2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

Program Comprehension: Past, Present, and Future

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

***Abstract*—Program comprehension is the main activity of the software developers. Although there has been substantial research to support the programmer, the high amount of time developers need to understand source code remained constant over thirty years. Beside more complex software, what might be the reason? In this paper, I explore the past of program-comprehension research, discuss the current state, and outline what future research on program comprehension might bring.**

I. Introduction

Program comprehension is an important cognitive process in software development, because developers spend most of their time with understanding source code [36], [38], [40]. Despite more than 30 years of research in software engineering, this amount still has not decreased notably. Of course, software is getting more and more complex, and there have been advancements, such as integrated development environments (IDEs), source-code visualizations, and new programming languages, which have made the job for developers more comfortable. However, the understanding of how and why these advancements help developers is rather limited.

One reason for our limited understanding of the programmer is that in the mid-90s, research ceased to make progress for over a decade. Thus, many of the approaches to support the developer are not properly evaluated regarding their effect on developer activity. This has led to a feature creep software-engineering research, meaning that new tools and extensions are developed almost on a monthly basis, but they are only rarely evaluated sufficiently. For example, together with colleagues, I developed the tool View Infinity to support program comprehension for feature-oriented software development [37]. However, our evaluation was based on seven developers, who solved several tasks in 30 minutes. This is hardly a sufficient evaluation, but falls more in the category of "I showed it my friends and they liked it". Thus, we cannot know whether and how View Infinity supports program comprehension in typical every-day tasks. Another problem is that tools and extensions often support only a small part of developers' every-day task. For example, the main purpose of View Infinity was to help developers get an overview of a feature-oriented software system, which is only a fraction of developers' every-day tasks. As a consequence of the feature creep, developers are overwhelmed by the availability of support and often only use their preferred IDE in conjunction with external tools [23].

To provide optimal support for the developers' tasks, and ultimately support developers in writing better software, we need to understand how they complete their tasks. Thus, the programmer needs to get more focus again in software-engineering research, not only the approaches that support the programmer. In this paper, I highlight past successful theory-driven research on program comprehension and efforts to support the programmer, discuss the current state and its problems, and outline where future research might be directed to.

II. Past

In the past, there has been extensive ground research on program comprehension, leading to different models explaining it. In this section, I concentrate on how program comprehension was measured and the models that emerged based on that research. Furthermore, I briefly highlight programming languages and tools that were developed to support the developer.

A. Measuring Program Comprehension

Research on program comprehension started more than 30 years ago [2], [24]. During that time, numerous studies have been conducted to investigate how developers understand source code. Most of these studies used rather small programs, compared to the size of typical programs of today. To measure program comprehension, researchers used different approaches: think-aloud protocols, memorization, and comprehension tasks.

1) Think-Aloud Protocols: Think-aloud protocols, or introspection, have been used to observe cognitive processes for more than 140 years [41]. In such studies, participants verbalize their thoughts, which are typically audio- or video-taped. These tapes are transcribed, and the transcriptions are used to analyze the thought process of the participants. Back then, the think-aloud protocol was a common technique to observe the comprehension process. As an example, Shaft and Vessey gave programmers the task to understand a program while thinking aloud, which gave an impression of how developers proceed when working with source code of a familiar and unfamiliar domain [26]. As a result, the authors found that developers state hypotheses when they are familiar with a program's domain and inferences when they are not familiar. In another study, Mayrhauser and Vans let developers think aloud while they completed a maintenance task with a comprehension component [39]. They used a larger software system of more than 40 thousand lines of code and found that developers build different models of the source code and switch between these models. To support this process, developers need natural-language support to better complete their tasks. The drawback of using think-aloud protocols is the

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.35

effort that comes with them, as the data needs to be recorded, transcribed, and then analyzed, which is very tedious. Thus, many researchers eschew this effort.

2) Memorization: Another way to measure program comprehension was to test free recall of source code. Letting developers recall source code to measure whether they comprehended it seems weird today, however, memorization and subsequent recall of source code (or any other material) is easier if it was comprehended first [28]. Shneiderman, who has published several influential articles on program comprehension during that time, likened the ability of programmers to that of musicians, who can memorize every note of thousands of songs or symphonies: He suggested programmers obtain the same ability to memorize entire programs in exact detail [29].

