The Hebrew University of Jerusalem

Faculty of Science
The Rachel and Selim Benin School of Computer Science and Engineering

Understanding Large-Scale Software: A Hierarchical View

Author: **Omer Levy** (03150130)

Supervisor: **Prof. Dror G. Feitelson**

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science

November 2018

Acknowledgements

First, to my advisor, Prof. Dror G. Feitelson of the Hebrew University, for his devoted and attentive supervision for this project. He guided me through the difficulties of exploring an uncharted territory, and the hours he invested in this project are well beyond what could be expected from a Master's degree supervisor.

Dr. Neta Kligler-Vilenchik of the Hebrew University provided us with invaluable guidance in the methodology of qualitative research. Bareket Henle assisted with development of the initial interview plan.

I would like to thank the 11 participants interviewed in this thesis, and who shall remain nameless. In addition to the time spent in the interview itself, they provided hours of interesting conversation on topics of program comprehension, during and outside the interviews. Special thanks to those who helped arrange the interviews and find the right participants.

Finally, to my family. To my loving spouse Shelly who not only provided never-ending support and encouragement, but also reviewed the drafts and provided insightful comments. And to my beloved daughters, Gali and Inbar, without whom this project would have been completed significantly earlier.

Abstract

Program comprehension accounts for a large portion of software development costs and effort. The academic literature contains research on program comprehension in 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. System comprehension is a continuous, iterative process, which mixes white-box comprehension and black-box comprehension, at different layers of the system. Developers combine both bottom-up and top-down comprehension strategies to understand a system.

Contents

1 Introduction **1**

1.1 Terminology 3

1.2 Related Work 3

2 Methodology **5**

2.1 The Interview Plan 6

2.2 The Participants 7

3 Depth of Comprehension **9**

3.1 Levels of Comprehension 9

3.2 The Desire Not to Comprehend 11

3.3 Top-Down vs. Bottom-Up 13

3.4 Aids to Comprehension 14

3.4.1 Documentation, Sample Code, and Tests 17

3.5 Proof of Comprehension 18

3.6 System Comprehension Specialization 19

4 Hierarchical Comprehension **22**

4.1 Function 22

4.2 Class 24

4.3 Package 27

4.4 System 28

4.4.1 What is a Software System? 28

4.4.2 What is System Comprehension? 29

4.4.3 Architecture 31

5 Threats to Validity **35**

6 Conclusion **36**

6.1 Future Research 36

Appendix A Architecture Statements - Results **38**

References **39**

1 Introduction

Software maintenance is responsible for the majority of the development costs of a software system [2]. There is a well-established direct relationship between the ability to comprehend the software code and the costs of software maintenance [3]. Program comprehension is therefore a key element in the software development life cycle. Indeed, many tools that are available to the software engineer today, as well as the 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.

The research in the area of program comprehension focuses on understanding how developers understand code, either by analyzing the cognitive models that are being employed in code comprehension[6] [15] [23], by suggesting tools and methodologies to improve code comprehension [12] [10], or by analyzing the impact of code elements on the comprehension [11] [1]. Such studies typically focus on short code segments.

When considering comprehension of a single function with a relatively average complexity, it is expected that an expert may be able to evaluate every possible code path, understand the meaning of each variable, and predict the output of the function given a particular input with high success rate. Yet due to the volume of code in a large software system, it is not possible for a single person to understand the entire system in the same way one understands a single function in the code. One person cannot possibly comprehend a software system comprised of hundreds of thousands of lines of code, if not more, at this level of detail. While the key to understanding code 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 is supported by 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 understand the entire system? To what extent do they understand it? 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? There is very little research on these questions.

This thesis attempts to understand what is the meaning of "program comprehension" in the context of understanding a complete, large-scale software system. Through a series of interviews with experienced developers, we try to uncover the differences between understanding a single function to understanding a software system. We study the different levels of comprehension and the different strategies for comprehension that are required for different tasks, the role of documentation in comprehension, and the special skills required for system comprehension. Some of the main findings of the research are:

- System understanding is largely detached from code, it is detached from programming language, and it 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 at different layers and several comprehension strategies.
- There is a mutual interplay between software comprehensibility and software quality. Software quality is partially defined by the comprehensibility of the software components. Conversely, quality impacts the comprehensibility and the need to understand as well.
- Understanding a system requires a different skill-set compared to the skills required for programming.

1.1 Terminology

Throughout the thesis 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). In this research, we refer to a class as a simplified model of a collection of functions (methods) with common state variables (data members), and with a differentiation between public methods that can be used from outside the class and private methods that cannot be used from outside the class. Under this simplified definition, which does not require inheritance, many non-object-oriented programming language allow similar constructs (such as modules in C modular programming) and apply to the discussion below.
- **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. We elaborate on this discussion in section 4.3. 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. We discuss this definition in section 4.4.

