University of Bremen

Universität Bremen

Faculty 3 — Mathematics & Computer Science

Master's Thesis

Building Code Cities using the Language Server Protocol

— DRAFT —

Falko Galperin

1. Reviewer Prof. Dr. Rainer Koschke
Working group *Software Engineering*

2. Reviewer Prof. Dr. Ute Bormann
Working group *Computer Networks*

January 14, 2025

Abstract

In this master's thesis, we investigate how to generate code cities by using the *Language Server Protocol* (LSP), and more specifically, how to integrate it into the code city tool SEE. Apart from investigating how feasible this is, another research question is whether code cities are a suitable means to present LSP information as compared to traditional IDEs.

After giving a short overview of LSP and SEE, the first main part of the thesis deals with the implementation of LSP into the code city tool SEE, with a special focus on how code cities can be algorithmically generated using the information provided by the protocol. We also conduct a brief technical evaluation to benchmark the performance of the city generation algorithm, with the result that using the algorithm is feasible for projects of up to around 100 000 lines of code, but takes too long for projects bigger than that, assuming no node and edge types are filtered out.

The second main part is about evaluating the second research question in a user study, which compares SEE against the popular IDE VSCode across three program understanding tasks. We evaluate the two tools on the dimensions of correctness, time, and usability. For usability, we use the *After-Scenario Questionnaire* (ASQ) as a post-task questionnaire, and the *System Usability Scale* (SUS) as a post-study questionnaire. Each of the $n = 20$ participants solved the tasks in an online study using both SEE and VSCode,with a different software project per tool.

Our analysis revealed significantly ($p < 0.05$) better results for VSCode in terms of speed, usability as measured by the SUS, and usability as measured by the "effort" question of the ASQ for some tasks, with other factors having no significant differences between SEE and VSCode. As a result, we conclude that code cities are a suitable means to present LSP information, but developers work faster and report higher usability when using traditional IDEs, at least in the case of SEE versus VSCode. The reasons for this are not fully apparent from the data we collected and may be a good avenue for future research.

Declaration

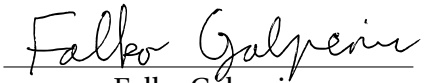
I hereby affirm that I have written the present work independently and have used no sources or aids other than those indicated. All parts of my work that have been taken from other works, either verbatim or in terms of meaning, have been marked as such, indicating the source. The same applies to drawings, sketches, pictorial representations and sources from the Internet, including AI-based applications or tools. The work has not yet been submitted in the same or a similar form as a final examination paper.

Additionally:

- I agree that my thesis may be viewed by third parties in the university archive for academic purposes.
- I agree that my thesis may be viewed by third parties for academic purposes in the university archive after 30 years (in accordance with §7 para. 2 BremArchivG).
- I agree that the work I have submitted and written will be stored permanently on the external server of the plagiarism software currently used by the University of Bremen, in a library belonging to the institution (accessed only by the University of Bremen), for the above-mentioned purpose.

With my signature, I confirm that I have read and understood the above explanations and confirm the accuracy of the information provided.

Bremen, January 14, 2025



Falko Galperin

Acknowledgements

First and foremost, I want to thank Rainer Koschke for his general help, quick answers to any questions that I had, and availability for detailed discussions on various aspects of both the implementation and user study of this thesis. Additionally, I am thankful for every participant who took part in my user study, with additional gratitude towards those who even shared the link to the evaluation in their own circles. Finally, I want to thank all proofreaders (**TODO:** Add proofreader names here) who, despite its length, read through this thesis to catch any mistakes. If you're reading this right now, you must be a proofreader, so good luck, and thanks!

Contents

TODO: Go through all LaTeX warnings

1 Introduction

Language Servers have become an increasingly popular way to offer "code smarts," such as code autocomplete or the highlighting of errors. However, even though the Language Server Protocol has been written to be broadly compatible with any development tool, in practice, it has only been used for traditional IDEs and editors instead of more modern and unconventional approaches like code cities. I am of the opinion that code cities can benefit from making use of this protocol, so the core focus of this master's thesis is going to be combining it with code cities, specifically SEE.

The main novel contribution will be a way of creating code cities using only information provided by the Language Server Protocol, while additional contributions consist of the integration of Language Server Protocol-based functionality into code cities as well as their integration into SEE's code windows. Finally, the penultimate Chapter 4 describes a controlled experiment (with $n = 20$ participants) in which code cities are compared to traditional IDEs via a user study—this serves as an evaluation for this thesis and for code cities as IDE replacements in general.

In this chapter, apart from explaining some formatting semantics, we will examine the motivation and basics behind each of the central concepts (i. e., code cities, the Language Server Protocol, and the integration of the two), name the goals and research question of the thesis, and finally describe the structure of upcoming chapters.

1.1 Format

This document uses many technical terms that not every reader may know. To remedy this, starting from the next section, whenever a technical term or acronym appears for the first time, it will be printed in this color, and an explanation of that term will appear in a footnote of the same color. Terms that are already explained in the text itself will not receive such a footnote. These terms and acronyms are collected within the glossaries in

Appendices C and D—all mentions of such terms also link (in the digital version of this thesis, at least) to the corresponding part of the glossary. Hence, if you come across a technical term or abbreviation you are not familiar with, simply try clicking on it. In total, the following colors are used to convey specific meanings:

- **Maroon** for the introduction of a glossary term or acronym,
- **Fuchsia** for internal links (e. g., to other sections),
- **Blue** for external links (e. g., to web pages),
- **Green** for cited literature, and
- **Cyan** for references to attached files (see Appendix E).

1.2 Motivation

As mentioned at the beginning of this chapter, this master's thesis is about *integrating* the *Language Server Protocol* into *Code Cities*. I will motivate each of these italicized central points individually in the following sections. Note that these will get more thorough explanations in Chapter 2.

1.2.1 Code Cities & SEE

Visualization in general often helps facilitate the understanding of complex systems by representing them with a simplified visual model. This can be especially useful in the area of software engineering, where it is often hard to get an intuitive overview of large software systems when only equipped with standard tools, like Integrated Development Environments (IDEs)*. One such software visualization—called Software Engineering Experience (SEE)—is being developed at the University of Bremen and will be a focal point of this thesis.

