Intuitive Source Code Visualization

Amy Zhu
University of Washington
Seattle, WA, USA
amyzhu@cs.washington.edu

Audrey Seo
University of Washington
Seattle, WA, USA
alseo@cs.washington.edu

Sahil Verma
University of Washington
Seattle, WA, USA
vsahil@cs.washington.edu

Thomas Schweizer
University of Washington
Seattle, WA, USA
tschweiz@cs.washington.edu

1 INTRODUCTION

To develop or maintain software (e.g., create a new feature, fix bugs, refactor code), developers build mental models of the context related to their task. The context is often a mix of local information (e.g., method callers, class attributes, local documentation) and global information (such as class structure, design pattern, source code architecture, dependencies). Modern IDEs have relatively good support to navigate the local context with documentation pop-ups and shortcuts to jump to callers, callees, or the class hierarchy. However, when we need to access a more high-level view, we often find ourselves looking at the project's directory structure - which offers limited information and can be hard to read at a glance - or look at the available class diagram (when it was not too out of date) which demands a costly mental context-switch. When the former options are not available, one must rely on exploring the code manually, which is tedious and mentally draining as one also has to keep the local context in mind or rebuild it when one is done exploring.

The core of the problem stems from an awkward mapping between the programmer's mental model which can encode the interconnections between different levels of abstraction in the code, and the source code which encodes that information sequentially in arbitrary files composed of lines of characters. Understanding the file and class organization becomes crucial to making edits to the code. Additionally, reconstructing the mental model of the original developer of the software is tedious and error-prone, especially when it was developed by others but also when returning to one's own code after not looking at it for a month. How often, when making a change to a system, does a developer have to click through multiple files, locate relevant-sounding methods, read method arguments (each step possibly spawning its own recursive search that moves through different levels of abstraction and multiple class files), and scour the documentation – if it exists – in order to construct what they believe to be a correct change? Such code spelunking consumes valuable time and mental energy. We feel that we need to somehow preserve the fidelity of the original abstraction and be able to explore it at multiple levels of granularity without requiring expensive cognitive overhead.

1.1 The "Zoom" idiom

Finding an elegant adapter between these two realms is a challenging problem. Looking at other domains such as industrial design and architecture, we believe a solution must exist. Research on visual programming suggests a promising solution that adds a layer between the developer and the source code. For example, a zooming interaction idiom provides a very natural, lightweight mechanism for moving between levels of abstraction. When you "zoom-out" of a class, you get a relevant representation of the subsystem in which the class resides without changing context (e.g., dataflow, dependencies, architecture).

We imagine you might be able to zoom out from the granularity of a function or code snippet to a package overview, and you might want to perform this "semantic zoom" for different views of your system.

For example, it might be useful to have a highlighting visual idiom to focus on the underlying logging structure, or all the pieces of your system that have to do with remote procedure calls (RPC). Adding the ability for developers to navigate from concrete syntax constructs (e.g., a method, a class) to abstract concepts (e.g., a module's architecture, a class' dependencies on other classes) as easily as clicking through a function without requiring context switches would make developers more agile and precise when exploring what they are currently working on, similar to an architect that can, at a moment's notice, zoom in and out of the 3D model of their building, going from a bird's eye view of the house to an overall view of the floor plan to a detailed view of the basement's stairs. And just as the architect can switch back and forth from these two views easily, our visualization tool will permit developers to switch between different levels of abstraction.

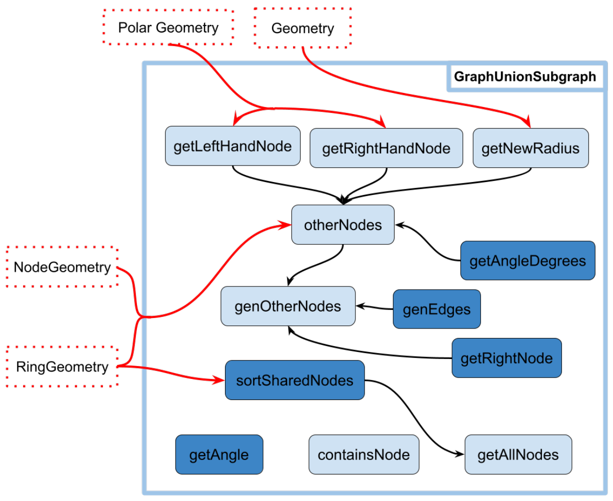

Figure 1: Zoomed-out view of an example class `GraphUnionSubgraph`. Rectangles represent classes and rounded rectangles represent methods. Black links show method calls inside the class, and red links and boxes represent calls to methods from other classes. Private methods are represented with a darker shade than public methods.

Overall, the main challenge is to disrupt a tradition of purely text-based source code for an abstract-based representation and navigation that is only possible through dedicated software. We aim to evaluate whether this kind of approach is conducive to simplified developer workflows or improved ability to understand software, thus increasing productivity.

1.2 Use cases

We envision this tool to have numerous uses, from helping developers to understand large and entangled codebases so that they can make changes quickly and accurately, to helping users of a library explore its inner workings. Getting a new developer up to speed and onboarded is a painful process for both the developer and the rest of the team, exacerbated by complex, bloated legacy code that often relies on institutional knowledge to be understood anywhere near approaching completely. For teams that do not have access to many resources, both human and computing, such as research groups or small startups, onboarding new members can be too costly to even attempt, which especially in the case of research projects, can result in the lack of maintenance and eventually disuse.

Understanding the relationships between the methods of a given piece of code beyond one degree of separation, i.e., immediate callers and callees, can be difficult. Previously, one of the authors had drawn such a sketch shown in Figure 1, in order to understand code written by another developer who was no longer with the company. The same author, while creating the rough draft of Figure 1, realized that the method `#getAngle()` was *private* and also *not called by any methods in the class.* It struck them that they had meant to delete this method a while ago, but the graph in Figure 1 made it obvious from a glance that `#getAngle()` was depended upon by no methods, and thus could be deleted without worrying about breaking the code.

1.3 Changing the paradigm of purely text-based source code

While there is extensive work on software visualization[7, 16, 18, 27], our approach differs in that we propose an integrated visualization that can seamlessly change the granularity of the information (i.e., the source code) without changing the location within the code. Additionally, our visualization removes information that is not relevant to the context and generate representations that stay consistent when changing granularity or editing source code.

Traditional search methods require considerable cognitive energy to reconstruct the structure of the program and the relationship between entities. This effort is repeated for each view of the system. Visualization can be used to offload this cognitive effort into less costly perceptual effort. We want to use filter and navigate as interaction idioms to achieve these tasks. Zoom will be used to shift between different levels of granularity; filter will be used to focus on different cross-cutting concerns.

In summary, we propose to improve program comprehension through a variable granularity visualization that is integrated to the developer's development environment. The visualization will focus on showing the relationships between the classes and methods. Specifically, we are interested to show the calls between methods inside and in-between classes to help developers explore related methods and classes, and understand structural and dynamic relationships between classes. Regarding the granularity, we define the three levels of granularity in the context of this project: The first level, or base level, of granularity is the local source code of the method or class. The second level of granularity is a visualization of the relationship between methods for the class, an example of which is illustrated in Figure 1. The third level of granularity is the same visualization as previously but aggregates the relationships between classes only.

1.4 Research Contribution

Our project aims to have the following research contributions, all of which, as previously discussed, are as yet unaddressed by the existing literature.

(1) Providing a framework to augment developers' workflows by enabling access to high-granularity relationship information that preserves and better reflects the context of the code.
(2) Learning to manage visualizing enough information about the source code while avoiding information overload but still including scope relevant information such that it is useful to the developer. (A classic problem in graph visualizations is what to do when there is too much data. We need to solve this problem in the context of program comprehension.)
(3) Understanding whether variable-granularity visualization techniques improve developers, comprehension of software programs.