1.2 Related Work

Von-Mayrhauser et al. [29] provides a thorough survey of cognitive models used for program comprehension. In analyzes the various maintenance tasks, and maps each of them to the cognitive models.

There has been some recent work in measuring code comprehension in quantitative experiments in short code snippets. Ajami et al. [1] measured comprehension by measuring 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. [4] performed a controlled experiment that showed that in some contexts, single letter variable names can be meaningful. Similar studies perform controlled experiments measure the impact that trace visualization [9], reactive programming [21] or UML object diagrams [28] have on program comprehension.

On large-scale software, Petersen et al. [19] performed a case survey on selection of components to integrate into a system. 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 [14] also discuss problems with package reuse practices and the importance of structured decisions. [20] and later [24] provided methods for identifying the important parts of a large software system as a means for system comprehension.

Kulkarni [13] 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 understand the system, and eventually located about 25,000 lines of code that were critical to understanding the entire system.

Störrle [27] performed a controlled experiment on UML comprehension, and concluded 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.

Börstler et al. [5] combined qualitative and quantitative research on how developers perceive code quality. The definitions of quality among the developers that participated in the study are numerous and are mostly comprised of defining the attributes of code quality – readability, structure, testability etc.

2 Methodology

The concept of comprehension is difficult to measure. Several methods for measuring different possible aspects of comprehension exist, and controlled experiments in small-scale code comprehension have also been performed to measure comprehension. However, the meaning of "comprehension" at the software system level is not well-defined. Clarifying the meaning of comprehension at the system level is one of the targets of this research. Given that the research is concerned with human understanding processes and tools, we looked for tools in the toolbox of social sciences. Specifically, we opted to use the tools of qualitative research. Since some of the basic terminology required to described system comprehension are missing from the academic research, as well as models to describe comprehension, we take an approach of an exploratory research in this thesis. Such an approach is targeted to establish those models, and seed future empirical and 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, about 60 minutes long each, were recorded and then transcribed.

The analysis of the interviews followed common text analysis procedures. Both my instructor and I 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 "peer debriefing", and is used in order to increase research validity. We searched for common themes that arose in several places in a single interview or in multiple interviews, including contradictions between those instances. We 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 in the form of ideas that are widely agreed upon. We found that outliers should not be discarded, quite the contrary. We found that experienced developers develop their own models of system comprehension, but rarely discuss them. In some cases, a participant used a wording that shed light on ideas that were latent in our original models and in other interview. Therefore, ideas that are expressed by single individuals 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 research are translations of the original quotes into English, with minimal rephrasing for readability and flow.

2.1 The Interview Plan

The interview is a semi-structured interview which is designed to take about 60 minutes. The main interview structure contains 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 defined a "software system" as a collection of packages that create an end-user product, where some of the packages may be imported from external sources. 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 / flow 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 common methods resonate with actual ways of testing comprehension?
- Importance of various aids for comprehension in the different layers (see section 3.4).
- 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.

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 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 of experience, with programming experience in C and C++, and managerial experience in areas of embedded programming and algorithms; works for company A.
- *[ES]*, a software team manager with 20 years of experience, with 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 a 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.

3 Depth of Comprehension

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. In the context of a single function, this comprehension level means to comprehend the function's prototype. It includes comprehending the arguments (inputs) to the function, and being able to predict the expected output of the function given a certain input. In a class, 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 the data flows between and within those classes.

The difference between these two levels of comprehension strongly relates to the concept of "information hiding" [18]. 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 by the maintainer of the component or anyone who might need to delve into the implementation details of the component.

However, a few participants indicated additional levels of comprehension. *[ES]* pointed to a level where you would comprehend "all sorts of implicit and explicit assumptions that whoever wrote the function had, that are related in some broader context". We call this level of comprehension the **Unboxable Comprehension**. This level of comprehension cannot be reconstructed from the code itself, and requires some other source of knowledge – typically a documentation of intent, or a discussion with the original code developer. In the common case where documentation does not exist or is low-quality or outdated, this level of comprehension is what is really lost when a developer moves away from a development team, and is related to the difficulty in replacing programming personnel. "There are parts in the code that nobody knows very well today. They can still be maintained, if there are bugs you can still fix them, but nobody has the understanding [...] of the philosophy of this specific module" *[BG]*. Such a level of comprehension is required in common scenarios, but its role as part of the comprehension process is overlooked in many cases.

[AO] pointed to a level of comprehension that includes the external interactions of a function beyond the code itself: "... the comprehension of how it is built under the hood. The classic example in C++, is that given a class with virtual functions, how the memory layout looks like. 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 the **Out-of-the-Box Comprehension**. This level of comprehension is required in rather rare conditions, where the level of optimization that is required 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 debugging task, white-box comprehension is required. Some participants stated that white-box comprehension in itself can be performed at several levels. It may be possible to fix simple bugs with only a superficial level of white-box comprehension – for example, a null pointer exception can in many cases be easily traced and fixed without really understanding the code of the function. In non-trivial bugs, a deeper, more 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 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 occurrences the subjects mentioned a desire *not* to comprehend.

