Understanding Large-Scale Software – A Hierarchical View

Omer Levy Dror G. Feitelson
Department of Computer Science
The Hebrew University of Jerusalem, 91904 Jerusalem, Israel

***Abstract*—Program comprehension accounts for a large portion of software development costs and effort. The academic literature contains 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 interview experienced developers, architects, and managers in the software industry and open-source community, to uncover the meaning of program comprehension at the system level. The interviews demonstrate, among other things, that system comprehension is detached from code and programming language, and includes scope that is not captured in the code. It focuses on the structure of the system and less on the code itself. This is a continuous, iterative process, which mixes white-box and black-box approaches at different layers of the system, and combines both bottom-up and top-down comprehension strategies.**

I. Introduction

Software maintenance is responsible for the majority of the development costs of a software system [3]. There is a well-established direct relationship between the ability to comprehend the software and the cost of software maintenance [4]. 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 UML architecture diagrams, through modern programming languages, and on to design patterns and coding practices—are all targeted at improving the comprehensibility and maintainability of software, and thus reducing maintenance costs [2], [32].

Research in program comprehension focuses on how developers understand code, by analyzing the cognitive models that are employed in code comprehension [6], [18], [27], suggesting tools and methodologies to improve code comprehension [14], [11], or analyzing the impact of code elements on comprehension [12], [13], [1]. 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. And while the key to understanding a function is understanding the programming language's syntax and the semantics created by the developer, the key to understanding larger volumes of code is understanding abstractions and concepts. This requires understanding that is not embedded in the code itself.

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 small 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, 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 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 affects the need to understand.
- Understanding a system requires a different skill-set compared to the skills required for programming.

A. Terminology

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

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).

- **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 system.
- **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.

B. Related Work

Several recent works measured code comprehension in quantitative experiments using short code snippets. Ajami et al. [1] 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. Beniamini et al. [5] showed that in some contexts single-letter variable names can be meaningful. Other studies performed controlled experiments to measure the impact that trace visualization [9], reactive programming [26] or UML object diagrams [33] have on program comprehension.

Xia et al [35] measure 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. Von-Mayrhauser et al. provide a thorough survey of cognitive models used in program comprehension [34]. They analyze various maintenance tasks, and map each of them to the cognitive models. Razavizadeh et al. [24] suggest aiding comprehension by providing multiple viewpoints to the architecture. Kulkarni performed a case study of comprehending a large software system (500,000 lines of code) 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 [15]. Other works also provided methods for identifying the most important parts of a large software system as a means for system comprehension [25], [28].

Petersen et al. performed a case survey on selection of components to integrate into a system [23]. 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 [16].

Störrle [31] 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.

II. 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 in 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, we looked for tools in the social sciences toolbox. Since some of the basic terminology required to describe system comprehension is missing from software engineering research, as well as models of what such comprehension means, we take an 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 participant's 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 program 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. The interviews, each about an hour long, were recorded and then transcribed.

The analysis of the interviews followed common text analysis procedures. 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 [7]. 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 in order to arrive at a joint analysis of the text.

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.

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.

A. 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 I-A. 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.
- 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 relation to system comprehension.

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

B. 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 corporate 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 corporate 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.

We opted to interview developers on the more senior end of the scale, assuming senior developers have experience with large-scale system and perspective on the comprehension process. The relatively low number of participants is acceptable in this type of research [10], especially given that the interviewer (the first author) had a close association with them being an architect with 18 years experience himself.

III. Depth of Comprehension

A. 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 can be easily 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, and how data flows through those classes.

The difference between these two levels of comprehension strongly relates to the concept of information hiding [21]. 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. For non-trivial bugs, a deeper, more complete level of white-box comprehension is needed.

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". We call this 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]*. This level of comprehension may be required in common scenarios, but its role as part of the comprehension process is often overlooked.

[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.

B. 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. 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.

Comprehending a software component is a hard task, and developers prefer to avoid comprehending as much as possible. Design concepts such as information hiding and modularity were created specifically to shield the developer from having to know about other software components [22]. Code quality and design quality are measured 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]*. Understandability becomes 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]*.

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 method 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. While this is not considered a recommended method (*[YI]*, *[BY]*), 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". A similar approach is promoted by software engineering gurus, e.g. in refactoring tasks [19].

C. Top-Down vs. Bottom-Up

The academic discussion of cognitive models of software comprehension has a long history, circling around two well-known main models: the "top-down" model [6] and the

"bottom-up" model [18]. Several combinations or variations of these models have also been proposed [30]. It has also 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. [15]).

