software engineering researchers have noted that expert programmers make better use of strategies and are more fluid in adapting strategies when compared to novice programmers. Previous theories of program comprehension have proposed two primary mechanisms: top-down and bottom-up program comprehension. *Top-down comprehension* is a hypothesis-driven process, in which developers initially form hypotheses about source code and, by looking at more and more details, refine these hypotheses subsequently, until they form an understanding of the program (Brooks, 1978). The programmer is guided by using cues called *beacons* that are similar to *information scents* in information foraging theory (Pirolli & Card, 1999). With *bottom-up comprehension*, developers start with details of source code and group these details to semantic chunks, until they have formed a high-level understanding of the program (Shneiderman & Mayer, 1979). With *opportunistic* and *systematic strategies* (Littman, Pinto, Letovsky, & Soloway, 1987), programmers either systematically examine the program behavior or seek boundaries to limit their scope of comprehension on an as-needed basis. Von Mayrhauser and Vans offered an integrated metamodel (von Mayrhauser & Vans, July 1993) to situate the different comprehension strategies in a single model scheme. Finally, Murray and Lethbridge (Murray & Lethbridge, 2005) have proposed that programmers develop and make use of specific strategies for specific types of problems rather than conforming to a general framework (e.g., the strategy needed for understanding the cultural practices of a team is distinguished from getting the big picture of a program).

Evidence for different strategy usage by experts and novices has been observed in several studies. Shaft and Vessey evaluated how program-comprehension strategies depend on expertise in a program's domain, and found that depending on familiarity, programmers use completely different strategies (Shaft

& Vessey, 1995). These results were consistent with top-down (familiar domain) and bottom-up (unfamiliar domain) comprehension models that were state of the art during that time. There are several studies that evaluated these comprehension models and how context drives the comprehension process that developers choose (Brooks, 1978; Widowski, 1987; Pennington, 1987a). In fact, some studies suggest that the ability for an expert to switch strategies is a more important factor than knowledge representation. For example, Widowski (Widowski, 1987) compared novice and expert programmers working on unplan-like and plan-like programs of varying semantic and syntax complexity. Again contradicting Soloway and Ehrlich's results (Soloway & Ehrlich, 1984), experts actually did better on unplan-like programs. Further analysis of protocols found that experts were able to adjust their strategy (structure-oriented vs. variable-oriented) more than novices in order to do account for unplan-like programs. Finally, Vessey (Vessey, 1987) observed when debugging a program, unsuccessful programmers used an erratic mix of reinspections and navigations, whereas successful programmers maintained a smooth progression through the program's execution. Further, effective strategy use could account for 74 % of debugging time, whereas chunking ability could only account for 31 %.

In conclusion, when considering programming plans and strategies, Gilmore (Gilmore, 1990) has argued that "plans" are insufficient for explaining expertise alone, and instead, that multiple factors, including practice and strategy acquisition must be examined:

> *Expertise is not as simple as we might sometimes think. Although high-level, efficient representations of programming knowledge develop with experience, it seems that this knowledge is not the sole determinant of programming success. Besides chunking of knowledge structures, experts seem to acquire a collection of strategies for performing programming tasks, and these may determine success more than does the programmer's available knowledge...Programming may be rather like riding a bike, or some other motor skill, without practice it cannot be mastered.*

3. How to Find a Way into the Light

Thus, with everything that we learned, we can sketch a roadmap toward a thorough understand of programmer expertise. We learn from cognitive psychology the role of deliberate practice, indicating that how novices work with code plays a crucial role in their learning process. Neuroscience shows that experts use their brain differently when completing a task they have experience in. Research in software engineering revealed different aspects of code (e.g., plan-like vs. unplan-like) and knowledge of programmers determine how programmers work with code. Thus, combining these three areas will give us unique opportunities to understand the nature of programmer expertise.

Based on the insights from the different disciplines, we have derived hypotheses that we think are the most important to address in future projects. In this section, we discuss these hypotheses and outline experimental designs to evaluate them.

