# How Do Developers Approach Their First Bug in an Unfamiliar Code Base? An Exploratory Study of Large Program Comprehension

**Andreas Bexell**
Ericsson | Lund University
andreas.bexell@ericsson.com

**Emma Söderberg**
Lund University
emma.soderberg@cs.lth.se

**Christofer Rydenfält**
Lund University
christofer.rydenfalt@design.lth.se

**Sigrid Eldh**
Ericsson | Mälardalen Uni. | Carleton Uni.
sigrid.eldh@ericsson.com

## Abstract

Program comprehension is a significant part of developing software; studies suggest that developers spend 50-70% of their time comprehending program code. Program comprehension and code comprehension have been the topics of numerous research studies, but the vast majority of these studies focus on the comprehensibility of statements or functions, and give little guidance for how to support comprehension of the large programs common in the industry.

In this paper, we present an exploratory study focusing on practitioners' comprehension of large programs in the context of approaching their first bug in an unfamiliar code base. We carried out a study where we interviewed five professional programmers with experience in software development of large programs. The interviews were focused on the subjects' attitudes, their strategies, frustrations they experienced and any opportunities in this area. We found that our participants employ several different strategies, including for example reproduction, localization and simulation, when approaching an unfamiliar code base. We see a potential relationship between these strategies and factors such as professional experience. Our results indicate that large program comprehension may be a fruitful area for further study, and we outline and discuss some of these opportunities.

## 1. Introduction

Program comprehension is an important part of the work of a programmer. Studies based on instrumentation of tools used by practitioners report time spent on program comprehension in the range of approximately 58% (Xia et al., 2018) to 70% (Minelli, Mocci, & Lanza, 2015). Program comprehension takes substantial time while developing software. Improved assistance in this area has the potential to create a significant impact for programmers.

Understanding large programs poses additional challenges to understanding medium-sized programs or individual statements of code. It is reasonable for the goals of many program comprehension studies to address code snippets, which have the size of approximately 10-100 lines of source code (LoC) and can be viewed as self-contained programs. However, there is little guidance to be found when seeking to understand more about the comprehension of larger programs typically found in the industry. How do programmers orient themselves in such large systems? For instance, Von Mayrhauser et al. (Von Mayrhauser, Vans, & Howe, 1997) describe comprehension of sufficiently large programs as *"understanding will, of necessity, be partial"*. They further question whether code comprehension studies of novices working on a general understanding of small programs apply to *"professional maintenance tasks [...] on large scale software"*. We believe that it is reasonable that comprehension of large programs may differ from comprehension of code snippets, and that this is an area worthy of more study.

In this study, we specifically address comprehension of large programs, where the size of the program could for example be the Linux kernel that consists of 30 MLoC. Such a large program has complex dependencies and relations between components, which is necessary and a prerequisite to comprehend before many programming tasks can be carried out. Programmers often need to understand large programs in unfamiliar code bases numerous times during their careers, e.g., when joining a new team or integrating with an unfamiliar system. We address the research question (**RQ**) *How do professional programmers approach their first bug in an unfamiliar large program?* We seek an answer to this question via semi-structured interviews with five practitioners, where we focus on attitudes towards large program comprehension, strategies, frustrations, and opportunities for improvement.

Our contribution is that we find several different strategies employed by these practitioners when starting to comprehend a large program in an unfamiliar code base. We also find indications that strategies change and evolve with experience. Furthermore, we find that there may be an untapped potential for improved tools support for large program comprehension.

The rest of this paper is structured as follows: we start with a description of the method in Section 3, then the results in Section 4, before we cover threats in Section 5, discuss our findings in relation to related work in Section 6, and conclude in Section 7.

## 2. Related work

Approaching one's first bug in an unfamiliar large program is an activity that requires acquiring a certain level of comprehension of the program and its structure, as well as being able to locate the bug in it. In this section, we present an overview of related work in the areas of program comprehension and code comprehension, with a brief comparison between these closely related topics. Next, we present related work on bug (fault) localization.