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 would start with the "top-down" approach—get a system overview, 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. 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. Through this 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 [...] of a function, while a single person is not likely to reach the same level of comprehension in an entire system [...] This entire discipline of software engineering is a matter of volume".

D. 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 documentation and function header documentation, Training materials, and Design document. 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 Figure 1. The cumulative distribution functions for the rankings of each element in each of the questions is shown in Figure 2.

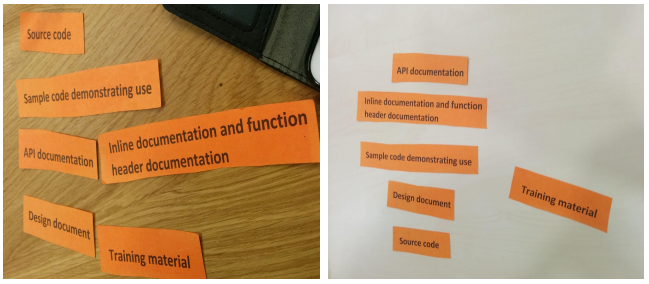


Fig. 1. Example rankings of comprehension aids for functions (left) or classes (right).

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 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, and 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 partic-

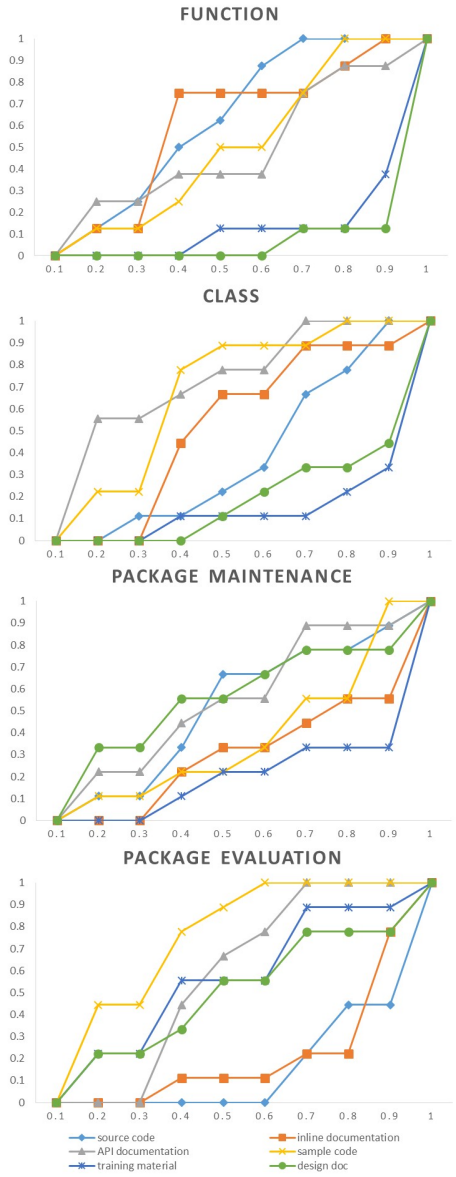


Fig. 2. Cumulative distribution functions for the rankings provided for each comprehension aid in different contexts. The X axis is the ranking, normalized to [0,1] because participants sometimes used different numbers of distinct ranks. Lower X values indicate a higher ranking. An aid appearing at top left of a diagram (e.g. the yellow line for "sample code" in the bottom diagram) was ranked higher than other aids.

ipants 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.
- **Debugging**: debugging helps gets acquainted with 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 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, non-trivial inline documentation is largely considered crucial in understanding 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 is not part of the compiled code, and therefore "my attitude to comments is respect them and suspect them" *[ES]*. In other words, comments may quickly become stale and not in sync with the actual code. Many of the participants therefore refrain from depending on the documentation, and a few of them prefer to avoid reading documentation altogether.

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.

E. 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 is 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 is completely detached from the code itself.

Unfortunately, the skills required for system comprehension—advanced design principles, a top-down approach to systems, managing a large software system—are seldom taught in most software engineering university-level schools, even though it is a critical skill for every software team.

IV. Hierarchical Comprehension

We will now break down the meaning of comprehension to the different levels—function, class, package, and system. 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.

A. 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 comprehension. This is tightly coupled with the idea of information hiding and Meyer's contracts [21], [20]. 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]*.

B. 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]*.

Participants 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 the formal contract. "Why do we use the class, when do we use the class" *[DL]*, "it's 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" [29]. "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.