3.1. How Does Programming Shape the Expert's Mind?

Research Hypothesis: Experts and novices have a different neuroanatomical representation, such that novices use a different network of brain areas.

Rationale: From brain imaging studies of experts in various fields, we know that experts in different occupations use different neural mechanisms and the insights we get from one kind of expert may not generalize to other types of experts. Different studies suggest that experts have encoded the process more efficiently in a single/few distinct brain area, possibly one that is typically used for other processes (like the fusiform face area, which is used by bird experts also to recognize birds). Programming is a more complex task, which consists of, among others, understanding code and writing code. Both processes are different, so we expect differences in brain activation. Thus, we expect that the study of expert programmers will reveal unique patterns of brain activations.

Design: To understand the neuro-anatomical representation of programmer expertise, we need to study

novice and expert programmers. A large portion of programming activities consists of, in essence, reading code. For this reason, we focus initially on code comprehension. A promising option is to build on the snippets that have already been designed for neuro-imaging studies, such as in reported in several brain imaging studies (Siegmund, Kästner, et al., 2014; Crk & Kluthe, 2014). Another way is to let participants categorize source code snippets based on functionality (Adelson, 1981), or identity correct or wrong implementations. Finally, another option is to use set of validated assessment instruments called concept inventories (Adams & Wieman, 2011), which provides the ability to assess if someone grasps the necessary concepts for expertise in a given field or domain.

We will collect block-based fMRI brain scans of expert programmers and novice programmers with programming and control tasks. Block designs are well suited to localize functional areas and study steady state processes, such as attention and comprehension, because they maximize the hemodynamic responses associated with cognitive processes, allowing for improved ability to locale brain areas. Using control tasks improves the ability to further isolate essential areas of cognitive for programming rather than supporting areas (e.g., eye movement while reading or locating syntax errors).

Objectives: The study and analysis will attempt to achieve the following objectives.

- *Identify brain regions associated with expertise in programming.* Understanding how programming shapes the expert's mind will help us to get a deeper understanding of the nature of programmer expertise. We expect that we will see differences in activation between novices and experts, such that experts recruit one or a few areas to understand and create source code, while novices require a wide-spread network of areas. Furthermore, we expect also a difference between experts and 10xers, for example, regarding activation in an even more concise area. This will help us to spot experts and 10xers more easily, and light the path that novice programmers have to follow to become an expert.
- *Identify cortical differences associated with expertise in programming.* Structural MRI brain scans of expert programmers and control participants will be analyzed using voxel-based morphometry, a method that allows an automatic whole-brain analysis of gray-matter volume. Compared with controls, we expect that expert programmers will have greater gray-matter volume in the previously identified brain areas. This analysis will further validate our results, as it will demonstrate that expert programmers not only use a distinct set of brain areas, but that years of programming experience has indeed shaped the expert's mind.

3.2. Which Theories of Programmer Expertise Can Be Supported?

Research Hypothesis: Top-down comprehension and bottom-up comprehension are represented in different networks of brain areas.

Rationale: Different representation of top-down and bottom-up processes have been observed in cognitive psychology and neuroscience. Since both processes share similarity with their psychological and neuro-scientific counterparts, we expect to observe a similar distinction in brain activation. Specifically, we expect memory-related activation, because top-down comprehension requires concepts to be stored in memory, and these should be retrieved during comprehension.

