Empirical Software Engineering (2021) 26: 48
https://doi.org/10.1007/s10664-021-09938-8

Understanding large-scale software systems – structure and flows

Omer Levy[1] · Dror G. Feitelson[1]

Accepted: 7 January 2021 / Published online: 31 March 2021

Abstract

Program comprehension accounts for a large portion of software development costs and effort. The academic literature contains mainly research on program comprehension of short code snippets, but comprehension at the system level is no less important. We claim that comprehending a software system is a distinct activity that differs from code comprehension. We interviewed experienced developers, architects, and managers in the software industry and open-source community, to uncover the meaning of program comprehension at the system level; later we conducted a survey to verify the findings. The interviews demonstrate, among other things, that system comprehension is largely detached from code and programming language, and includes scope that is not captured in the code. It focuses on one hand on the structure of the system, and on the other hand on the flows in the system, but less on the code itself. System comprehension is a continuous, unending, iterative process, which utilizes white-box and black-box approaches at different layers of the system depending on needs, and combines both bottom-up and top-down comprehension strategies. In summary, comprehending a system is not just comprehending the code at a larger scale, and it is not possible to comprehend large systems at the same level as comprehending code.

Keywords Program comprehension · System comprehension · Code structure · Program flows

Communicated by: Federica Sarro and Foutse Khomh

This article belongs to the Topical Collection: *International Conference on Program Comprehension (ICPC)*

Dror Feitelson holds the Berthold Badler Chair in Computer Science. This research was supported by the ISRAEL SCIENCE FOUNDATION (grants no. 407/13 and 832/18). This paper is an invited extended version of a paper from ICPC 2019.

✉ Dror G. Feitelson
feit@cs.huji.ac.il

Omer Levy
lomer1978@gmail.com

[1] Department of Computer Science, The Hebrew University of Jerusalem, 91904 Jerusalem, Israel

1 Introduction

Software maintenance is responsible for the majority of the development costs of a software system (Austin and Samadzadeh 2005). There is a well-established direct relationship between the ability to comprehend the software and the cost of software maintenance (Banker et al. 1993). Program comprehension is therefore a key element in the software development life cycle. Indeed, many tools and practices that were developed in the software development world over the last few decades—from modern programming languages, through UML architecture diagrams, and on to design patterns and coding practices—are all targeted also at improving the comprehensibility and maintainability of software, and thus reducing maintenance costs (Arisholm et al. 2006; Tichy 2011).

Research in program comprehension focuses on how developers understand code, by analyzing the impact of code elements on comprehension (Jaffe et al. 2018; Jbara and Feitelson 2014; Ajami et al. 2019; Medeirios et al. 2019), suggesting tools and methodologies to improve code comprehension (Kosar et al. 2018; Feng et al. 2018), and analyzing the cognitive models that are employed in code comprehension (Brooks 1983; Letovsky 1987; Siegmund et al. 2012). Such studies typically employ short code segments. When considering comprehension of a single function with an average complexity, it is expected that an expert may be able to evaluate every possible code path, understand the meaning of each variable, and successfully predict the output of the function given a particular input.

Yet due to the volume of code in a large software system, it is not possible for one person to understand the entire system in the same way one understands a single function. The key to understanding a function is understanding the programming language's syntax and the semantics created by the developer. But the key to understanding larger volumes of code is understanding abstractions and concepts. This requires understanding that is not necessarily reflected in the code itself. Von Mayrhauser and Vans write in this context about "understanding software, rather than merely programs (code)" (von Mayrhauser and Vans 1996).

Large complex software systems are planned, developed, and maintained regularly in the software industry. However, it is not clear how software systems are comprehended based on the existing software comprehension literature. Do the developers of these systems really understand the entire system? To what extent do they understand it? What aids do they use to better understand the system? More generally, what is the meaning of the term "program comprehension" in this context, and how does it differ from the comprehension of smaller segments of code?

We try to answer these questions through a series of in-depth interviews with experienced developers. We asked about the different levels of comprehension at different granularities of code, the comprehension strategies that are required for different tasks, the role of documentation in comprehension, and the special skills required for system comprehension. Some of our main findings are:

- System understanding is largely detached from code and programming language, and includes scope that is not captured in the code. Understanding a system is not just understanding more code.
- Understanding at a larger scope shifts focus from the code to its structure (architecture): the components, their connections, data flow, and also the considerations that led to this design.
- There are different levels of comprehension. In particular, there is a significant difference between black-box and white-box comprehension.

- Understanding a software system is a lengthy and iterative process, which includes a mix of several levels of comprehension and several comprehension strategies.
- There is a mutual interplay between software comprehensibility and quality. Comprehensibility is an element of software quality. Conversely, quality improves comprehensibility and may alleviate the need to understand.
- Understanding a system requires a different skill-set compared to the skills required for programming.

This paper is an extended version of a paper from the 27th *International Conference on Program Comprehension* (Levy and Feitelson 2019). The original paper focused on the hierarchical structure of systems. This version extends the discussion to include flows as well, adds a survey of experienced developers to validate the results, and also adds multiple examples and observations from a repeated analysis of the interviews.

1.1 Software System Model

Throughout the paper we discuss a hierarchical model of a software system that includes the following layers:

- **Function:** as common in most modern programming languages (such as C/C++/Java).
- **Class:** as common in most modern object oriented languages (such as C++/Java). We use a simplified model where a class consists of a collection of functions (methods) and common state variables (data members), with a differentiation between public methods that can be used from outside the class and private methods which are internal. This simplified definition, which does not require inheritance, includes many non-object-oriented programming languages with similar constructs (such as modules in C modular programming).
- **Package:** we use the term “package” to loosely refer to a collection of classes that provide a cohesive functionality. Some of the research participants objected to this term and proposed other terms or definitions. The term is meant to provide an additional hierarchical level between a class and a full system. Packages are the basic unit of reuse in that they are self-contained.
- **Subsystem:** a part of the system that is an independent deployable unit, when parts are deployed in different places. For example, a distributed system may have a client subsystem and a server subsystem; an embedded system may have multiple controllers communicating with one-another, where each controller is a separate subsystem.
- **System:** we use the term “system” to refer to an entire software system. The exact definition of a system is debatable and its boundaries may also be understood in different ways.

In parallel to this hierarchy of layers that describes the system structure, we also consider a description of the system through its flows—that is, the way the system control shifts between different parts of the code over time. Importantly, this description of the system through flows directly relates to describing the system via use cases. Each flow starts with an entry point to the system.

The two different views of the system are related in that flows can be described among subsystems at the system level, among packages in the subsystem or system level, among classes at the package level, among functions in the class level, and even between basic blocks within a single function (see Fig. 1).

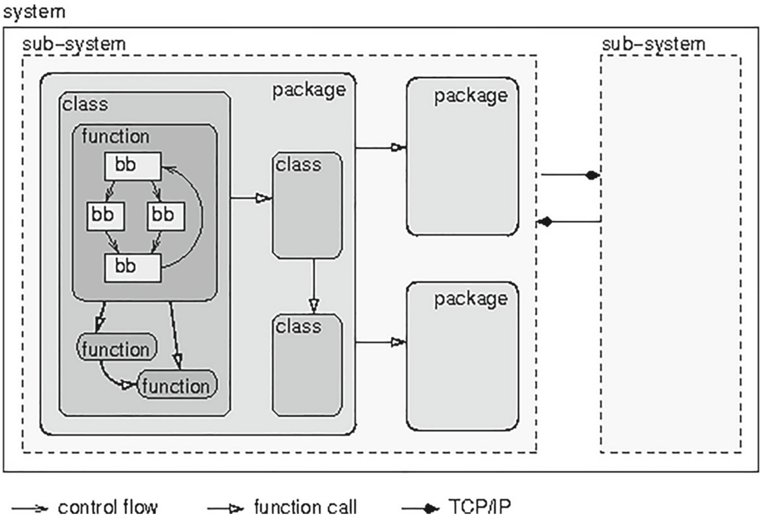


Fig. 1 Flows at different levels of a system's structure hierarchy. bb: basic block

In many cases a flow that is described at the system level can be "zoomed into" to be viewed at the subsystem, package, or class level. The passing of control between packages is accomplished by a function belonging to a certain class in one package calling a function belonging to some class in another package—most probably a function that is part of the package's API. But flows between subsystems are different, as they typically involve the definition of a communications protocol. Likewise, control flow within a function in also different.

Interestingly, several common complexity measures are tightly related to this type of analysis of a system through flows. For example, the McCabe Cyclomatic Complexity metric (MCC) (McCabe 1976) characterizes the paths of control flow within a function. The Fan-in/Fan-out metric (Henry and Kafura 1981) characterizes the call graph between functions. This suggests that flows embody complexity, and one must understand the flows in order to understand the system.

1.2 Related Work

Controlled experiments have been used to study code complexity and comprehension for a long time. In early work Weissman identified multiple possible factors affecting comprehension, including aspects of readability, control flow, and data flow, and developed the experimental methodology to study them (Weissman 1974). Several recent works measured code comprehension in quantitative experiments using short code snippets. Ajami et al. (2019) measured the time it took experimental subjects to answer questions related to code snippets, and showed, among other results, that loops are harder to understand than conditional statements. Other studies performed controlled experiments to measure the impact that trace visualization (Cornelissen et al. 2011), reactive programming (Salvaneschi et al. 2017), the naming of variables (Beniamini et al. 2017; Avidan and Feitelson 2017), or UML object diagrams (Torchiano et al. 2017) have on program comprehension.

In a related vein, Störrle (2014) performed a controlled experiment on UML comprehension, and derived guidelines for UML diagram layout and diagram size. He shows a negative correlation between the experiment score and the diagram size. This may be related to the complexity of the architecture conveyed by the UML diagram, implying that larger more complex systems are harder to understand.

There have been several works focused on understanding large-scale software. Petersen et al. performed a case survey on selection of components to integrate into a system (2018). This topic is a particular facet of system comprehension—it requires understanding an unknown software component and how it relates to the system requirements. In 9 of the 22 cases, the final decision was perceived negatively. This may point to the difficulty in system comprehension and the decisions in that area. Kulkarni and Varma also discuss problems with package reuse practices and the importance of structured decisions (2017).

Feng et al. present a tool they call SAGE to aid developers in comprehending code (2018). This tool analyzes execution traces to identify frequent behaviors at different levels of granularity, and label them with comprehensible names. Note, however, that it assumes that a high-level understanding of the functionality of the system is known apriori. It's goal is to map this understanding to the code.

Brunner and Porkoláb, on the other hand, note that understanding legacy software is especially hard. They suggest to incorporate information from the source control system in addition to looking at the code (Brunner and Porkoláb 2019). This can help identify diverse code elements that were developed together, and therefore are parts of the implementation of a specific feature.

Kozaczynski et al. take a systematic approach to design a tool to aid in software systems comprehension (1991). They start by mapping the questions developers need to answer, and design the tool to have capabilities needed to answer each one of them. This is based on parsing and analyzing the code base to create a system model, which is stored in an accessible repository.

Alomari et al. suggest that visualizations can be important for understanding large systems (2016). They propose a tool to visualize slices, for example using bipartite graphs to visualize the definition-use coupling due to different variables. This is expected to aid in impact analysis when contemplating a change to the code. Similarly, Moonen and Yazdanshenas developed a tool to visualize the flows between the constituents of a component-based system (2016). Both these works testify to the importance of flows for understanding.

Maletic et al., in contradistinction, suggest a tool they claim is scalable for visualizing the structure of the code (2011). Wettel and Lanza, in a series of papers, suggest the city metaphor to visualize the structure of a software system (2007a, b). In this visualization classes are represented by buildings, where the dimensions of the building reflect code metrics such as the number of methods or data members. Packages are represented by city blocks. This allows for an overview of the system structure, and draws attention to special cases: interface classes with numerous methods stand out as skyscrapers, whereas parsers that set numerous variable look like parking lots. But there is no indication of what the system does or how. Panas et al. also use a city metaphor, but at a finer granularity (buildings are functions) with additional attributes shown, which can become somewhat cluttered (2007). Maletic et al. have analyzed software visualizations with a focus on their use for understanding (2002).

The need for tools that aid comprehension is a result of the difficulty to understand large code bases. Hwa et al. consider the hierarchical structure of object-oriented systems (modules, classes, and the inclusion and referencing relationships between them) (2009). They postulate a set of metrics for system understandability based on accepted concepts like size, encapsulation, coupling, and cohesion. However, they do not address the basic issue of what does it mean to understand a system.