SEE is an interactive software visualization tool using the code city metaphor in 3D, developed in the Unity game engine. It features collaborative "multiplayer" functionality across multiple platforms[1], allowing multiple participants to view and interact with the same code city together.

[1] Notably, besides usual desktop and touchscreen-based environments, virtual reality (e. g., via the *Valve Index*) is supported as well.

*IDE: Editor for source code with features that are useful for development (e. g., highlighting errors). Examples are VSCode or IntelliJ.

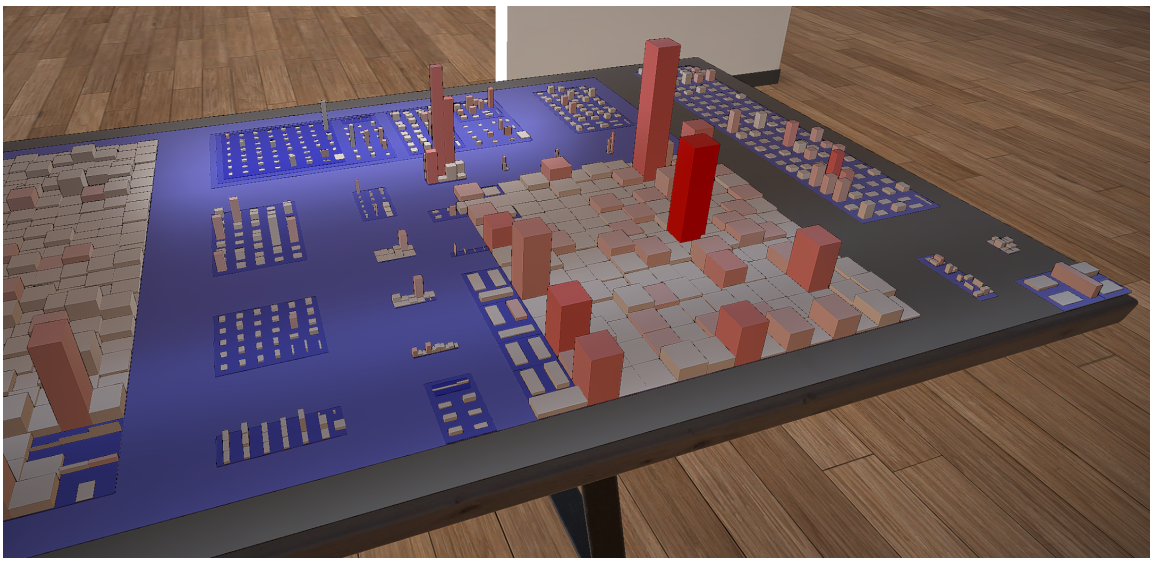


Figure 1.1: A code city visualized in SEE.

In the code city metaphor, software components are visualized as buildings within a city. Various metrics from the original software can then be represented by different visual properties of each building—for example, the Lines of Code (LOC)* within a file might correlate to the height of the corresponding building. Relationships between software components, such as where components are referenced, are instead represented by edges drawn between the respective buildings. The exception to this are part-of relations, that is, relations that describe which component belongs to which other component. These are instead visualized in SEE by buildings being nested within their corresponding "parent" building. In this way, the data model of SEE can be represented as a graph in which the software components are the nodes and the relationships are the edges. We will go into detail on this view of SEE's data model in Section 2.1.2.

For example, in Fig. 1.1, we can see the source code of the SpotBugs project [**spotbugs**] rendered as a code city. A few very tall buildings—indicating that the respective component is very big and that a refactoring into smaller pieces may be in order—immediately jump out. Additionally, this visualization also makes the number of methods readily apparent: the redder a node, the higher its method count. Figure 1.2 instead visualizes the modeled architecture of a very small system, as compared to a city "empirically" generated by an implementation like in the previous example. Here, we can also see yellow edges between the components, in this case representing desired references that should be present between components.

*LOC: The number of lines in a source code file.

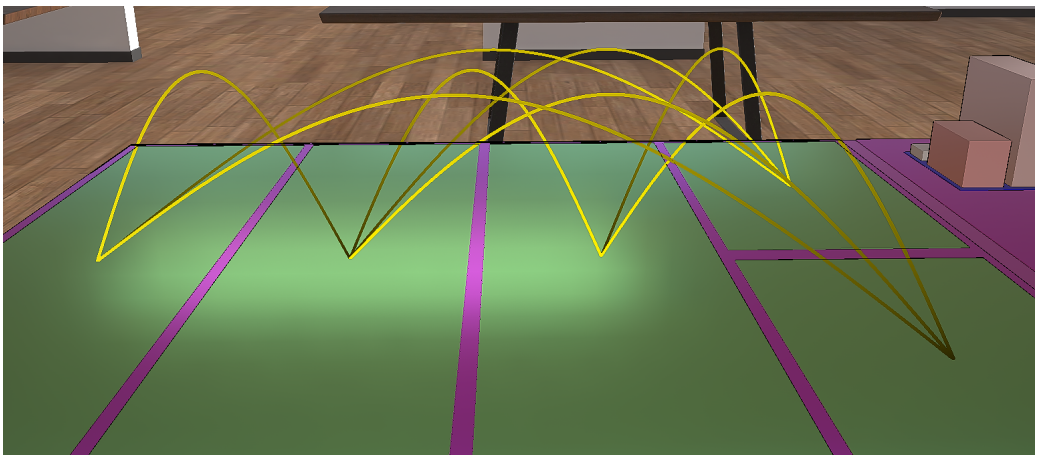


Figure 1.2: An example of what edges can look like in SEE.