Design: To differentiate between top-down and bottom-up comprehension, we need to decompose each process into its stages. For example, bottom-up comprehension requires perceiving and processing a line of source code (visual perception, reading comprehension), retrieving meaning from the source code (problem solving), chunking statements to semantic units (synthesis), and finally concluding the purpose of source code (induction). To observe these stages in novice and expert programmers, we will show both groups obfuscated code, preventing experts from applying their knowledge about plans, strategies, and domains. Regarding top-down comprehension, programmers need access to their knowledge (memory retrieval), deduce the high-level purpose of source code (deduction), and confirm the purpose by examining details (memory retrieval, matching). To enable novices to actually use a top-down

approach, we intend to train them with specific snippets and according plans (e.g., that a decreasing loop counter often indicates a reversing operation) and then observe both novices and experts with the same snippets. This will also allow us to observe how expertise and the comprehension strategies interact with different brain activation. Possibly, we observe four different patterns depending on expertise (novice, expert), and strategy (top-down, bottom-up).

Objectives: The study and analysis will attempt to achieve the following objectives:

- *Identify strategy differences between experts and novices according in the comprehension strategy.* We will use multivoxel pattern analysis and hidden semi-Markov models (Anderson, Pyke, & Fincham, 2016) to identify distinct stages associated with program comprehension. This analysis will allow us to find differences in how long experts and novices spend time in different cognitive stages, and whether novices or experts introduce any distinct stages or ordering in their problem solving.
- *Identify differences in neural efficiency between experts and novices.* We examine deactivation strengths between different cortical areas in order identify neural efficiency (Durning et al., 2015). For example, during cognitive tasks, the default network is often deactivated in order to enable concentration on the task; the strength of the deactivation correlates with the difficulty of the task (Buckner, Andrews-Hanna, & Daniel, 2008). Analyzing the deactivation, such as the default network, allows us to observe when participants need to concentrate more in order to solve a task. Hence, we expect differences in deactivation patterns between experts and novices. Furthermore, we expect a stronger deactivation for bottom-up comprehension, compared to top-down comprehension.

4. Discussion

We discuss several implications and challenges for future research on programmer expertise.

4.1. Implications

In software engineering, in a classic study, Sackman evaluated typical tasks during programming, such as implementing and debugging code, and measured the time and correctness developers needed to complete such tasks (Sackman et al., 1968). He found that even for expert programmers, there are huge differences in performance, up to a factor of 25. On average, the efficiency varied by a factor of 10, leading to the term "10xer" for developers who are exceptionally good at their job. If we can understand how experts deliberately practice programming, we can devise more efficient training strategies for novice programmers and professional programmers to adapt their skills to new technologies.

There are several ideas that are promising. In the Pragmatic Programmer (Hunt, 2000), Hunt proposed the idea of practicing a *code kata*, which was an exercise in programming that can help a programmer practice a programming skill on a daily basis. Similarly, there has been a proposal to teach software engineering concepts via athletic "cross-fit" programming exercises (Hill, Johnson, & Port, 2016). The insight is that in cross-fit training, you try to best your previous performance record for a particular workout. Likewise, students learn to practice programming exercises (workout of the day), but as they practice the exercise, they measure their performance relative to an "expert". In class, they be assessed with a new challenge and only receive credit if they could complete exercise within 20 minutes. Overall, the idea of targeted practice on a set of programming skills is consistent with effective training many other domains, such as chess, where deliberate practice on specific openings, and practicing generating moves for critical board positions was found to be more effective than just straight game play.

4.2. Challenges

Measures of Expertise: It is critical for the success of the study to choose appropriate and representative programmers for the novices and experts group in order to clearly distinguish cortical and strategic differences in expertise. However, it is a challenge to establish a sound measure of programming expertise, which does not simply rely on a rudimentary measure of experience. Many studies have used rather superficial and ad-hoc measures to define programmer expertise. For example, Soloway and Ehrlich categorized undergraduate students as novices, and graduate students as experts. However, a deeper investigation into the expertise levels of their participants did not happen. This makes it rather difficult

to fully understand how expertise actually influenced the study results.