In a study, Shneiderman compared two versions of a program, one in which the statements are in an executable order, and the other in which statement order was scrambled [28]. He found that the version with the executable order could be recalled better, and this also correlated with programming experience (the more experience participants have, the better they are at recalling the version with the executable statement order). Soloway and Ehrlich gave programmers small source-code snippets [35]. As a task, developers should memorize the source code and recall it verbatim. As result, the authors found that experienced programmers rely on coding conventions (e.g., variable names according to their content) and are as slow as beginning programmers when the coding conventions are violated. Pennington also used memorization to measure program comprehension and evaluated how priming affects response time [20]. Priming means that, if you see a target stimulus, you respond faster to it if you previously saw another related stimulus. Since both stimuli are stored close together in your memory, the related stimulus also activates the target stimulus, so you can react faster. Participants in the study should decide whether statements, presented sequentially, were part of the source code they studied. The response time was faster if a statement was preceded by a related statement.

3) Comprehension Tasks: In the study of Soloway and Ehrlich, another task for programmers was to fill in a left-out line to correctly complete the source code, referred to as “fill in the blanks”. Developers can only correctly complete the program if they understood it first [35]. Boysen let participants decide whether simple expressions, such as `x < 5`, were true or false, or determine the value of a variable for more complex statements, such as `if x < 5 then y = 1 else y = 2` [2]. He measured correctness and response time of participants. Since this task explicitly requires participants to understand source code, it is more direct than requiring participants to memorize code. He found that different operators lead to different response times, and that statements that are true are processed faster.

Most of the approaches of these studies to measure program comprehension seem unusual and inappropriate today, because today’s activity of developers does not seem to have much in common with these studies. Nevertheless, back then, these studies captured the comprehension process quite well and led to models of program comprehension, as discussed next.

B. Modeling Program Comprehension

Basically, program-comprehension models comprise two processes: top-down and bottom-up processes. There are top-down and bottom-up comprehension models that describe program comprehension, as well as integrated models that combine both processes.

1) Top-Down Comprehension Models: If programmers are familiar with a program’s domain, they use a top-down approach [39]. Top-down comprehension means that first, a general hypothesis about a program’s purpose is derived. This can only be done if programmers are familiar with a program’s domain, because only then can they use their knowledge of examples and properties of that domain. This way, programmers can compare the current program with other programs they know. During deriving this general-purpose hypothesis, programmers neglect details, but only concentrate on relevant aspects for building the hypothesis.

Once a hypothesis of the general purpose of a program is stated, this hypothesis is evaluated. To this end, programmers look at details and refine the hypothesis stepwisely by developing subsidiary hypotheses. These subsidiary hypotheses are refined further, until the programmers have a low-level understanding of the source code, such that they can verify, modify, or reject the hypotheses. During the refinement process, programmers look for *beacons*, which are “sets of features that typically indicate the occurrence of certain structures or operations in the code” [4]. Similar to beacons, programming plans (i.e., “program fragments that represent stereotypic action sequences in programming”) are used to evaluate hypotheses about programs [35].

2) Bottom-Up Comprehension Models: With insufficient domain knowledge, programmers cannot look for beacons or programming plans, because they do not know what they look like. In this case, programmers need to examine the code closely to be able to state hypotheses of a program’s purpose. In this case, programmers start to understand a program by examining its details first: the statements or control constructs that comprise the program. Statements that semantically belong together are grouped into higher-level abstractions, referred to as chunks. If enough chunks are created, programmers leave the statement level and integrate those chunks to further higher-level abstractions. For example, programmers examine several methods of a program. If they discover that some of these methods have a higher-level purpose, for example, sorting a list of data-base entries, they group them to chunks. Now, they do not think of these methods as single entities anymore, but as the chunk that “sorts data-base entries”. They examine the program further and discover further chunks, for example, inserting and deleting entries of a data base. Now, these chunks are integrated into a larger chunk, which the programmer refers to as “data-base-manipulating” chunk. This process continues until the programmers have a high-level hypothesis of a program’s purpose. Different bottom-up models differ on the kind of information that is integrated to chunks. Pennington states that control constructs (e.g., sequences or iterations) are used as base for chunking [20], whereas Shneiderman and Mayer say that chunking begins on the statements of a program [30].

3) Integrated Models: Simple top-down and bottom-up models often cannot describe the comprehension process sufficiently. Most of the time, developers use both processes, which is described in integrated models [39]. For example, if programmers have domain knowledge about a program, they form a hypothesis about its purpose. If they encounter fragments that they cannot explain with domain knowledge, they start to examine the program statement by statement and integrate the newly acquired knowledge into their hypotheses about the source code. Usually, programmers use top-down comprehension where possible and bottom-up comprehension only when necessary, because top-down comprehension is more efficient than examining the code statement by statement [26].