Comprehending a software component is a complex task, and the developers prefer to avoid comprehending as much as possible. Design concepts such as information hiding or modularity were created in order to support this desire to comprehend as little as possible: modularity is intended to shield the developer from having to know about other software components. In fact, code quality and design quality are measured by the ability to understand them with minimal effort: "The more understanding a function 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 becomes part of the definition for quality.

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.

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 integrating external software packages into the system, and evaluating such packages for integration. A few subjects indicated some 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 method that allows developers 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 can be seen promoted by software engineering gurus, even in refactoring tasks [16].

3.3 Top-Down vs. Bottom-Up

The academic discussion around the cognitive models used by software developers to comprehend software has a long history, circling around two main models: the "top-down" model [6], which suggests understanding by breaking problems into smaller sub-problems, and continuing to refine as needed; and the "bottom-up" model [15], which starts by locally understanding the details and then builds up the understanding to larger and larger concepts. Several combinations or variations of the top-down and bottom-up models have been proposed [26].

Other studies have showed 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 (for example [13]). This conclusion is confirmed in the interviews. Moreover, the interviews show a pattern of usage for these approaches. Activities related to comprehending functions or classes were mostly associated with the "bottom-up" approach – reading the code, refactoring the code etc. But when asked about comprehending a package or a system, the participants described activities that are more related to a "top-down" approach.

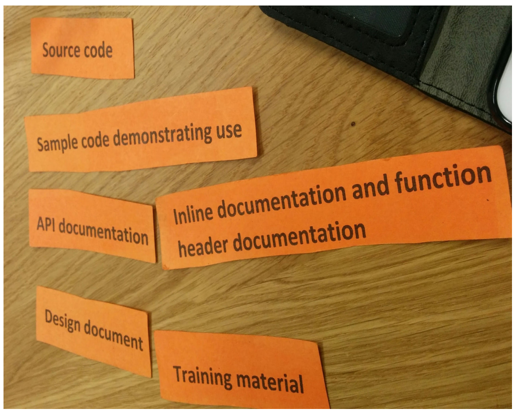
When asked how they would recommend a new member of the team comprehend the system, many participants described a process where they first try to establish a "top-down" understanding of the system – provide the new member an overview of the system, encourage them to try to run sample code, read the architecture documentation if available, trace the code from the main loops and review the interfaces of the first level of classes or modules. However, this approach has its limitations. The amount of new information for a newcomer at this stage is overwhelming, and it is hard to understand without spending some time "in the trenches", developing and debugging code. The participants therefore suggest to assign a small "bottom-up" task to the new member related to some sub-component. This could be fixing a bug, adding a small feature, or studying a piece of code and generating appropriate documentation. Through this task, the new member develops familiarity with the sub-component and its related sub-components, and gradually develops comprehension of that sub-component and its place in the system. In an iterative fashion, the developer then gets familiar with more components and combines it with the "top-down" comprehension of the system. So the learning of the large software component is an iterative process that might span a long period, and it combines switching back and forth between "top-down" and "bottom-up" comprehension activities.

A few participants made statements in the spirit of "I prefer a top-down approach but some people prefer to develop an understanding from the details", 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 discussion above, it appears that the 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, such as writing a function or debugging a class. 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" approaches include understanding of 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 the developer to avoid understanding many of the details by understanding abstractions. As described by one of the participants, "it is more likely that a person knows all the execution possibilities, all the possible states of an execution and reaches full comprehension of a function, while a single person is not likely to reach the same level of comprehension in an entire system with a million lines of code. Then they need to work with abstractions. This entire discipline of software engineering is a matter of volume" *[MN]*.

3.4 Aids to Comprehension

In one of the questions in the interview, 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 material" and "Design document". The participants were asked to organize the paper slips in a way


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

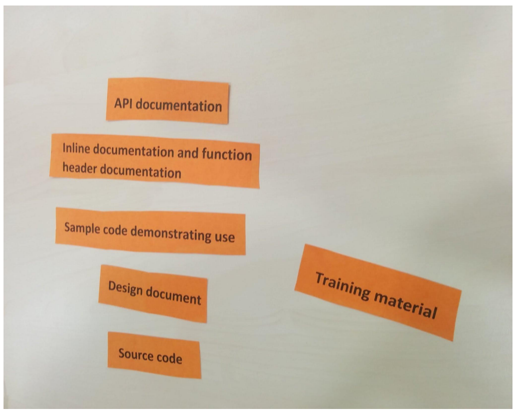

API documentation
Inline documentation and function header documentation
Sample code demonstrating use
Design document
Training material
Source code

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

that shows the relative importance of each of the elements in each of the following tasks: understanding a function; understanding a class; understanding a package for the purpose of using it, as part of a package evaluation effort; and understanding a package for the purpose of maintaining it. Examples are shown in Figure 1.