2 TECHNICAL APPROACH

In order to implement a visualization tool that meets the goals outlined in Section 1.4, there are several challenges that need to be overcome in the implementation. First, a method of parsing and analyzing large codebases in a tractable way needs to be devised. Second, there are numerous difficulties that can arise while generating graphs that also have to be human-readable. Third, the visualization tool needs to be engineered in such a way that it does not unduly burden the developer's workstation, which could be a problem especially with large codebases. Lastly, the visualization will need to easily integrate into the developer's existing workflows.

2.1 Parsing and static analysis

Regardless of whether the visualization is integrated, we will need to take several steps to analyze the code so that there is data to visualize. The first step is to collect enough information about the source code to generate the visualization. We only aim to capture the static behaviour of the program, so we will design or retrofit a parser and static analyzer, so that we can capture the entire structure of the code, which is paramount. We also want to supplement this analysis with the output from other static analysis tools so we can discover and reconstruct other information, including method callers and view-specific interactions between entities. Some examples of attributes we would like to analyze include

- relationships between caller and callee methods
- relationships between classes
- relationships between packages.

For parsing and analyzing huge projects, with tens of thousands or even hundreds of thousands of lines of code, we will investigate two approaches. In the first, we would use a greedy-style algorithm that parses and scans everything beforehand. In the second, we would use a sort of lazy parsing where we would only analyze methods, classes, or packages (depending on the level and granularity of the analysis) as needed, and cache results, only re-parsing if the code changes.

2.2 Zoom levels

In Google Maps, complexity and information density are largely managed by your zoom position: when you see a country, you can see high-level details such as main highways and borders but not the low-level details such as street names.

For this project, we define three level of adaptive zoom:

- **Baseline**. This level is the traditional level with which developers interact with the code using a text editor. It contains very detailed information for a given file at the time.
- **Low level**. This level shows the relations between the methods of one class and its external relations as illustrated in Figure 1. This visualization can be used to explore relations between methods of the class and outside dependencies. Clicking on an outside class dependency will shift the focus towards the new class.
- **High level**. This level shows the relations between classes. At this level, we cannot see individual methods. This visualization can be used to navigate between groups of class and hierarchies in the source code.

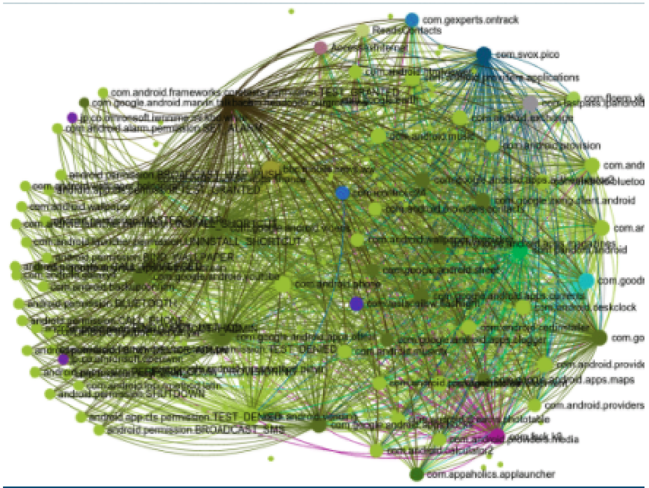

Figure 2: An extreme example of visualization pitfalls: there is too much information at once and the labels are not easily readable.

Each level of visualization is interactive: you can zoom in and out, and pan around. Clicking on an element, moves the focus to it (i.e., center on the screen). Hovering on an element provides additional information based on a tool-tip. Double-clicking drill down on an element, changing the level of zoom.

2.3 Technical challenges with visualization

Overall, we face several interesting difficulties with respect to creating the most intuitive visualization possible.

One of the major technical problems we will face is that node graph visualizations tend to become unreadable as the amount of data in them grows as illustrated in fig. 2. We must avoid cluttering the visualization and reduce noise while still keeping enough information to be informative and valuable to the developer. It is commonly observed, e.g. in the *Deutsch limit*, that there is a concrete, fairly low limit to how many nodes can effectively exist in the graph when performing visual programming[2]. Filtering by different views and using a metaphorical zoom to expand on different parts of the system already naturally minimizes elements, and some additional strategies for reducing noise include: showing only close neighbours determined by a distance function and grouping redundant information and relationships originating from common sources together. We could alternatively use this distance metric to create clusterings of items. Candidates for a distance heuristic include semantic (scored by some kind of text-based analysis), structural (how far away is it in the file structure), or something more intensive (collecting dependencies as edges and using the length of a minimum spanning tree).

Another challenge will be to organize the remaining information onto a 2D plane so that it is not only pleasing to the eye but also deterministic. The algorithm should be deterministic so that visual elements are not moved to different places every time the user changes granularity, i.e., if we start from the class view given in Figure 1 and then zoom out to the software level or zoom in to source code level, and then return to the class view, the nodes in the graph should not have changed positions drastically. If we don't pay enough attention to this visual tracking, users will become disoriented easily. A further complication is that as the code base is changed, the visualization should also update accordingly, but without unexpectedly moving preexisting elements of the visualization.

We also need to resolve the distinction between visual zoom and metaphorical zoom (if users wish to zoom in on the visual elements versus wanting to perform a zoom on the program structure), decide on which of highlighting or filtering is more perceptually effective, and tackle tracking items when switching across views. These are some examples of smaller issues that will certainly arise.

These concerns are rooted in visualization literature, and we expect to be able to find inspiration for solving them there.

2.4 Architecture

One of the challenges in introducing a new tool to an existing workflow is the resistance to change, making it hard for people to adopt a new tool. To mitigate this and encourage developers to use our approach, we build our tool to maximize interoperability to existing (and future) development environment by decomposing our approach into three parts as illustrated on Figure 3.

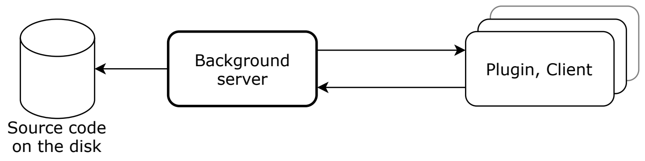

Figure 3: Overview of the technical components in our proposed approach.

The main element is the *background server* (center) which has the dual responsibility to 1) parse the *source code* (left) to find relationships and observe file changes, and 2) communicating with a *client, plugin* (right) to provide visualization depending on the developer's action (i.e., zooming in and out, or opening a file). The background server is the "brain" of the approach, handling all the algorithmic complexity and source code parsing[1]. The plugin and client are small add-ons to an IDE, terminal or text editor to show the visualization to the developer while staying in context, and send context information about the developer's actions to the background server. This architecture also give the flexibility for the client to be a standalone visualization application for debugging or monitoring purposes.

The communication between the background server and the plugin or client is implemented using a RESTful API. We might encounter problems with networking delays using our proposed architecture for the system. To mitigate these issues, we can expose an RPC interface as a communication channel, and send patches or deltas when hot reloading the system.

[1]If the client supports it, the parsing can be delegated to it since there is a two-way communication channel between the background server and the client

2.5 Developer workflow integration

IDE integration is ideal for encouraging software developer adoption of our tool (which has been found to be crucial [21]), so we will aim to create a plugin for at least one IDE.

We chose Visual Studio Code as our initial IDE to deploy our visualization tool on for several reasons. First of all, Visual Studio Code plugins can be written in JavaScript and use web technologies, so the D3 code can be used as long as we provide the library. Secondly, this also makes the visualization more universal from the beginning, since most people have access to a web browser even if they use a different IDE, so this approach will have the broadest impact.