Zhang et al. suggest that the complexity of a system's structure can be quantified based on the networking concept of k-cores (2010). This also defines a hierarchy on the system's components, but in this case the hierarchy is based on how central a component is. They show that diverse systems may have common characteristics in terms of their structure, but do not make explicit connections between these characteristics and the difficulty of understanding the systems.

A possible approach to alleviate the need to understand code is to provide code summaries. Several tools that generate such summaries automatically have been proposed (Alon et al. 2019; Haiduc et al. 2010; Rodeghero et al. 2015). Using eye tracking to see where developers focus during the execution of a summarization task helps identify what they find the most important elements of the code in terms of comprehension.

Turning to case studies of understanding real systems, Xia et al. (2018) measured the different computer-mediated activities related to program comprehension, finding they account for 58% of the time in real-life environments. They also show that senior developers spend less time on program comprehension than juniors.

Kulkarni undertook the comprehension of a 500,000 lines software system for the purpose of reuse. He used a mix of top-down and bottom-up approaches to eventually locate about 25,000 lines of code that were critical to understanding the entire system (Kulkarni 2016). Other works also provided methods for identifying the most important parts of a large software system as a means for system comprehension (Salah et al. 2006; Şora 2015).

The academic discussion of cognitive models of software comprehension has a long history, mostly circling around two well-known main models mentioned above: the "top-down" model (Brooks 1983) and the "bottom-up" model (Letovsky 1987). Several combinations or variations of these models have also been proposed (Storey 2006; von Mayrhauser and Vans 1995; Fekete and Porkoláb 2020).

The most significant body of work on understanding software systems is by von Mayrhauser and Vans from the 1990s (1994a, b, 1995, 1996, 1997, 1998). Importantly, their work is based on analyzing maintenance sessions of professionals working on large-scale full systems. For example, in von Mayrhauser and Vans (1996) they report in detail on the protocol analysis of a 2-hour session comprising a maintenance task to port client programs to a new operating system. In von Mayrhauser et al. (1997) they report on two 2-hour program enhancement sessions by two different developers in the context of different projects. One of them was debugging the new code he had added to the system, while the other was trying to identify the correct location to add his code.

The gist of von Mayrhauser and Vans work was to suggest and demonstrate the integrated model of program comprehension, which includes top-down and bottom-up components (von Mayrhauser and Vans 1994a; 1995). Their findings indicate that "understanding is built at all levels of abstraction simultaneously rather than level by level. This necessitates frequent switches between code, design, and application domain knowledge during the cognition process" (von Mayrhauser and Vans 1994a). However, our work indicates that actual practice is nearly entirely focused on the code. Design documents often do not exist, and even when they do exist, they may be considered to be outdated and untrustworthy. The underlying difference between our results and those of von Mayrhauser and Vans is that they postulate a continuum from understanding short code snippets to full systems. We favor a view in which system comprehension is inherently different from code comprehension.

We have not found any focused discussion of what comprehension actually means at the system level, and possible differences between understanding code and understanding a system. Kruchten's 4+1 model of architecture is a way to achieve a comprehensive description of a system (1995). It includes the system's structure and behavior, and also its logic and its layout, all unified by how they serve the system's use cases. This may be used as a guideline to what should be understood to claim a full understanding of the system. Razavizadeh et al. (2009) also suggest aiding comprehension by providing multiple viewpoints to the architecture. However, it seems that developers tend to focus exclusively on the structure, and this is what is typically meant when discussing program comprehension.

2 Methodology

The concept of comprehension is difficult to measure. Several methods for measuring different possible aspects of comprehension exist, and have been used in controlled experiments on small-scale code comprehension. However, the meaning of "comprehension" at the software system level is not well-defined. Clarifying this is the main objective of our work. Given that this question concerns human understanding, and since some of the basic terminology required to describe system comprehension is missing from software engineering research, we take an initial approach of exploratory qualitative research. Such an approach is targeted to establish basic models, and seed future empirical and more quantitative research efforts.

After defining our own perceived model of system comprehension, we constructed a semi-structured one-on-one interview plan. This was intended to uncover the participants' existing thoughts on system comprehension, while allowing further discussion beyond the interview plan in case the participant raised thoughts and ideas that we found to be important to system-level comprehension.

Since we are interested in software system comprehension as it is commonly practiced in the field, we performed interviews with 11 experienced developers, managers, architects, and entrepreneurs from different companies with different company profiles. This is usually called "expert interviews" in the social sciences literature, and is considered an efficient way to gather data in exploratory research (Bogner and Menz 2009). A potential problem with expert interviews is a possible imbalance between the interviewees and the interviewer. This may lead to reduced cooperation and inefficiency. However, in our case the interviewer was also an experienced software professional, and as a result the interviewees appear to have considered the interviews as a discussion among peers.

The interviews, each about an hour long, were recorded. They were then transcribed by an office services company. Finally, the text was corrected by the interviewer in cases the transcribers didn't understand technical jargon.

We followed common basic procedures of text analysis. Both authors carefully read each interview transcript, separately. Each one of us highlighted any quote that was related to system comprehension, specific activities used for comprehension, ideas about measuring and proving comprehension, or connections between different aspects of comprehension. This method of duplicate, separate analysis is called "analyst triangulation", and is used in order to increase research validity (Carter et al. 2014). Inspired by the grounded theory approach (Glaser and Strauss 1967), we searched for common themes that arose in several places in a single interview or in multiple interviews, including contradictions between those instances. We also looked for thoughts that supported or contradicted our initial perceived model. Finally, we compared our notes and discussed them at length in order to arrive at a joint analysis of the texts. We ended up with a refined model for system comprehension, which was somewhat different from the model we started with.

It should be noted that unlike common text analysis methodology, we are not necessarily seeking common ground such as ideas that are widely agreed upon. We found that outliers should not be discarded, quite the contrary. Experienced developers develop their own models of system comprehension, but rarely discuss them. In some cases, a participant used wording that shed light on ideas that were latent in our original model and in other interviews. Thus ideas that are expressed by a single individual sometimes inspire a modification in the model and allow us to view things differently.

For this extended version of the paper we performed a second round of analysis of the interviews. This second analysis revealed that some aspects of system comprehension were missing or not emphasized enough in the original model. Examples include the identification of the subsystem structural level, and the flows comprising a different viewpoint on the system. In the following, these new findings are integrated with the original ones.

The interviews were conducted in Hebrew. The citations from the interviews quoted in the following are translations of the original quotes into English, with minimal rephrasing for readability and flow. Note that these quotations were selected to illustrate the results and the utterances from which they were derived, but they are not a comprehensive list of all supporting utterances. The conclusions derive from the entire analysis process, not only from these quotations.

2.1 The Interview Plan

We used a semi-structured interview plan designed to take about an hour. It included the following discussion topics:

- Brief introduction.
- The participant's professional experience and programming-related history.
- Introduction of the hierarchical view of a software system (function, class, package, and system) as defined in Section 1.1. We solicited feedback on this definition.
- Definition of "comprehension" in the context of the various layers—what does it mean to understand a function? What does it mean to understand a class? etc.
- The various levels of comprehension required for different programming tasks in the different software layers—can you use a function/class/package without understanding it? What level of understanding is required to use a function/class/package? What level of understanding is required to debug or maintain a function/class/package?
- Inter-dependencies between the comprehension of different layers: does one need class level understanding in order to understand a method in the class?
- Comprehension process for the different layers—how do you go about understanding a function/class? What is the first thing you look at when you evaluate a package?
- Proof of comprehension in the different layers: how do you test one's comprehension of a function/class/package/system? What methods from the research literature resonate with actual ways of testing comprehension?
- Importance of various aids for comprehension in the different layers (different types of documentation, source code, usage samples, etc.).
- Roles in the company that are related to software comprehension: who are the people that have a system level comprehension? What is their percentage in the development group? What are their roles?

- Definition of system architecture—how would you define system architecture? We provided some general statements on architecture and solicited feedback.
- The development process of system architecture—is this an individual effort or a group effort? Are there documents or any other forms to communicate the architecture?
- Importance of system architecture understanding and its relation to system comprehension.

The full plan is available online at http://tiny.cc/interview-plan.

2.2 The Participants

We interviewed 11 experienced participants from different companies and roles:

- *[AK]*, a developer and architect with 20 years of experience, mostly in C in embedded programming and system applications; works for an international corporation of 10,000+ employees (company A).
- *[MN]*, a developer and architect with 20 years of experience, mostly in C in embedded programming and system applications, also experienced in C++ and Java; works for company A.
- *[GA]*, a software team manager with 23 years experience, including programming experience in C and C++ and managerial experience with embedded programming and algorithms teams; works for company A.
- *[ES]*, a software team manager with 20 years experience and programming experience in C, C++, and C#, mostly in desktop and web applications; works for an international corporation of 10,000+ employees (company B).
- *[AO]*, a developer and architect with 20 years of experience, mostly in C++ and C# desktop and web applications; works for company B.
- *[SN]*, a developer with 8 years of experience, mostly in C in system applications; works for company B.
- *[YI]*, a software manager with 6 years of experience, founder and CTO of a start-up company of 100-200 employees developing mobile applications (company C).
- *[DL]*, a team leader with 13 years of experience, mostly in C, C++ and python; works for company C.
- *[BY]*, a team leader with 4 years of experience, mostly in C and objective C; works for company C.
- *[BG]*, a developer with 17 years of experience, works at a small start-up company of 10-20 employees and is also a renowned developer in the open-source community.
- *[AR]*, a university professor and researcher with 28 years of experience, and industry experience as founder and CTO of multiple start-up companies; also taught a course centered on understanding a large-scale open source system.

We opted to interview developers on the more senior end of the scale, assuming senior developers have experience with large-scale systems and a perspective on the comprehension process. The relatively low number of participants is acceptable in this type of research (Crouch and McKenzie 2006), especially given that the interviewer (the first author) had a close association with them being an architect with 20 years experience himself.

2.3 Validation Survey

As part of extending the study we set out to ratify the model using a validation survey. The survey was designed to reflect some of the findings from the interviews, and the model for software comprehension that arose from those findings. It consisted of 32 questions, where each question is a statement that reflects the system comprehension model we developed based on the original interviews.

We asked the participants to rate their agreement with these statements on a 7-point scale. The scale points were designated as strongly disagree, disagree, tend to disagree, neutral, tend to agree, agree, and strongly agree. In addition a don't know / no opinion option was given. The survey was expected to take about 10–15 minutes to complete.

Following the content questions we added some general demographic questions, such as job title, professional experience, employment country, and whether or not they participated in the original study. No additional personal information was collected.

The survey was implemented on the SurveyMonkey platform. A link to the survey was sent out to experienced professionals based on personal acquaintance, with a request to forward it to other experienced professionals as well.

56 participants responded to the survey, 8 of whom had also been interviewed in the original study. 23 of the participants self-identified as "Software Developer / Software Engineer", and the rest identified as software architects, team leaders, managers, or other. All participants had 4+ years of professional experience in the software world, with a median of 15 years; more than a third had 20+ years of experience. 81% of them were from Israel.

3 Depth of Comprehension

We now begin the presentation of our findings regarding the understanding of large-scale software systems.

3.1 Levels of Comprehension

Almost all participants indicated that there are several levels to the comprehension of a software component. The participants distinguished between at least the following two levels of comprehension:

- **Black-Box Comprehension**: this is the basic level of comprehension. At this level the subject comprehends *what* is the functionality of the given component. At the function level, this equates to comprehending the function's prototype, its arguments (inputs) and the expected output of the function given a certain input. At the class level, this comprehension level is equivalent to comprehending the class's public interface. This notion can be extended to a package or a system black-box comprehension in a similar manner.
- **White-Box Comprehension**: at this level of comprehension, the subject would comprehend *how* the component implements its functionality. They would be able to describe the flow that leads from the input to the output. In a function, this level of comprehension is equivalent to comprehending the code of the function. In a higher layer, such as in a package, this level of comprehension is equivalent to describing the classes in the package that are involved in the implementation of a particular package API, the role of each class, and how data flows among these classes.

[MN] explained this distinction as an issue of viewpoint and scope: "You draw a [UML] sequence diagram where [module name] is a single line. The fact that internally A talks with B, opens a file, [etc.] ... keeps us busy and keeps us awake at night, but actually it may

not interest someone who is building a bigger story, where [the module] just performs some task, and how it performs the internal division of the work is not interesting".

The difference between these two levels of comprehension strongly relates to the concepts of modularization and information hiding (Parnas 1972). A well-designed component that employs proper information hiding can be well-comprehended for most common uses using black-box comprehension only. White-box comprehension would be required only for maintenance or refactoring.