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 is more efficient for comprehension. 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 in the API and methods' contracts, and understanding these contracts is the most important part of the 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

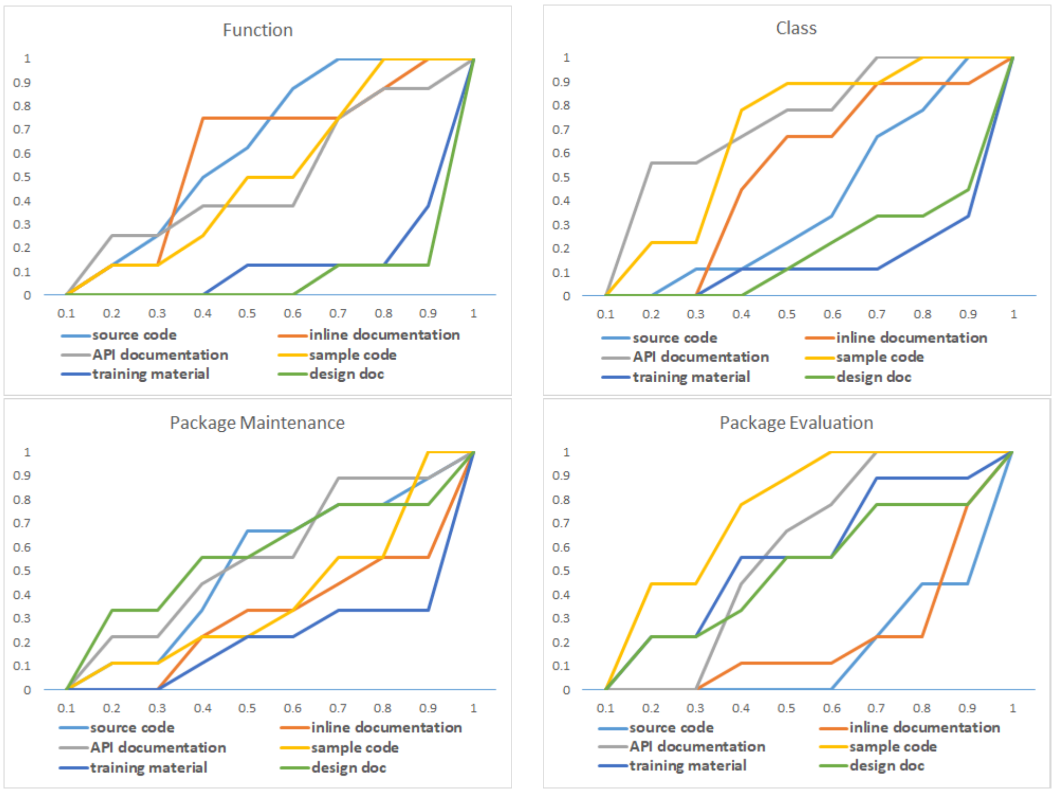


Figure 2: Cumulative distribution functions for the rankings provided for each element in each question. The x axis is the ranking - lower x values indicates that the participants ranked the element in a higher importance. So an element appearing at the top in the left side of each diagram (for example, the gray line for "API documentation" in the class diagram), is an element that was generally ranked higher than other elements.

design document. When asked about understanding a package for evaluation purposes, the most significant element was "sample code demonstrating use". Far behind, "training material" comes in second place, and "design document" as a close third. Here we see that the task at hand has an impact on the comprehension process.

The cumulative distribution functions for the rankings of each element in each of the questions is shown in 2.

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**: having consistent coding conventions and naming 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 simple tasks such as fixing a simple bug or adding a simple feature can help the developers focus on the important parts of the code, especially when addressing larger volumes of code. Creating documentation that is missing or explaining the code to others is also considered a task that focuses the developers and builds 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]*.

- **Drawing analogies to familiar things**: as part of understanding a large amount of code.

- **Exercising the code**: calling the code while conjecturing its expected behavior. Adding tests to a component is a particular way of doing this *[ES]*.

3.4.1 Documentation, Sample Code, and Tests

The discussion around the role of documentation in the comprehension process generated ambiguous sentiments by the participants. On one hand, the participants acknowledge the fact that understanding a software system from the code alone is a difficult process. Many assumptions made by the developers are not embedded 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 seen 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. Event in functions that could be really trivial like arithmetic operations, you always have corner cases" *[YI]*.

On the other hand, documentation is not part of the compiled code, and therefore "respect for comments should be taken with a grain of salt" *[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 provides code that compiles and demonstrates how a software component should be used, and is a good source for 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 actual code.

3.5 Proof of Comprehension

The participants were presented with the question of how is it possible to test if 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. The participants were asked which methods resonate with their experience.