### 2.1. Program Comprehension & Code Comprehension

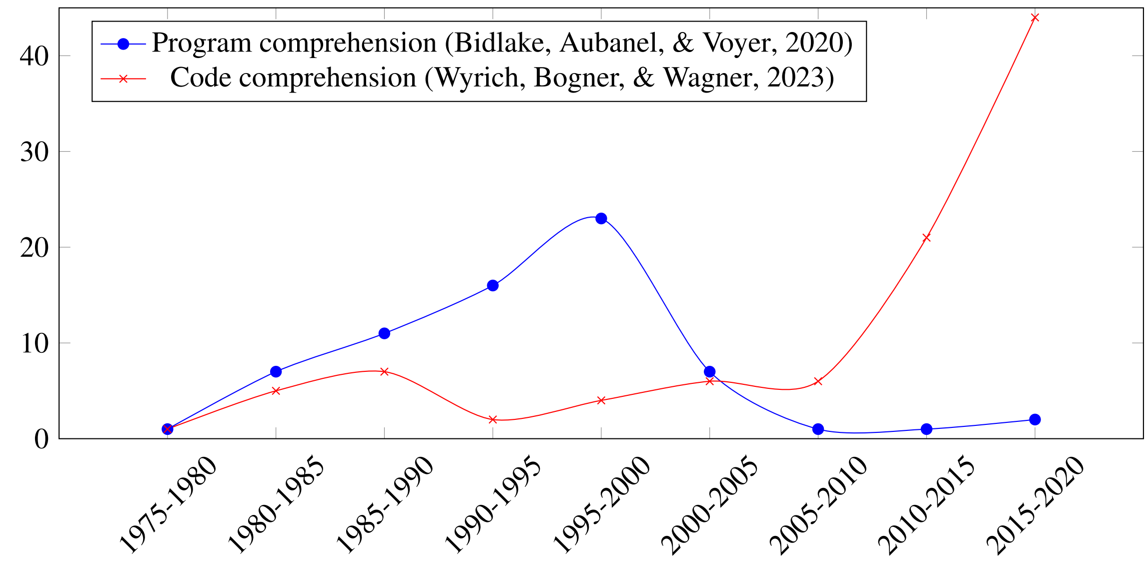

*Figure 1 – A graph of publication dates of included papers in the meta-studies of "mental representations of programs" and "code comprehension" indicate a turn in research attention from the first towards the latter.*

Studies on program comprehension tend to focus on the programmers' mental models and understanding of an entire program (Bidlake et al., 2020), while code comprehension studies tend to use code snippets selected to fit the needs of the study, for instance, to strike a balance between simplicity and complexity (Wyrich et al., 2023). Recent mapping studies of program and code comprehension indicate a turn in the field of software comprehension from mental representations of programs, reported by Bidlake et al. (Bidlake et al., 2020), to code comprehension, reported by Wyrich et al. (Wyrich et al., 2023); see Figure 1. This study falls in the domain of program comprehension, rather than in code comprehension, with a special focus on large programs.

Siegmund (Siegmund, 2016) summarizes program comprehension to these points; (1) getting an overview of a large program or software architecture, (2) understanding type structures and call hierarchies, (3) understanding the relationship between components, and (4) identifying the developers who are responsible for a component. Our study focuses on the first and third items.

Siegmund describes the top-down comprehension model, bottom-up comprehension model, and the combination of the two in the integrated comprehension model. She notes that if programmers are encountering areas they are not familiar with, they resort to executing the program sequence by sequence. However, she does not explore how programmers create their own orientation in the large program, and how they explore unfamiliar areas in the program to solve a particular task.

Kulkarni provides a case description by narrating their experience of starting work with a new large code base. (Kulkarni, 2016). They reflect on their use of bottom-up and top-down comprehension, and on their conscious use of Brook's, Soloway's and Shneiderman's models (Brooks, 1978; Soloway & Ehrlich, 1984; Shneiderman & Mayer, 1979) Kulkarni concludes that no one strategy is all-embracing, but that several approaches are needed.

Wuilmart et al. (Wuilmart, Söderberg, & Höst, 2023) interview four professionally active programmers and indicate that to gain sufficient comprehension of a large program, programmers may need to consult with other programmers and sources outside of the source code. They further found that the interviewed programmers would typically approach a new code base top-down followed by experimentation with different ways of running the program.

### 2.2. Bug localization and Fault localization

When beginning work with a new code base, a common problem is to find a good starting place. (Tempero & Ralph, 2018) discusses the "where to start" question. They note that most "program comprehension" research is done on an "implementation" level – the understanding of statements. This is contrasted with "design comprehension", as in understanding the internal relations of a program.

Locating a bug in a large program requires specific strategies. This has been studied by Katz and Andersson (Katz & Anderson, 1987), by Vessey (Vessey, 1985), and Decasse and Emde (Decasse & Emde, 1988), who each enumerate a set of strategies employed by programmers when locating bugs in software programs. Later, Romero et al. (Romero, Du Boulay, Cox, Lutz, & Bryant, 2007) has coded strategies from observations of programmers' behaviour when debugging software.

In the 1990s, research on bug localization took a turn for the study of automatic bug localization techniques see (Wong, Gao, Li, Abreu, & Wotawa, 2016) and (Wang, Galster, & Morales-Trujillo, 2023), and new theories on strategies for human debuggers after that are scarce. The gap between recent graduates' experience with large code bases compared to the industry's expectations is addressed in (Shah, Yu, Tong, & Raj, 2024). The fact that there is an expectation from the industry may indicate that experienced software developers engage with unfamiliar large code in a manner different from students. We study professionally active developers working with large programs.