Some participants also noted that white-box comprehension itself has several levels. It may be possible to fix simple bugs with only a superficial level of comprehension—for example, a null pointer exception can in many cases be easily traced and fixed without really understanding the code of the function. According to *[BG]* such cases are even common. For non-trivial bugs, a deeper, more complete level of white-box comprehension is needed. But being able to extract the needed information to trace bugs is also a unique skill: "there were people in organizations I managed, that could very easily debug systems that they knew very little about" *[GA]*.

A few participants suggested additional levels of comprehension. *[ES]* pointed to a level where you would comprehend "implicit and explicit assumptions that whoever wrote the function had, that are related to some broader context". *[BG]* gave an example of a file system, where the implementation can be based on blocking or on non-blocking system calls, and it might be important to know the considerations leading to a specific choice. More generally, this may relate to required domain knowledge: "If you are writing a financial system, ... you can't say you understand the system if you don't understand the financial processes that it implements" *[MN]*. At a deeper level, Ko claims that the value proposition of the system also deeply affects the code, but is hard to articulate and is largely hidden from view. But engineers need to know what generates value and use it as a guideline in their daily decisions, else the system will fail in the marketplace (Ko 2017).

We call the level of comprehension represented by these examples the **Unboxable Comprehension**. Such comprehension cannot be reconstructed from the code itself, and requires some other source of knowledge—typically a documentation of intent, or a chat with the original code developer. In the common case where documentation is outdated, low-quality, or missing, this level of comprehension is what is lost when a developer leaves a project. "There are parts in the code that nobody knows very well today. [...] If there are bugs you can still fix them, but nobody has the understanding [...] of the philosophy of this specific module" *[BG]*. But this level of comprehension may be required in common scenarios, especially to avoid introducing new bugs when changing or adding to the code. However, its role as part of the comprehension process is often overlooked.

While the static hierarchy of the system (function, class, package, and system) is evident in the code, the flow description of the system is not easily visible in the code, and is therefore also part of the unboxable comprehension of the system. A common example of such an unboxable attribute is concurrency—the way different threads in the system interact. "We are still in a situation where in many cases you cannot guarantee thread safety by just looking at a function, and knowing whether it executes synchronously or asynchronously... so you must still have this documentation" *[YI]*. Another common example is system flows related to memory management. "Obviously it is more convenient that you allocate something and return it because you need less to handle someone else's memory... but due to memory consumption considerations which is something that is typically not a module's responsibility but a system responsibility... suddenly it could be important to define some pool that enforces that you have to pass pointers to buffers" *[YI]*. Indeed issues of concurrency and memory management are considered topics that are hard to understand in a system (see for example Cornelissen et al., 2009).

[AO] pointed to a level of comprehension concerning the external interactions of a function beyond the code itself: “There’s the comprehension of how it is built from below. The classic example in C++, is that given a class with virtual functions, how the memory layout looks. This doesn’t always sound very important. Once you try to understand really how it works, and use this understanding for optimization, memory efficiency, performance, etc., this becomes a very important understanding”. We call this level of comprehension **Out-of-the-Box Comprehension**. It is needed in rather rare conditions where the required level of optimization justifies it.

The different levels of comprehension are required for different tasks, and they drive different strategies that developers apply to comprehend unfamiliar code, depending on the task they need to perform. When one wants to use a function or a class that they did not write themselves, they would typically strive for black-box comprehension only. If the black-box is properly built (meaningful name, clear arguments, header documentation if needed), this is, in many cases, all that is needed. If not, they would defer to white-box comprehension.

If the task at hand is a maintenance task like debugging or refactoring, white-box comprehension is required. In non-trivial cases a deep, complete level of white-box comprehension is needed to actually understand the flow and how it provides the desired functionality.

Unboxable comprehension is not always available, but it is desirable when the developer needs to make significant changes in the component, refactor it, or solve more complex bugs. Finally, optimization tasks, and especially more complex optimizations, might call for comprehending the external dependencies as mentioned in the context of the out-of-the-box level of comprehension.

3.2 The Desire Not to Comprehend

While the goal of the interviews was to discuss the processes required to comprehend software, in many cases the subjects mentioned a desire *not* to comprehend. Comprehending a software component is a hard task, and developers prefer to avoid comprehending as much as possible. The whole reason for APIs is to allow use without knowing—let alone understanding—the implementation details. In addition, achieving full comprehension may just be impossible. Comprehending a complete software system, including all of its flows and corner cases, becomes impractical when the system grows beyond a certain volume of code, or when there are more than a few developers of the system.

Similar observations have been made in the past. More than 50 years ago, Sackman et al. observed that one of the benefits of the shift to on-line work (as opposed to batch processing) was the acceleration due to using a trial and error approach (1968). More than 30 years ago, Littman et al. identified the “as-needed” program comprehension strategy as an alternative to the “systematic” strategy (1987). Roehm et al. specifically state that trying to avoid comprehension is common practice in the industry (2012). Design concepts such as information hiding and modularity were created specifically to shield the developer from having to know about other software components (Parnas et al. 1985). And even within a component, one can make do with comprehending only what is needed for the task (von Mayrhauser and Vans 1998): “for certain bugs, you need to understand at least the pertinent flow” *[AK]*.

Code quality and design quality are measured, inter alia, by the ability to understand them with minimal effort: “The more understanding it is a more trivial task, I consider it a better function” *[GA]*. “A good architecture is one that you do not need to understand the architecture to do things, and the system drives me to the right thing that I need to do" *[BG]*. Understandability is effectively part of the definition for quality.

The subjects also mentioned the futility of trying to comprehend given the rate of changes: "Things are usually so fluid, and projects [progress] in such a pace such that it is meaningless to study [them] deeply" *[AK]*. "Our world is a very dynamic world with systems that change all the time. Also if it's a system you develop with 50 other people and they all change parts, so you need to learn on demand and dig deep [only] as needed" *[AO]*.

An interesting contrast between industrial software development and the open source community arises at this point. Industry software it often pressured in its development schedule, and the developers have to compromise and reconcile their desire to comprehend with the schedule constraints. Open source developers are relieved from that pressure: "At work you want to provide value and be practical, when you write open source many times you can indulge yourself, you want to dig deep and understand really why things are as they are" *[BG]*.

The interviews included a discussion on evaluating and integrating external software packages into a system. A few subjects mentioned reliability indicators—stars on github, popularity of the package *[BG]*, or the identity of the package developer *[SN]* —as factors in such an evaluation. *[SN]* also explained that "If it is a source that I trust, that is, for example, open source code, it has a relatively high level of reliability. Why? Because many people look at it, in many cases libraries that are standard libraries are used by lots and lots of people and then you know that the reliability of this thing is high. [...] So as the reliability of the code is higher, you can trust it blindly and what you are interested in is the interface". So the perceived quality is also used as a justification to avoid comprehension. In other words, the reliability indicators serve as a proxy to the quality of the software package, and this allows the developer to skip parts of the understanding process.

Another way to avoid understanding is using unit tests. A good set of unit tests allows an outsider developer to debug and modify the code without fully understanding it: they just modify the code as they think needed, and then run the unit tests to make sure nothing else was broken. As put by *[AO]*, unit tests can provide an "automation of understanding". While a few of our interviewees said this is not considered a recommended method (*[YI]*, *[BY]*), this approach is in fact promoted by software engineering gurus, especially for refactoring tasks (Metz 2014; Martin 2015). In the words of Sandi Metz, "You want to get to the point where you have confidence that you can safely refactor, and then it means that you never have to understand that code".

3.3 Top-Down vs. Bottom-Up

The academic discussion of cognitive models of software comprehension has a long history, mostly circling around two well-known main models: the "top-down" model (Brooks 1983) and the "bottom-up" model (Letovsky 1987). Several combinations or variations of these models have also been proposed (Storey 2006; von Mayrhauser and Vans 1995; Fekete and Porkoláb 2020). In particular, it has been shown that when a software developer tries to comprehend a larger software component (a package or an entire system), a combination of the two approaches is useful (e.g. Kulkarni, 2016).

The interviews confirm this conclusion, and show a pattern of usage of the two approaches. Activities related to comprehending functions or classes were mostly associated with the "bottom-up" approach. But when asked about comprehending a package or a system, the participants described activities that are more related to a "top-down" approach.

A typical system comprehension effort would start with the “top-down” approach—get an overview of the system, run sample code, read the architecture documentation, trace the code from the main loops, and review the interfaces of the first level of classes or modules. But this approach has its limitations. “We build systems that are very involved and very complicated, and working top down on a large system does not hold water” *[AO]*. The amount of new information for a newcomer may be overwhelming, and it is hard to understand without spending some time “in the trenches”, developing and debugging code.

The participants therefore suggested to take up a small “bottom-up” task related to some sub-component—fixing a bug, adding a small feature, or studying a piece of code and generating appropriate documentation. “It is when people work with their hands that somehow this mechanical memory causes the information to be assimilated” *[YI]*, reflecting the Confucian saying “I do and I understand”. By performing a task, they develop familiarity with the sub-component and its related sub-components, and gradually comprehend that sub-component and its place in the system. In an iterative fashion, the developer then gets familiar with more components and combines this with the “top-down” comprehension of the system.

A few participants made statements implying that there is an element of personal inclination between the approaches. As *[AO]* described it: “Some people tend to dive deeper. [...] They tend to open the hood and understand how the engine works. Others prefer to sit behind the steering wheel and drive”.

Following the above discussion, it appears that developers that have a tendency towards a “top-down” approach have a better chance to comprehend large volumes of code. The “bottom-up” capabilities are basic capabilities that are required for everyday programming tasks, but tasks related to larger volumes of code, such as package comprehension or system architecture, require a combination of those “bottom-up” capabilities with “top-down” capabilities. This is related to the amount of details: “bottom-up” is based on understanding the details, and when the volume of code gets larger, the amount of details exceed what can be digested by a single person. The “top-down” approach allows a developer to avoid understanding many of the details by understanding abstractions. As described by *[MN]*, “it is more likely that a person knows all the execution possibilities, all the possible runtime states of a function, and achieves full understanding, while a single person is not likely to reach the same level of comprehension in an entire system that has a million lines of code. Then he needs to deal with abstractions”.

3.4 Aids to Comprehension

In one of the interview questions the participants were given 6 slips of paper denoting elements that may aid in comprehension: Source code, API documentation, Sample code demonstrating use, Inline and function header documentation, Training materials, and Design documents. The participants were asked to organize the paper slips in a way that shows the relative importance of each element for understanding a function, understanding a class, understanding a package for the purpose of using it (as part of package evaluation), and understanding a package for the purpose of maintaining it. Examples are shown in Fig. 2. The cumulative distribution functions for the rankings of each element in each of the questions is shown in Fig. 3.

For understanding a function, the elements that appeared most at the top of the list were Source code and Inline documentation. This is not surprising and matches the notion that for a single function, a bottom-up approach to comprehension is more efficient. The general assumption is that in order to comprehend a single function, everything is there in the code

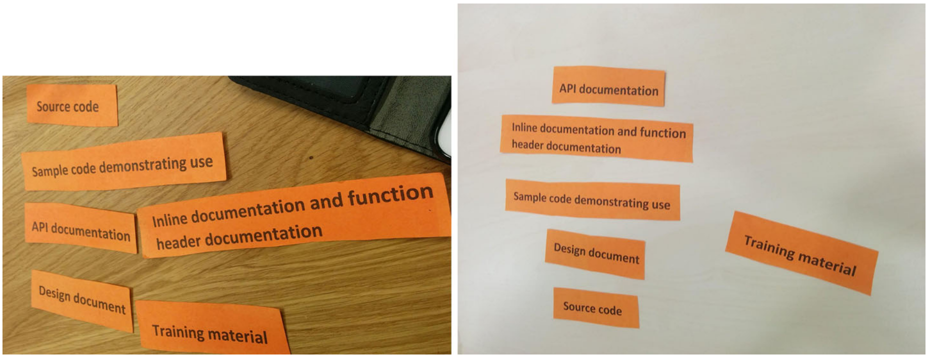

Source code
Sample code demonstrating use
API documentation
Inline documentation and function header documentation
Design document
Training material
API documentation
Inline documentation and function header documentation
Sample code demonstrating use
Design document
Training material
Source code

Fig. 2 Example rankings of comprehension aids for functions (left) or classes (right)

itself, and whatever is missing from the code is expected to be documented inline in the function.

For understanding a class, the most significant elements the participants noted were API documentation and Sample code demonstrating use. Here we see that the class is mostly defined by the API and methods' contracts, implying that understanding these contracts is the most important part of class comprehension.