The above described models explain the comprehension process as developers often used it in the past. However, the models seem incomplete when it comes to explaining program comprehension of today: The developers' every-day tasks seem to be more complex, because programs are larger and evolve, so that after a few years, nothing of the original source code may exist anymore. But before discussing the present, the next section highlights the history of programming languages.

C. Programming Languages

The first programmable computers were programmed with machine code, that is, sequences of 1's and 0's (the presence/absence of holes in punched cards). Now imagine you were to give an instruction to your computer and had to work only with 1's and 0's—it would be very difficult to memorize what the sequence `0000 0100 1100 1011` means and not confuse it with other sequences. Thus, assembly languages have been developed, which allowed developers to use mnemonics, so that developers could work with instructions, such as `add $t1, $t2, $t3`, instead of `0000 0100 1100 1011` [25]. However, this still required low-level programming close to the hardware, that is, moving numbers around the registers of a CPU.

Thus, higher-level programming languages were developed, such as COBOL, Fortran, Java, or C#. These allow developers to more directly translate their programming problem to a programming language. Thus, instead of directly working with CPU registers, developers can simply write `if a = 1 then t1 = t2 + t3`, which better reflects the intention of what programmers want to do.

To ease programming, APIs have been developed, which allowed programmers to reuse often used features without having to implement them. For example, the Java API provides often used String operations, and the Android API allows developers to implement Apps for Android devices and interact with hardware, such as the camera. Thus, APIs provide support for specific scenarios.

Similar to APIs, domain-specific languages (DSLs) provide concepts for a specific domain. DSLs often contain a restricted set of instructions tailored to a certain domain. This reduces the expressive power of a programming language, but makes it easier to learn and write according programs, because there are not that many instructions to be learned. For example, the structured query language (SQL) contains instructions for handling typical data-base operations, such as storing and loading data. With the simple statement `select [column] from [table] where [condition]`, many common data-base queries can be expressed.

D. Programming Tools

In the early days, source code was often written with a standard editor, such as Emacs or vi. However, as the complexity of programs grew, a standard text editor was not sufficient anymore,[1] so the need for integrated development environments rose quickly. Newman describes one of the first successful IDEs, which integrates the typical tools and extensions that developers need for their every-day task, such as an overview of all data and tools for synchronizing design and implementation level [19].

In addition to IDEs, there was research on how code layout affects program comprehension [15], [22], [31]. For example, Shneiderman and McKay found that, with increasing program complexity, indentation improves program comprehension [31]. Miara and others evaluated how different indentation spaces (0, 2, 4, or 6 white spaces) affect comprehensibility. They found that indentation of 2 to 4 white spaces was most beneficial for program comprehension. [15]. Rambally evaluated how different color-coding styles affect comprehensibility [22]. Specifically, he compared how color coding control structures (e.g., scope of a loop) vs. different kinds of statements (e.g., I/O, variable declarations) affects comprehension. He found that coloring different kinds of statements lead to better comprehensibility than coloring control structures (which in turn was better than no color coding). Thus, the need to improve comprehensibility of programs was the target of early research.

To summarize, there was extensive research in the past on how to support the programmer's life. Based on this research, we know, for example, that we should use useful variable names, so that they can serve as beacons and developers can use the fast top-down comprehension process, which is supported by the results of Shneiderman [31]. We know that developers need IDE support and that code layout can significantly affect program comprehension. This research helped us to understand the programmers of the 90s. However, this research somehow ceased in the mid-90s, unlike the research on software engineering. Thus, there now is a gap between

[1] Of course, there are developers who might not agree. However, many text editors, such as vi, notepad++, and Emacs, allow developers to integrate compiler, profiler, debugger, etc., but then, vi is more like an IDE than a text editor.

our understanding of programmers and the challenges they face today. Where did this lead us to?

III. Present

Today, software is everywhere, and there are hundreds of programming languages, source-code-layout guidelines, and IDEs that have been designed to support the programmer. However, since research on program comprehension almost vanished,[2] there is now a mismatch between our insights on program comprehension and state-of-the-art software development, so that it is difficult to understand the cognitive processes of developers when they work with new a programming language, source-code layout, or IDE. If we do a study now, for example, to evaluate whether a new IDE extension really supports developers in comprehending their programs, there are so many options to consider to truly measure program comprehension and not something else, for example, the familiarity of a programmer with the IDE or a programming language. In this section, I discuss the present challenges to measure program comprehension and discuss the development of programming languages and tools.