Visual Studio Code is a popular IDE, though there are also other alternatives such as Eclipse and IntelliJ IDEA. While it would also be great to target these IDEs, within the scope of this course, we thought that creating a plugin for Visual Studio Code would be a good starting point for our prototype and will also have the lowest startup cost in human hours, since one of the authors is already familiar with D3.

3 IMPLEMENTATION

In this section, we describe the solution, named CodeMap, we developed based on the design discussed in Section 2. Our implementation is open-source and available on `https://github.com/amyjzhu/503-hacking/`.

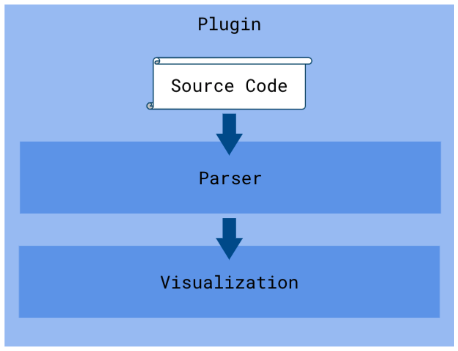

Figure 4: Caption

We give an overview of the implementation in fig. 4. Following the precepts of our original design discussed in Section 2, the background server, visualization and plugin are separate entities. Keeping the different component separate did allow us to iterate faster because we were able to develop and test the plugin, visualization, and background server separately. In this instance, the background server manages our parsing solution exclusively because the source code is only parsed once ahead of time in a JSON file. The parser and the visualization are held together by the plugin that acts as a clueless intermediary: it has no domain knowledge.

3.1 Background server

The background server for CodeMap is written in Java, and uses the library JavaParser [24] to parse the files of a given project. JavaParser is a powerful source code parser that is used in many projects, both open source and industrial, and is relatively easy to use. It support symbolic resolution which we use to extract relationship between software artifacts (i.e., packages, classes, methods). Additionally, JavaParser can works on single files, meaning we can use it to do lazy parsing in the future where we the background server would be able to parse files only as needed and cache the results. We leverage JavaParser to obtain the following:

(1) For each file, all of the classes (including inner classes) in the file.
(2) For each class, all of the declared methods.
(3) For each method declaration with a body (i.e., not an `abstract` method), the set of all of the method calls.

Because JavaParser not only finds method calls, but also where they were declared (i.e., the method's enclosing class), we are able to obtain the *fully qualified signature* of both the declared methods and the method calls, which enables us to link a method call c made in method m of class A to the method m_c declared in class B, where c is calling the method m_c. This data is used to generate the different levels of the CodeMap visualization.

While working on the parser, we found a bug with JavaParser. We reported the bug to the authors. While, we're not the first one to report it, we are the first to submit a minimal reproducible bug report [2].

3.2 Visualization

We use D3 to power the visualization in CodeMap. D3 is an excellent framework for building interactive applications with great velocity, and it afforded us the ability to develop and experiment with the visualization in a browser completely separately from the IDE integration. In particular, we used D3-force in order to discover a reasonable layout for each node. Then, we implemented scrolling with D3-zoom such that visually zooming in beyond a threshold zoom level also performed a semantic zoom (e.g. after zooming in 4x, we switched from a class view to a method view). Under this scheme, classes are zoomed in (large) enough at the threshold such that a class becomes the macro universe that contains each method. Once the layout is decided, it is fixed and does not act under any forces afterwards. These are efforts to address some of the visualization-focused technical challenges laid out in the previous section.

Our design was informed by the many technical challenges of visualizing software of "real sizes". When there were over 9000 nodes, and far more links, drawing links was burdensome, both cognitively and on the processor. For one, even with nodes laid out sensibly, the visual clutter of thousands of links across the canvas was overwhelming and made it impossible to differentiate the real source and target of each. Also, redrawing thousands of elements made panning, and especially zooming between layers, unacceptably slow for an interactive tool (in the order of seconds). Initially, to manage which elements to visualize, we rely on the borders of the SVG canvas. However, it was clear that we would have to adopt additional ideas and idioms. First, we disabled animations; each frame incurred massive lag and sometimes stuttering so the full animation would not even have played out after one minute. Instead, we instantly calculated the final locations of all entities and drew them once. Second, we implemented a "draw distance", similar to what's used in video games — all the entities were deterministically laid out, and only the ones that were within the SVG window (i.e. within the width and height of the canvas once the canvas had been translated and scaled) were rendered. Third, we did not draw links automatically. Instead, we drew the links on demand, meaning that the user must click on a class to see the outgoing links. This significantly reduced the visual clutter and facilitated more focused exploration.

Another issue with large codebases was that oftentimes following links meant heading to parts of the visualization that were distant from the originating entity. In order to make following links even possible, we allowed users to pan in alternate ways that wouldn't accidentally trigger distractions, using the mouse wheel and right mouse button. We also ensured that links from an entity persisted until the user clicked that entity again to deselect them. To circumvent endless scrolling to really distant target nodes, we implemented an auto-pan for links; double-clicking a link brings you to the node at the other end. Each of these decisions was made to make navigating the cosmic acreage of codebases gentler, both visually and conceptually.

We implemented highlighting to guide the developer and help them keep focus. When they hover an entity (i.e., package, class, method), the entities that are unrelated have a reduced opacity, making it easier to see and locate related entities.

Figure 5 and Figure 6 are screenshots of CodeMap in action. Figure 5 showcases the visualization one can obtain when zoomed at the class level (high-level). Figure 6 showcases the visualization at the method level (low-level).

It's important to note here that low-level corresponds to being more zoomed in: we've "peeled back more layers" to reach the low-level content. One might remark that a tension arises because low- and high-level are idiomatic ways to describe levels of abstraction in software systems, but in visualization and map contexts, zooming in means a higher level of zoom, means more detail, and consequently means lower-level (though the zoom indicator appears *higher*). We choose to preserve these idioms to create the most natural descriptors, and maintain intuition for reasoning about the system.

3.3 Plugin

We developed a Visual Studio Code (VS Code) extension for CodeMap to display the visualization described previously. We chose VS Code because the platform is very popular and well-documented. Additionally, it uses the html/css/javascript web stack that all authors were familiar with and that the visualization also uses, simplifying the integration.

The plugin ties the parser and the visualization together, and ensures the developer has an agreeable user experience. The developer is able to parse and start visualizing classes directly within the IDE.

The plugin integrates the visualization as part of a *Webview*[3]. We inject the visualization code directly inside the Webview. The background server is integrated through a script call from VS Code's

[2] https://github.com/javaparser/javaparser/issues/2738#issuecomment-799060362

[3] More information about VS Code Webviews can be found at https://code.visualstudio.com/api/extension-guides/webview.