When moving on to a package, we see a higher diversity in the answers. When asked about understanding a package for maintenance purposes, the most significant element is Design document, followed by API documentation and Source code. We see that when approaching a larger volume of code, the participants opt for a top-down approach and prefer a well-written design document. When asked about understanding a package for evaluation purposes, the most significant element was Sample code demonstrating use. Far behind it come API documentation, Training materials, and Design document. Here we see that the task at hand has an impact on the comprehension process.

Additional elements were also mentioned in the interviews as aids to comprehension:

- **Naming** (especially in functions): a meaningful name for a function is one of the first things developers look for.
- **Coding and naming conventions**: consistent conventions can help parsing and provide context to the code.
- **Minor maintenance tasks**: when discussing the training of new members in a development team, a few participants recommended giving them tasks such as fixing a simple bug or adding a simple feature to help them focus on the important parts of the code. Creating missing documentation or explaining the code to others are also considered tasks that build understanding.
- **Asking others**: involving other members of the team in the understanding process, including asking questions, sharing thoughts, or reviewing the code, can be helpful.
- **Exercising the code**: single-stepping through the code using a debugger exposes flow and behavior of code, as opposed to a static view. One participant mentioned adding "throw" statements in the code in order to pause the execution and examine the call stack *[BG]*. Adding tests is also a way to exercise the code.
- **Drawing analogies to familiar things**: as part of understanding a large amount of code.

The role of documentation in the comprehension process generated ambivalent sentiments by the participants. On one hand, participants acknowledged the difficulty of

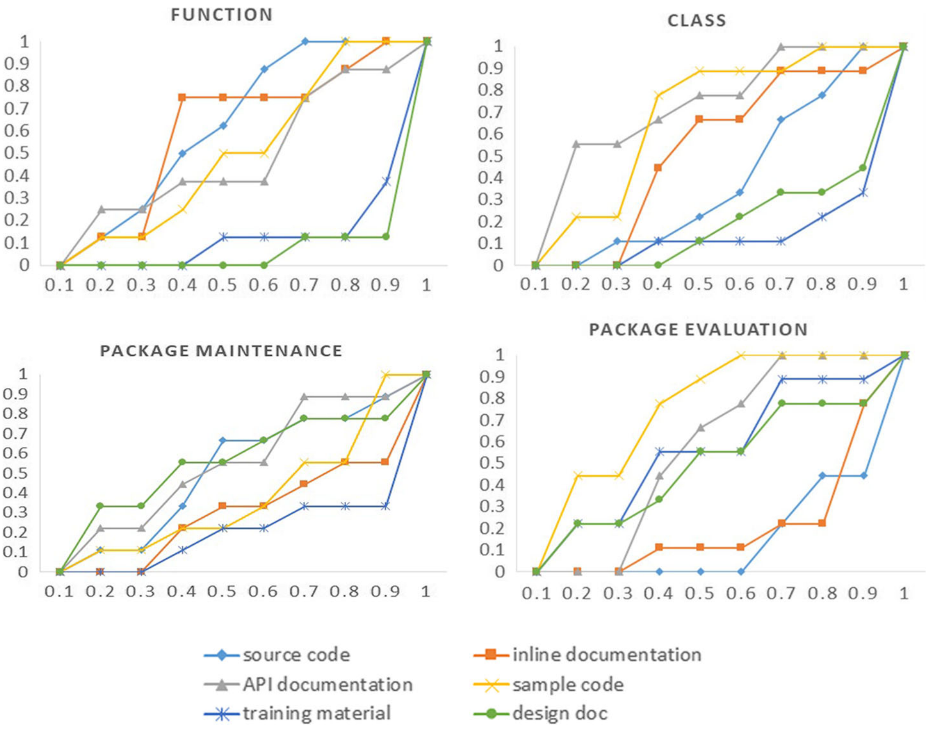


Fig. 3 Cumulative distribution functions for the rankings provided for each comprehension aid in different contexts. The X axis is the ranking, normalized to [0,1] because participants sometimes used different numbers of distinct ranks. Lower X values indicate a higher ranking. An aid appearing above and to the left of others (e.g. the yellow line for "sample code" in the bottom right diagram) was ranked higher than other aids; one appearing below and to the right (e.g. the green line for "design doc" in the top left diagram) was ranked lower

understanding a software system from the code alone. Many assumptions made by the developers are not exposed in the code, and documentation is the best source for understanding those assumptions. This includes architecture documentation providing a system-wide overview, or low-level design documentation for a single class or function. Even at the code level, inline documentation is largely considered crucial in understanding non-trivial hidden assumptions and developers' intents that cannot be gleaned from the code itself. "When you use a function without reading its documentation, it is mostly guessing that something behaves as you want, meaning you make up some reality for yourself and hope for the best. Even in functions that could be terribly trivial like arithmetic operations, you always have corner cases" *[YI]*.

On the other hand, documentation may quickly become stale and not in sync with the actual code. "My attitude to comments is respect them and suspect them" *[ES]*. Many of the participants therefore refrain from depending on the documentation, and a few of them prefer to avoid reading documentation altogether. This relates to previous work that identified face-to-face communication as preferred over documentation (Roehm et al. 2012).

In contrast, sample code that compiles and demonstrates how a software component should be used is in effect live documentation of the code. However, "it is very hard to generate good sample code when [demonstrating use of a] class, unless it is a lot of sample code” *[AO]*. Sample code demonstrates a use case, and a class, especially when it is complex, participates in many use cases, making it harder to demonstrate.

A particular type of sample code is test code, and in particular, tests that are part of a package’s continuous integration test suite, which gates updates to the code repository. “The documentation may be maintained or may not be maintained, the tests are surely maintained to the most recent state because otherwise the continuous integration [tests] wouldn’t pass” *[BG]*. Continuous integration tests are therefore the highest form of live documentation—it is code that compiles and is in absolute synchronization with the system’s code.

3.5 Proof of Comprehension

The participants were presented with the question of how is it possible to test whether one has comprehended a function. They were then presented with the following ways to test function comprehension, which are taken from academic writing on code comprehension: using the function; adding a feature; debugging; rewriting code from memory; describing function flow; describing the function interface; describing how it is used; fill in the blanks (cloze test). The participants were asked which methods resonate with their experience.

Most participants resonated with the functional methods to prove comprehension. But in doing so they consistently differentiated between black-box comprehension (describing how a function is used, describing its interface) and white-box comprehension (adding a feature, debugging it, and describing its flow). A recurring theme was that it shouldn’t be too trivial—you need to select the code and the task for the test to be meaningful. Some participants also specifically suggested asking about edge cases.

Two of the methods offered—rewriting the code from memory and filling in the blanks—stem from cognitive psychology research (Shneiderman 1976; Cook et al. 1984) as known methods to measure comprehension. They are based on the claim that these actions are much easier if one understands the code, as opposed to trying to perform them mechanically. However, these approaches did not strike a chord with most of the participants (other than in extreme cases: “If I send someone to space and I want to make sure that if need be that person could maintain the function, I would probably make him rewrite the function [...] but [otherwise I wouldn’t do it] to test their knowledge. It is a return-on-investment question – it is a very serious overkill but I can imagine situations where I would choose even this overkill” *[GA]*). Perhaps some of the resistance was also due to not knowing the theory behind these tests: “I personally don’t connect to... try to reconstruct exactly what was in the original function like a parrot” *[SN]*, and “that you can remember things by heart still doesn’t mean you understand something” *[YI]*.

Additional methods and refinements were proposed by a few participants. One was to suggest what could be improved in the function and how *[GA]*. Another was to suggest how the function should be tested *[AO]*, the idea being that a good test plan should reflect the potential weaknesses of a function, and being able to identify such trouble spots reflects deep understanding.

The participants were then asked to propose methods to test one’s comprehension of a package or a system. Descriptive “top-level” approaches (“describe the system functionality”) were prominent, however in many cases the participants wanted to test “bottom-up” understanding as well, such as “fix a bug” or “describe in depth a particular component”. Of course the participants acknowledged that it is impossible to cover the entire system and prove complete system comprehension with this approach, however it ratifies that system comprehension includes understanding at multiple levels.

In addition, participants expressed reservations regarding the concept of a general test of comprehension, saying that it depends on the level of comprehension you want. This also depends on the level of code: for a function you may want to understand the dark corners, but "as you go up the hierarchy it is more important to understand some model, some concepts that hopefully exist in the code" *[MN]*. Also, "it becomes less and less binary. Maybe you understand some part better, and you don't understand so well another part" *[AK]*. Finally, *[BG]* said that the best way to know if someone understands something is to just ask them. If the situation is not adversarial, they will most probably admit to not understanding things they do not understand.

4 Hierarchy of Comprehension

We will now break down the meaning of comprehension to the different levels—function, class, package, and subsystem. Although several concepts are similar at all the levels, such as the distinction between black-box and white-box comprehension, there are important differences that help shed a light on the meaning of system comprehension. These are summarized in Table 1. Flows and system-level comprehension will be addressed later.

4.1 Function

The participants were asked to define what does it mean to understand a function. A few participants pointed out that this is an artificial situation, because in practice understanding a function is always part of a larger context, and that understanding a function outside of any context resembles "a job interview situation" *[DL]*, *[ES]*. "A function [...] is part of some whole. It affects the [class's] state and is affected by the [class's] state, so you need some more general understanding of the class" *[ES]*. On the other hand, *[BG]* mentioned that in a code-review situation—common in open source development—you sometimes do consider a function in isolation outside of its context.

Most participants defined understanding a function as understanding its "contract": its parameters, its return values *[AK]*, its name, and its functionality *[SN]*. Others mentioned the function's prerequisites, how it works in different conditions, what is the system state before and after the function. This is what was earlier referred to as black-box

Table 1 High-level summary of the main elements of comprehension at different levels of the system hierarchy

Level	Essence	Black-box	White-box
Function	Provide functionality	How to use based on function signature	How functionality is implemented
Class	Provide state and encapsulation	How to use based on API	Data flow between functions
Package/ subsystem	Independent and exchangeable units	How to use based on API	Design patterns and interactions between classes

See text for detailed discussion

comprehension, and is tightly coupled with the idea of information hiding and Meyer's contracts (Parnas 1972; Meyers 1992).

The participants described white-box comprehension as the "deeper" level of comprehension. This means "understanding how the function does what it does" *[SN]*, "understanding how it works and does what it is supposed to do" *[AK]* or "actually understanding the logical order of operations and why this is the order" *[DL]*.

The level of understanding that is required depends on the task at hand, what one is planning to do with the function. Most participants agreed that it is very possible to use a function without "fully" understanding it—i.e., black-box comprehension is required, but not necessarily white-box comprehension. *[DL]* gave an example of a face recognition function: most developers who use the function do not have any understanding of how the function works.

Unfortunately, functions may have properties that impact their black-box behavior, but are in many cases overlooked as being part of the function's contract. This forces developers wanting to use the function to dive deeper into the implementation details and to defer to white-box comprehension. Such properties include:

- **Side effects:** a situation in which the function changes a state that is outside its local environment. The participants identified this as a pain point in function comprehension: "The difficulty in functions is side effects" *[MN]*. Indirect side effects can be even harder to track, such as when a function depletes resources from a global resource pool.
- **Reentrancy or thread synchronization:** The developer of a function may not be aware, at the point of writing the function, what are the synchronization requirements of the function, or those might change later.
- **Performance:** Consider for example a function that implements a core algorithm. The function's asymptotic complexity or actual performance become a critical property of the function, and one might consider them part of the function's contract. However, these cases are rare. In most cases the runtime of the function is considered part of the function's implementation details (white-box) *[YI]*, and not part of the function's contract. "The understanding and the need to understand the function depends on what you are looking for, what your constraints are. If you care about performance you need to understand the performance of the function. If you care about [...] communication bounds then you care about that" *[BG]*. "The level of understanding will change depending on the level of optimization you need to perform" *[AO]*.

4.2 Class

A class is a collection of functions (methods) and data members (which hold each instance's state). "When you work with a function that is part of a class, [...] the object itself provides you with context and a translation of the problem to entities that are part of the problem's solution" *[YI]*.

The participants once again distinguished between a black-box comprehension and a white-box comprehension of the class. But they mentioned various aspects of black-box comprehension that are related to understanding the wider context of the system and the place of the class in the system, beyond its public interface. For example, how the class integrates with other classes in terms of inheritance and implementing design patterns. "Understanding a class is actually understanding the abstract concept that it represents, and when can I use it or when is it the right thing for me to use this concept" *[DL]*, "its limitations—when can it not be used" *[DL]* , "why is it there, why isn't it part of some other object, why isn't it split into two" *[YI]*. As *[YI]* coined it, this is all part of the *intent* of the developer that is reflected in the class design. The intent is not part of the code. Ideally it would be part of a design document. This is a particular example of what we earlier called "unboxable comprehension".