In industry, technical interviews are an expensive and often unsatisfactory process. Alternative methods, such as standardized tests, requires considerable effort to develop and validate. For example, Bergerson and others describe an approach to develop such an instrument, which is based on established techniques that are rooted in psychology (Bergersen, Sjøberg, & Dybå, 2014). This test comprises several programming tasks in Java and has been validated with 65 programming experts over two days. Extending approaches for standardized measures of experience levels of programmers (Siegmund, Kästner, Liebig, Apel, & Hanenberg, 2014) is a start. Additionally, screening participants with technical interviews and standardized tests of programming expertise (Bergersen et al., 2014) can be a further filter. Conservatively, only programmers who pass all screening measures should be admitted to the study's experts group.

A related challenge is to select 10xers from expert programmers. We expect the comparison between 10xers and experts will show further differences beyond the contrast of experts and novices. Thus, it is interesting to also include 10xers and compare them to average expert programmers. Of course, finding 10xers under experts is a similarly difficult task. The previous selection process might not be enough to identify 10xers from a group of expert programmers. To this end, recruiting participants with high ranks from leaderboards, such as HackerRank, or through supervisor's recommendations can be promising.

Material and Task Design: For creating code, specifying the programming problem will be challenging. Focusing on too small problems (e.g., factorial), it may be difficult to see differences between novices, experts, and 10xers, but too large problems may again be impossible to implement, because they require too much time or lead to source code that is too long. Further, depending on the presentation and context of the programming problem, some programmers may adapt a different strategy when solving, which should be anticipated and controlled for. Using pilots to design programming tasks with a known set of strategies and suitable difficulty outside a fMRI scanner is a good initial step.

5. Conclusion

Different insights from cognitive psychology, cognitive neuroscience, and classic studies in software engineering can be combined in order to answer important challenges in understanding expertise of software developers. Research on cognitive psychology tells us that deliberate practice plays an important part toward becoming an expert. Neuroscience tells us that experts have specialized areas that support a very efficient representation of cognitive processes. Software engineering studies have demonstrated the importance of mental representation and strategies used by experts.

Studies in expertise are useful in guiding the program-comprehension research community. For example, we could examine more closely how overloading the working-memory capacity affects program comprehension or whether 10xers have above-average working memory, language skills, can better concentrate, or whether they use completely different approaches when working with source code. We may be able to find alternative ways to identify experts and even support training. For example, we could expect that programming experts have a certain activation pattern when working with code. Thus, if we find such a pattern in a job applicant, we can assume a certain level of expertise. Furthermore, this could facilitate educational interventions to improve desired regional brain activation in order to reduce cognitive errors and increase programmer expertise. In the long run, ideally these kinds of studies can finally contribute to answer long-asked and sometimes heatedly-discussed questions: Should we start programming education with objects or functions first or teach generic language skills? What effect does the mother tongue have on learning programming? What makes programmers efficient? Can anyone become a good programmer? What does it take to become an exceptionally good programmer? How can we help novices to overcome the typical obstacles of learning to program? Of course, fMRI and also other neuro-imaging techniques are not a "silver bullet" that can definitely answer these questions, but they give a promising new and complementary perspective to our current understanding of program comprehension and the ongoing endeavor to improve the life of programmers.

6. References

Adams, W. K., & Wieman, C. E. (2011). Development and validation of instruments to measure learning of expert?like thinking. *International Journal of Science Education*, *33*(9), 1289-1312.

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory & Cognition*, *9*(4), 422–433.

Anderson, J. R., Pyke, A. A., & Fincham, J. M. (2016). Hidden stages of cognition revealed in patterns of brain activation. *Psychological Science*, *27*(9), 1215-1226.

Bellamy, R., & Gilmore, D. (1990). Programming plans: Internal or external structures. *Lines of thinking: Reflections on the psychology of thought*, *2*, 59–72.

Bergersen, G., Sjøberg, D., & Dybå, T. (2014). Construction and Validation of an Instrument for Measuring Programming Skill. *IEEE Trans. Softw. Eng.*, *40*(12), 1163–1184.