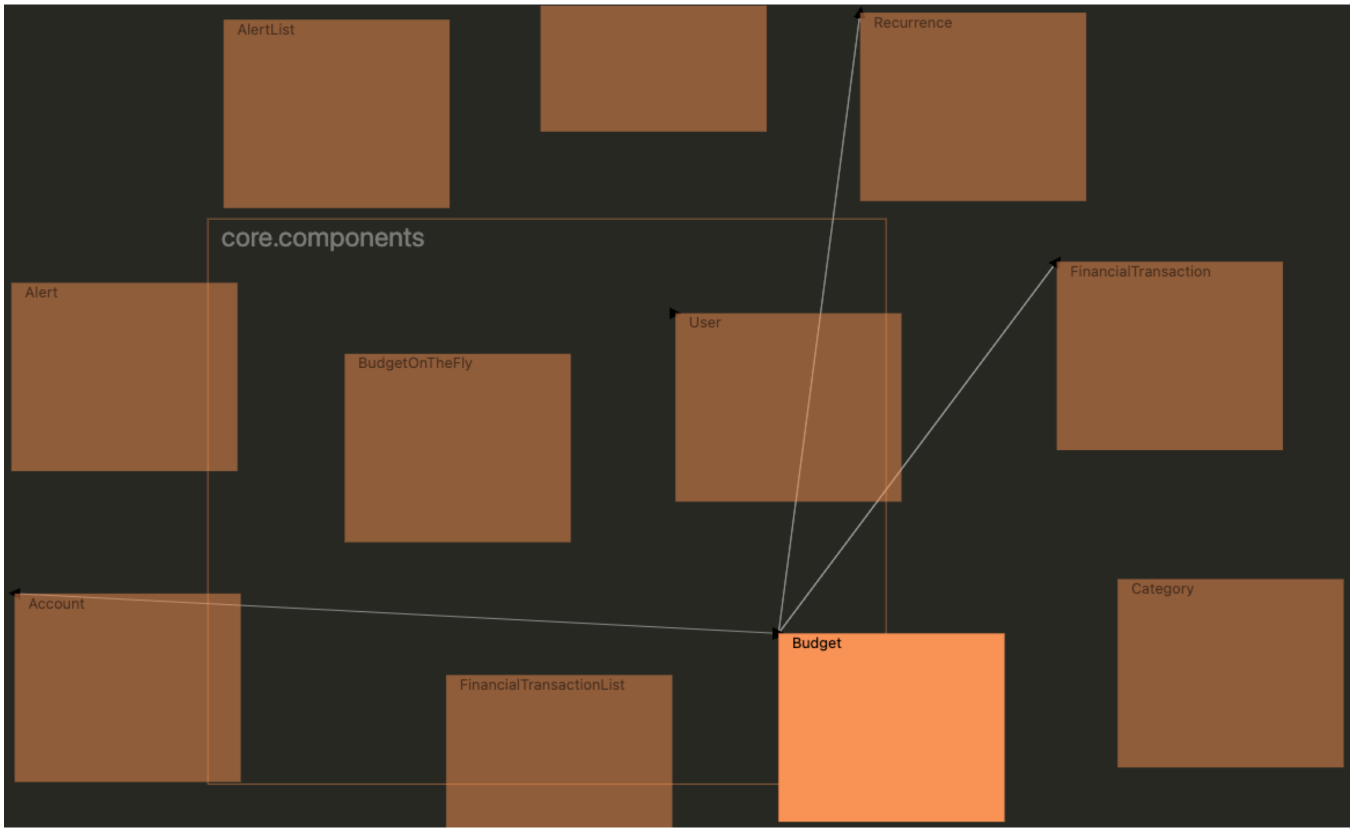

Figure 5: Semantically zoomed onto classes and class relationships. We can see that the `Budget` is located inside the package `core.components` and uses co-located classes `FinancialTransaction`, `Account`, and `Recurrence` and that it doesn't use any outside classes.

embedded terminal: the plugin calls bash script that loads the background server packaged as a JAR file. The plugin provides the arguments to the background server based on the plugin's context (i.e., which project is open).

3.4 CodeMap in action

In this section, we walk through an example scenario of using CodeMap:

The developer starts up VS Code opened on their Java project. Fist they invoke the Command Palette with SHIFT + CMD + P / CTRL + SHIFT + P and type `codemap.parse`to tell the background server to parse their project.

Once done, the developer navigate to the public class they want to look at or edit. They open the visualization using the shortcut CTRL + CMD + V / CTRL + ALT + V (or through SHIFT + CMD + P / CTRL + SHIFT + P, `codemap.parse`). The visualization opens centered on the class they were just looking at previously in their text editor.

From there, the developer can pan around to explore the neighbouring classes, clicking on them to discover dependencies. Alternatively the developer can zoom-in to see the methods inside a class. As expected, if the developer clicks on a method, it shows links to the method it depends on (i.e., its callees). Using DOUBLE CLICK on a link, follows the dependency: it centers the visualization to its target. DOUBLE CLICK again on the link, brings the visualization back on the the source method.

Zooming back at the class level, the developer can CMD + CLICK / CTRL + CLICK on the class to jump to the source code to see implementation details and make edits. Jumping back to the visualization is easy, using the shortcut CTRL + CMD + V / CTRL + ALT + V will reopen the visualization centered on the same class.

4 EVALUATION

The goal of our evaluation is two-fold:

(1) Evaluating if the visualization helps programmers in their development tasks, i.e., rejecting the hypothesis that a visualization is at best equal to do a search or "click-through" methods in an IDE to find relationship between files in order to build a mental model to understand the program.
(2) Whether the visualization is perceived useful by the programmers.

The latter is important because our work aims to integrate visualization as a first-class citizen into software development compared to contemporary approaches that are only supplemental to current development practices. Additionally we want to start a discussion with developers that re-imagines software development so that text

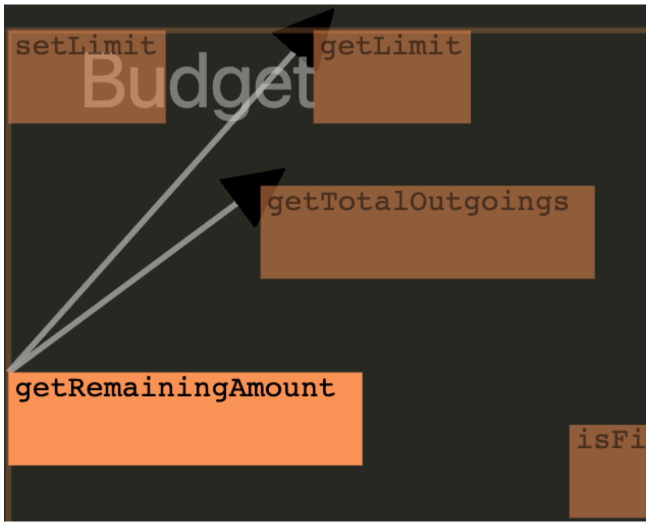

Figure 6: Semantically zoomed onto methods and method relationships. We can see the method `#getRemainingAmount()` from class `Budget` calls the methods `#getLimit()` and `#getTotalOutgoings()`, but has no other dependencies.

files are not the only way to interact with source code while editing software, and consider other medias also as first-class citizens.

Specifically, we want to answer the following research questions:

RQ1 Does an integrated visualization with variable levels of information shown help users understand code bases?

RQ2 Do developers like the visualization, find it intuitive, and think that it helps them understand high-level relationships in source code (e.g, between the methods in a class or between classes)?

To answer these questions we propose to do a user study to measure the performance difference for completing development tasks (e.g., fixing a bug, adding a feature) between a regular development environment and the access to the tool. We will draw from previous empirical studies in the program comprehension and software visualization communities [5, 27]. After that, we would interview the participants using a Likert-scale survey and a few open questions to understand their perception of the tool. This will help us understand their concerns, and determine if visualization actually helps program comprehension compared to doing a search for related methods in the IDE or the terminal.

4.1 Evaluating visualizations

We looked at recent and seminal work in the software comprehension and visualization research community to familiarize ourselves with common and recommended evaluations strategies to evaluate software visualization approaches.

We looked at visualization papers from the proceedings from the IEEE Working Conference on Software Visualization (VISSOFT)[25] from last 5 years. We retained the papers that featured a visualization and described how they evaluated their approach. We also took into account meta-papers that studied how visualization approaches are evaluated.

We observed that some papers set expectations through a focus group (sometimes called a wish list) to list features they think the tool should have [10]. We did something similar during the inception of the project where we discussed about the features our tool should have, and should not have to facilitate software comprehension through visualization. We detail these reflections in Sections 1 and 2.