Most participants resonated with the functional methods to prove comprehension: describing how a function is used, adding a feature, debugging it, and describing its flow, its interface, or how it is used. Two of the methods offered – rewriting the code from memory and filling in the blanks ("cloze procedure") – stem from cognitive psychology research [22] [8] 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 situation where I would choose even this overkill" *[GA]*).

Additional methods and refinements were proposed by a few participants, such as what could be improved in the function and how *[GA]*, or how would the function be tested *[AO]*.

The participants were then asked to propose methods to test one's comprehension of a package or a system. The variation at the higher levels increases. 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.

3.6 System Comprehension Specialization

At this stage it is apparent that comprehension at the system level is a unique specialization, which we shall call "system comprehension", that is distinct from the baseline specialization of a software developer. It is a combination of a "top-down" understanding of the system, together with some of the relevant "bottom-level" details that might have impact on the top-level breakdown of the system. 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, other than my post-doctorate. [...] My industry background is mostly as an entrepreneur, I started a software company that was pretty large at one stage [...] but I was not involved in the development of code, I was the CTO from algorithmic point of view and as an entrepreneur. [...] The original idea was in fact an architectural idea, so I invented the main system architecture. But it was at a fairly high level and not at the code level, not even at the object level. Just at the idea level. What modules the software has and how you interact with it in general" *[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. There is complete detachment between the architecture and the code itself.

In many cases, however, system comprehension is a discipline derived from the discipline of software engineering. The participants noted that the number of programmers in a single software team that possess a system-level comprehension is one or two, as low as 10-15% of the developers in the entire system. In fact when the system is very large, the knowledge of the entire system may be split among several developers and architects, such that no single person has complete understanding of the entire system. In a typical project structure, each team is responsible for one package in the system, and a technical leader in the team understands the system enough to communicate with the other teams. The participants noted some correlation between system comprehension capabilities and seniority and years of experience.

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%.

Unfortunately, the skills required for system comprehension – advanced design principles, top-down approaches 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.

4 Hierarchical Comprehension

This chapter will now break down the meaning of comprehension to the different levels – function, class, package, and system. Although there are a few concepts that are similar at all levels, such as the distinction between black-box comprehension and white-box comprehension, there are important differences that help shed a light on the meaning of system comprehension.

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 a 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 you sometimes do evaluate a function outside of its context.

Most participants defined understanding a function as understanding the "contract" defining the function: understanding its parameters, its return values *[AK]*, its name, and its functionality *[SN]*. "A good name is one of the most difficult things in computer science, but in the end if the name is good enough it already gives you some confidence" *[YI]*. 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. *[MN]* described this as "effectively being able to describe in human language the behavior of the function without describing the implementation". This is tightly coupled with the idea of information hiding and Meyer's contracts [17].

The participants described white-box comprehension as the "deeper" level of comprehension. This means "understanding how the function does what is does" *[SN]*, "understanding how it works and does what it is supposed to do" *[AK]* or "understanding the different operations that the function performs including their order and the reason for that order" *[DL]*. *[ES]* separated the white-box comprehension into two levels – a superficial level which is enough to do simple technical debugging, i.e. find a null pointer reference and fix it, and a deeper level as required for a full redesign.

The level of understanding that is required depends on the task at hand, what one is planning to do with the function. "First I look at the function's signature. Its documentation, if available. The function's name, the in and out parameters. If I only want to use the function that should usually suffice. If I want to maintain it I need to read it's code, understand all of the possible flows, like how the parameters are used, what the function does with these parameters, what does it do with the state, what are the possible return values and when is each one returned" *[AK]*. Indeed 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 there are a few properties that impact the "black-box" behavior of the function, but are in many cases overlooked as being part of the function's contract by the developer. This forces another developer wanting to use the function to dive deeper into the implementation details and forces them to defer to "white-box" comprehension.

One such property is the function's side effects, that is, a situation in which the function changes a state that is outside its local environment (and in particular, when the function changes global variables). The participants identify this as a pain point in function comprehension: "Understanding a function includes understanding its side effects" *[DL]*, "The difficulty in function is side effects" *[MN]*. Indirect side effects can be even harder to track, such as when a function uses resources from a global resource pool (allocates memory or uses operating system resources). "Even a well-packaged black-box that appears to work in a vacuum, in the end it can impact lots of other parts of the system without intending to do so. They could for example exhaust a global thread pool that you depend on and cause all sorts of deadlocks" *[YI]*.

Another property of a function that impacts its black-box behavior but is often overlooked relates to 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. *[YI]* described this as an example of a property that ideally would be part of the function's signature and supported as a guarantee of the programming language.