C. Package

The term "package" is ill-defined. For the following discussion, we define it 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. 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".

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 III-B.

D. System

What is System Comprehension? We asked our research participants 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 includes 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]*.

Another common reference made 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]*. "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 rises—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 of the system, and understanding the rationale of the structure—are completely unrelated to the code. This conclusion is opposed 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.

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.

In addition, sometimes the architecture documentation reflects the original design and not the design that evolved.

One reservation to the above is the law of leaky abstractions (Section IV-B): 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 [17]. 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.

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 I.

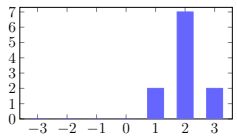
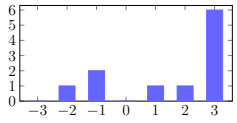
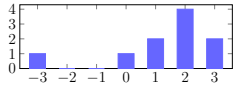
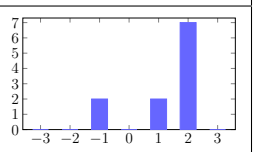
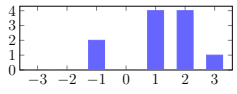
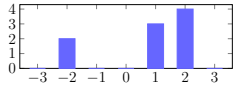
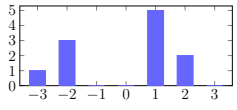
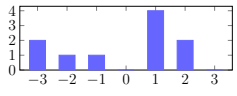
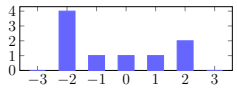
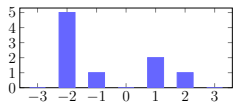
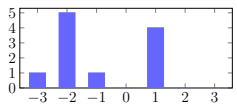
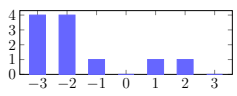
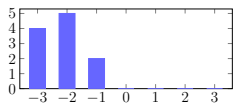
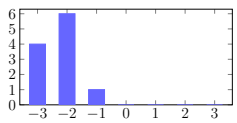
All the participants agreed that every system has an architecture, whether it was designed by a conscious set of decisions or not. In other words, the architecture exists independently of the architecture definition process. The documents and diagrams that support the architecture (hopefully) reflect 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 architecture, and between the design documents and the actual code, may be reasons for the difficulties in comprehending the architecture.

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. 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 decision 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.

TABLE I
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 ($\mu = 2.0$)	
Architecture reflects design decisions ($\mu = 1.545$)	
Architecture is the product of a design process ($\mu = 1.3$)	
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 ($\mu = 0.0$)	
Architecture defines internal interfaces between teams in the development group ($\mu = -0.1$)	
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 ($\mu = -1.64$)	
Architecture is a UML description of the system ($\mu = -2.18$)	
Architecture defines only the internal structure of the system ($\mu = -2.27$)	

Most participants acknowledge that there is some relationship between the architecture and the organizational structure (a soft version of "Conway's Law" [8]). 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.

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.

V. 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 VI). The participants are highly experienced professionals, many working in multi-national companies, who interact with developers from other cultures. Some of their responses were common enough to lead us to believe they can be reproduced in a larger, more representative sample. 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.

VI. 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 existing research which focuses on program comprehension in 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. We show that program comprehension is in fact comprised of two separate and distinct activities—code comprehension and system comprehension. 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 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.

Future Research: 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 (implicit assumptions) and out-of-the-box comprehension (required for optimization). Another possible research could 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).

The world of open source is touched upon in this paper, to show the differences in system comprehension compared to the industry. Further research is required to understand system comprehension in the open source world, which is typically more evolutionary and less planned than industrial software.

Acknowledgements

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

[1] Shulamyt Ajami, Yonatan Woodbridge, and Dror G Feitelson. Syntax, predicates, idioms: what really affects code complexity? In *Proceedings of the 25th International Conference on Program Comprehension*, pages 66–76. IEEE Press, 2017.

[2] Erik Arisholm, Lionel C. Briand, Siw Elisabeth Hove, and Yvan Labiche. The impact of UML documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering*, 32(6):365–381, Jun 2006.

[3] MA Austin and MH Samadzadeh. Software comprehension/maintenance: An introductory course. In *Proceedings of the 18th International Conference on Systems Engineering*, pages 414–419. IEEE, 2005.

[4] Rajiv D Banker, Srikant M Datar, Chris F Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–95, 1993.

[5] Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G Feitelson. Meaningful identifier names: the case of single-letter variables. In *Prodeedings of the 25th International Conference on Program Comprehension*, pages 45–54. IEEE, 2017.