It is interesting to look at the effect of the class on the comprehension of its methods. Assuming a class has cohesion, its components are not independent: methods and data members know of each other and interact. Thus a method is related to the class's state, and cannot be comprehended by the contract alone, without the context of the class.

Finally, as we go up to the level of a class and beyond, it may be impossible to maintain complete abstraction of the interfaces from the details. This is Joel Spolsky's "Law of Leaky Abstractions" (Spolsky 2002). "From some level of complexity abstractions always leak. [...] there is always a situation where the abstraction is supposed to tell you, you don't care what's inside, but it would actually break and you would care" *[ES]*. To some extent this conclusion contradicts the "black-box" vs. "white-box" distinction: you need to understand the details in order to understand the abstractions. This might explain why system comprehension is hard, and why it requires a combination of a top-down view of the system with a bottom-up understanding of the details and how they impact the system and its breakdown.

4.3 Package and Subsystem

This level of the hierarchy was meant to represent a first breakdown of the system into software components, some significantly large volume of code that is not the entire system. We originally used the term "package" to loosely refer to a collection of classes that provide a cohesive functionality. For example, this could describe the use of dynamic link libraries (DLLs) in a Windows environment. A few of the participants expressed objections to this definition. *[MN]* mentioned that the term "package" has a specific technical meaning in the Java programming language (Wikipedia 2018). He preferred the use of "deployment package" for the context of application servers. *[SN]* pointed out the fluidity of the term: for example, in one system it may refer to services in the system, and in another it may refer to individually compiled executables.

[AO] suggested that "package" should be augmented with "subsystem", in the sense that in a distributed system, multiple subsystems may be communicating with each other, while internally they may each be structured using software packages. In the following we adopt this distinction (which was also used above in Fig. 1).

A distributed system with several deployable components or subsystems is in fact a "system of systems". Each subsystem can be comprehended separately, and viewed with its own function/class/package hierarchy. In addition, there is another layer of the larger system, and flows that exist between the subsystems.

Typically, the communication between layers of a system is done via a programming API. But communication between subsystems in a distributed system is done using a communication protocol. While conceptually similar, a communication protocol is typically outside the scope of a programming language, and cannot be simply defined within the scope of the code. This might represent a disadvantage, in that it is harder to understand a protocol without looking outside the code. On the other hand, it might serve as an advantage, in that a strict protocol documentation is more likely to exist and define the protocol characteristics. Communication protocols are free from issues such as memory management and concurrency between the two sides, issues that are commonly harder to understand in API definitions.

Returning to packages, we define a package as an “atomic unit of reuse”. While classes and interfaces are intended to be reusable, the reality is that in many cases functionality is divided between inter-dependent classes that each have a single responsibility—and as such they cannot really be used independently of supporting classes or related classes. For example, this is at the center of the concept of design patterns. The package is the smallest unit of code that can be extracted and reused in a different context. This is especially true for packages that are developed with the intention to be reused, such as open-source libraries or services or their commercial counterparts.

In the context of comprehension, this means that a package can be comprehended outside the context of the system. “Third-party packages are perhaps the best example. There is a company that develops the package so that it would not be dependent of the system, and so usually it is really independent of the system. [...] I think of packages in our system that we developed, there we created more dependencies to other things, whether we intended to do so or not” *[ES]*.

Under this definition, a package is the only level that can really be comprehended as a black-box. If the reuse is “as is”, with no modifications to fit the specific system, this reduces the comprehension effort as the package is comprehended at a black-box level only. This is especially true when the package performs a complex functionality that requires an expertise that is not available in the hosting system’s development team, such as an algorithm implementation. “If you need a library that will implement ‘zip’ or ‘unpack’ [...] it is a black box, you don’t care how it works. It just needs to have the right interface and then you just use it” *[SN]*.

In the context of evaluating packages to be reused, factors such as reliability and popularity become significant as proxies to the quality of the code, and therefore to the need to comprehend them, as previously discussed in Section 3.2.

5 Flows: Understanding System Dynamics

The hierarchical layers in the previous section define a static structure: it describes the components of the system and how they are encapsulated in one another. However, in order to understand the system, it is required to also understand the dynamic aspects of the system—its behavior. “I think that describing a function’s flow fits the definition of understanding *how* it does things, and describing the function fits *what* it does” *[BY]*—the flow comprehension is related to the deeper white-box understanding of “how”. Littman et al. put it slightly differently, saying that the flows represent the causal interactions between the functional components (1987). Flows are also important with regard to non-functional requirements, such as reliability or performance.

Understanding system flows has several distinct but related aspects:

- The flow of the control during execution;
- The flow of data from one component to another;
- The system state and how flows change it.

System states are often used to analyze system properties, e.g. to show that certain states will occur repeatedly (liveness) or will never happen (safety) (Alpern and Schneider 1987). As each system state is a combination of the states of all the system elements, the number of states is exponential in the size of the system. Brooks notes the myriad possible states of a system as a major reason why software development is hard (1987).

While system states and state transitions describe flows in the system, this seems to be a higher level of abstraction that is more distant from the code. The alternative is to describe the system's functionality using use cases, with details given in the form of entry points and flows. A few participants in the interviews indeed described the product they work on through its use cases (for example *[YI]*). *[BG]* mentioned studying a flow—not a component—as a way to start learning about a new open source system you want to join. *[AO]* described a project where the division of labor was based on scenarios or features rather than on components. This was claimed to have two major advantages. First, it is more interesting to the developers. Second, it produces better developers, because they need to know all the different components they work with and need to have a level of system-wide understanding. As a result they become more independent and more interdisciplinary.

At the bottom level, the highest level of understanding a function is "the level of understanding every flow" *[MN]*. This is also why debugging is a good way to learn about code: "in debug you see everything: you see the code, but you also see the flow, because when you put a breakpoint you can see the whole stack and then you see how this thing behaves completely" *[SN]*.

For a class, flows are combined with states: this "is something that has a lifecycle and you have to understand its lifecycle. Instantiation, destruction, and all the possible sequences" *[MN]*. System state may be a concern in re-entrant code. "The function does not do all its work in one call. ... you need to cope if maybe between [calls] the state of the system has changed" *[MN]*. More generally, *[AK]* mentioned state as an element of comprehension at the system level, but not at lower levels.

While we did not delve into the topic of flows in the interviews, in retrospect some interesting observations can be made. Flows may be described at different levels of the hierarchy (see Fig. 1): between subsystems, packages, classes, or functions. Within each level, flows may define a "service hierarchy" of components. This is related to the direction of function calls. Typically newly written software calls many utilities and depends on them. But understanding the software does not require a deep understanding of the utilities. Compare this with single-stepping in a debugger, when looking at a high-level function and skipping over service functions that are not part of the flow being debugged. And these utilities often call and depend on still other utilities. This happens at all levels of the structure hierarchy:

- Functions that implement use-cases call service routines;
- User-facing front-end classes call on classes that implement business logic and back-end functionality;
- Packages that constitute the system being built use external packages from third-party vendors.

The exception to this directionality is callbacks, used in event-driven programming. We did not explicitly raise this issue in our interviews.

It is also interesting to note that some of the most complex elements of software comprehension are related to flows in the system:

- Memory management: coordinating the allocation and freeing of memory and avoiding memory leaks.
- Recursion: where a function (perhaps indirectly) calls itself.
- Concurrency: multithreading, obtaining and freeing locks, and avoiding deadlocks.

As part of the flow analysis of the system, these elements are typically not visible in the code, and are part of the "unboxable" comprehension of the system. This on its own may be part of the difficulty in understanding these elements. *[DL]* gave an example of a networking package with a function they had to debug that was hard due to asynchronous interactions of two threads and accessing unallocated memory.

6 System Comprehension

Given all the above, we can now focus on the issue of understanding software at the system level.

6.1 What is a Software System?

In the interviews we presented the definition for a "software system" as follows:

> We define a software system as a collection of packages that create an end-user product or end-user experience. A company may be creating a system that contains packages that are developed within the company, or imported from other companies. Alternatively, a company may be developing packages to be integrated into customer systems.

While overall the participants agreed with the gist of this definition, some of them challenged it. First, not all participants agreed that every software system creates an end-user experience, or at least suggested that in some cases this end-user experience is hidden. A recurring example of such a system was a network router or network firewall, that are software systems that have impact on the end-user, but this impact is hidden and even transparent.

Second, a few participants challenged the hierarchical approach defining a software system as a collection of packages. This feedback stems to some extent from the disagreement on the term "package" as discussed in Section 4.3, and the fact that some systems are distributed systems where there are several "subsystems" or "deployment packages" in the system and each of those subsystems is a separate executable that might be comprised of software packages.

Third, a few participants pointed to the fact that software systems interact with each other, such that the boundaries of the "system" becomes unclear. If several software systems interact with each other, are they really subsystems in a larger software system? Or is there a larger concept such as "system of systems"? This point is tightly related to the previous comment and the concept of "deployment packages".

Fourth, the dynamic nature of a system is not reflected in the definition. In addition to the packages that comprise the static structure of the system, the system does not have meaning without discussing the use cases, the entry points, and the flows.

Finally, *[AR]* challenged the fact that the term needs to be explicitly defined. "A software system is source, any system that has software. It does not need to be defined in a more complicated fashion.“ To the follow-up question asking what is the distinction between a software library and a system he responded: "These are distinctions that do not need to be defined. One cannot define such a thing because 'system' is just a word. [...] If you ask most people they will say that a single line of code is not a system and a million lines is a system, and as happens with anything related to words, the boundary cannot be accurately defined". Likewise, *[BG]* said "I think that these things, it's very hard to define them. And also when you define them usually there isn't a lot of benefit in a precise definition." And when *[MN]* discussed what should be defined as a package he mentioned “not getting in trouble with [philosopher] Bertrand Russell”.

We conclude that indeed it is difficult to provide a precise definition of a “software system”. But despite the above feedback, we believe that the definition we provided, possibly with some adjustments to clarify the “end-user” aspect and the “package” definition, will be acceptable by most as capturing the essence of a software system.

6.2 What is System Comprehension?

We asked our interviewees what does it mean to comprehend a system. Many participants described the structure of the system as the key to comprehension: “Intuitively, comprehending a system is [...] its modules and how they communicate with each other, what is the role of each one, and how they play together to create something” *[BY]*. This definition includes both the static structure and the key flows in the system (the flow of data between modules): “In a system you look at how the objects come together, it is more a view of pipes of how the data flows from here to there. [...] Something comes in from one side, it splits into three copies here, it goes through some processing, the processing is synchronous or asynchronous [...] and what comes out from the other side” *[YI]*; “We talk not only about input and output, but more about data flow” *[AO]*.

Beyond the structure and data flows, system understanding means a deep understanding of the interactions and inter-dependencies between components. *[GA]* gave an example of a tracking system that did not perform as well as the requirements demanded. Investigating the root cause of the problem led to inadequate adjustment to varying lighting conditions. So the solution was not to improve the tracking, but to improve the exposure subsystem. Similarly, *[AO]* mentioned a client server system where the communication latency affected the user experience. The problem was not in the client software or in the server software, it was the latency—and by implication, the architecture.

Beyond the inter-dependencies among components, there may also be a dependence on the system’s deployment environment. *[AO]* described a web-based application, and said that “in a distributed system world, looking at the software system without considering the underlying hardware and its topology, its capabilities, its limitations is not enough. ... The hardware stack is not just hardware... to start to understand the service [in the application] you need to start by understanding how data centers work, where they are connected, what the parameters of the communication network are. Moreover, there are parameters of billing what costs me. In a world of service cogs it becomes something you can’t ignore when you consider software design.”

Another common reference made in the interviews was to intent. This could refer to the intent of the system as a whole: “Understanding a software system is first and foremost understanding its grand objective, what service it provides” *[DL]*. But it could also refer to the intent behind the structure of the system, the considerations that led to this particular structure: “I also want to understand all sorts of considerations why, [...] why are these the system’s modules” *[DL]*. Important considerations at the system level are the system’s non-functional requirements. “Non functional considerations start to be part of the comprehension. As the system is larger and more complex, the need to understand its non functional aspects grows—if it’s regarding where this system is vulnerable, where are its reliability parameters, its performance behavior” *[AO]*.

These two traits of system comprehension—understanding the structure and flows of the system, and understanding the rationale for this design—are completely unrelated to the code. An example of this observation is the teaching of operating systems to computer science undergraduates. While early academic work on Unix was based on the source code (Lions 1996), later textbooks supported a good understanding of this system—including processes, scheduling, the file system, and memory management—at the conceptual level, with little or no reference to the underlying code (McKusick et al. 1996; Bach 1986).