A more subtle function property that is often overlooked relates to performance. Consider for example a function that implements a core algorithm. In this case, 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. Consider, for example, a pseudo-random number generator function that generates one bit at a time. At a normal usage the performance is negligible and therefore the performance is not considered part of the function's contract. However when put in a system that uses many 1024-bit pseudo-random numbers, this function might become a system bottleneck. This forces the user of the function that depends on the pseudo-random generator function's performance to dive deeper to a white-box comprehension. "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]*.

A function can also have properties that affect its white-box comprehension, the code readability. "Sometimes there are functions that are written in a clever way, ...that don't work by most code developers' line of thought. That could impact because you don't know why they did what they did" *[AK]*.

4.2 Class

A class is a collection of functions (object methods) together with internal data members, which are the 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]*. Therefore, the comprehension of a class differs from the comprehension of a function in that "understanding a function is a matter of flow, understanding a class is a matter of state" *[BY]* , and a class implements a state and operations to manipulate that internal state. "Several issues are added in a class [over a function]. One, the issue of state" *[ES]*.

The participants once again distinguished between a black-box comprehension and a white-box comprehension of the class. The black-box comprehension refers to the public interface of the class – its name, its public methods with their contracts, and any documentation around the public interface. Ideally, this level of comprehension should suffice in order to use the class. This notion is strongly coupled with the concept of "information hiding" that is at the core of object oriented programming. So is the concept that "in a class you can implement an aggregation that makes sense, that people understand [...] in many times the objective of classes is to make people understand things" *[BY]*. That is to say, the concept of the class itself, when designed correctly, is targeted towards improving the comprehension of the code. "A class is supposed to do a simple thing, [something that is] hard to implement it but its interface is simple and straight forward" *[GA]* – the complexity is hidden in the implementation details, so as to allow using the class based on minimal black-box comprehension.

Participants mentioned various aspects of the black-box comprehension that are related to understanding beyond the formal contract, and relate to understanding the wider context of the system and the place of the class in the system. "Why do we use the class, when do we use the class" (and also in comparison to alternative solutions that perform similar functionality) *[DL]*, "it's limitations – when can it not be used" *[DL]* , "why is it there, why isn't it part of another object, why isn't it split into two" *[YI]*, "Understand the [class's] role in the system" *[AK]*, "The importance is not to know the class's code, rather that you understand the approach towards the problem" *[BG]*. As *[YI]* coined it, this is all part of the **intent** or the developer that is represented in the class design. The intent is not part of the code. Ideally it would be part of a design document, although as stated earlier, "the [design document] is typically not trustworthy" *[AO]*. This is a particular example of what was earlier called "unboxable comprehension".

It is interesting to look at the comprehension of a method, a function within a class. As discussed in the previous paragraph, one of the attributes of a good class design is that it can not be logically split in two. Its components are not independent – methods and data members know of each other and interact. As such, "a class is a black box that does something, but as opposed to a function [...] it has internal state, [...] it's a box that behaves more independently than a function" *[SN]* (a method is related to the class' state, so not completely black-box), "You need to understand something more general about the class to understand a method in it" *[ES]*. The class provides context to the methods and therefore a method cannot be comprehended by the contract alone, without the context of the class. *[AO]* provides an example where a method called "notify" can mean different things in different classes, and hence even the name of the method lives within the context, and you must have the context to comprehend the method. For example, classes that implement the "observable" design pattern may have a "notify" method to call the watcher class callback; in contrast, the Java language uses the "notify" method to wake a single thread that is waiting for a certain object.

Not all methods in the class are created equal – some are more important than others in the context of comprehension. "A class has a shared core, a collection of methods [...] and if you understand the core [...] you can understand them without understanding most of the class. You can know the core and zero other methods" *[GA]*. The constructor is a particular public method that many see as a good entry point to the class comprehension. "The class constructor gives a lot of information on the class behavior" *[YI]*.

Finally, as we go up to the level of a class and beyond, the practice is that it is impossible to maintain complete abstraction of the interfaces from the details. This is Joel Spolsky's "Law of Leaky Abstractions" [25]. "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

We use the term "package" to loosely refer to a collection of classes that provide a cohesive functionality. A few of the participants expressed objection to this definition. *[MN]* mentioned that the term "package" has a meaning in the Java programming language to mean something else [30]. He preferred the use of "deployment package" for the context of application servers. *[AO]* preferred the term "subsystems", in the sense that in a distributed system, multiple subsystems may be communicating with each other. Internally they may each be structured using software packages. *[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.

Indeed the original intent of the term was to depict a first breakdown of the system into software components, some significantly large volume of code that is not the entire system. As *[SN]* mentioned, this may have different names in different environments, and different meanings. Therefore, the challenges presented by the participants to the terminology are viable. In the context of program comprehension and the original intent, the term "package" could be interchangeable with "subsystem", "component", "service", "deployment package", "executable" or something else, all depending on the context of the specific system. We believe that this distinction is not significant to the comprehension process.