Collecting data from 102 professional developers, Hirsch and Hofer suggest that localizing bugs in programs is more time-consuming than fixing them. They conclude that more research may be needed specifically on bug localization. (Hirsch & Hofer, 2021)

Alaboudi and LaToza perform an analysis of the activities developers engage in during video-recorded debugging sessions (Alaboudi & LaToza, 2023). This results in a comprehensible enumeration of what observable activities programmers engage in (*Edit, Navigate, Test*). This study collects retold experience, and may additionally capture some of the strategies that may result in the activities enumerated by Alaboudi and LaToza.

## 3. Method

To address our research question, we designed an interview study focused on the following topics in relation to large program comprehension; the *attitudes* with which programmers approach an unfamiliar code base, what *strategies* they employ to orient themselves to solve their first bug there, what *frustra-*

*Table 1 – The mapping of topics and questions in the interview protocol. See also Appendix A.*

| **Topic** | **Question(s)** |
|---|---|
| Attitudes | *"[...] how do you feel?"* |
| Strategies | *"What do you do [...]?"* |
| | *"What strategies do you employ [...]?"* |
| | *"What tools do you employ?"* |
| Frustrations | *"What trouble do you get into?"* |
| Opportunities | *"What would you want [...]?"* |

*Table 2 – Overview of interview participants. Exp - years of professional programming experience, CB - number of new code bases the participant has been introduced to, Lang - number of programming languages the participant has professional capacity in.*

| **Participant** | **Age** | **Gender** | **Exp** | **CB** | **Lang** |
|---|---|---|---|---|---|
| S1 | 40 | non-binary | 27 | 15 | 15 |
| S2 | 47 | male | 20 | 15 | 5 |
| S3 | 43 | male | 18 | 30 | 5 |
| S4 | 26 | female | 2 | 2 | 6 |
| S5 | 37 | male | 22 | 20 | 5 |
| *Avg/Med* | 38/43 | | 17/22 | 16/15 | 7/5 |

*tions* they encounter, and what *opportunities* for improvement they identify.

### 3.1. Data Collection

We used *semi-structured interviews* (Robson, 2011) as a means to collect the data needed to address our research question. The interview protocol was designed to enable *directed content analysis* (Hsieh & Shannon, 2005) to collect information about the participants' professional experience and their experience of large program comprehension with a focus on attitudes, strategies, frustrations, and opportunities (Table 1). The interview protocol is included in Appendix A.

We recruited five participants, described in Table 2. The participants were all programmers with 2-27 years of professional software development experience from more than one company. All five have started working with new large programs at least twice during their respective careers. All of them are Swedish and have worked in large software development companies in Sweden.

Before the interviews, the participants were informed of the topic of the study and the members of the research group. Participants were promised that any publications would be in a manner so that the identity of the participants would be protected. The raw data is available only to external auditors. The participants were informed of their right to opt-out of the study at any time, without penalty. Each of the participants signed a consent form to this effect.

The interviews were executed in Swedish. They were conducted via remote link with audio and video. The interviews were recorded (except the first interview, when the recording failed). In addition, physical notes were taken during the interviews.

### 3.2. Data Analysis

Data extraction was carried out by the first author. The recordings were automatically transcribed using Microsoft Teams auto-transcription. The transcriptions were used as a guide to listen to the recordings to extract quotes according to directed content analysis, related to the topics in Table 1). Initially, at least one quote per participant and topic was extracted. During this process, the quotes were translated from Swedish to English.

The extracted quotes were reviewed and discussed together with the second author. Especially the

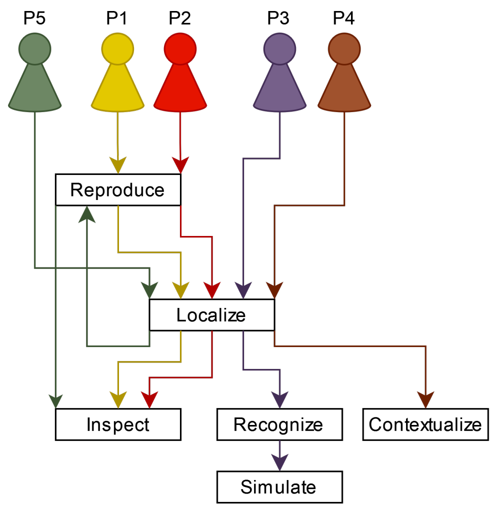

*Figure 2 – The activities mentioned by the participants, with order corresponding to the order in which each activity was mentioned in the interview.*

quotes connected to strategies were reviewed in more detail as it became clear that participants undertake several different activities. The interviews were revisited by the first author to code the activities using *conventional content analysis* (Hsieh & Shannon, 2005). The coded activities of bug localization in an unfamiliar large program were mapped to activities described in the literature (see Table 3).