A. Challenges of Measuring Program Comprehension

What might be a reason for why research on program comprehension, and the programmer in general, stopped in the mid-90s? Based on a recent survey among members of the program committees of the major software-engineering venues, we suspect that one reason is that empirical evaluations of program comprehension (and of the human factor, in general) are difficult to conduct and also lost their appreciation [34]. We found a huge difference in the opinion on what makes a good empirical study, so that there is a high risk of getting an empirical paper rejected several times, depending on the reviewers' opinions.

To illustrate the different points of view on a good empirical study, I present an example study of 2010 [9]. This study evaluated the effects of static and dynamic type systems on development time. Before I proceed, I ask you to think about how you would design such a study and consider the following questions: Would you use existing programming languages, such as Java and PHP, to compare the effect of type systems on development time? But what about other differences between both programming languages, could they have an effect on development time? And would you use an existing IDE, such as Eclipse? What about the features that Eclipse provides, what effects do they have on development time? What participants would you recruit? How do you make sure their experience and preferences do not bias development time? As you might suspect, there is not *the* correct answer to these questions—instead, there are different alternatives that each have benefits and drawbacks. Hanenberg chose an approach in which he controlled the influence of these confounding factors. He developed a new programming language in two variants, one with a static and one with a dynamic type system. This way, the programming language differed only in the type system. He also developed a new IDE with a very limited set of features (a class browser, a test browser for running and testing the application, and a console window to print strings). Thus, there are no additional features of the IDE with which one participant is more experienced than another. For participants, the author conducted extensive interviews to balance their experience to avoid bias on development time. Thus, Hanenberg designed a highly controlled setting in which he excluded the influence of many confounding factors (i.e., he maximized internal validity). While this is a well-designed study, of course the generalizability of results to real programming languages, IDEs, and experienced developers is very limited (i.e., has very limited external validity).

This trade-off between control and generalizability is inherent in empirical research: Researchers always face the decision how much control they want in an empirical setting vs. how general the results should be. This is even harder today, because there are so many factors to consider. In a study of papers published from 2000 and 2010 in several (empirical) software-engineering venues, we found 39 such factors to consider when conducting experiments on program comprehension, including the above discussed factors (i.e., programming language, IDE, programming experience) and factors that occur only because participants take part in an experiment, such as evaluation apprehension, motivation, or biased behavior [33]. In theory, it might be possible to have both, high control and generalizable results. However, in practice, it is impossible: Consider we want to evaluate the effect of two parameters, for example, kind of type system and programming experience, on development time. There are two kinds of type systems, static and dynamic, and two levels of programming experience (there could be more levels, but for simplicity, we use two). To test the effect of these two variables on development time, we need to test each of the four possible combinations (static/low experience, static/high experience, dynamic/low experience, and dynamic/high experience). And for each combination, we need a certain number of participants to gather meaningful data and be able to draw statistically sound conclusions. If we recruit 5 participants per combination, we would need 20 participants. What if we want to consider IDE support, as well? Assuming we use Eclipse and VisualStudio as representative IDEs, there now would be 8 combinations to consider. With 5 participants per combination, we would need 40 participants. Now, if we would consider all identified 39 parameters (with two variations per parameter), and we recruit 5 participants per combination, we would need $2^{39} \times 5 = 2.7487791 \times 10^{12}$ participants to cover all combinations. Since it is impossible to recruit this many participants, we need to make tradeoffs.

To learn how to manage such trade offs, we can look at other disciplines that also research the human factor, such as psychology or medicine. In these disciplines, it is standard to empirically evaluate theories before they are accepted or

[2] A notable exception is the International Conference on Program Comprehension, which was a workshop until 2005. However, also there, the programmer was not the focus, but approaches to support the programmer. Only in recent years, the programmer moved into focus again.

test medicine before it is approved for clinical use. There are, in essence, two differences of these disciplines to software-engineering research: First, there are guidelines and standards on how to conduct empirically sound studies. Unfortunately, there are no such standards or even guidelines in software-engineering research, which is why opinions of the key players on software engineering differ so much. Second, studies are replicated by independent researchers before results are accepted. However, in software engineering, replications are not appreciated and even seen as hunt for publication [34], and there are only very few of them: In a literature study, we found that only 8% of papers describe a replication study [34]. Thus, measurement of program comprehension, and more general the human factor in software engineering, is underappreciated and difficult, which is one of the reasons for why research regarding program comprehension stopped. Fortunately, researchers are overcoming these obstacles, so that the human is getting more attention again in recent years, which can be seen by the increasing number of studies.