1.2.2 Language Server Protocol

As stated on its website:

> Adding features like auto complete, go to definition, or documentation on hover for a programming language takes significant effort. Traditionally[,] this work had to be repeated for each development tool, as each tool provides different APIs for implementing the same feature.
>
> A Language Server is meant to provide the language-specific smarts and communicate with development tools over a protocol that enables inter-process communication.
>
> The idea behind the Language Server Protocol (LSP) is to standardize the protocol for how such servers and development tools communicate. This way, a single Language Server can be re-used in multiple development tools, which in turn can support multiple languages with minimal effort. ([**lsp**])

Since LSP[2] is a central component of my master's thesis, I have created a diagram in Fig. 1.3 in the hope to strengthen intuitions around the motivation and use of the protocol. While the LSP specification has originally been created by Microsoft, it is by now an open-source project[3], where changes can be actively proposed using issues or pull requests. Apart from the specification itself, a great number of open-source implementations of Language Servers for all kinds of programming languages from Ada to Zig exist. A partial overview of

[2] Note that I will often refer to the Language Server Protocol as just "LSP" instead of "the LSP" (e. g., "IDEs use LSP") from now on, as this is how the specification [**lsp**] does it as well.
[3] Available at `https://github.com/Microsoft/language-server-protocol` *(last access: 2024-09-11)*.

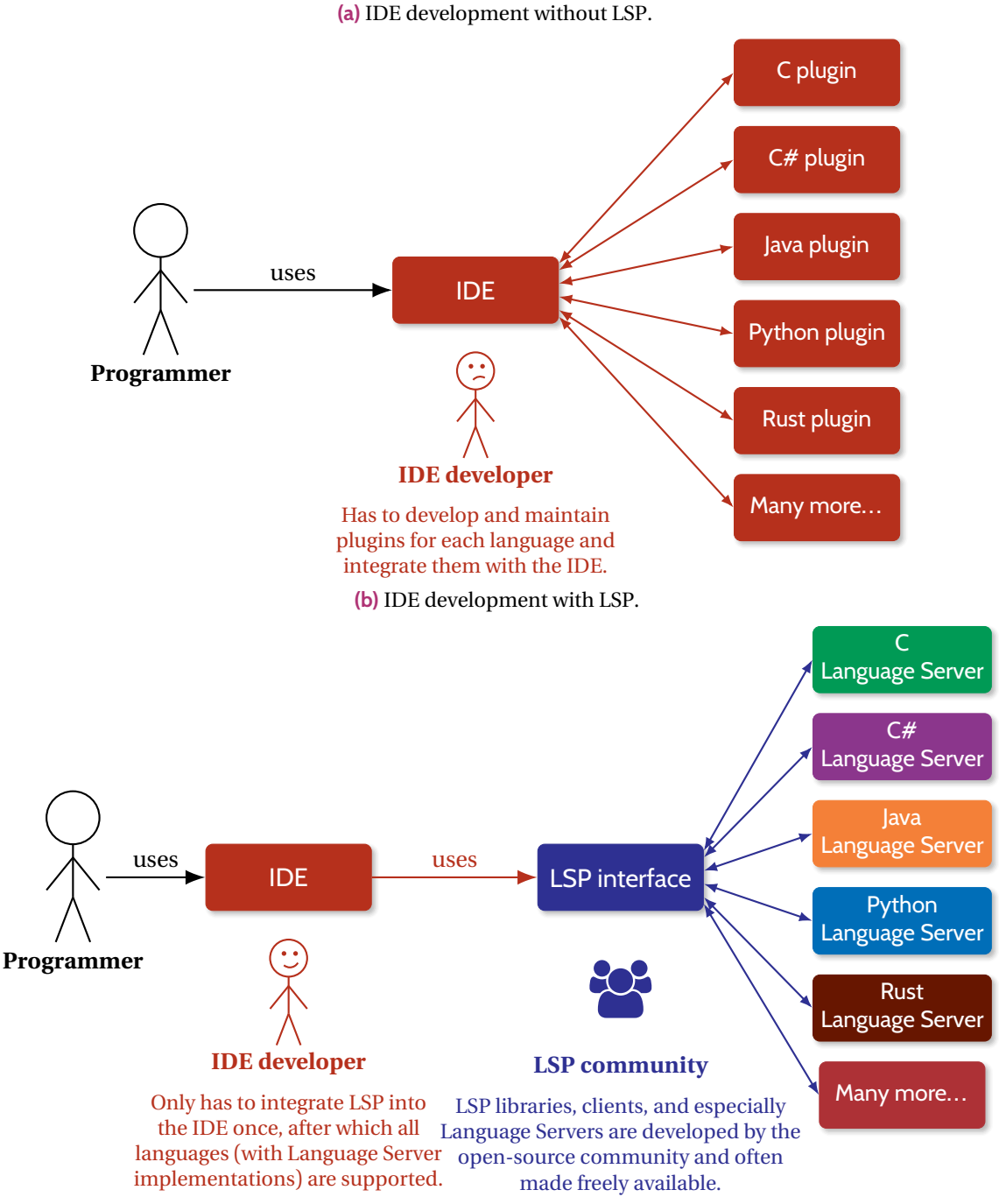


Figure 1.3: An illustration of how LSP can help simplify IDE development.

available implementations is listed at https://microsoft.github.io/language-server-protocol/implementors/servers/ *(last access: 2024-09-11)*.

The protocol introduces the concept of so-called capabilities, which define a specific set of features a given Language Server (and Language Client*) support. These include navigational features, like the ability to jump to a variable's declaration, and editing-related features, such as autocomplete. To give a specific example of what an LSP capability might look like in practice, the `texlab` [**forster2025**] Language Server for LaTeX—which I am using while writing this document—provides a list of available packages when one starts typing text after "`\usepackage{`". Additionally, for the currently hovered package, a short description of it is displayed. A screenshot of this behavior within the Neovim editor is provided in Fig. 1.4.

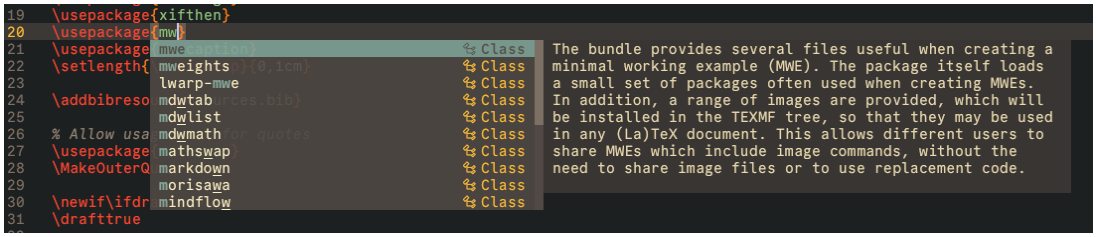


Figure 1.4: An example of the `texlab` Language Server running in Neovim.