In the interviews, *[AR]* went as far as to claim “to understand a system you don’t have to know to program at all.” This conclusion is contrary to the comprehension of lower layers of the system hierarchy, where the code and related elements, such as the contract, were critical to the comprehension. As we go higher in the system hierarchy, the focus of comprehension moves away from code and contracts, and towards discussion of general structures and data flows. An architect that is developing the structure of the system does not necessarily need to be familiar with the code to develop it. Similarly, a person trying to understand the structure of the system might not need the code to do so. As a result this comprehension level is also completely independent of the programming language used to create the system, and may be described in terms that are independent of the programming language.

One reservation to the above is the law of leaky abstractions (Section 4.2): that the details at lower layers tend to leak into the abstractions made at higher layers. This is inevitable as the system gets more complex, and may affect the system level (Lehman 1980). The meaning in this context is that sometimes implementation details of a certain function or class may lead to a particular design of the system structure. If this happens the system structure cannot be fully understood without appreciating the leaks that affected it.

As part of understanding the reasoning behind the system structure, the system’s history and evolution also play an important part. “If you observe Thunderbird, you first need to understand why they developed this project. It is very important to understand [...] the original objective, what need they tried to address. Would anyone today start this project? [...] You can read your email in a browser, why do you need a desktop client? It is debatable, but you need to understand why they did this in order to understand the system” *[AR]*.

In fact, the history and evolution of a system is one of the reasons understanding the system is difficult. The current structure of the system may reflect intentions, constraints, and choices that are no longer relevant. Systems that were originally well-designed may be modified to address unexpected needs and usages in such a way that makes them convoluted (reflecting Lehman’s second law of software evolution (1980)). Understanding the architecture may also be confounded by out-of-date documentation, which reflects the original design and not the design that evolved.

To summarize, understanding a software system appears to be profoundly different from just understanding its code. It even includes more than just understanding the design decision and the considerations that led to these decisions. The system level is where you see the big picture: the interactions, the dependencies, and how it all falls together.

6.3 System Comprehension Specialization

Based on the above, it appears that comprehension at the system level is a unique specialization, distinct from the baseline specialization of a software developer. It combines top-down system understanding with relevant low-level details that impact the high-level breakdown. It lives in the realms of documentation and block diagrams, and may be detached from the code itself and even from a particular programming language.

One extreme example is a participant who introduced himself as follows: “I do not have a very strong background in development [...] My industry background is mostly as an entrepreneur, I started a software company [...] but I was not involved in the development of code, I was the CTO [...] The original idea was in fact an architectural idea” *[AR]*. So, the main product of the company was an architectural innovation, by a person that was not involved in the development and does not have a strong programming background. The architecture was completely detached from the code itself.

The interviews included an explicit section about personnel: how many develop a package or a system, and how many of them have a deep understanding. One participant described a project where the developers worked in a “feature crew” model, where each team was responsible for development of a suit of features across all system packages *[ES]*. In such a mode of work, most developers are familiar with all system components and have a basic system understanding. Nevertheless, when complex system-wide issues are being handled, the number of developers that are capable of handling such issues decreases to about 50%.

When a more traditional component-based code ownership model is used, the reported numbers were lower. But it also depended on the size of the component and team. In teams of a few people (for a package) up to dozens of people (for a system) the most common estimate was that about 20% of them have a good understanding of the package or system. But in small teams of up to 4 developers it was thought that 2 or more of them may have a good understanding, which can be over 50%. The people that have such a high level of comprehension were identified as architects, technical leads, or just more senior developers.

Unfortunately, the skills required for system comprehension—advanced design principles, a top-down approach to systems, managing a large software system—are not always taught in higher education software engineering programs, even though they are critical skills for every software team to have. *[BG]* blamed this, together with the notion that architecture can’t be completely defined in advance, for cases where teams avoid doing any architecture at all.

The bottom line is that the difference between system comprehension and code development engenders a distinction between those who have the aptitude for system comprehension and those who don’t. In addition, practices such as confining developers to work on specific modules limit their ability to appreciate the intricacies of the full system. Taken together the result is that in large systems only a fraction of the personnel achieve good system comprehension.

6.4 Architecture

The term “architecture” is widely used in the context of system description, but its meaning may differ depending on context, organizational culture, and personal preferences. We presented the participants with statements related to architecture, and asked them to rate their agreement with these statements. The scale is -3 to 3, where -3 means “strongly disagree” and 3 means “strongly agree”. The results are shown in Table 2.

All the participants agreed that every system has an architecture, whether it was designed by a conscious set of decisions or not. In other words, the architecture exists independently of the architecture definition process. The documents and diagrams that support the architecture (hopefully) reflect the implemented architecture, but do not define it. There was strong agreement that the architecture is the system’s structure, but that a UML description of the system is not architecture.

Most participants agreed that having a design phase is important to an effective development process and that the architecture reflects the design decisions. However in many cases, as part of the system’s evolution, architecture “happens” and does not entirely reflect conscious decisions. It is also harder to maintain documents that reflect the architecture as it evolves over time. The constant drift between the conscious decisions and the actual

Table 2 Results regarding the Architecture Statements, in order of general agreement (not the order in which they were presented to participants); μ is the average of all responses

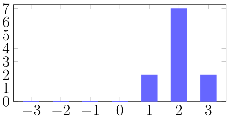
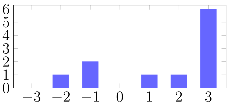
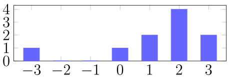
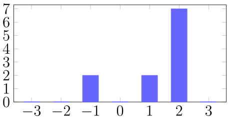
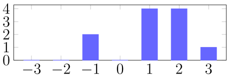
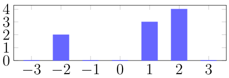
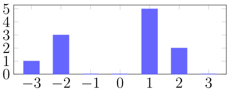
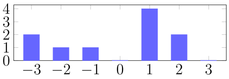
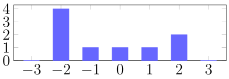
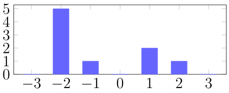
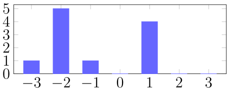
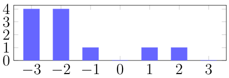
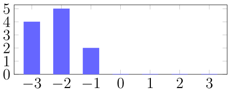
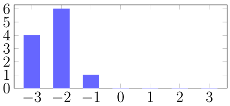
Every system has an architecture	$\mu = 2.00$	
Architecture reflects design decisions	$\mu = 1.55$	
Architecture is the product of a design process	$\mu = 1.30$	
Architecture is the structure of the system	$\mu = 1.27$	
Implementation changes over time as a result of architecture changes	$\mu = 1.18$	
Architecture defines internal interfaces between components in the system	$\mu = 0.77$	
Architecture reflects the company's organizational structure	$\mu = 0.00$	
Architecture defines internal interfaces between teams in the development group	$\mu = -0.10$	
Architecture changes over time as a result of implementation changes	$\mu = -0.44$	
Everyone in the development groups is familiar with the architecture	$\mu = -0.78$	
Architecture is defined in documents	$\mu = -0.91$	
Architecture defines only the externally visible properties of the system	$\mu = -1.64$	

Table 2 (continued)

Architecture is a UML description of the system	$\mu = -2.18$	
Architecture defines only the internal structure of the system	$\mu = -2.27$	

architecture, and between the design documents and the actual code, may be reasons for the difficulties in comprehending the architecture.

[BG] in particular lamented the rush to write code without thinking about the architecture and considering alternatives. He recalled giving a talk about a popular open-source library that he maintains, where he demonstrated how to write a front-end app using this library. “And all the questions I got after the lecture were, ‘so I should use Y instead of X for my architecture?’ Like it was really hard for people ... to understand like that they need to sit and think about their system.”

The architecture includes both the internal decomposition of the system and data pathways, and externally visible attributes of the system.[1] Architecture consists mostly of design decisions, and the considerations that led to the decisions (Kruchten 2004). The considerations and decisions could be conditioned upon the philosophy that drives the system goals *[BG]*. In many cases the design decisions are trade-offs between conflicting considerations, such as performance, physical resources, cost, or development effort *[AO]*. Elements that are not part of the designed system but surround it—such as the operating system, the underlying network, the selection of programming language and compiler—are all factors that impact the system design decisions and the architecture.

Most participants acknowledge that there is some relationship between the architecture and the organizational structure (a soft version of “Conway’s Law” 1968). In some cases the system is decomposed to teams based on the teams’ location or expertise. The architecture may also put special emphasis on internal interfaces that reflect interfaces between teams, especially if they are geographically or organizationally distant. There are also instances in which the architecture drives the organizational structure (instead of the other way around).

This discussion on architecture highlights the relationship between system comprehension and architecture. Many elements discussed as part of the definition of system comprehension (as discussed above) appear in the definition of architecture. The structure of the system—the main components and the communication paths between these components—plays a significant role in understanding the system, and is also a key part of the architecture. Understanding the intent of the system, of its main components, through understanding the system usages and the system’s history and evolution, are also significant in both system comprehension and the architecture.

[1] In the hardware world there is a distinction between the terms ‘architecture’ and ‘micro-architecture’, reflecting the externally visible attributes of the system vs. its internal design. Software organizations in hardware corporations sometimes adopt this terminology. However in the software world the term ‘architecture’ usually refers mainly to the internal design.

Yet most participants indicated that understanding the system is more than understanding the architecture: it also includes understanding some of the important low-level details of the system. Those details may have an impact on the larger structure of the system, or may not—and therefore are not part of the architecture definition—but they are significant in understanding the system. Thus, system comprehension requires both views of the system: the top-down view provided by the architectural elements, and the bottom-up view provided by the low-level details. Obtaining both views takes time and requires a combination of architecture tasks, such as reading documentations and API definition, and maintenance tasks, such as fixing bugs or adding features.

Another difference between architecture and system understanding concerns domain knowledge. *[BG]* gave an example of deep learning. “I can understand the whole architecture and not understand at all what the system does. [...] Really understand how each layer talks with another layer in my network, ... [that] there is marshaling and unmarshaling and things that run on the GPU and all those things, and still I will not have a clue what my code does.” In other words, even if you understand the technicalities, this is not a real understanding of the essence.

To recap, architecture plays a key role in system comprehension. It can be said to encompass the top-down structural design aspects, with an emphasis on the technical side. But full system comprehension is wider then that.

7 Validation Survey Results

Most of the questions in the validation survey were designed to reflect our observations and conclusions from the interviews, so high scores may be expected. However, in some cases the answers reveal that our conclusions are not universal.

Table 3 shows the answers we received to questions about comprehension in general. Survey participants strongly agreed that there are different possible levels of comprehension, and that it is important to understand the system components and flows.

Participants also agreed that information required for understanding may exist outside of the code (what we called unboxable comprehension above). Specifically, they tended to agree that the assumptions of whoever wrote the code are important, and that this makes concurrency and memory management hard.

Concerning the distinction between black-box and white-box comprehension, participants agreed that white-box reflects deeper understanding. They also generally agreed that components can be used with little understanding of their internals. But at the same time, the tended to agree with statements implying that understanding is more than APIs. Interestingly, the explicit question on whether white-box reflects a deeper understanding than black-box was the only one that elicited multiple “don’t know” responses.

Interestingly, participants were somewhat reluctant to admit that they themselves try to understand only as much as is needed to complete a task. And they tended to disagree with the statement that one can be an excellent developer without understanding the complete system. This may be a difference between an online survey and an interview. In the interviews such controversial statements were often discussed, and the interviewees sometimes pointed out intricacies that confound the issue or even changed their minds. But in a survey what you get is typically the subject’s initial reaction.