## 4. Results

Here, we present the results of our interview study. We focus the analysis of the interview material on the four topics presented in Section 3; attitudes, strategies, frustrations, and opportunities.

### 4.1. Attitudes

We found that our participants approach an unfamiliar codebase with a mix of **anticipation and caution**. Several participants described feelings of vulnerability, like a feeling of being overwhelmed (*"It's a mix of fun and overwhelming"*, **S1**) or nervous (*"Until I've checked out [the code] I'm quite calm. [...] That is something I have learned with time because you are very, very nervous. What am I getting into?"*, **S5**). One participant mentioned that a mentor can help to relieve the **feeling of being lost** (*"When you have a mentor, it's easier [...] Otherwise, just 'here's the code base, here's a computer, find out what's happening' that's much more frightening, because I'm, like, what file should I even open first?"*, **S4**).

We also noticed that there appears to be a risk of friction in terms of **expectations** (e.g., *"many places have these - 'we've always done like this' [...] can drive you crazy in the beginning"*, **S1**). To spare some frustration, participants recount strategies to contain expectations, for instance, to refrain from a personal investment to reduce the **feeling of ownership** (*"May be a good idea to cool it in the beginning, to wait and see - there may be a good reason"*, **S1**. The structure of the work may already create some of this distance. For instance, one participant described the role you have as a consultant where you are more a visitor in the code base which may create a different relation to code ownership (*"As a contractor, you have the benefit of not being a part of company politics, but you can be objective and try to solve the problem. The client's problem, that is."*, **S3**).

### 4.2. Strategies

Our participants mention strategies composed of a series of activities, illustrated in Figure 2 with the order they were mentioned by participants. Some of the activities are shared between participants, while others are mentioned to a lesser extent and some are mentioned only by a single participant.

**Reproduce**: To "reproduce" a bug means to be able to provoke and observe a bug, ideally in a predictable manner. This is a common entry point ( *"It's important to know exactly what the bug is [...] So that's the first thing, to find out exactly what the bug results in and see if it can be reproduced."* **S5**). Reproducing a bug can either be by running a **test case** that fails or by following a **specific procedure** ( *"We need to start by reproducing this bug I'm meant to solve [...] best case, there is a failing test case ... or maybe I need to use the product according to some instructions to make it exhibit this problem."* **S1**, *"I mean, it's a bug, it's reproducible. Yes: you press a button or play a video or something [...]. And you see: this happens, that shouldn't. [...] Then I can dig from there."* **S2**). How to reproduce a bug is often a core part of a bug report, and several subjects state they start their understanding of the bug by reproducing it. This might also provide an opportunity to introduce oneself to the interaction with the program.

**Localize**: When finding a bug in a large and unfamiliar code base, it is important to try to shrink the area of investigation to a manageable size. "Localization" is the act of trying to limit the area of interest and demarcate a part of the code that is relevant to a specific task (*"try to come down from the problem is located somewhere in these hundred thousand lines to [...] ... 1000. A workable number"*, **S1**). We find a mix of different approaches to localization: **S1** and **S4** search for **string constants** (*"Search for strings or something [...] they are searchable and somewhat unique. When you don't know naming, schema, folder structures and so on."* **S1**) and **keywords** (*" I search the code base for keywords."* **S4**) in the source code and work from there, in a **bottom-up** approach. **S3** starts with a **top-down** approach from **architectural overviews** but is ready to switch to a bottom-up approach (*"If you're lucky, there's some kind of architecture you can look at [...] Or you have to discuss it with someone or ask [...] otherwise, you'll have to search the files manually"*, **S3**). **S2** focuses on the **data flow** and tries to trace it through the program to localize its source (*"The incoming data is wrong. OK, then we need to find who feeds that data and what is wrong with it."*, **S2**) , while **S5** focuses on the **control flow** and tries to locate a central point of the system to trace their way outwards from there. (*"I mean, a juncture, a manager, whatever where you don't need to go further backwards or forwards. Here I can start working without familiarizing myself with too much."*, **S5**).

**Inspect**: The source code of a software system is only part of what is needed to understand how it behaves. To be able to alter the behaviour of the running software, or identify anomalies, the data the code operates on needs to be observed. To systematically observe a part of the code along with its data, as it runs is to "inspect" the program. The subjects refer to **stepping** and **breakpoints**, that is, using a **debugger** to execute the program one instruction at a time, being able to inspect changes in its state for each step (*"I step and see what value all the variables have and let it run and [...] begin clicking a few buttons to see where the flow takes me."*, **S5**; *"I've got a point before the crash/error/bug manifests. If possible, one can set a breakpoint there"* **S1**). However, using a debugger is not always possible (*"Systems very seldom are like 'well, you just click these buttons in the IDE and run this and then switch to a debug perspective' "*, **S2**). **S1** and **S2** express experiences of situations where a debugger is unavailable and the, in their opinions, less ideal strategies of using **printouts** of partial states while the program runs in a normal manner (*"otherwise, one may go old school and print to trace out some data and see how it changes"*, **S1**; *"Sadly, it often becomes printf. [...] We suspect these variables have the wrong values, and in the end, well, we have to print the values and look at them"*, **S2**).