Regarding the evaluation of the approach, we encountered two strategies. The first one was to conduct a controlled experiment in the form of a user study [5, 6, 9–11, 27]. The second one was a qualitative analysis in the form of a case study [1, 15, 26]. We observed that most of the papers would do a user study while others would opt for a case study. Some papers did both.

In controlled experiments, participants are asked to complete some tasks and the researchers measure their performance through *task completion* and *task correctness* dependant variables. A survey would be then filled out by the participant with questions intended to inform on the *Perceived Cognitive Load*, *Perceived Usability*, and *System Understanding*. In some cases [9–11], the researchers would use the *System Usability Score* (SUS) [3] to gather formalized feedback on the usability of their approach. The tasks the participants go through were generally very simple. For example, in [9], one of the tasks is "Find the weekdays when the most issues were closed," which is conceptually simple to understand, but can be hard to execute without a dedicated tool or relevant knowledge on this task. We found that most of the studies had only one treatment: the participants would be measured on tasks only when using the proposed approach. A few studies had two treatments, the second being a replication of a previous approach with similar objectives. When the study had one treatment, the results are compared against hypothesis or a baseline.

In case studies, the authors evaluated their approach when applied to one or more projects or use case. They manually inspected the results given by their approach and discussed how their approach worked, or did not work, in various situations. In some cases, objective quality indexes are used to measure properties of the proposed approach.

4.2 Case study

Within the time constraints, we were able to conduct an informal case study with colleagues as participants. We asked them to give their thoughts on the tool, and observe how they used it to complete some simple tasks.

4.2.1 Protocol. To conduct our case study, we chose the code base JFreeChart [14], which is a mid-sized, mature code base with about 135,000 lines of code and over 4000 commits, and has been in active development for the past fourteen years. We think this is representative of many systems that developers are expected to work on when, for example, starting a new job. JFreeChart has only one main developer, but it has had many contributors, so it exhibits ad hoc additions and exhibit changes to a system over time. We chose four common tasks that model questions developers might ask themselves or others when beginning to work on new features or bug fixes. We also selected tasks that were fairly independent and dealt with different parts of the system, so that knowledge from a previous task does not affect the participant's performance.

These were the tasks:

Task1 Does the class `CyclicXYItemRenderer` use any classes from the same package? Does it have any dependencies on classes in other packages of JFreeChart?

Task2 Imagine that something is wrong with the class `StandardXYZToolTipGenerator`. Which entities might be at fault?

Task3 In which package is the class or interface `ImageFormat`?

Task4 What happens inside the code when we add a title to a chart with the class `JFreeChart`? What entities are involved?

For each session , we first had the participant install VS Code, clone the code base of JFreeChart, and install CodeMap. Each participant was given access to the document containing common shortcuts for VS Code and listed the ways one could interact with CodeMap, as well as descriptions of the tasks to complete[4]. At this time, we also ask the participant for their consent to participate to the study and be recorded.

Then we walked the participant through a short tutorial of how to use CodeMap as well as some useful VS Code shortcuts. After the tutorial, the participants were asked to complete the tasks.

As the participant completed the tasks, we recorded their screen and audio, and timed each task. The dependent variable is time taken to complete the task. The independent variable is the treatment: whether CodeMap is used for the task. The confounding variables are their programming experience in school and in industry, whether they prefer an IDE, or another solution such as emacs or vim, and if they have used Visual Studio Code before. We also encouraged them to be verbose in explaining their thought process. We took notes on what we observed and also watched the recordings to add to our notes. Given the short time frame of this project, we were unable to complete a more in-depth analysis of the recordings and how each session went.

It may have been beneficial to collect some other metrics about how participants interact with software: for example, collecting biometrics data like where developers look when they first enter a codebase, or tracking the number of clicks made or time spent reading any particular file. However, these are more logistically challenging, and it is unclear how to compare these metrics between the two setups.

For all participants, the tasks were run in the same order. However, we had two treatments in order to counteract the effects of becoming more familiar with the JFreeChartcode base. Half of the participants used CodeMap to answer the first two questions and did not use it for the latter two questions. It was inverted for the other half of the participants. We organized the tasks so that odd numbered tasks (i.e. 1 and 3) were easier and the even numbered tasks (i.e. 2 and 4) were more difficult. Given the small number of participants in this study, the goal is more to see how they think about the tool and what they think should be improved or changed, rather than to assert any certainties about the tool's effectiveness.

If the task completion time with access to our tool is lower (and statistically significant) than task completion time without access to our tool, the answer to RQ1 is yes, an integrated visualization helps developers understand code bases. Time is a proxy for understanding; generally, if a task is more difficult to reason about, it should take more time to complete. Given that the small number of participants in this initial study, this case study cannot fully answer RQ1, but instead will serve as a motivation for future work. Due to the long loading time needed for the tool, we did not include that time in the time taken to complete the task.

After the study was completed, we sent each participant a short survey (see Section 4.4) containing closed questions (to be marked on a Likert scale), open-ended questions, and background questions to determine the answer to **RQ2**.

4.3 Results from case-study

We recruited seven participants for this case study; however, one participant was unable to run CodeMap due to a bug in our implementation when used on the Windows operating system. This left us with six participants, whom we will refer to with the letters A through F. All the data for time spent on each task can be found in Fig 7.

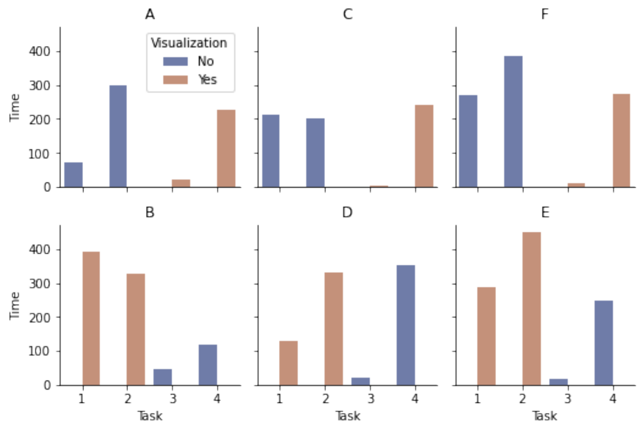

Figure 7: The time it took for each participant to complete each task.

Fig. 8 shows the timing averaged across all the participants, along with error bars. Due to the small nature of our case study, this plot cannot be used to conclude whether using CodeMap improves developers' comprehension or not. Additionally, even when a participant finishes a task quickly, their answer may not be correct. Therefore, the majority of this section will be spent doing a qualitative analysis of how each participant performed on each task with and without the tool. **Answer to RQ1:** Inconclusive.

4.3.1 ***Examples of incorrect attempts.*** Participant A finished Task 1 without using CodeMap very quickly, but made an error in determining whether or not there were any dependencies on classes from the same package. Participant A only looked at package imports, and due to the similarity in the name of the package for `CyclicXYItemRenderer` and one of the classes that `CyclicXYItemRenderer` imports, concluded that it did use classes from the same package. This is actually true, but those classes would not have to be imported in Java.

Several participants had problems with understanding what the questions in the tasks were asking. Participant B took a long time

[4]These documents may be found in Appendix A.