The counterparts to Language Servers are the Language Clients: These are the IDEs and editors that incorporate the Language Server into themselves. Examples for IDEs that support acting as a Language Client in the LSP context include *Eclipse, Emacs, IntelliJ, Vim*, and *Visual Studio Code (VSCode)***, the latter of which becomes relevant in Chapter 4.

1.2.3 Integration

Currently, code cities in SEE are rendered by reading pre-made Graph eXchange Language (GXL)† files, which can be created by the proprietary Axivion Suite. This approach has the disadvantage of only supporting languages supported by the Axivion Suite, as well as making regenerating cities (e. g., if the source code changed) fairly cumbersome. Another current shortcoming of SEE is that information about the source code available to the user is limited when compared to an IDE—for example, quickly displaying documentation

*Language Client: A development tool, such as an IDE, that supports LSP and can hence integrate language-specific features into itself using compatible Language Servers.

**VSCode: A proprietary, but free IDE developed by Microsoft with a plugin system from which LSP originated. See `https://code.visualstudio.com` *(last access: 2024-10-05)*

†GXL: A file format for graphs, used in SEE for representing dependency and hierarchy graphs of software projects.

for a given component by hovering over it is not supported. This is where the Language Server Protocol can help. Figure 1.5 visually explains what the integration done in this thesis entails.

1.3 Goals & Research Questions

The goal of this master's thesis—as outlined in Section 1.2.3—is to integrate the Language Server Protocol into SEE by making it a Language Client, then evaluate this implementation by comparing it with traditional IDEs in a user study. To this end, the main contribution is a way of generating code cities using the Language Server, where all the information obtainable by relevant LSP capabilities should be manifested[4] in the city in a suitable way. This is an unintended (or at the very least, unusual) use of LSP and may require some experimentation.

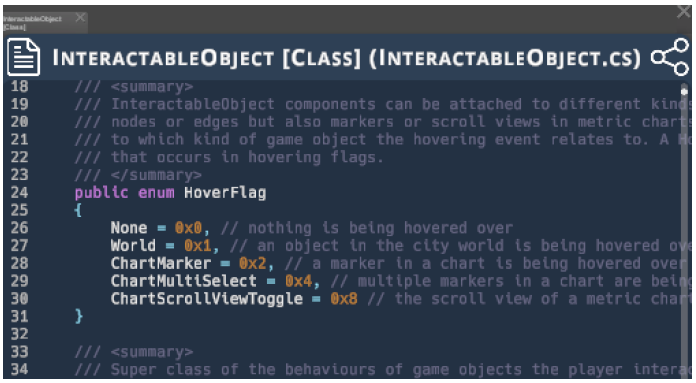


Figure 1.6: A code window.

Apart from the code cities, SEE also provides the so-called code windows, in which the source code of a specific component can be viewed in a similar way as in an IDE. This can be seen in Fig. 1.6. An additional goal of this master's thesis is to enhance the functionality of the code windows by implementing more IDE-like behavior into it (e. g., allowing users to go to a variable's declaration, or displaying diagnostics inline) using the Language Server.

It should be noted that any LSP capabilities which involve modifying the underlying software project is out of scope for this master's thesis. Rewriting the code windows to be editable (in a way that distributes the edits over the network) is complex enough to warrant its own thesis [**moritz**], not even taking into account that this would also more than double the number of capabilities that would then become useful to implement. As such, only capabilities that are "read-only" (i. e., that do not modify the source code or project structure in any way) will be implemented as part of the thesis. Consult Section 2.2.2 for a full list of LSP capabilities implemented as part of this thesis.

[4] Since there is a lot of diverse data available via LSP, it makes sense to only immediately display the most pertinent information and make the rest of it available upon request within SEE's user interface.