**Recognize**: Software source code often forms patterns. Many parts of the code may do similar things, and will over time tend to do similar things in similar ways. Common expressions of code are sometimes referred to as "idioms". Some idioms have proved to be based on misunderstandings and may not work as intended; such are sometimes referred to as **"anti-patterns"**. A sufficiently experienced programmer may be able to *recognize* anti-patterns or mistakes in idioms simply by looking at them (*"... And there you say 'This can probably go wrong' [...] 'This is the special case' [...] that's experience, right?"*, **S3**).

**Simulate**: Every programmer needs to make a mental model of the execution flow of a program to be able to understand it. This mental model can range from superficial to detailed. Making an exhaustive analysis of the program flow under different circumstances amounts to *simulating* the program execution in one's mind (*"You more or less run the code in your head, and then you see it."*, **S3**). A programmer with a sufficiently detailed mental representation of the program may be able to simulate several program runs and thereby reveal conditions under which the program executes abnormally or in an undesired manner.

**Contextualize**: To understand what goes wrong in a piece of code, it may be helpful to know in what environment that code is written and how it connects to other parts of the system. A panoramic overview of the *context* of the code can be beneficial to large program comprehension. **S4** collects and documents the context of the source code and draws a **map** for further reference (*"I paint a small picture of that area. Step one is getting an overview of the full code base. [...] What does a flow look like? So it became a small flow chart. On paper."*, **S4**).

*Table 3 – Bug localization strategies*

| | (Katz & Anderson, 1987) | (Vessey, 1985) | (Decasse & Emde, 1988) | (Romero et al., 2007) |
|---|---|---|---|---|
| **Reproduce** | *"testing [the] system"* | *"Determine problem"* | *"checking computational equivalence of intended program and actual one"* | *"Following execution"* |
| **Localize** | *"locating the erroneous component of [the] system"* | *"Locate error"* | *"filtering"* | *"Causal reasoning"* |
| **Simulate** | *"Mentally process data through program"* | | | *"Hand simulation"* |
| **Inspect** | | *"Examine program control"* | | |
| **Contextualize** | | *"Gain familiarity"* | | *"Comprehension"* |
| **Recognize** | | | *"recognizing stereotyped errors"* | |

## 4.3. Frustrations

Two points of frustration stood out in the interviews:

**Tedious Tool Setup**: A common frustration among our participants concerns the tool setup where they expect to encounter trouble, for instance, regarding how to set up a functioning **build process** (*"Oftentimes it may be hard to understand how to even build the software"*, **S3**). A related point of frustration is lacking support for **debugging** (*"It's very uncommon with systems where you just click a few buttons in the IDE, and it starts and runs and you can just switch to a debug perspective."*, **S2**; *"Integrated debuggers are still not standard, surprisingly"*, **S1**), or the need for a **time-consuming setup** to get working debug support (*"Sometimes it takes time to get the break-point functionality up and working, so that the code stops where you are interested. I really miss that"*, **S5**). More generally, developers may feel **abandoned** when it comes to tooling (*"Tools and that kind of stuff are often kind of neglected."*, **S2**).

**Lack of Visualization**: Another point of frustration mentioned was connected to support for visualization and the lack of having up-to-date versions of unified modelling language **(UML) diagrams** (*"The way I see it, to have a UML diagram and keep it updated can for a team be hard"*, **S4**). In some cases a solution may be available but company policies connected to protection of **intellectual property** may prohibit their use (*"I have looked a lot at available [UML generator] tools. But I never quite understand if they upload stuff, you know, they may be online, and then I'm not allowed to use them."*, **S4**).

## 4.4. Opportunities

Related to the frustrations several intervention opportunities emerged from the analysis of the interviews.

**Quick and Easy Developer Tool Setup**: Many of the more experienced programmers in this study have experienced situations where the tool chain is incomplete or dysfunctional. It may be hard to build the project from the IDE, it can be hard to trigger the test cases, or complicated to get code indexation and debugging to work. A more ready-to-go and **simpler development setup** is an area of opportunities for improvement, and might over a sufficiently large developer base provide a good return on investment (*"I mean, pull up visual studio code, check out the code, press the button with the icon and then run in debug mode. [...] I think that would have made things easier."*, **S2**).

**Backward-Stepping Debugging**: A debugger is a tool that allows running a program one step at a time, and allows for inspection and alteration of its state between the steps. Typically, a debugger can only run the program forwards along a single path. **Backing up**, to see for example where a state originated from, is usually not possible. Yet, this could make an appreciated feature (*"I think some kind of runtime that could record an execution. [...] It would be like single-stepping after the fact. [...] Back and forth"*, **S2**).