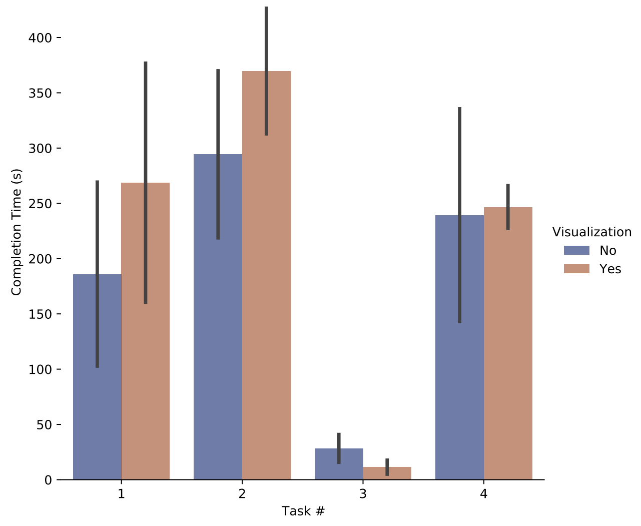

Figure 8: Grouped box plot showing how quickly participants were able to complete each task, with and without the use of our visualization.

to complete Task 1 due to a small screen size that made using CodeMap more difficult and also due to misunderstanding the task. Task 1 only asks for existence, but participant B attempted to list all the classes that `CyclicXYItemRenderer` uses. Participant F misunderstood Task 1 as asking for which classes depend on `CyclicXYItemRenderer`, instead of the other way around.

4.4 User survey

4.4.1 Protocol. After completing the user study, we gave participants an additional survey that addresses **RQ2**. The questions in this survey aim to measure how useful the participants found CodeMap in bolstering their program comprehension, and how intuitive it was to use. We also ask several open-ended questions, which we coded for common themes using an open-coding approach. Lastly, we collected some non-identifying data about each participant so we can better understand their study results if needed. See Appendix A for a full list of the questions.

Demographic. All of our participants were proficient with software engineering, and had at least a Bachelor's or higher in computer science or a related field. All had been or were currently pursuing graduate-level studies. Experience in industry as a software engineer varied from 0 to around 4 years (7 including years in research labs); however experience with programming through primary, secondary, and undergraduate studies ranged from 6 to nearly 20 years. Java experience ranged from 0 to around 18 years, and one participant noted that they only used it in university. 83% of the participants reporting having used `VS Code` before, but only 3 (of 6) said that they used it to develop regularly. Unfamiliarity with the chosen IDE means participants probably took longer to complete the tasks, even the ones not using the visualization.

4.4.2 Results. Overall, we find results to be in favour of using our tool. Participants also made several valuable suggestions that we plan to incorporate.

Likert responses. We aggregate each of the responses to the Likert scale questions and present them in Figure 9. The overall sentiment is very encouraging towards our proposed paradigm. On about half the questions, participants indicated they agreed or better on each of the questions (on a scale of -2 to 2, we had values of 1, 1.16, and 1.8). Question 3, "Source code is more aligned to how I conceptualize software", was the only negative response on average, and had no Strongly Agrees, which supported our hypothesis that source code was a subpar representation of complex codebases. Some questions participants were more hesitant or unsure about, possibly because they made stronger, more difficult to support claims or were more open-ended. Ultimately, all averaged responses, save Question 3's, were positive (Neutral or better). The most bolstering find was the strong, unanimously positive support for the ideas that variable-granularity was something participants thought was useful when inspecting programs, and that this approach was better at visualizing program structure than participants' existing workflow. **Answer to RQ2:** Yes (modulo the number of participants).

Open-ended questions. We also collected data from some open-ended questions. These questions asked

Q1 Could you imagine incorporating this tool into your workflow? How would you best utilize this tool in everyday software development? **Feedback:** All of our participants were willing to integrate a more-polished version of CodeMap in their workflow. They agreed that it was useful to explore an unfamiliar codebase. One participant added that they would like to use CodeMap side-by-side with the codebase.

Q2 Which aspects of this visualization were the most helpful, if any? **Feedback:** Most participants mentioned the links between various entities of the code were most useful. Other participants liked the clicking feature, which took them to the entity they were connected with.

Q3 Which aspects of this visualization were not useful, if any? **Feedback:** The color of classes, the over-crowded number of methods, large text size, long loading time.

Q4 What visualization features would be most helpful, if any? **Feedback:** The participants would have liked to be able to see visualization and code at the same time, and there needed to be more of a separation between the package and class level.

5 THREATS TO VALIDITY

5.1 Construct validity

The user study tasks may be inherently biased towards completing it with visualization or without. To mitigate this, we tried to avoid too local or global tasks and select non-trivial issues that would better approximate actual developer workflows.

5.1.1 Implementation limitations. Generics in Java were only introduced in Java 1.5, and sometimes, old legacy Java code will include *raw types*, which is the use of a type that is generic, such as `List<T>`, but without specifying the type parameter(s). For example, in the

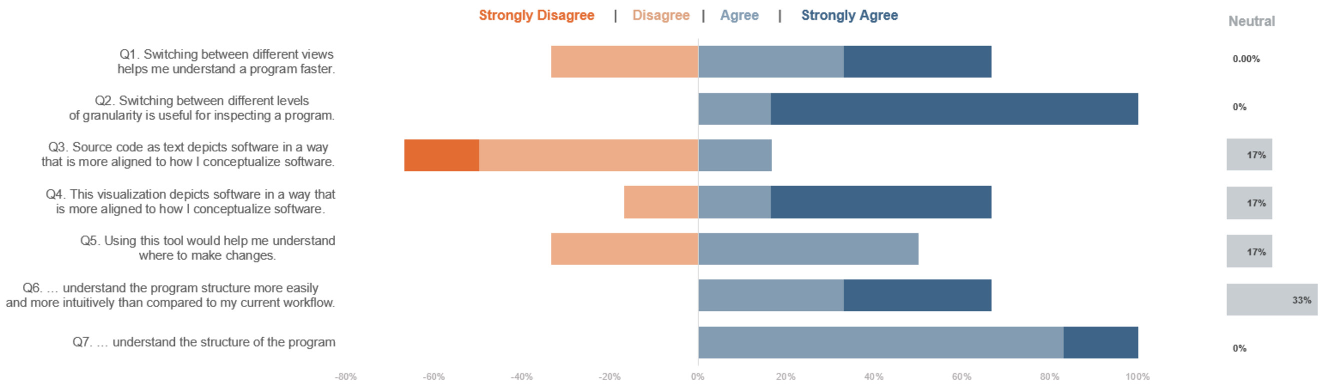

Figure 9: Sentiments collected from the survey we asked the participants to complete directly following the study.

code `List myList = new List();`, the object `list` has a raw type. Raw types still compile because the Java compiler supports legacy code, but JavaParser, which we use to power the majority of our parsing, cannot determine what the type of an object with a raw type will be, and fails to parse resolve the fully qualified signature for any methods called on those objects. Therefore, for projects that contain legacy Java code that use raw types, CodeMap cannot show all of the caller-callee relationships. However, on JFreeChart, we found that this only happened 61 times, and often for the same method calls, and the parser for CodeMap was able to find 12,354 total resolved method calls, so at least for JFreeChart, this seems to be a relatively small problem.

5.2 Internal validity

The internal validity is threatened by the static task order. Because we asked the participants to perform the tasks in the same order, we introduce a confounding factor: it is possible that the two first tasks are inherently easier or harder than the second tasks. Also, because we do the two tasks with the tool before (in group A) or after (in group B) the tasks without the tool, we cannot fully account for the fact that the participants may become more familiar with the code as they finish each task, meaning any participant will do better on the last set of tasks. The latter concern should be mitigated by the fact that the two treatments alternate which set of tasks comes last, but it is possible a more sophisticated task order, and interleaving, would produce a less biased result.

5.3 External validity

External validity is threatened by the small participant population that have mostly academic experience and a couple of years of industry experience. Thus, we cannot claim that our approach has an effect on the global population of developers as the bulk is mostly in industry.

Additionally, external validity is weakened because we study one Java Project using one platform: we don't know if the participants would have a different experience with a different language, or IDE (or no IDE).

6 RELATED WORK

Lemieux et al. [17] and Merino et al. [19] propose a comprehensive literature survey on software visualization.

6.1 Program comprehension

There has been work in program comprehension dating back to 1970s. We point the readers to relevant work for program comprehension using visualization, specifically for object-oriented programs.