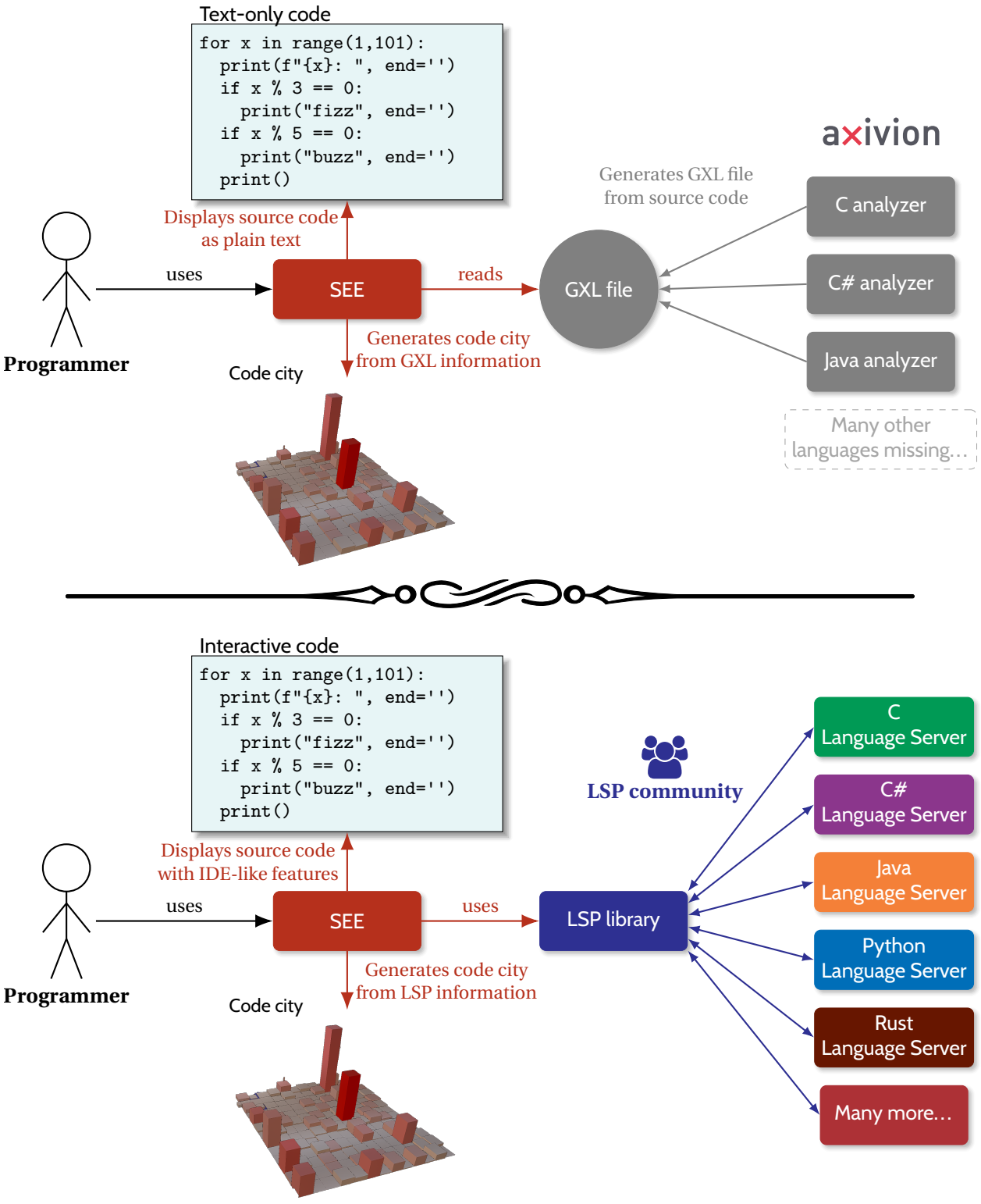


Figure 1.5: A simplified illustration of how SEE usually gains and uses information about software projects (top half), and how the integration of LSP changes that (bottom half).

Additionally, I will not implement the C# interface to the LSP (i. e., translating between C# method calls and JavaScript Object Notation—Remote Procedure Calls (JSON-RPCs)*). There already exist well-made interfaces for this purpose[5], and the focus of the thesis will be on the integration of the protocol's *data* into SEE's code cities and code windows, not the integration of the protocol itself.

Hence, the goals for this thesis can be summarized as follows:

- Integrating an LSP framework into SEE and allowing users to manage Language Servers from within SEE
- Making SEE a Language Client, such that:
    1. Code cities can be generated directly from source code directories, using Language Servers that SEE interfaces with,
    2. Code windows gain "read-only" IDE-like functionality, covering behavior of capabilities listed in Section 2.2.2, and
    3. Code cities gain similar functionality (where applicable), such as displaying relevant documentation when hovering over a node.
- Evaluating the above empirically in a controlled experiment via a user study.

The main research questions that I want to answer in this thesis are as follows:

RQ1 Is it feasible (i. e., realistically doable with usable results) to generate code cities using the Language Server Protocol?

RQ2 Are code cities a suitable means to present LSP information to developers as compared to IDEs + tables (on the dimensions of speed, accuracy, and usability)?

1.4 Thesis Structure

We begin by examining the Language Server Protocol and the concept of code cities along with SEE more closely in Chapter 2. Next, in Chapter 3, we take a look at my implementation of the LSP-based code city generation algorithm, alongside additional contributions to visualize this information in the cities and code windows of SEE. In the course of this, we will answer RQ1. To answer RQ2, I will carry out a user study comparing an LSP-enabled

[5] Such as OmniSharp's implementation of LSP in C# [**csharplanguageserverprotocol2023**], which I will use.

*JSON-RPC: A remote procedure call protocol that uses JSON as its encoding, supporting (among other features) asynchronous calls and notifications. It is used as the base for LSP (even though LSP is technically not a remote protocol).

IDE (specifically, VSCode) with an LSP-enabled code city visualization (specifically, SEE) and report on its results in Chapter 4. Finally, we wrap up in Chapter 5 by summarizing the thesis's results and providing an outlook on additional ideas for the implementation, while also listing some possible avenues of further research.

2

Concepts

Outlining the central concepts behind this thesis is a crucial first step before tackling the implementation, so that we can form a concrete idea of which parts of LSP are well-suited to being integrated into code cities. Thus, we will examine the concept of code cities, where we will use SEE as a concrete implementation (which we do in Section 2.1), as well as the Language Server Protocol (which we do in Section 2.2). For the former, we will go over the essential features that we need to work with in the implementation, with a special focus on the project graph. As for the latter, we will go over the available capabilities and take a look at both existing Language Servers and Language Clients to get an idea of what the Language Server Protocol offers and how it is most commonly used. For both topics, we will focus on the parts relevant to the implementation and evaluation—for example, we will only explore those capabilities in detail that actually end up being used in this thesis.

2.1 SEE

As explained in Section 1.2.1, SEE is an interactive software visualization tool using the code city metaphor in 3D, with the aim to make it easier to work with large software projects on an architectural level. To name a few example scenarios in which using code cities could be useful, compared to using traditional IDEs:

- Senior developers may use it in its "multiplayer" mode to explain the structure of their software to newcomers.
- Project planners could visualize code smells* to find candidates for refactoring [**galperin2021**].

*Code smell: certain structures in source code that suggest that a refactoring is in order, such as duplicated code, or a very long method [**fowler2019**].

- Software architects can find violations of the planned architecture using SEE's reflexion analysis*.

SEE is an open-source project, currently hosted on GitHub[6]. It is a research project at the University of Bremen, where it has been in development since 2017 [**ganser2017**], and where it is frequently offered as a bachelor or master project for the students. While it was at first a project for the *Unreal Engine*, the game engine has been switched to *Unity* relatively early, mainly because its editor eased development due to the ability to reload the User Interface (UI) without having to restart the whole engine.