**Runtime Visualization**: When using a debugger, it is important to continuously evaluate the **state** of the running program, but this could be further aided by tools (*"Some kind of visualization tool [...] like being able to see how the data changes"*, **S1**). Furthermore, it is important to keep track of where, in the source code, the program is currently executing, and what **path** it has taken to get to this point (*"See that OK, we enter this function with erroneous parameters, why is that? Who is calling?"*, **S1**; *"I think the data is wrong, but who is calling with this data? I sort of want to find the call graph, and that can be rather tricky."*, **S2**). Visualizations of a program's runtime paths and state change behaviour could provide an accessible overview and elevate a programmer's understanding of the runtime behaviour of the code.

**Static Visualization**: A programmer working on a large program needs a mental model of how the parts of that program interconnect. When probed about the process of creating such a model, the participants in this study mentioned mental visualizations with relations, in the shape of UML and **block diagrams** (*"You know, a big part is some kind of visualization of the architecture, so you can see how the big blocks are connected ... like a block diagram."*, **S3**). One subject express a wish for a tighter integration of UML into the IDE to be able to use it for navigation (*"In a dream world, I would have an interactive UML diagram that is already in place. [...] Be able to click around in the UML diagram instead."*, **S4**).

**Event Tracking**: An event-based architecture differs from an object-oriented or a functional architecture in that functions may alter their behaviour in response to the properties of the data — "event" — being processed. These events are typically queued before being sent for processing. When looking at the runtime behaviour of such code, we will see multiple calls to the event processing function from the queue handler, but it will often be hard to know where the event was sent from. When debugging, an erroneous event may be identified, but tracing the origin of an event may be hard. Marking events with their **origin** may simplify tracing it (*"An easy way would be. [...] I queue the [event] object, and later when the event gets [processed], you can see the object that created it*, **S5**).

### 4.5. Summary

**RQ**: How do programmers approach the task of resolving their first bug?

We find that different programmers may have different strategies. While there is a significant overlap in the activities they mention, it is clear, even from this small sample, that not all programmers approach bug fixing in an unfamiliar large code base the same way.
We also find that four of the five programmers in our study think they would benefit, in this task, from visualization tools aiding navigation or run time monitoring.

## 5. Threats to Validity

We base our analysis and definitions of threats to validity on Feldt and Magazinius (Feldt & Magazinius, 2010).

*Conclusion validity.* We conduct an exploratory study, meaning our conclusions should be taken as indicators and inspiration for further study, rather than as verifiable truth.

*Construct validity* We explore initial code base comprehension by conducting interviews. This limits the results in several ways: *a)* it only collects strategies that are *conscious*, *b)* it only collects actions the subject can *remember*, *c)* it only collects actions the subject *wants to talk about*. A method to investigate strategies in initial code base comprehension is to observe the programmers' behaviour in the wild, as done in for example (Alaboudi & LaToza, 2023), however, to uncover strategies this may need to include think-aloud methods, that may in turn be invasive and affect the results.

*External validity/transferability.* The interview subjects were arbitrarily selected from the network of the first author in the geographical region of the authors, using convenient sampling. As such, they might be influenced by local factors and practices. The interview study is further quite small: only five subjects. The results from the interviews may not generalize to other contexts. The results of this exploratory study should be considered inspirational, rather than a ground truth of the experiences of a programmer community.

*Credibility.* The credibility of our findings comes down to the credibility of our subjects. As such, the credibility hinges upon the internal validity (in particular response bias) and the construct validity. Since the topic and questions are uncontroversial, we have no reason to believe our subjects have systematically tried to mislead us.

*Dependability.* We are confident that a similar *Problem Exploration* study would lead to similar diversity in results in terms of *attitudes*, *strategies* and *frustrations*, but we expect there is more to find here. A set of subjects with a different background may see or emphasize other *opportunities*.

*Confirmability.* The results from the *Problem Exploration* are influenced by the interview guide, but the answers are the subjects' own. The data analysis is done responsively and reactively, carefully considering what the subjects have said and how the answers correlate.

## 6. Discussion

Approaching a new large code base in a professional setting is an endeavour. The programmers we talked to find it *fun and overwhelming*, but it is also something they describe as *frightening* and that makes them *very, very nervous*. A software programmer needs to learn to orient – to find their way around – the new code base. It is unsurprising, then, that they express the need for *someone to [...] guide* them.

The programmers we have interviewed seem to have well-formed strategies made up of a string of activities they follow when they undertake to fix their first bug in a new code base. Some activities are shared among the programmers, but not all. It is clear, even from this small convenient sample, that not all programmers employ the same strategy.