Bilalić, M., Langner, R., Ulrich, R., & Grodd, W. (2011, July 13). Many Faces of Expertise: Fusiform Face Area in Chess Experts and Novices. *The Journal of Neuroscience*, *31*(28), 10206–10214.

Bower, G. H., Black, J. B., & Turner, T. J. (1979). Scripts in memory for text. *Cognitive psychology*, *11*(2), 177–220.

Brooks, R. (1978). Using a Behavioral Theory of Program Comprehension in Software Engineering. In *Proc. int'l conf. software engineering (icse)* (pp. 196–201). IEEE.

Brooks, R. (1983). Towards a Theory of the Comprehension of Computer Programs. *Int'l J. Man-Machine Studies*, *18*(6), 543–554.

Buckner, R., Andrews-Hanna, J., & Daniel, S. (2008). The Brain's Default Network: Anatomy, Function, and Relevance to Disease. *Annals of the New York Acadamy of Sciences*, *1124*, 1–38.

Camerer, C. F., & Johnson, E. J. (1991).
In (p. 195-217). New York, NY, US: Cambridge University Press.

Charness, N., Krampe, R., & Mayr, U. (1996).
In (p. 51-80). Hillsdale, NJ, US: Lawrence Erlbaum Associates, Inc.

Crk, I., & Kluthe, T. (2014). Toward Using Alpha and Theta Brain Waves to Quantify Programmer Expertise. In *Engineering in medicine and biology society (embc), 2014 36th annual international conference of the ieee* (pp. 5373–5376). IEEE.

De Groot, A. D. (1978). *Thought and choice in chess* (Vol. 4). Walter de Gruyter GmbH & Co KG.

Durning, S. J., Costanzo, M. E., Artino, A. R., Graner, J., van der Vleuten, C., Beckman, T. J., ... Schuwirth, L. (2015, Mar 29). Neural basis of nonanalytical reasoning expertise during clinical evaluation. *Brain Behav*, *5*(3), e00309.

Elstein, A. S., Shulman, L. S., & Sprafka, S. A. (1990). Medical problem solving: A ten-year retrospective. *Evaluation & the Health Professions*, *13*(1), 5–36.

Ericsson, K. A., & Lehmann, A. C. (1996). Expert and exceptional performance: evidence of maximal adaptation to task constraints. *Annual review of psychology*, *47*(1), 273–305.

Gauthier, I., Skudlarski, P., Gore, J. C., & Anderson, A. W. (2000, Feb). Expertise for cars and birds recruits brain areas involved in face recognition. *Nature Neuroscience*, *3*(2), 191-197.

Gilmore, D. J. (1990). Expert programming knowledge: a strategic approach. *Psychology of programming*, 223–234.

Gilmore, D. J., & Green, T. R. G. (1988). Programming plans and programming expertise. *The Quarterly Journal of Experimental Psychology Section A*, *40*(3), 423-442.

Harley, E. M., Pope, W. B., Villablanca, J. P., Mumford, J., Suh, R., Mazziotta, J. C., ... Engel, S. A. (2009, Nov 25). Engagement of fusiform cortex and disengagement of lateral occipital cortex in the acquisition of radiological expertise. *Cereb Cortex*, *19*(11), 2746-2754.

Hill, E., Johnson, P. M., & Port, D. (2016, Jan). Is an athletic approach the future of software engineering education? *IEEE Software*, *33*(1), 97-100.

Holding, D. H. (1992). Theories of chess skill. *Psychological Research*, *54*(1), 10–16.

Hunt, A. (2000). *The pragmatic programmer*. Pearson Education India.

Linger, R. C., Mills, H. D., & Witt, B. I. (1979). Structured programming: theory and practice.

Littman, D., Pinto, J., Letovsky, S., & Soloway, E. (1987). Mental Models and Software Maintenance. *J. Systems and Software*, *7*(4), 341–355.

Maguire, E. A., Woollett, K., & Spiers, H. J. (2006, December 1). London taxi drivers and bus drivers: a structural MRI and neuropsychological analysis. *Hippocampus*, *16*(12), 1091–1101.