After going over the basics of how SEE works, I will give an explanation and formalization of the project graph—as this is the central component in the LSP-based code city-generation algorithm—followed by a brief overview over other relevant features for this thesis.

2.1.1 Basics

As mentioned before, when analyzing a software project in SEE, its source code structure is reflected in the rendered buildings, where connections between the components, such as references, are displayed using edges rendered as splines (this will be elaborated in Section 2.1.2), as can be seen in Fig. 1.2, for example. Metrics of the project, such as the McCabe complexity [**mccabe**], can then be mapped onto visual properties of this rendering, such as its color, width, or height.

So far, the kind of visualization described here would still be possible in two dimensions, but adding a third dimension gives us the benefit of having another axis to map metrics onto, as well as making it a more natural environment for humans to interact with. By giving users the ability to walk, move the camera around, and so on, the code city can be navigated in a richer and more intuitive manner than if it were just a simple 2D image. This also makes it possible to offer SEE as a virtual reality application. Virtual reality gives users an even more immersive and natural way of interacting with the environment, leading to improved spatial memory of the city [**tavanti2001**]. It also makes it a more fitting environment for multiple people. Each player can have their own avatar placed in the scene, interacting with one another and the city, communicating via voice chat.

As an interactive game-like project such as this comes across more clearly in a video than in mere text and disconnected snapshots, readers can take a look at this introductory video

[6] `https://github.com/uni-bremen-agst/SEE` *(last access: 2024-09-18)* (but note that, to clone this repository, you additionally need access to the Git LFS counterpart hosted at the University of Bremen's GitLab, as this is where paid plugins for SEE are hosted.)

*Reflexion analysis: The process of comparing the architecture and implementation of a software project and finding incongruencies between the two.

for an overview of SEE: https://www.youtube.com/watch?v=fYMRpmuk7To *(last access: 2025-01-06)*.

2.1.2 Project Graph

We will now take a look at the graph representing the source structure of the analyzed software projects in SEE. One of the main goals of this thesis is generating such a graph using LSP, as opposed to the currently available methods of generating this graph, which require access to the proprietary Axivion Suite to create the GXL files used in SEE.

These graphs can be formalized as $G = (V, E, a, s, t, \ell)$, with:

- V being a set of nodes and E being a set of edges.
- $a : (V \times \mathcal{A}_K) \rightharpoonup \mathcal{A}_V$ associating nodes and an attribute name ($\in \mathcal{A}_K$) with a value ($\in \mathcal{A}_V$). Note that this is a partial function.
- $s : E \to V$ denoting the source node of an edge.
- $t : E \to V$ denoting the target node of an edge.
- $\ell : E \to \Sigma$ providing a label for an edge over some alphabet Σ. Also, $\texttt{partOf} \in \Sigma$ such that the subgraph $(V, \{e \in E \mid \ell(e) = \texttt{partOf}\}, a, s, t, \ell)$ is a polytree*.

Explained in natural language: The graph's nodes (V) can be connected by directed edges (E)—these are not necessarily unique for each possible tuple of nodes, meaning that there can be more than one edge between the same two nodes. A node represents a component in the source code (e. g., a class), while edges represent relationships between them (e. g., inheritance). This is an attributed graph. Specifically, each node can have multiple attributes (distinguished by attribute keys), while each edge always has one label that denotes the type of its relationship. The label `partOf` additionally has a special meaning: edges with this type induce the hierarchy of the source code (e. g., classes contained in files). Hence, if we look at the graph that removes all edges except for those with label `partOf`, and we make these edges undirected, we get a tree.

To illustrate, a few examples of attributes are:

- `Source.Path`: the path to the file the element is contained in.
- `Source.Name`: the name of the element within the source code.
- `Type`: the type of the element (e. g., "method").

*Polytree: A directed acylic graph which also has no cycles in undirected form.

Another attribute that is also relevant for LSP is the `Source.Range`, which is often used in the upcoming Section 2.2 and Chapter 3. It describes a contiguous portion of source code. We formally define the domain of ranges $\mathcal{R}$ as the Cartesian square of positions $\mathcal{P}^2$, whereas the domain of positions $\mathcal{P}$ is defined as the Cartesian square of natural numbers (including zero), that is, $\mathcal{P} = \mathbb{N}_0^2$. Hence, as a whole, $\mathcal{R} = \mathcal{P}^2 = \mathbb{N}_0^4$. Semantically, a position $(l, c) \in \mathcal{P}$ describes a zero-indexed line l and a zero-indexed character offset c, relative to the beginning of the line. A range $(b, e) \in \mathcal{R}$ can be understood as a beginning position b (inclusive) and an ending position e (exclusive). We will also occasionally "decompose" those positions and refer to a range r in decomposed form as $r = (b_l^r, b_c^r, e_l^r, e_c^r)$. For example, the interval of lines that the range r (partially or completely) covers is then given by $[b_l^r, e_l^r)$.

I should note that the model presented here is a simplification of SEE's actual data model for source code graphs. For example, in reality, edges can have multiple attributes, there can be multiple edge types other than `partOf` that have the polytree property, and so on. This is just what is needed to understand the LSP-based city generation algorithm presented in Section 3.2.

2.1.3 Other Relevant Features

Apart from the project graph, there are a few other features of SEE we need to go over, as they become relevant when integrating LSP into code cities (in Section 3.3) and code windows (in Section 3.4).

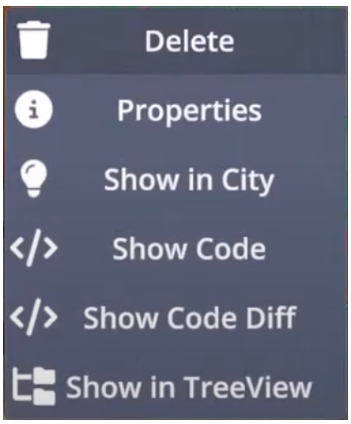


Figure 2.1: A screenshot of SEE's context menu before the changes in Chapter 3.

Context Menu When right-clicking any node or edge, a menu opens with several context-dependent options, shown in Fig. 2.1. These include the option to delete the

element, to highlight it within the code city, to open its corresponding code window, and others. We will expand this menu with LSP-specific actions in the course of Section 3.3.