CODECRAWLER [16] (CC) is a language independent, interactive, information visualization tool. It is mainly targeted at visualizing object-oriented software, for languages like C++, Java, and Smalltalk. CC generates polymetric views of the source code – where nodes represent the entities and edges represent the relations between entities. In polymetric views, more dimensions can be captured with information like node size, node color, and node position as software metrics. It provides vizualization in several views like coarse-grained view, fine-grained view, evolutionary views, and coupling views. Our approach is different because CC does not allow a programmer to navigate through the code. CC only provides visualization for the code (which are pretty dense in the examples views in the CC paper) based on software metrics, whereas our approach provides a easy switch from source code to views and back providing smooth navigation. Also, our views uses considers distances between elements in order to present them in a way that is comprehensible and not very dense, which CC does not.

JIVE [8] provides runtime visualization for object-oriented programs, specifically Java. It treats objects as environments, captures history of execution, supports forward and backward execution, and queries on runtime state (for debugging). Jive also supports multiple views, which it illustrates thorough a binding example of a binary tree: 1. Detailed view, 2. Compact view, 3. Sequence diagrams, 4. Call-path and minimized views. JIVE uses the Java Platform Debugger Architecture for implementation. Our tool differs in that JIVE captures dynamic state from the program execution, rather than capturing static information about the structure of the codebase.

ExploreViz [7] provides a live landscape view of large codebases in a 3 stage hierarchical view. First, the system consists of servers, the second is node groups for servers running the same applications, the third is the thickness of connection between nodes according to the communication between them. ExploreViz mostly targets cloud computing software and offers views at multiple levels, starting with a high level application view, and zooming in can provide details about classes and their communication links on demand. The links between nodes are generated based on program trace information that is collected from running systems; therefore, ExplorViz is dynamic visualization tool. The ExplorViz visualization provides a high level view of the application and its classes. It also provides metrics like CPU and RAM utilization for the application at runtime. Our approach differs in that it provides the capacity for navigation, uses static analyses, and the zooming goes down to the much lower level of methods and relations between classes which is not available in ExploreViz.

Extravis [12] which is used for visualization of runs of a program. Extravis motivates the problem stating that even a couple of minute runtime of a large codebase can contain gigabytes of information about method calls and understanding them is a huge time consuming task for a programmer. Extravis offers two views: Hierarchial Edge bundles (HEB) and Massive Sequence View (MSV) to handle to problem. The HEB shows the calls from one hierarchy element to another (which is inferred from the directory structure of the Java code). It in arranged in a circular view, which calls bundled from one element to another using an algorithm. There is a highlighting feature in HEBs which highlights calls in a given timeframe. This helps in reducing visual clutter. MSV on the other hand show each call, and don't arrange them by hierarchy, and it faces scalability issues. Our approach differs in providing navigation, static program visualization, and the ability to zoom in at levels of granularity.

Jayaraman et al. [13] proposed an algorithm for compaction of sequence diagrams for visualization using JIVE. Sequence diagrams are basically object diagrams at different timestamps of the program as it runs. Due to the enormous interactions and size of codebase, the sequence diagram can quickly become too big to visualize efficiently, and this work addresses this specific problem. Our approach, on the other hand is a novel method for program visualization and might at a later stage use such a compaction algorithm.

Cornelissen et al. [4] provides a setup for evaluating the benefit of the visualization tool, Extravis for programmers. It measures if using Extravis, decreases the program comprehension time and increases the correctness in understanding. The experiments take place with a group of 14 Ph.D. students at a university and the results are statistically significant in showing a decrease in time and increase in correctness in program comprehension. We also aim to perform such an user-study to evaluate the effectiveness of the visualization tool we are developing.

Ho-Quang et al. [11] uses *role-stereotypes* to enrich the structural representation in systems with information about the type of functionality that a class has in the system as well as the types of collaborations with other classes that it typically has.

6.2 Visual programming

Visual programming languages allow a programmer to write a computer program using visual elements like icons, blocks, and arrows instead of text. Popular visual programming languages include Scratch for teaching programming to children [22], Unity for game development [23], and Orange for machine learning and statistics [20]. The goal of our project is not to develop a visual programming language, but enhance comprehension of object-oriented programs using an interactive visualization approach.

7 DISCUSSION

This project made us realize that building a visualization tool and evaluating it is complicated and time-consuming. Furthermore, working in the medium of visualization offers an almost infinite potential for modification. Every aspect can be tweaked to change its perception in the hope that it's conducive to a better understanding for the user. Each iteration of this process takes time, much more than anticipated. In this section, we reflect on how the project went through three aspects: Limitations, Challenges, and Future Work.

7.1 Limitations and challenges

As is inevitable, although we made our best engineering efforts, we have several limitations within the current approach, as well as some existing challenges that should be addressed.

7.1.1 Visualization: entity sizes. In terms of the visualization, one more challenge that will have to inform future designs is choosing the right size of boxes in the visualization. Boxes that are too small won't contain all its children properly, and may cause collisions or elements overlapping, but choosing a box size that fits the maximal entity means that many classes are empty, and as sparsity increases, so does the tedious amount of panning. One idea to make this problem manageable would be variable-sized boxes, with the size acting as a proxy for "amount of information", whether LOC, or number of methods. However, this also has drawbacks, as it adds perceptual complexity to a relatively unimportant (in the context of codebase navigation) variable. N.B. that while this idea has merit broadly (see Section 7.2, we wanted to focus on the effect of variable-level granularity in this work. Another would be to decouple the zoom level with the semantic zoom level so that users could pan around faster while following links, and then zoom back in.

7.1.2 Performance: Moving around and initial loading. As we described in section 3.2, we faced several performance challenges when we first tried to visualize a larger codebase such as JFreechart. We managed to reduce performance issues such that things were smooth after a long initial load. At least one participant, however, noted that this wait time was very long. Pre-calculating the positions (and other metadata) of entities, or adding a loading icon are two simple ideas for improving the user experience. Another idea would be a more sophisticated lazy calculation strategy. Furthermore, we may be able to borrow some ideas from how Google Earth handles navigating by requesting new, more detailed tile information based on the area a user is zooming in on.

7.1.3 Deutsch Limit. Performance concerns subsumed the goal of designing a way to tame complexity. However, none of the participants mentioned that they felt there were too many entities or that they were too overwhelmed to tease out information from the tool. We think this might be due that classes and packages have a tendency to naturally be organized in clusters manageable by developers during development.

7.1.4 Finding the right parsing tool kit. Originally, when we were looking for parser tool that we could adapt to our needs, we chose Doxygen[5], since it can take files from many languages, including C++, C, Java, Python, etc. It also produces XML, which is structured and therefore we thought it would be easy to use for our purposes of capturing the relationships between packages, classes, and methods. However, it turned out that we had to obtain these separately by re-parsing the XML with a Python script using regexes. Since Doxygen did not identify the different parts of the Java syntax and resolve the declarations of method calls, it became clear that Doxygen would not work as our parser, which led us to use JavaParser [24].

7.1.5 Parsing raw types. Even though JavaParser can only parse Java code, the provided APIs make resolving method calls very easy. Their symbol solver takes care of all of the hard work of finding where a method was declared, which allows us to find the types of the parameters so as to match it up with an actual declaration. However, JavaParser is unable to resolve the types of the method.

7.1.6 Planning. If we had to do this again, we would want to spend much more time to scope the project and how we would implement things. The regular progress deadlines proposed through the quarter helped us a lot but were ultimately not enough for this project with many moving pieces and its exploratory nature.

7.2 Future Work

In this section, we present avenues we would like to explore and we think would improve our tool in terms of performance and productivity for users.