For example, Röhm and others used the think-aloud method to observe professional programmers during their work [23]. As a result of this study, the authors derived several hypotheses about programmer behavior, for example, that they focus on getting a task done instead of understanding source code. We evaluated how background colors can improve comprehensibility of source code that makes heavy use of preprocessor statements [6]. As result, we found that for familiarizing with a new program, background colors can significantly support a programmer. In addition to such conventional techniques, researchers are also adopting novel measurement techniques to gain new perspectives on program comprehension, as I explain next.

B. New Approaches to Measure Program Comprehension

Researchers have discovered that with neuro-imaging techniques, new insights on the programmer could be gained. For example, we conducted a study in which we let programmers understand source code while we observed them with functional magnetic resonance imaging (fMRI) [32]. fMRI allows us to observe which brain regions get activated when participants complete defined tasks, such as understanding source code. Based on this paradigm and more than 20 years of fMRI studies (one of the first was published in 1991 [1]), brain regions are associated with different cognitive processes. Thus, we can associate the activated areas during program comprehension with cognitive processes. In a nutshell, we found several areas that are related to language processing, which is evidence that program comprehension is also a language-processing process. While this result does not seem surprising, there now is empirical evidence on the language-processing nature of program comprehension.

Other researchers have also started to use neuro-imaging techniques. For example, Nakagawa and others used near-infrared spectroscopy to measure changes in blood flow while programmers mentally executed source code. They found activation in the prefrontal cortex (a brain area that is necessary for higher-order cognitive processes), which correlated with the difficulty of a task [18]. Kluthe used electroencephalography (EEG) to measure program comprehension of participants with varying levels of expertise [12]. He let participants mentally execute the code and asked them to determine the output of source-code snippets. He found that, with lower expertise, program-comprehension tasks were more difficult to solve, indicating a higher cognitive load, which was reflected in the EEG signals. Thus, EEG could be a reliable way to measure cognitive load of programmers. In a similar way, Fritz and others used three psycho-physiological measures—eye tracker, electrodermal-activity sensor, EEG—to predict the difficulty of programming tasks [7]. Participants were required to mentally execute code that drew rectangles and decide whether rectangles overlap or determine the order in which rectangles were drawn. The authors found that these measures are promising to predict task difficulty. In addition to neuro-imaging techniques, researchers have been using eye tracking for some time. For example, Sharif and Maletic evaluated the effect of `under_score` vs. `camelCase` style on recognition time [27]. As result, they found that identifiers in `under_score` style are recognized faster than identifiers in `camelCase`.

In essence, the programmer is beginning to get into the focus of software-engineering research again, with conventional research methods as well as with new research methods.

C. Programming Languages

Today, there are a plethora of programming languages available, each with different features and focus to support programmers. Programming languages can have a static or dynamic type system, can be imperative or functional, be general-purpose or domain-specific, etc. Often, features of a programming language are claimed to support a programmer, but there are hardly evaluations of these features. For example, the study of Hanenberg evaluated the effect of static vs. dynamic type systems [9], however, under a highly controlled setting. There are no studies that generalize the results of this study, so it is not clear under which circumstances the type system has no effect on development time.

Also, new programming paradigms have emerged to support programmers, for example, feature-oriented or aspect-oriented programming. There are programming languages that implement these programming paradigms and that extend existing programming languages (e.g., AspectJ extends Java). However, they have not been evaluated sufficiently. Hanenberg and others conducted a study to evaluate how aspect-oriented programming affects the development speed of crosscutting concerns [10]. The task for participants was to add a crosscutting concern to an existing application, and the result showed positive as well as negative effects, depending on different types of tasks. But again, the generalizability of this result is unclear.

To summarize, new programming languages, paradigms, and extensions emerge frequently, but they are often not evaluated sufficiently, so that their effect on program comprehension is not clear. For example, up to this day, there is a heated discussion on how to teach programming, that is, whether starting with a functional programming language or starting with an object-oriented language is more beneficial. However, there is only anecdotal evidence. In a nutshell, there are so many programming languages, paradigms, and extensions that no one can possibly know which the optimal programming language is for the current task or state of mind.