A common frustration seems to be that current tools are perceived as unreliable – or at least as cumbersome to get and keep in a reliable state. They seem to have limited support for the different strategies employed by the programmers. Indeed, several programmers mention activities specifically executed to work around the limitations in the tool support.

The tools available to the programmers do not feature integrated visualisation to support the construction of mental models or aid navigation in code bases. Reliable integrated visualisation of static and dynamic aspects of the code is requested as a means of facilitating code base comprehension.

### 6.1. Directions for Future Work

We see several possible directions for future work:

**Effective visualizations for large program comprehension**. The experience of starting working with a new code base may be comparable to finding oneself in a new and unfamiliar physical space. To find one's way around a new and unfamiliar landscape, it comes as no surprise that many of the programmers we interviewed want a map to navigate by. What should such a map look like in practice? What kinds of questions should it be capable of answering? While UML is mentioned by some of the subjects in this study, there are more alternatives. Hawes et al. use a territory map metaphor to visualize large code bases and their internal dependencies (Hawes, Marshall, & Anslow, 2015). Mortara et al. use the city as a metaphor (Mortara, Collet, & Dery-Pinna, 2021), and Hori et al. visualize source code structure in as a house (Hori, Kawakami, & Ichii, 2019).

**Understanding the experience of shared code spaces**. With the analogy of travelling, some of the programmers we interviewed plan to stay a while in a new code base, while others clearly see themselves as temporary visitors. Perhaps in relation to this intention, we saw a variation of frustration with a slight tendency towards an increase when the code base is about to become a new "home". We see an interesting parallel to the work by Church et al. (Church, Söderberg, & Höst, 2023), in how ownership plays a role in the relationship to the shared space of a code base. How do programmers relate to code ownership? How does a feeling of code ownership emerge?

**Understanding how experience shapes preferences**. Our participants steered towards UML when asked about what they visualized or wanted to be visualized. With our sample of programmers being concentrated to a certain geographical region, where the likelihood of a similar educational background is high, we speculate that this affinity to UML may be due to a similar educational background. The notion that our experiences in the programming context will bias our preferences, is not new. Meyerovich et al. (Meyerovich & Rabkin, 2012) have studied this aspect in the setting of programming language adoption. How do programming experience influence preferences for programming tools?

**Effect of live programming on code base comprehension**. Live programming (Tanimoto, 2013; Church, Söderberg, Bracha, & Tanimoto, 2016), focused on immediate feedback about program state and runtime information, provides properties that may assist based on what we found in our problem exploration. Our participants expressed a need for visualization and seeing connections between the visualisation and the code. They further expressed a need for debugging support for a deeper understanding of code behavior. Live programming has been found to have a positive effect in an educational setting (Huang, Ferdowsi, Selvaraj, Soosai Raj, & Lerner, 2022). Provided a functioning workflow integration, would live programming support provide effective support for code base comprehension?

### 6.2. Limitations

This paper describes an exploratory study into programmers experiences with initial code base comprehension. The study is based on a small sample from one geographical area and the participants have similar educational and professional backgrounds. To make the study more generalized it would be possible to, for instance, carry out an observational study in the wild with a larger set of participants.

## 7. Conclusions

We have presented the results of an exploratory study focused on understanding code base comprehension. We found that the programmers in our study approached a new code base with a mix of anticipation and dread while expecting a lack of functioning tool support. The most apparent lack in tool support we found concerned easy tool setup, debugging, and visualization support. We find these results encouraging and an indication that code base comprehension is an understudied area worthy of more attention.

## 8. Acknowledgements

This work has been supported by Ericsson AB, the Swedish Foundation for Strategic Research (grant no. FFL18-0231), the Swedish Research Council (grant no. 2019-05658), ELLIIT – the Swedish Strategic Research Area in IT and Mobile Communications, and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Special thanks to Lo Heander and Peng Kuang for inspiration and discussions.

## 9. References

Alaboudi, A., & LaToza, T. D. (2023). What constitutes debugging? an exploratory study of debugging episodes. *Empirical Software Engineering*, *28*(5), 117.

Bidlake, L., Aubanel, E., & Voyer, D. (2020). Systematic literature review of empirical studies on mental representations of programs. *Journal of Systems and Software*, *165*, 110565.

Brooks, R. (1978). Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd international conference on software engineering* (pp. 196–201).

Church, L., Söderberg, E., Bracha, G., & Tanimoto, S. (2016). Liveness becomes entelechy-a scheme for l6. In *The second international conference on live coding.*

Church, L., Söderberg, E., & Höst, M. (2023). My space, our space, their space: A first glance at developers' experience of spaces. In *Companion proceedings of the 7th international conference on the art, science, and engineering of programming* (pp. 48–53).

Decasse, M., & Emde, A.-M. (1988). A review of automated debugging systems: Knowledge, strategies and techniques. In *Proceedings.[1989] 11th international conference on software engineering* (pp. 162–163).