Table 4 shows questions related to comprehension aids. The survey respondents gave their strongest support to good names. We also see very high support for usage examples,

Table 3 Results to questions about comprehension in general; ? shows “don’t know” or no response; μ is the average of all other responses

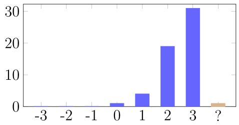
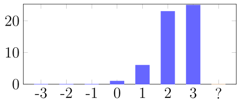
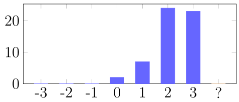
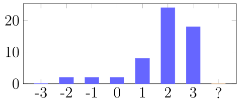
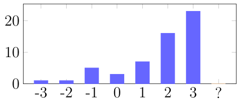
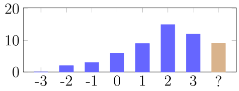
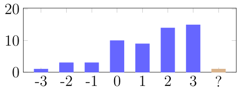
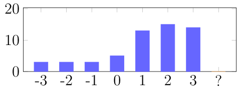
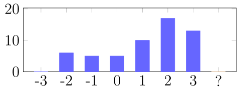
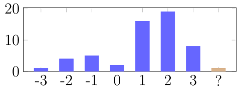
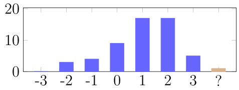
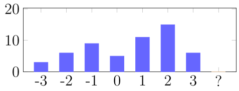
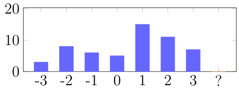
There are different levels of understanding of a software component	$\mu = 2.45$	
When understanding a software system, it is important to understand the main flows of the system	$\mu = 2.31$	
When understanding a software system, it is important to understand the main components of the system	$\mu = 2.21$	
When understanding code, there may be information outside the code that is required for understanding	$\mu = 1.86$	
A well designed software component can be used with little understanding of its internals	$\mu = 1.75$	
White-box comprehension reflects deeper understanding than black-box comprehension	$\mu = 1.45$	
From my experience, I can use a software component without understanding how it works	$\mu = 1.27$	
To understand code one needs to understand the assumptions of whoever wrote it	$\mu = 1.20$	
Source code is not needed in order to evaluate a package, if you just want to use it	$\mu = 1.18$	
Topics such as concurrency and memory management are hard to understand because they are not clear from just reading the code	$\mu = 1.13$	
Understanding classes and packages means understanding their APIs	$\mu = 1.02$	
One can understand a software system without knowing much about the structure of its code	$\mu = 0.53$	
Before integrating a component I didn’t develop, I learn only what I need for the purpose of integration	$\mu = 0.49$	

Table 3 (continued)

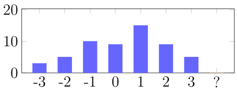
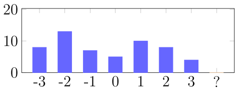
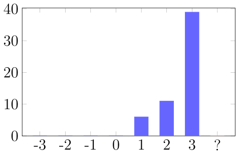
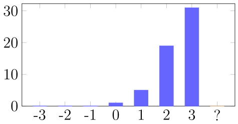
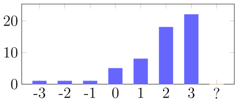
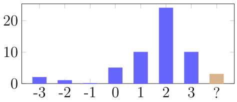
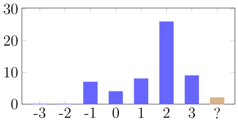
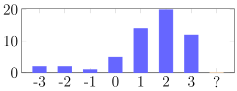
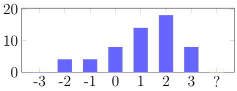
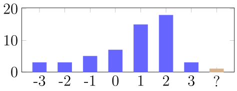
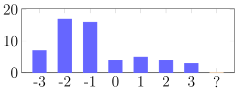
When fixing a defect, I learn the code only as much as needed to perform the task	$\mu = 0.34$	
One can be an excellent developer without good understanding of the complete system	$\mu = -0.35$	

Table 4 Results to questions on aids to comprehension; ? shows "don't know" or no response; μ is the average of all other responses

Good names are important for understanding code	$\mu = 2.59$	
Usage examples are very useful for understanding a class	$\mu = 2.43$	
In-line documentation is useful for understanding code	$\mu = 1.86$	
When understanding a software system, I combine reading the architecture documentation with reading the code	$\mu = 1.54$	
A good suite of unit tests can help with understanding a software component	$\mu = 1.48$	
Design documents are useful for understanding packages and systems	$\mu = 1.41$	
To really understand a function I read the code	$\mu = 1.11$	
I do not always trust documentation	$\mu = 0.74$	
Design documents are not useful for understanding code [note inverted scale]	$\mu = -0.88$	

relatively high support for inline documentation, and medium support for test suites as comprehension aids.

The attitude towards documentation was somewhat ambivalent, as was expected. On the one hand respondents reported that they read the architecture documentation together with reading the code when they need to understand a software system. They also agreed that design documents are useful for understanding packages and systems. But they also agreed that they don't always trust documentation. Note the reversed scale in last question: the disagreement essentially means that design documents are indeed useful for understanding code. Interestingly, the statement "To really understand a function I read the code" received only moderate support.

Table 5 shows answers to questions concerned with the comprehension strategy. There was relatively high uncertainty about these concepts, especially the common definition of bottom-up as a comprehension strategy based on generalizing from the code. Still, respondents tended to agree that a top-down strategy is required in order to understand large volumes of code, and that it is beneficial to combine this with a bottom-up approach.

Finally, Table 6 presents the answers to questions about flows. As noted above, there was strong agreement that flows are important to the comprehension process. However, respondents also agreed that flows can be difficult to understand. This may be related to the two questions which indicate that developers are somewhat unsure about how to deal with system dynamics: there was only weak agreement with the claims that dynamics are best understood based on system states and their transitions, or that they are embodied in flows.

8 Threats to Validity

The participants interviewed in this research are not a representative sample. Their number is small, 10 out of 11 are male, and all are from a small set of companies in one country. However, in exploratory qualitative research a representative sample is not necessarily required, and in fact the target population may not be well defined. The goal of the research was to raise the appropriate questions and terminology in order to seed future experimental research (see Section 9). The participants are highly experienced professionals, many working in multi-national companies, who interact with developers from other cultures. Some

Table 5 Results to questions on comprehension strategies; ? shows "don't know" or no response; μ is the average of all other responses

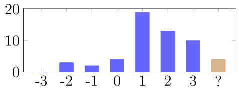
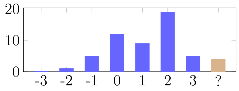
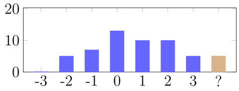
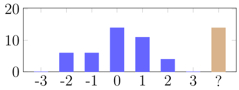
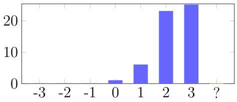
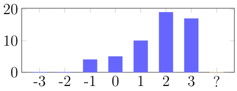
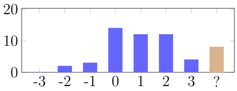
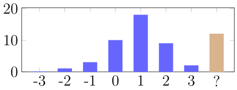
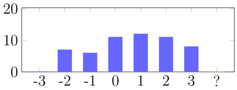
Understanding large volumes of code requires a "top-down" approach	$\mu = 1.31$	
To understand a system one needs to combine "top-down" and "bottom-up" approaches	$\mu = 1.08$	
System understanding is achieved top-down, starting from documentation and verifying that the code complies with it	$\mu = 0.56$	
"Bottom-up" comprehension helps understanding the system by generalizing from understanding the code	$\mu = 0.02$	

Table 6 Results to questions related to flows; ? shows "don't know" or no response; μ is the average of all other responses

When understanding a software system, it is important to understand the main flows of the system (repeated from Table 3)	$\mu = 2.31$	
Flows can be difficult to understand from just reading the code	$\mu = 1.73$	
The dynamics of a system are best understood using the system's states and state transitions	$\mu = 0.87$	
The dynamics of a system are embodied by its flows	$\mu = 0.86$	
Flows are best understood using documentation	$\mu = 0.69$	

of their responses were common enough to lead us to believe they can be reproduced in a larger, more representative sample. The validity check survey indeed shows some indication to this fact and increases the validity of those results. Nevertheless we believe wider studies must be performed to ratify the results of this research.

The first author (and interviewer) is an experienced professional in the field of system architecture. This is an advantage in terms of being familiar with the terminology and the work processes of the participants. However, a valid concern to the research validity is whether preconceived notions and personal biases were mixed with the interview and the analysis. We addressed this concern by using general, open-ended questions and allowing the participants to challenge whatever definitions or constructs were proposed in the interview. In the analysis phase, we both performed the interview analysis separately and reached conclusions only after a joint discussion ("analyst triangulation", a known method to increase research validity). The analysis was based on the textual analysis of the interview audio recording transcripts, rather than on personal impressions (an "audit trail", also a known method to increase validity). Nevertheless and as aforementioned, replication studies are required to ratify the results of the research.

9 Conclusions

Program comprehension accounts for a large portion of any software development effort. We attempt to understand the processes that are related to comprehension of a software system as a whole, as opposed to most existing research which focuses on program comprehension on a small scale. Through a series of semi-structured interviews with experienced software developers, architects, and managers, we drew the distinction between system comprehension and code comprehension.

The analysis of the interviews demonstrates that such a distinction indeed exists, and that code comprehension and system comprehension are in fact separate and distinct activities.

System comprehension requires different skills and experience compared to code comprehension. System comprehension involves both top-down and bottom-up comprehension strategies, and developers apply different strategies depending on the tasks they need to perform. System comprehension shifts focus from the code to its structure, and to considerations that are not reflected in the code at all. It is a continuous, iterative effort. Not all skilled software developers have the skills required for system comprehension, but these skills are highly required by at least a portion of the development team.

One of our results was to extend the conventional distinction between black-box and white-box comprehension. Based on the interviews, we identify two additional levels of comprehension as summarized in Table 7. The "unboxable" comprehension, in particular, is especially relevant to system understanding, as it refers to knowledge that cannot be extracted from the code itself. Additional interesting insights are listed in Fig. 4.

As an exploratory research, we offer models that can be used in future quantitative research. For example, in the context of code comprehension, it would be interesting to investigate unboxable comprehension. One aspect of this is the assessment of the detrimental effects (if any) of not knowing implicit assumptions. Future research could suggest better programming language constructs to make such assumptions explicit, or methods for identifying explicit assumptions and documenting them in order to improve effectiveness. Likewise, to gain a better focus on code comprehension per se, it is needed to be able to distinguish between comprehension problems due to the code, and comprehension problems due to lacking domain knowledge.

In addition, the identification of out-of-the-box comprehension (required for optimization) opens the door for new empirical research on encapsulation. Instances of maintenance that require out-of-the-box comprehension are indicators of leaks, where the required knowledge is not properly encapsulated. Assuming these are relatively rare, identifying and studying them can be instrumental for better understanding of encapsulation and its limitations.

Another possible research direction is to build controlled experiments that measure some of the findings of this research, such as the effectiveness of code samples for package-level comprehension. One could also explore what attributes allow developers to circumvent the understanding process (like developer's reliability or package popularity). Research in this direction could suggest practical methods for package developers to increase the usability of their packages.

The difference between system understanding and architecture also deserves further study. Our interviewees emphasized the understanding of structural aspects of the system.

Table 7 Levels of understanding in relation to code

Black box	Understanding what the code does (its functionality), as reflected by its API	Needed to use the code
White box	Understanding how the code performs its function	Needed to change the code
Out-of-the-box	Understanding the intricate interactions of given code with other parts of the system	Needed for specific optimizations
Unboxable	Understanding the assumptions and considerations that led to designing the code this way	Needed for non-functional requirements and to avoid introducing new defects

- Comprehension is not a goal but a means for performing a task. Comprehension has a context.
- Low-level code (methods and classes) is not stand-alone, and cannot be fully understood without its program context.
- In reality, black boxes often leak. Code may have unintended dependencies.
- Understanding flows provides a deeper level of comprehension than knowing the code's structure.
- Regression tests can serve as "live" documentation: they are always up to date.
- Comprehensive unit tests can save the need to fully understand changed code.
- Trust in the source of imported code can replace the need to understand it.
- Three main levels of understanding code are use, change, and optimize.
- Bottom-up comprehension can be used for small amounts of code, but large scope requires top-down comprehension.
- There are things that are important to know that cannot be learned from the code. Example: developer intent.
- Understanding a system does not necessarily require understanding its code. However, code issues may affect the whole system design.
- A large-scale system cannot be fully understood.

Fig. 4 Top insights identified in the interviews with experienced developers. Note that these are not necessarily universal, and may represent thought provoking opinions

But Kruchten's 4+1 model for viewing system architecture suggests that system aspects even further removed from the code should also be considered, including system states, development process management, and physical infrastructure. It would be interesting to survey active developers about the relevance of all these additional aspects of the architecture in their routine work. Ultimately this research direction could help standardize the ways architecture is developed, documented, and communicated to programmers or users.

Finally, the world of open source is touched upon in this paper, indicating that there may be differences in system comprehension compared to the industry. But further research is required to understand system comprehension in the open source world. This is especially interesting given that open source development is typically more evolutionary, less planned, and less documented than industrial software. It therefore seems to present a greater challenge for system comprehension.