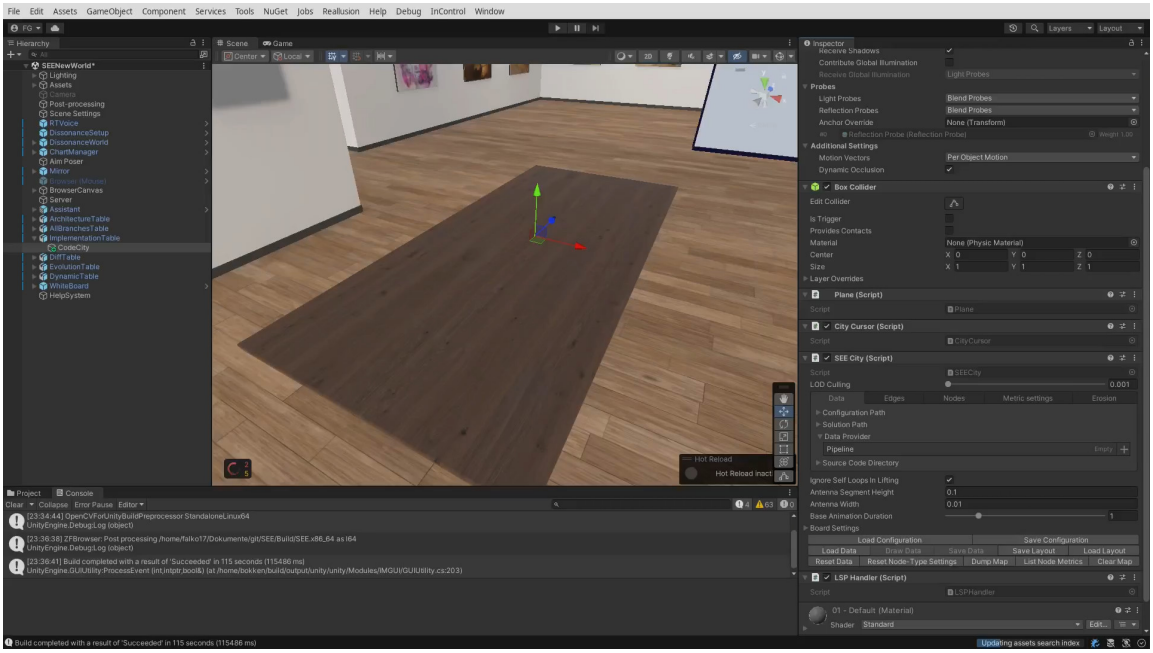


Figure 2.2: The Unity Editor with a scene in SEE opened.

City Editor To actually generate a code city—assuming one has a GXL file for this purpose—a customized UI component within the Unity Editor* exists in SEE. Here, a variety of options can be configured, such as the layout of the city, the mapping of metrics to visual attributes, or the graph providers, which create a project graph based on optional input parameters (such as the aforementioned GXL file).

Figure 2.2 shows what the Unity Editor looks like in SEE: On the left are the objects of the scene, the middle shows a preview, the bottom left displays log messages, and finally, the components (including the city editor component) can be seen on the right. After implementing the city-generation algorithm for LSP, a new graph provider with its own Unity Editor UI shall be implemented as well.

Code Windows As explained in Section 1.3, there is also the option of opening code windows to view the source code of a component, an example of which is shown in Fig. 1.6. Currently, these windows do little more than lexer-based syntax highlighting, so a goal is to include more IDE-like behavior by using functionality offered by LSP.

*Unity Editor: The main UI of the Unity game engine, in which scenes can be set up, components can be configured, the game itself can be run, etc. Note that it is only used for development purposes, and hence not included within generated builds of a game.

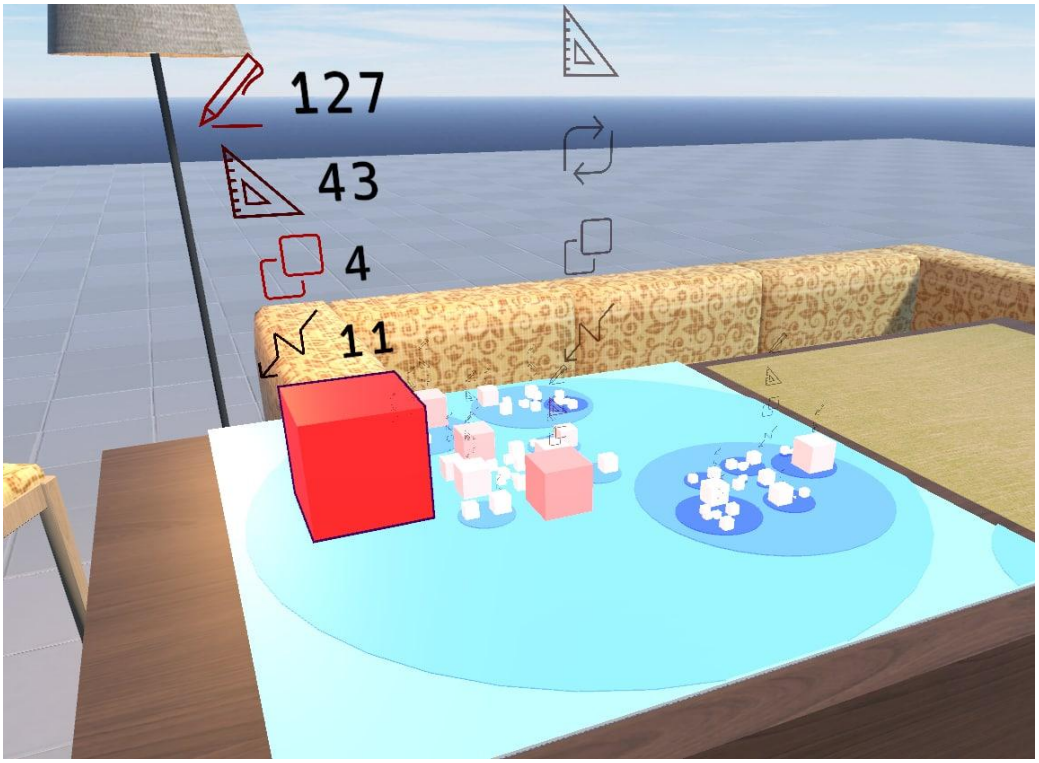


Figure 2.3: An example of code smells visualized as erosion icons in SEE.