7.2.1 Lazy parsing. Lazy parsing, or essentially just-in-time parsing where the code base is parsed on an as-needed basis, would significantly improve both the tool and its performance. The result of the initial parse could be saved and added to as the developer explores more of the code base. As code is updated, the tool could detect these changes and re-run the parser only on the changed files.

7.2.2 Persistent visualization. Currently, the visualization uses a physics force simulation to determine the placement of methods, classes, and packages. This however does not meet our ideal criteria of having a deterministic way to visualize a codebase, and one that would not change very much with the addition of either new methods, classes, or packages. The CodeMap prototype used a force simulation as a way to get the visualization up and running quickly, and due to the constrained time frame, we were unable to explore other options.

[5] https://www.doxygen.nl/index.html

7.2.3 Dynamic shapes. We are interested to see if changing the shape of the classes and the packages in function of their properties (e.g., the number of methods or lines of code in a class) would be more intuitive to developers rather than fixed-shaped as it is now. Previous work, notably software cities[27], use this principle already; the representation of the classes and packages change in function of their measured attribute (e.g., lines of code, complexity, number of bugs).

8 CONCLUSION

This project was challenging to complete but we learned a lot along the way. The technical part was more difficult than expected, especially with multiple moving parts using technologies we did not have experience with. Nonetheless, we were able to make the most of it and learn to find alternatives when we hit roadblocks.

Despite a small population size in our user study, making quantitative results difficult to leverage with certainty, we found compelling evidence in the participants feedback in favor of our approach. Notably, we observed that the participants largely enjoyed using CodeMap despite the latest implementation's flaws. They found it useful to understand horizontal relationships (i.e., method to method, class to class) and to get an overview of the structure of the source respective to each level.

Working in a group containing more than half the class's students came with its advantages and its disadvantages: while we were able to push the project further than in a group of two and work on multiple components at once, there was a non-negligible overhead required to plan, organize, and coordinate the project.

Overall, we are quite pleased with our final result, and we are eager, based on the excellent feedback we received from conducting this case study, to explore more aspects of interactive source code visualizations and refine our solution further.

ACKNOWLEDGMENTS

We want to thank the CSE503 W21 class and PLSE lab members for their participation in the user study, their interest, and their feedback on our project.

REFERENCES

[1] Carol V Alexandru, Sebastian Proksch, Pooyan Behnamghader, and Harald C Gall. 2019. Evo-clocks: Software evolution at a glance. In *2019 Working Conference on Software Visualization (VISSOFT)*. IEEE, 12–22.

[2] A. Begel and M. Resnick. 2000. LogoBlocks: A Graphical Programming Language for Interacting with the World.

[3] John Brooke. 1996. Sus: a "quick and dirty'usability. *Usability evaluation in industry* 189 (1996).

[4] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. 2011. A Controlled Experiment for Program Comprehension through Trace Visualization. *IEEE Trans. Softw. Eng.* 37, 3 (May 2011), 341–355. https://doi.org/10.1109/TSE.2010.47

[5] Haci Ali Duru, M. Çakir, and Veysi Isler. 2013. How Does Software Visualization Contribute to Software Comprehension? A Grounded Theory Approach. *International Journal of Human–Computer Interaction* 29 (2013), 743 – 763.

[6] Florian Fittkau, Santje Finke, Wilhelm Hasselbring, and Jan Waller. 2015. Comparing trace visualizations for program comprehension through controlled experiments. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 266–276.

[7] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. 2017. Software landscape and application visualization for system comprehension with ExplorViz. *Information and Software Technology* 87 (2017), 259 – 277. https://doi.org/10.1016/j.infsof.2016.07.004

[8] Paul Gestwicki and Bharat Jayaraman. 2005. Methodology and Architecture of JIVE. In *Proceedings of the 2005 ACM Symposium on Software Visualization* (St.

Louis, Missouri) *(SoftVis '05)*. Association for Computing Machinery, New York, NY, USA, 95–104. https://doi.org/10.1145/1056018.1056032

[9] Johann Grabner, Roman Decker, Thomas Artner, Mario Bernhart, and Thomas Grechenig. 2018. Combining and Visualizing Time-Oriented Data from the Software Engineering Toolset. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 76–86.

[10] Elke Franziska Heidmann, Lynn von Kurnatowski, Annika Meinecke, and Andreas Schreiber. 2020. Visualization of Evolution of Component-Based Software Architectures in Virtual Reality. In *2020 Working Conference on Software Visualization (VISSOFT)*. IEEE, 12–21.

[11] Truong Ho-Quang, Alexandre Bergel, Arif Nurwidyantoro, Rodi Jolak, and Michel RV Chaudron. 2020. Interactive Role Stereotype-Based Visualization To Comprehend Software Architecture. In *2020 Working Conference on Software Visualization (VISSOFT)*. IEEE, 122–132.

[12] Danny Holten, Bas Cornelissen, and Jarke Wijk. 2007. Trace Visualization Using Hierarchical Edge Bundles and Massive Sequence Views. *VISSOFT 2007 - Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 47–54. https://doi.org/10.1109/VISSOF.2007.4290699

[13] Swaminathan Jayaraman, Bharat Jayaraman, and Demian Lessa. 2016. Compact visualization of Java program execution: Compact visualization of Java program execution. *Software: Practice and Experience* 47 (01 2016). https://doi.org/10.1002/spe.2411

[14] JFree. 2021. JFreechart. https://github.com/jfree/jfreechart

[15] Rainer Koschke and Marcel Steinbeck. 2020. Clustering Paths With Dynamic Time Warping. In *2020 Working Conference on Software Visualization (VISSOFT)*. IEEE, 89–99.

[16] Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger. 2005. CodeCrawler: An Information Visualization Tool for Program Comprehension *(ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 672–673. https://doi.org/10.1145/1062455.1062602

[17] François Lemieux and Martin Salois. 2006. Visualization techniques for program comprehension. *New Trends in Software Methodologies, Tools and Techniques (eds. H. Fujita and M. Mejri)* (2006), 22–47.

[18] S. Jayaraman B. Jayaraman D. Lessa. 2016. Compact visualization of Java program execution. In *Journal of Software: Practice and Experience*, Vol. 47. Springer. https://doi.org/10.1007/s12650-020-00727-x

[19] Leonel Merino and Oscar Nierstrasz. 2018. *The medium of visualization for software comprehension*. Ph.D. Dissertation. Universität Bern.

[20] Orange Data Mining. [n.d.]. Orange Data Mining – Data Mining. https://orangedatamining.com/

[21] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720

[22] Scratch. [n.d.]. Scratch - Imagine, Program, Share. https://scratch.mit.edu/

[23] Unity Technologies. [n.d.]. How to make a video game without any coding experience. https://unity.com/how-to/make-games-without-programming

[24] Danny van Bruggen. [n.d.]. JavaParser. https://javaparser.org/

[25] VISSOFT. 2013. *VISSOFT IEEE Working Conference on Software Visualization.* https://vissoft-conferences.dcc.uchile.cl/

[26] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *2020 Working Conference on Software Visualization (VISSOFT)*. IEEE, 110–121.

[27] Richard Wettel, Michele Lanza, and Romain Robbes. 2011. Software Systems as Cities: A Controlled Experiment. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 551–560. https://doi.org/10.1145/1985793.1985868

A LINKS

- **Project's repository** https://github.com/amyjzhu/503-hacking/
- **Experimental Protocol** https://bit.ly/3ltanLY
- **Treatment A** https://bit.ly/3vBNMSd
- **Treatment B** https://bit.ly/3cJxFcE
- **Analysis Notebook** https://bit.ly/3qUYSyh
- **Presentation** http://bit.ly/3tyDf8G
- **Survey** https://forms.gle/avgqPEa6rV7c7Vbn9