From a different direction, a package could be defined as an "atomic unit of reuse". While classes and interfaces are intended to be reusable, the reality is that in many cases classes have inter-dependencies 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 the package can be comprehended outside the context of the system. "Third-party packages are the best example. There is a company that develops packages in the intention that it would be independent of the system, and so usually it is really independent of the system. [...] I think of packages in our system that we developed – 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.

Packages are reused in order to reduce the development effort. In some cases they are reused partially with the intent of making modifications to fit the specific system, but in many cases they are reused "as is". In the latter cases, the reuse also reduces the comprehension effort – the package is comprehended at a black-box level only. This is especially true when the package performs a complex functionality that requires the 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 a significant factor of the evaluation as proxies to the quality of the code, and therefore to the need to comprehend them, as was previously discussed in Section 3.2.

4.4 System

4.4.1 What is a Software System?

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

> 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 definition or parts of it, we did receive several challenges to this definition. First, not all participants agreed that every software system creates an end-user experience, or at least that in some cases this end-user experience is hidden. A recurring example to 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 the previous section, 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".

Lastly, *[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".

Indeed it is difficult to provide a precise definition, but despite the above feedback, we believe that the definition we provided for software system, 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.

4.4.2 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: "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]*. Another response: "We talk not only about input and output, but more about data flow" *[AO]*. Or: "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]*.

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 system code. This conclusion is opposed to the comprehension of lower layers of the system hierarchy, where the reasoning existed as part of the comprehension, but the code and elements of the code, 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 (which is an abstraction provided by the programming language), 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 structure of the system, the history and evolution of the system also play a part in the comprehension. "If you observe for example Thunderbird, you first need to understand why they developed this project. It is very important to understand [...] what was the original objective, what need they tried to address. Would anyone today start this project? [...] You can look at your email in a browser – why do you need a desktop client? This is debatable, but you need to understand why they did this in order to understand the system" *[AR]*. In reality, 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.

In section 4.2 we described "the law of leaky abstractions", stating that details at lower layers tend to leak into the abstractions made at higher layers. This is inevitable as the system gets more complex, and applies to the system level as well. 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. Understanding the system in terms of the system structure as described above does not provide a complete view of the system, only of its structure.

4.4.3 Architecture

The term "architecture" is widely used in the context of system description, but its meaning may differ depending on the context, organizational culture and personal preferences. We presented the participants with statements related to architecture, and asked them to rate their levels of agreement to those statements. The statements ranged from those related to the content of architecture ("architecture is the structure of the system", "architecture defines only the internal structure of the system", "architecture is a UML description of the system"), to statements referring to the architecture definition process ("architecture reflects design decisions", "architecture changes over time as a result of implementation changes"), to general statements about architecture ("every system has an architecture", "architecture reflects the company's organizational structure"). The complete list of statements and the participants' responses are brought in appendix A.

The participants generally agreed that every system has an architecture, whether it was designed by a conscious set of decisions or not. In other words, the architecture is an entity that exists independently of the architecture definition process. The documents and diagrams that support the architecture only reflect the implemented architecture (hopefully) but do not define it. The definition of the architecture is the structure that is actually implemented in the code.

Most participants agreed that having a design phase is important to an effective development process and that the architecture reflects those 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. This constant drift between the conscious decisions and the actual architecture, and the drift between the design documents and the actual code, may be one of the difficulties in comprehending the architecture.

In some environments design decisions are regularly maintained in documents, and in others there is little to no documentation, and it is not maintained over time. In the latter group, the lack of documentation is sometimes regretted by the team members, but in some cases it is a conscious decision to avoid the overhead of documentation and a reflection of the realistic difficulty to keep the documents alive. This topic is especially interesting in the open source community, where the architecture in many cases evolves by multiple community members, and does not reflect a rigorous decision making process. Therefore, there is little architecture documentation and the system is learned by new members solely from the code.

The architecture includes both externally visible attributes of the system, and the internal decomposition of the system and data pathways. In the hardware world there is a distinction between the architecture, which reflects the externally visible attributes of the system, and the micro-architecture, which is the internal design of the system. In some cases, software organizations in large hardware corporations adopt this terminology. However this terminology is not widely acceptable in the software world, and the term "architecture" usually refers to both the external attributes and the internal design.

Architecture consists mostly of design decisions, and the considerations that led to the decisions. Such consideration could be the "philosophy" that drives the system goals *[BG]*. In many cases the considerations that lead to the design decisions are trade-offs between system resources, 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 – all impact the system and should be part of the decision factors on the architecture. Understanding the design decisions and the considerations behind them are key factors in understanding the architecture, and comprehending the system.