D. *Programming Tools*

The same as for programming languages counts also for programming tools: There are so many tools to support the programmer, and for each tool, there are numerous extensions, which often focus on one aspect of supporting the programmer. For example, Whyline supports developers during debugging and lets them ask "Why did/didn't" questions (e.g., "Why did the window not resize?"), which is more natural to a programmer's thinking and can make debugging faster [13]. Code Bubbles helps developers to define working sets for defined tasks, which are displayed as bubbles, so that they can concentrate on the relevant files only [3]. There are also approaches to let developers adjust their IDE to their current situation, such as moldable development tools, which let developers write their own source code with a domain-specific language to provide different views on programming objects [5]. However, it is unclear whether and how these tools help in practice and to what extent. There are studies, but they are often limited and they are hardly ever replicated. Furthermore, Röhm and others found that developers often use additional tools to their IDE, although the IDE has similar features, of which developers are often not even aware [23]. Also, developers hardly use advanced comprehension support, such as source-code visualization.

Thus, there is effort to improve the programmer's life, which, however, lacks sufficient empirical evaluation. As a consequence, the feature creep is going around in software-engineering research. Currently, there is a gap of our knowledge on how programmers complete their every-day task and the current state of the art in software engineering. This gap might well be one of the reasons why software projects are late or fail completely and critical software errors, such as the Heartbleed bug or Android's Stagefright exploit, occur so frequently.

IV. Future

What might the future hold for program-comprehension research and, more general, the programmer's life?

A. *Near future*

In the near future, there will be more and more sound empirical studies. Already in recent years, the programmer and, more general, the human factor in software engineering, is getting attention again: In 2005, Sjøberg and others found that only 1.9 % of papers conducted an empirical study (with or without human participants), whereas we found in 2015 almost every paper included an empirical study, and more than 20 % consider the human factor [34]. Furthermore, the education of researchers regarding empirical studies will improve. Several universities already offer courses especially for empirical evaluations in software engineering. This way, the expertise regarding empirical research in program committees will also improve, as new researchers with these skills will be invited to program committees. Based on their expertise, they can value sound empirical studies without being (mis-)guided by their preferences. Thus, the risk of getting a sound empirical study rejected is reduced. Furthermore, this will also have a positive effect on the number of empirical studies.

With increasing frequency of empirical studies of program comprehension, the old models of program comprehension will be extended to capture current aspects of program comprehension. For example, memorization as Shneiderman described it to measure program comprehension might not be part of the models. Today's programs are so large that developers cannot reasonably memorize them. Of course, memory still plays a role, but in the sense that developers use their domain knowledge to understand a program with a top-down approach. The understanding of program comprehension and according models to explain it will evolve to explain program comprehension of today and tomorrow. For example, Fritz and Murphy interviewed professional developers and found that they also ask questions about who is working on what (e.g., "What are colleagues working on right now?", "How much work have people done?"), code changes, broken builds, or test cases [8]. LaToza and Myers found that developers often ask reachability questions (e.g., "Why is calling method m necessary?") [14]. This new understanding of a programmer's life will be abstracted and combined to a contemporary model of program comprehension, which is not focused only on the source-code level anymore. Program-comprehension tasks of today's developers also include:

- Getting an overview of a large program or software architecture
- Understanding type structures and call hierarchies
- Understanding the relationship between components
- Identifying the developers who are responsible for a component

For example, to fix a bug, developers start to get an overview of a system, identify components and their relationships. Then, developers get closer to the source-code level and understand which method calls which method and which classes inherit from which other classes. If, during that process, developers get stuck, they seek help and identify according developers and communicate with them. Based on an updated program-comprehension model, research on program comprehension can also be theory-driven again, so that we have a clearer path to follow. For example, we can help developers to identify developers who can help them by extracting networks of developers and highlight key developers [11] or to identify developers of self-admitted code hacks [21].