Feldt, R., & Magazinius, A. (2010). Validity threats in empirical software engineering research-an initial survey. In *Seke* (pp. 374–379).

Hawes, N., Marshall, S., & Anslow, C. (2015). Codesurveyor: Mapping large-scale software to aid in code comprehension. In *2015 ieee 3rd working conference on software visualization (vissoft)* (pp. 96–105).

Hirsch, T., & Hofer, B. (2021). What we can learn from how programmers debug their code. In *2021 ieee/acm 8th international workshop on software engineering research and industrial practice (ser&ip)* (pp. 37–40).

Hori, A., Kawakami, M., & Ichii, M. (2019). Code house: Vr code visualization tool [Conference paper]. In (p. 83 – 87).

Hsieh, H.-F., & Shannon, S. E. (2005). Three approaches to qualitative content analysis. *Qualitative Health Research*, *15*(9), 1277-1288. (PMID: 16204405)

Huang, R., Ferdowsi, K., Selvaraj, A., Soosai Raj, A. G., & Lerner, S. (2022). Investigating the impact of using a live programming environment in a cs1 course. In *Proceedings of the 53rd acm technical symposium on computer science education-volume 1* (pp. 495–501).

Katz, I. R., & Anderson, J. R. (1987). Debugging: An analysis of bug-location strategies. *Human-Computer Interaction*, *3*(4), 351–399.

Kulkarni, A. (2016). Comprehending source code of large software system for reuse [Conference paper]. In (Vol. 2016-July).

Meyerovich, L. A., & Rabkin, A. S. (2012). Socio-plt: Principles for programming language adoption. In *Proceedings of the acm international symposium on new ideas, new paradigms, and reflections on programming and software* (pp. 39–54).

Minelli, R., Mocci, A., & Lanza, M. (2015). I know what you did last summer-an investigation of how developers spend their time. In *2015 ieee 23rd international conference on program comprehension* (pp. 25–35).

Mortara, J., Collet, P., & Dery-Pinna, A.-M. (2021). Visualization of object-oriented variability implementations as cities [Conference paper]. In (p. 76 – 87).

Robson, C. (2011). *Real world research.* John Wiley & Sons.

Romero, P., Du Boulay, B., Cox, R., Lutz, R., & Bryant, S. (2007). Debugging strategies and tactics in a multi-representation software environment. *International Journal of Human-Computer Studies*, *65*(12), 992–1009.

Shah, A., Yu, J., Tong, T., & Raj, A. G. S. (2024). Working with large code bases: A cognitive apprenticeship approach to teaching software engineering [Conference paper]. In (Vol. 1, p. 1209 – 1215).

Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, *8*, 219–238.

Siegmund, J. (2016). Program comprehension: Past, present, and future. In *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)* (Vol. 5, pp. 13–20).

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on software engineering*(5), 595–609.

Tanimoto, S. L. (2013). A perspective on the evolution of live programming. In *2013 1st international workshop on live programming (live)* (pp. 31–34).

Tempero, E., & Ralph, P. (2018). Towards understanding programs by counting objects [Conference paper]. In (p. 1 – 10).

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, *23*(5), 459–494.

Von Mayrhauser, A., Vans, A. M., & Howe, A. E. (1997). Program understanding behaviour during enhancement of large-scale software [Article]. *Journal of Software Maintenance and Evolution*, *9*(5), 299 – 327.

Wang, D., Galster, M., & Morales-Trujillo, M. (2023). A systematic mapping study of bug reproduction and localization. *Information and Software Technology*, 107338.

Wong, W. E., Gao, R., Li, Y., Abreu, R., & Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering*, *42*(8), 707–740.

Wuilmart, P., Söderberg, E., & Höst, M. (2023). Programmer stories, stories for programmers: Exploring storytelling in software development. In *Companion proceedings of the 7th international conference on the art, science, and engineering of programming* (pp. 68–75).

Wyrich, M., Bogner, J., & Wagner, S. (2023, nov). 40 years of designing code comprehension experiments: A systematic mapping study. *ACM Comput. Surv.*, *56*(4).

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., & Li, S. (2018). Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, *44*(10), 951-976. doi: 10.1109/TSE.2017.2734091

## A. Interview guide

*The subjects were reminded that the interview was recorded and that the data is processed with informed consent according to research standards.*

### Personal information

- Alias
- Age
- Gender identification

### Professional Experience

- How would you describe your experience in the software industry?
- Number of workplaces have you worked in? (A new assignment in the same company may be a new workplace - extract the number of new codebases the subject worked with!)
- Number of programming languages?
- Years in the business?
- Anything else?

### Starting a new job

- When you start in a new workplace - how do you feel?
- What do you do when you are assigned your first bug in a new workplace?
- What strategies do you employ – to fulfil what goals
- What trouble do you get into?
- What tools do you employ? Why?
- What would you want to support you in your work with your first bug?