Most participants acknowledge that there is some relationship between the architecture and the organizational structure (a soft version of "Conway's Law" [7]). In some cases the architecture 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 in section 4.4.2), appear in the definition of architecture. The structure of the system – the main components and the communication paths between those components – play a significant role in understanding the system, and are 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 acknowledged that understanding the system has more than understanding the architecture. Understanding the system also includes understanding some of the important low-level details of the system. Those details sometimes have an impact on the larger structure of the system, and sometimes do not – and therefore are not part of the architecture definition – but they are significant in understanding the system. For this reason, 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 – reading documentations and API definition; and maintenance tasks – fixing bugs or adding features.

5 Threats to Validity

The surveyed sample of participants interviewed in this research is not a representative sample. The number of participants is small, 10 out of 11 participants are male, all from one country, and from a small sample of companies. However, it should be noted that a representative sample is not a goal of the research, being an exploratory qualitative research. The goal of the research was to raise the appropriate questions and terminology in order to seed future experimental research (see section 6.1). 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 author is himself 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. On the other hand, 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 (my instructor and I) performed the interview analysis separately and reached conclusions only after a joint discussion (this method is known in the qualitative research methodology as "peer debriefing", one of the methods used to increase research validity). The analysis was based on the textual analysis of the interview audio recording transcripts, rather than on personal impressions. This is also a known method to increase the validity ("audit trail"). Nevertheless and as aforementioned, replication studies are required to ratify the results of the research.

6 Conclusion

Program comprehension is an important activity that accounts for a large portion of any software development effort. Software methodologies and tools were developed by the academic and industrial communities in recent decades in order to reduce the effort required for program comprehension, acknowledging its importance.

This thesis attempts to understand the processes that are related to comprehension of a software system as a whole, unlike the 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 have at hand. 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.

6.1 Future Research

As an exploratory research, the thesis offers models that can be used in future quantitative research. For example, the thesis discussed methods to verify whether one comprehended a system. Such methods can be used to measure the percentage of developers that comprehend the system in an existing environment, or to measure the impact on comprehension of elements such as documentation, sample code, and the code itself.

Another possible research could build controlled experiments that measure some of the findings of this research, such as the conditions that require software developers to have a deeper understanding of a system (for example, a performance optimization scenario). It is also possible to measure the conditions where the developers prefer to avoid understanding (for example, package evaluation scenario). One could further explore what attributes allow developers to circumvent the understanding process (such as reliability of the developer or package popularity).

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

A Architecture Statements - Results

We presented the participants with statements related to architecture, and asked them to rate their levels of agreement to those statements. The table below provides the raw data to their responses.

The scale is -3 to 3, where -3 means "strongly disagree" and 3 means "strongly agree". Empty cells indicate that a clear response to the question was not provided as part of the interview.

	GA	ES	AO	SN	AK	MN	YI	DL	BY	BG	AR
Architecture is the structure of the system	-1	2	-1	1	2	2	2	2	2	2	1
Architecture reflects design decisions	-2	3	3	-1	2	-1	1	3	3	3	3
Architecture defines only the internal structure of the system	-3	-2	-2	-2	-3	-2	-2	-1	-3	-2	-3
Every system has an architecture	1	2	2	2	3	2	3	2	1	2	2
Implementation changes over time as a result of architecture changes	-1	1	1	2	1	2	2	2	1	-1	3
Architecture defines only the externally visible properties of the system	-2	-3	-2	2	-3	1	-2	-1	-3	-3	-2
Architecture is defined in documents	1	1	-2	-2	1	1	-2	-1	-2	-3	-2
Architecture is a UML description of the system	-3	-2	-2	-2	-3	-1	-2	-1	-3	-4	-2
Architecture defines internal interfaces between components in the system	1	2		1	2	2	-2	-2	2		1
Architecture changes over time as a result of implementation changes	0	2	1	-2	-2	-2	-2		-1		2
Architecture reflects the company's organizational structure	1	1	2	-2	-2	1	2	1	-3	-2	1
Architecture defines internal interfaces between teams in the development group	-2	1	-1		-3	1	2	2	-3	1	1
Everyone in the development groups is familiar with the architecture	-2	-1	1		-2	-2	2	1	-2		-2
Architecture is the product of a design process	-3	1	2	2	3	2	0	3	1		2

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

[3] 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.

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

[5] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. I know it when i see it perceptions of code quality: ITiCSE'17 working group report. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*, pages 70–85. ACM, 2018.

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

[8] Curtis Cook, William Bregar, and David Foote. A preliminary investigation of the use of the cloze procedure as a measure of program understanding. *Information Processing & Management*, 20(1-2):199–208, 1984.

[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] Yang Feng, Kaj Dreef, James A Jones, and Arie van Deursen. Hierarchical abstraction of execution traces for program comprehension. 2018.

[11] Alan Jaffe, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *International Conference on Program Comprehension (ICPC). ACM*, 2018.

[12] 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*, pages 1–30, 2018.

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

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

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

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

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

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

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

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

[30] Wikipedia. Java package. https://en.wikipedia.org/wiki/Java_package. Accessed: 2018-10-31.