Acknowledgments Many thanks to Neta Kligler-Vilenchik who provided us with invaluable guidance in the methodology of text analysis. Bareket Henle assisted with development of the initial interview plan.

References

Ajami S, Woodbridge Y, Feitelson DG (2019) Syntax, predicates, idioms — what really affects code complexity? Empirical Softw Eng 24(1):287–328

Alomari HW, Jennings RA, Virote de Souza P, Stephen M, Gannod GC (2016) vizSlice: Visualizing large scale software slices. In: IEEE Working Conf. Softw. Visualization, pp 101–105

Alon U, Brody S, Levy O, Yahav E (2019) Code2seq: Generating sequences from structured representations of code. In: International conference learning representations (7)

Alpern B, Schneider FB (1987) Recognizing safety and liveness. Distributed Comput 2(3):117–126

Arisholm E, Briand LC, Hove SE, Labiche Y (2006) The impact of UML documentation on software maintenance: An experimental evaluation. IEEE Trans Softw Eng 32(6):365–381

Austin MA, Samadzadeh MH (2005) Software comprehension/maintenance: An introductory course. In: Proc. 18th International conference systems engineering. IEEE, pp 414–419

Avidan E, Feitelson DG (2017) Effects of variable names on comprehension: An empirical study. In: International conference program comprehension (25)55–65

Bach MJ (1986) The Design of the UNIX operating system. Prentice-Hall

Banker RD, Datar SM, Kemerer CF, Zweig D (1993) Software complexity and maintenance costs. Comm ACM 36(11):81–94

Beniamini G, Gingichashvili S, Klein Orbach A, Feitelson DG (2017) Meaningful identifier names: The case of single-letter variables. In: International conference program comprehension (25)45–54

Bogner A, Menz W (2009) The theory-generating expert interview: Epistemological interest, forms of knowledge, interaction. In: Bogner A, Littig B, Menz W (eds) Interviewing experts. Palgrave Macmillan, pp 43–80

Brooks FP Jr (1987) No silver bullet: Essence and accidents of software engineering. Computer 20(4):10–19

Brooks R (1983) Towards a theory of the comprehension of computer programs. Intl J Man-Machine Studies 18(6):543–554

Brunner T, Porkoláb Z (2019) The role of the version control information in code comprehension. In: IEEE International science and technology conference. Informatics (15)219–224

Carter N, Bryant-Lukosius D, DiCenso A, Blythe J, Neville AJ (2014) The use of triangulation in qualitative research. Oncol Nurs Forum 41(5):545–547

Conway ME (1968) How do committees invent. Datamation 14(4):28–31

Cook C, Bregar W, Foote D (1984) A preliminary investigation of the use of the cloze procedure as a measure of program understanding. Inf Process Manag 20(1-2):199–208

Cornelissen B, Zaidman A, van Deursen A (2011) A controlled experiment for program comprehension through trace visualization. IEEE Trans Softw Eng 37(3):341–355

Cornelissen B, Zaidman A, van Deursen A, Moonen L, Koschke R (2009) A systematic survey of program comprehension through dynamic analysis. IEEE Trans Softw Eng 35(5):684–702

Crouch M, McKenzie H (2006) The logic of small samples in interview-based qualitative research. Soc Sci Inf 45(4):483–499

Fekete A, Porkoláb Z. (2020) A comprehensive review on software comprehension models. Annales Mathematicae et Informaticae 51:103–111

Feng Y, Dreef K, Jones JA, van Deursen A (2018) Hierarchical abstraction of execution traces for program comprehension. In: International conference. program comprehension (26)86–96

Glaser B, Strauss A (1967) The discovery of grounded theory: Strategies for qualitative research. Sociology Press

Haiduc S, Aponte J, Marcus A (2010) Supporting program comprehension with source code summarization. In: 32nd Intl. conf. softw. eng. 2:223–226

Henry S, Kafura D (1981) Software structure metrics based on information flow. IEEE Trans Softw Eng SE-7(5):510–518

Hwa J, Lee S, Kwon YR (2009) Hierarchical understandability assessment model for large-scale OO system. In: Asia-Pacific softw. Engineering conferences (16)11–18

Jaffe A, Lacomis J, Schwartz EJ, Le Goues C, Vasilescu B (2018) Meaningful variable names for decompiled code: A machine translation approach. In: International conference Program Comprehension (26)

Jbara A, Feitelson DG (2014) On the effect of code regularity on comprehension. In: International conference Program Comprehension (22)189–200

Ko AJ (2017) A three-year participant observation of software startup software evolution. In: International conference Software engineering(39)

Kosar T, Gaberc S, Carver JC, Mernik M (2018) Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments. Empir Softw Eng 23(5):2734–2763

Kozaczynski W, Letovsky S, Ning J (1991) A knowledge-based approach to software system understanding. In: Ann. knowledge-based software engineering conference. (6)162–170

Kruchten P (1995) The 4+1 view model of architecture. IEEE Softw 12(6):42–50

Kruchten P (2004) An ontology of architectural design decisions in software-intensive systems. In: Groningen workshop on software variability management (2)54–61

Kulkarni A (2016) Comprehending source code of large software system for reuse. In: International conference program comprehension. IEEE, (24)1–4

Kulkarni N, Varma V (2017) Perils of opportunistically reusing software module. Software: Practice and Experience 47(7):971–984

Lehman MM (1980) Programs, life cycles, and laws of software evolution. Proc IEEE 68(9):1060–1076

Letovsky S (1987) Cognitive processes in program comprehension. J Syst Softw 7(4):325–339

Levy O, Feitelson DG (2019) Understanding large-scale software – a hierarchical view. In: International conference Program Comprehension (27)283–293

Lions J (1996) Lions' Commentary on UNIX 6th Edition, with Source Code. Annabooks

Littman DC, Pinto J, Letovsky S, Soloway E (1987) Mental models and software maintenance. J Syst Softw 7(4):341–355

Maletic JI, Marcus A, Collard ML (2002) A task oriented view of software visualization. In: International Workshop visualizing software for understanding and analysis (1)32–40

Maletic JI, Mosora DJ, Newman CD, Collard ML, Sutton A, Robinson BP (2011) MosaiCode: Visualizing large scale software. In: International Workshop visualizing software for understanding & analysis(6)

Martin RC (2015) Expecting professionalism. https://youtu.be/BSaAMQVq01E?t=2102. Accessed: 2020-05-15

McCabe T (1976) A complexity measure. IEEE Trans Softw Eng SE-2(4):308–320

McKusick MK, Bostic K, Karels MJ, Quarterman JS (1996) The design and implementation of the 4.4BSD operating system. Addison Wesley

Medeirios F, Lima G, Amaral G, Apel S, Kästner C, Ribeiro M, Gheyi R (2019) An investigation of misunderstanding code patterns in C open-source software projects. Empirical Softw Eng 24(4):1693–1726

Metz S (2014) All the little things. https://www.youtube.com/watch?v=8bZh5LMaSmE. Accessed 11 Aug 2018

Meyers B (1992) Applying "design by contract". Computer 25(10):40–51

Moonen L, Yazdanshenas AR (2016) Analyzing and visualizing information flow in heterogeneous component-based software systems. Inf Softw Tech 77:34–55

Panas T, Epperly T, Quinlan D, Sæbjørnsen A, Vuduc R (2007) Communicating software architecture using a unified single-view visualization. In: IEEE International conference Engineering Complex Comput. Syst. (12) 217–228

Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Comm ACM 15(12):1053–1058

Parnas DL, Clements PC, Weiss DM (1985) The modular structure of complex systems. IEEE Trans Softw Eng SE-11(3):259–266

Petersen K, Badampudi D, Shah SMA, Wnuk K, Gorschek T, Papatheocharous E, Axelsson J, Sentilles S, Crncovic I, Cicchetti A (2018) Choosing component origins for software intensive systems: In-house, COTS, OSS, or outsorcing? — a case survey. IEEE Trans Softw Eng 44(3):237–261

Razavizadeh A, Cimpan S, Verjus H, Ducasse S (2009) Software system understanding via architectural views extraction according to multiple viewpoints. In: On the move to meaningful internet systems: OTM 2009 Workshops, LNCS, vol 5872. Springer, pp 433–442

Rodeghero P, Liu C, McBurney PW, McMillan C (2015) An eye-tracking study of Java programmers and application to source code summarization. IEEE Trans Softw Eng 41(11):1038–1054

Roehm T, Tiarks R, Koschke R, Maalej W (2012) How do professional developers comprehend software? Intl Conf Softw Eng 34:255–265

Sackman H, Erikson WJ, Grant EE (1968) Exploratory experimental studies comparing online and offline programming performance. Comm ACM 11(1):3–11

Salah M, Mancoridis S, Antoniol G, Di Penta M (2006) Scenario-driven dynamic analysis for comprehending large software systems. In: Proc. 10th European conference software maintenance & reengineering. IEEE, p 10

Salvaneschi G, Proksch S, Amann S, Nadi S, Mezini M (2017) On the positive effect of reactive programming on software comprehension: An empirical study. IEEE Trans Softw Eng 43(12):1125–1143

Shneiderman B (1976) Exploratory experiments in programmer behavior. Intl J Comput Info Sci 5(2):123–143

Siegmund J, Brechmann A, Apel S, Kästner C., Liebig J, Leich T, Saake G (2012) Toward measuring program comprehension with functional magnetic resonance imaging. In: Proceedings of the ACM SIGSOFT 20th International symposium on the foundations of software engineering. ACM, p 24

Şora I (2015) Helping program comprehension of large software systems by identifying their most important classes. In: International conference on evaluation of novel approaches to software engineering. Springer, 122–140

Spolsky J (2002) The law of leaky abstractions. https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/. Accessed: 2018-09-26

Storey M.-A. (2006) Theories, tools and research methods in program comprehension. Past, present and future Softw Quality J 14(3):187–208

Störrle H. (2014) On the impact of layout quality to understanding UML diagrams: Size matters. In: International conference on model driven engineering languages and systems. Springer, 518–534

Tichy W (2011) The evidence for design patterns. In: Oram A, Wilson G (eds) Making Software. O'Reilly Media Inc., pp 393–414

Torchiano M, Scanniello G, Ricca F, Reggio G, Leotta M (2017) Do UML object diagrams affect design comprehensibility? results from a family of four controlled experiments. J Vis Languages Comput 41:10–21

von Mayrhauser A, Vans AM (1994) Comprehension processes during large scale maintenance. In: International conference Software Engineering(16)39–48

von Mayrhauser A, Vans AM (1994) Dynamic code cognition behaviors for large scale code. In: Workshop Program Comrehension (3)74–81

von Mayrhauser A, Vans AM (1995) Program comprehension during software maintenance and evolution. Computer 28(8):44–55

von Mayrhauser A, Vans AM (1996) On the role of hypotheses during opportunistic understanding while porting large scale code. In: Workshop Program Comrehension, (4)68–77

von Mayrhauser A, Vans AM (1998) Program understanding behavior during adaptation of large scale software. In: Workshop program comrehension, (6)164–172

von Mayrhauser A, Vans AM, Howe AE (1997) Program understanding behavior during enhancement of large-scale software. J Softw Maintenance Res Pract 9(5):299–327

Weissman L (1974) Psychological complexity of computer programs: An experimental methodology. SIGPLAN Notices 9(6):25–36

Wettel R, Lanza M (2007) Program comprehension through software habitability. In: International conference Program Comprehension, (15)231–240

Wettel R, Lanza M (2007) Visualizing software systems as cities. In: IEEE International workshop visualizing software for understanding & analysis, (4)92–99

Wikipedia (2018) Java package. https://en.wikipedia.org/wiki/Java_package. Accessed 31 Oct

Xia X, Bao L, Lo D, Xing Z, Hassan AE, Li S (2018) Measuring program A large-scale field study with professionals. IEEE Trans Softw Eng 44(10):951–976

Zhang H, Zhao H, Cai W, Liu J, Zhou W (2010) Using the k-core decomposition to analyze the static structure of large-scale software systems. J Supercomput 53(2):352–369

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Omer Levy is a senior software engineer at Intel Corporation Ltd. (Israel), specializing in system software architecture, embedded systems, and security architecture. Omer holds a B.Sc. in Mathematics and Computer Science and M.Sc. in Computer Science from the Hebrew University of Jerusalem, Israel.

Dror Feitelson has been on the faculty of the Department of Computer Science at the Hebrew University of Jerusalem since 1995, and heads the experimental systems lab. His research interests are in experimental computer science and empirical software engineering, with an emphasis on human aspects and experimental methodology. His current research focus is on program comprehension, and in particular on variable naming and what makes names “meaningful”.