B. Far future

With the improving quality of evaluation and new measurement techniques, there are whole new options to explore and support the programmer. For the far future, I envision a situation-dependent cognitive model of a single developer, which observes the current mental state of a developer and gives him or her the optimal support for the current situation. For example, the work by Fritz and others based on psychophysiological measures is a starting point in that direction, since these measures could predict task difficulty [7]. In the EEG study by Kluthe, the EEG signal was a predictor for cognitive load [12]. These devices will get cheaper in the near future, and also wearing them gets more comfortable than it currently is. Thus, developers can be observed during their every-day task. If it becomes clear that their cognitive load increases beyond its capacity, then IDE support should step in to support developers. For example, we know that the average working-memory capacity is 7 ± 2 items [16]. If developers have to keep in mind the value of too many variables at the same time, this is reflected in a higher cognitive load. In this case, a situation-aware IDE, which is coupled to the sensor data, can detect the increasing cognitive load and identify relevant variables based on eye-tracking data. Then, the IDE can automatically store the variable values and display them to the developer. Of course, with such a situation, there is the risk that developers feel monitored and patronized by their IDE. Thus, the IDE should be customizable to developers' preferences. Additionally, the sensor information could be used to assess the commitment and opinion of developers regarding their work. Hence, privacy issues may arise and become in conflict with the use of such sensors, such that developers are reluctant to use them. Similar debates are already occurring today, such as in customized adds depending on our browsing histories. Currently, we need to explicitly deactivate such data collection, and it is interesting to see whether this will also happen with sensor data. Another issue is that we need approaches to manage this big-data problem, such as machine-learning techniques to identify an impending overload of cognitive resources.

Going one step further, brain-computer interfaces might also become an interesting option to support programmers. Brain-computer interfaces help patients with locked-in syndrome to communicate with their outside world. To this end, an electrode is placed in the motor cortex (i.e., a brain area that is associated with intentional body movement), and by imagining moving their hand, patients can control the cursor of a mouse [17]. It might be interesting to use such brain-computer interfaces to let developers directly input source code, without the detour of using the keyboard (which includes recalling the letters of words and translating the letters to movements of the fingers). However, with such devices being rather invasive, it might be a far away before they will be considered for use by healthy humans.

In addition to better support for professional developers, I envision that educating new developers will be improved by individualized teaching methods. Learning programming is full of obstacles, so that many students do not proceed. With a dynamic cognitive model of program comprehension, we can recognize what students are stumbling upon in a situation, and a dynamic IDE can provide situation-specific support. Thus, learning to program will not be as difficult as it is today.

To summarize, I envision optimal, individualized support for developers, so that they can concentrate on the current task without being obstructed by the limitations of their mind. This way, developers can better use their cognitive resources and concentrate on the task at hand. In the long run, this will help to increase the quality of software and avoid critical errors.

V. Conclusion

Program comprehension has received a lot of attention in the past, but has lost the interest of the software-engineering researchers in the mid-90s. Fortunately, research on program comprehension, and the human factor in software engineering in general, is getting more attention again. Past research gives a good baseline for our current understanding of program comprehension, but needs to be extended to accommodate state-of-the-art software development. Conventional empirical methods as well as new techniques, which have proved successful in cognitive neuroscience, help us to gain a new perspective on the programmer's life. With these methods combined, we can provide optimal, situation-specific support for individual developers, so that they can concentrate on the task at hand. This way, software projects can be released on time and without critical errors.

Acknowledgements

I would like to thank Norbert Siegmund and Sven Apel for fruitful discussions and helpful comments on this paper.

References

[1] J. Belliveau, D. Kennedy, R. McKinstry, B. Buchbinder, R. Weisskoff, M. Cohen, M. Vevea, T. Brady, and B. Rosen. Functional Mapping of the Human Visual Cortex by Magnetic Resonance Imaging. *Science*, 254(5032):716–719, 1991.

[2] J. Boysen. *Factors Affecting Computer Program Comprehension*. PhD thesis, Iowa State University, 1977.

[3] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code bubbles: A working set-based interface for code understanding and maintenance. In *Proc. Conf. Human Factors in Computing Systems (CHI)*, pages 2503–2512. ACM Press, 2010.

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

[5] A. Chiş, T. Gîrba, and O. Nierstrasz. Towards Moldable Development Tools. In *Proc. Int'l Evaluation and Usability of Programming Languages and Tools (PLATEAU)*. ACM Press, 2015. To appear.

[6] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Softw. Eng.*, 18(4):699–745, 2013.

[7] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger. Using Psycho-physiological Measures to Assess Task Difficulty in Software Development. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 402–413. ACM Press, 2014.

[8] T. Fritz and G. Murphy. Using Information Fragments to Answer the Questions Developers Ask. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 175–184. ACM Press, 2010.

[9] S. Hanenberg. An Experiment about Static and Dynamic Type Systems: Doubts about the Positive Impact of Static Type Systems on Development Time. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 22–35. ACM Press, 2010.

[10] S. Hanenberg, S. Kleinschmager, and M. Josupeit-Walter. Does Aspect-Oriented Programming Increase the Development Speed for Crosscutting Code? An Empirical Study. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 156–167. IEEE CS, 2009.