[6] Ruven Brooks. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9(6):737–751, 1977.

[7] Nancy Carter, Denise Bryant-Lukosius, Alba DiCenso, Jennifer Blythe, and Alan J. Neville. The use of triangulation in qualitative research. *Oncology Nursing Forum*, 41(5):545–547, Sept 2014.

[8] Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[9] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 37(3):341–355, 2011.

[10] Mira Crouch and Heather McKenzie. The logic of small samples in interview-based qualitative research. *Social Science Information*, 45(4):483–499, Dec 2006.

[11] Yang Feng, Kaj Dreef, James A Jones, and Arie van Deursen. Hierarchical abstraction of execution traces for program comprehension. In *Proceedings of the 26th International Conference on Program Comprehension*, pages 86–96, 2018.

[12] Alan Jaffe, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th International Conference on Program Comprehension*, 2018.

[13] Ahmad Jbara and Dror G Feitelson. On the effect of code regularity on comprehension. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 189–200. ACM, 2014.

[14] Tomaž Kosar, Sašo Gaberc, Jeffrey C Carver, and Marjan Mernik. Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments. *Empirical Software Engineering*, 23(5):2734–2763, 2018.

[15] Aniket Kulkarni. Comprehending source code of large software system for reuse. In *Proceedings of the 24th International Conference on Program Comprehension*, pages 1–4. IEEE, 2016.

[16] Naveen Kulkarni and Vasudeva Varma. Perils of opportunistically reusing software module. *Software: Practice and Experience*, 47(7):971–984, 2017.

[17] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proc. IEEE*, 68(9):1060–1076, Sep 1980.

[18] Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 7(4):325–339, 1987.

[19] Sandi Metz. All the little things. https://www.youtube.com/watch?v=8bZh5LMaSmE. Accessed: 2018-08-11.

[20] Robert L Meyers III and Debra A Perelman. Risk allocation through indemnity obligations in construction contracts. *SCL Rev.*, 40:989, 1988.

[21] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[22] David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, Mar 1985.

[23] Kai Petersen, Deepika Badampudi, Syed Muhammad Ali Shah, Krzysztof Wnuk, Tony Gorschek, Efi Papatheocharous, Jakob Axelsson, Severine Sentilles, Ivica Crnkovic, and Antonio Cicchetti. Choosing component origins for software intensive systems: In-house, COTS, OSS or outsourcing?—A case survey. *IEEE Transactions on Software Engineering*, 44(3), 2018.

[24] Azadeh Razavizadeh, Sorana Cimpan, Hervé Verjus, and Stéphane Ducasse. Software system understanding via architectural views extraction according to multiple viewpoints. In *On the Move to Meaningful Internet Systems: OTM 2009 Workshops*, pages 433–442. Springer, 2009. LNCS vol. 5872.

[25] Maher Salah, Spiros Mancoridis, Giuliano Antoniol, and Massimiliano Di Penta. Scenario-driven dynamic analysis for comprehending large software systems. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, pages 10–pp. IEEE, 2006.

[26] Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering*, 43(12):1125–1143, 2017.

[27] Janet Siegmund, André Brechmann, Sven Apel, Christian Kästner, Jörg Liebig, Thomas Leich, and Gunter Saake. Toward measuring program comprehension with functional magnetic resonance imaging. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 24. ACM, 2012.

[28] Ioana Şora. Helping program comprehension of large software systems by identifying their most important classes. In *International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 122–140. Springer, 2015.

[29] Joel Spolsky. The law of leaky abstractions. https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/. Accessed: 2018-09-26.

[30] M-A Storey. Theories, methods and tools in program comprehension: Past, present and future. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 181–191. IEEE, 2005.

[31] Harald Störrle. On the impact of layout quality to understanding UML diagrams: Size matters. In *International Conference on Model Driven Engineering Languages and Systems*, pages 518–534. Springer, 2014.

[32] Walter Tichy. The evidence for design patterns. In Andy Oram and Greg Wilson, editors, *Making Software*, pages 393–414. O'Reilly Media Inc., 2011.

[33] Marco Torchiano, Giuseppe Scanniello, Filippo Ricca, Gianna Reggio, and Maurizio Leotta. Do UML object diagrams affect design comprehensibility? results from a family of four controlled experiments. *Journal of Visual Languages & Computing*, 41:10–21, 2017.

[34] Anneliese Von Mayrhauser and A Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, (8):44–55, 1995.

[35] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2018.