McConnell, S. (2010). What does 10x mean? measuring variations in programmer productivity. In A. Oram & G. Wilson (Eds.), *Making software: What really works, and why we believe it* (chap. 30). " O'Reilly Media, Inc.".

Milton, J., Solodkin, A., Hlustík, P., & Small, S. L. (2007, April 1). The mind of expert motor performance is cool and focused. *NeuroImage*, *35*(2), 804–813.

Murray, A., & Lethbridge, T. C. (2005). Presenting micro-theories of program comprehension in pattern form. In *Iwpc '05: Proceedings of the 13th international workshop on program comprehension* (pp. 45–54). Washington, DC, USA: IEEE Computer Society.

Neubauer, A. C., & Fink, A. (2009, July 10). Intelligence and neural efficiency. *Neuroscience and biobehavioral reviews*, *33*(7), 1004–1023.

Pennington, N. (1987a). Empirical studies of programmers: Second workshop. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), (pp. 100–113). Norwood, NJ, USA: Ablex Publishing Corp.

Pennington, N. (1987b). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, *19*(3), 295–341.

Pirolli, P., & Card, S. K. (1999). Information foraging. *Psychological Review*, *106*, 643–675.

Rich, C. (1981). *Inspection Methods in Programming* (Tech. Rep. No. TR-604). MIT. Retrieved from `http://hdl.handle.net/1721.1/6934`

Sackman, H., Erikson, W., & Grant, E. (1968). Exploratory Experimental Studies Comparing Online and Offline Programming Performance. *Commun. ACM*, *11*(1), 3–11.

Shaft, T., & Vessey, I. (1995). The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension. *Information Systems Research*, *6*(3), 286–299.

Shanteau, J., & Stewart, T. R. (1992). Why study expert decision making? some historical perspectives and comments. *Organizational Behavior and Human Decision Processes*, *53*(2), 95 - 106.

Shneiderman, B. (1976). Exploratory experiments in programmer behavior. *International Journal of Computer & Information Sciences*, *5*(2), 123–143.

Shneiderman, B., & Mayer, R. (1979). Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *Int'l J. Parallel Programming*, *8*(3), 219–238.

Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., ... Brechmann, A. (2014). Understanding Understanding Source Code with Functional Magnetic Resonance Imaging. In *Proc. int'l conf. software engineering (icse).* (To appear)

Siegmund, J., Kästner, C., Liebig, J., Apel, S., & Hanenberg, S. (2014, October). Measuring and modeling programming experience. *Empirical Softw. Engg.*, *19*(5), 1299–1334.

Simon, H. A. (1990). Invariants of human behavior. *Annual review of psychology*, *41*(1), 1–20.

Soloway, E., & Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *IEEE Trans. Softw. Eng.*, *10*(5), 595–609.

Soloway, E., Ehrlich, K., & Bonar, J. (1982). Tapping into tacit programming knowledge. In *Proceedings of the 1982 conference on human factors in computing systems* (pp. 52–57). New York, NY, USA: ACM.

Tanaka, S., Ikeda, H., Kasahara, K., Kato, R., Tsubomi, H., Sugawara, S. K., ... Watanabe, K. (2013, June 11). Larger Right Posterior Parietal Volume in Action Video Game Experts: A Behavioral and Voxel-Based Morphometry (VBM) Study. *PLoS ONE*, *8*(6), e66998+.

Vessey, I. (1987, July). On matching programmers' chunks with program structures: An empirical investigation. *Int. J. Man-Mach. Stud.*, *27*(1), 65–89.

von Mayrhauser, A., & Vans, A. M. (July 1993). From code understanding needs to reverse engineering tools capabilities. In *CASE'93* (pp. 230–239).

Widowski, D. (1987). Reading, comprehending and recalling computer programs as a function of expertise. In *Proceedings of cercle workshop on complex learning.*