[11] M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle. From Developer Networks to Verified Communities: A Fine-Grained Approach. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 563–573. IEEE CS, 2015.

[12] T. Kluthe. A Measurement of Programming Language Comprehension Using p-BCI: An Empirical Study on Phasic Changes in Alpha and Theta Brain Waves. Master's thesis, Southern Illinois University Edwardsville, 2014.

[13] A. Ko and B. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proc. Conf. Human Factors in Computing Systems (CHI)*, pages 151–158. ACM Press, 2004.

[14] T. LaToza and B. Myers. Developers Ask Reachability Questions. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 185–194. ACM Press, 2010.

[15] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman. Program Indentation and Comprehensibility. *Commun. ACM*, 26(11):861–867, 1983.

[16] G. Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review*, 63(2):81–97, 1956.

[17] M. Moore and P. Kennedy. Human Factors Issues in the Neural Signals Direct Brain-computer Interfaces. In *Proc. Int'l. Conf. on Assistive Technologies*, pages 114–120. ACM Press, 2000.

[18] T. Nakagawa, Y. Kamei, H. Uwano, A. Monden, K. Matsumoto, and D. M. German. Quantifying Programmers' Mental Workload During Program Comprehension Based on Cerebral Blood Flow Measurement: A Controlled Experiment. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 448–451. ACM Press, 2014.

[19] P. Newman. Towards an Integrated Development Environment. *IBM Systems Journal*, 21(1):81–107, 1982.

[20] N. Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychologys*, 19(3):295–341, 1987.

[21] A. Potdar and E. Shihab. An Exploratory Study on Self-Admitted Technical Debt. In *Proc. Int'l Conf. Software Maintenance and Evolution (ICSME)*, pages 91–100. IEEE CS, 2014.

[22] G. Rambally. The Influence of Color on Program Readability and Comprehensibility. In *Proc. Technical Symposium on Computer Science Education (SIGCSE)*, pages 173–181. ACM Press, 1986.

[23] T. Röhm, R. Tiarks, R. Koschke, and W. Maalej. How Do Professional Developers Comprehend Software? In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 255–265. IEEE CS, 2012.

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

[25] D. Salomon. *Assemblers and Loaders*. Ellis Horwood, 1992.

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

[27] B. Sharif and J. Maletic. An Eye Tracking Study on camelCase and under_score Identifier Styles. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 196–205. IEEE CS, 2010.

[28] B. Shneiderman. Exploratory Experiments in Programmer Behavior. *International Journal of Computer & Information Sciences*, 5(2):123–143, 1976.

[29] B. Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems (Winthrop Computer Systems Series)*. Winthrop Publishers, 1980.

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

[31] B. Shneiderman and D. McKay. Experimental Investigations of Computer Program Debugging and Modification. *Proc. Human Factors and Ergonomics Society Annual Meeting*, 20(24):557–563, 1976.

[32] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann. Understanding Understanding Source Code with Functional Magnetic Resonance Imaging. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 378–389. ACM Press, 2014.

[33] J. Siegmund and J. Schumann. Confounding Parameters on Program Comprehension: A Literature Survey. *Empirical Softw. Eng.*, 20:1159–1192, 2015.

[34] J. Siegmund, N. Siegmund, and S. Apel. Views on Internal and External Validity in Empirical Software Engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 9–19. IEEE CS, 2015.

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

[36] T. Standish. An Essay on Software Reuse. *IEEE Trans. Softw. Eng.*, SE–10(5):494–497, 1984.

[37] M. Stengel, J. Feigenspan, M. Frisch, C. Kästner, S. Apel, and R. Dachselt. View Infinity: A Zoomable Interface for Feature-Oriented Software Development. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 1031–1033. ACM Press, 2011.

[38] R. Tiarks. What Programmers Really Do: An Observational Study. *Softwaretechnik-Trends*, 31(2):36–37, 2011.

[39] A. von Mayrhauser and M. Vans. From Program Comprehension to Tool Requirements for an Industrial Environment. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, pages 78–86. IEEE CS, 1993.

[40] A. von Mayrhauser, M. Vans, and A. Howe. Program Understanding Behaviour during Enhancement of Large-scale Software. *J. Software Maintenance: Research and Practice*, 9(5):299–327, 1997.

[41] W. Wundt. *Grundzüge der Physiologischen Psychologie*. Engelmann, 1874.