EROSION ICONS Following my bachelor's thesis [**galperin2021**], SEE offers the possibility to indicate the number of code smells per component using the so-called *erosion icons*. These are essentially small icons that can be put above each node, as shown in Fig. 2.3. The size and color of these erosion icons can then indicate the quantity of code smells for that given node. A controlled experiment has suggested that this gives developers a quicker, more intuitive overview over the distribution of code smells within a project, compared to traditional (i. e., tabular) ways of displaying this information [**galperin2022**]. While LSP does not offer a standardized way of offering code smell information, we can use its diagnostics capability (see Section 2.2.2) for the same purpose.

2.2 Language Server Protocol

The very basic concepts behing LSP have already been explained in Section 1.2.2. The protocol aims to make it easier for IDEs to support more programming languages—specifically, to support language-specific capabilities, which we will go over in Section 2.2.2. It was originally developed for Microsoft's editor VSCode and was later converted into an open-source specification[7] (though there are still VSCode-specific extensions to LSP that are

[7] Available at `https://github.com/microsoft/language-server-protocol` *(last access: 2024-10-05)*.

not in the official specification today). LSP has found widespread use: An overview page by Microsoft lists at least 269 Language Servers[8] (i. e., servers offering support for some programming language) and 61 Language Clients[9] (i. e., IDEs or development tools). The current (as of January 14, 2025) version of the protocol is 3.17, with version 3.18 being under active development.

2.2.1 Basics

Messages in the Language Server Protocol are built using JSON-RPC, which uses JSON to encode both requests (consisting of a method name and parameters) and responses (consisting of either a result object or an error object) for procedure calls. Requests include an ID that a response by the server can then reference to match it up with the request it is a reply to. There also so-called *notifications*, which are in essence requests without an ID, intended to send a message that does not warrant a response [**jsonrpc**; **json**]. In LSP, any message sent between the Language Client and Language Server consist of a header—describing the length and type of the content—followed by the content, which is always a JSON-RPC payload. While the specification does not mandate it, it lists some recommended communication channels on which the protocol messages can be sent, those being `stdio` (using standard input/output), `pipe` (using Windows's pipes), `socket` (using a socket), and `node-ipc` (IPC communication over Node.js[10]).

Listing 2.1: Example specification of request and response objects for the Hover capability.

```
interface HoverParams {
  textDocument: string; /** The text document's URI in string form */
  position: { line: uinteger; character: uinteger; };
}

interface HoverResult {
  value: string;
}
```

The specification lists all available types for requests and responses as TypeScript interfaces. A sample type definition for the hovering capability, taken from the documentation, can be seen in Listing 2.1. From this example, we can also see that locations within documents[11] are described by a Uniform Resource Identifier (URI) representing the document along

[8] https://microsoft.github.io/language-server-protocol/implementors/servers/ *(last access: 2024-10-05)*
[9] https://microsoft.github.io/language-server-protocol/implementors/tools/ *(last access: 2024-10-05)*
[10] https://nodejs.org *(last access: 2024-10-05)*
[11] These documents must always be textual—there is no support for binary files.

with a range (which we have formalized in Section 2.1.2). The corresponding method name for this example is `textDocument/hover`.

The first request from the Language Client to the Language Server always has to be the `initialize` request, including the so-called client capabilities. These specify the capabilities that the Language Client supports. The response from the server will be a response object that includes the analogous server capabilities. The capability information being sent during this initial handshake is not just a pure list of the names of the corresponding procedure names, but also includes additional details on exactly which parts are supported (or, e. g., which encodings are used), specific to each capability. Afterwards, both parties will then restrict their usage of LSP to the subset of capabilities that both the client and server support.

The end of an LSP session is marked by the Language Client sending the Language Server a shutdown request. After the server confirms the success of the shutdown with a corresponding response, the client should send an exit notification that finally asks the server to quit their process.

For long-running operations, the specification also supports reporting progress on ongoing requests, and cancelling them. The progress reports not only allow indicating the status of the request to the user (e. g., by displaying it in the Language Client's UI), but also allows the server to return partial results to stream responses (e. g., showing the first few references to a variable while the rest are still loading).

The Language Client should also notify the Language Server whenever a document is opened or closed—that way, the server can, for example, start tracking diagnostics in the background as soon as a certain file is opened. There are also notifications related to modifications to the document that the Language Client should send, but these are irrelevant for us because modifying source code is out-of-scope for the LSP integration planned in this thesis.

Apart from describing fundamentals of the protocol like the above, the biggest part of the protocol's specification are the *capabilities*, that is, the JSON-RPC method names and types of the parameters and response objects for each feature that either the Language Client or Language Server can use [**lsp**].

2.2.2 Planned Capabilities

The capabilities I will make use of in SEE can roughly be grouped into the three categories *Navigation*, *Information*, and *Structure*. We will take a detailed look at each relevant capability below, along with how exactly they will be integrated into code cities. This level of

detail is justified in the fact that the integration of the capabilities is the main focus of this whole thesis. Note that not all Language Servers support all capabilities—for example, for a language without a hierarchic type system, the capability *type hierarchy* cannot really be sensibly implemented.

NAVIGATION This is the category containing the most capabilities that we can use for this thesis, since there are a lot of ways to navigate from one element within the source code to another. All of these take as input the position in a document, and return any number of locations, where the locations can contain a name, a file, and a range within the file.

All such available capabilities should appear in the context menu of SEE, either upon right-clicking a node, or right-clicking a code element in a code window. Selecting one of these navigation options from the menu should open a menu from which the user can select a single result. If there is only one result to begin with, this step should be skipped. Once a single result has been selected: If the request originates from a code window, the result should be opened and highlighted in that window. If the request instead originates from a code city, the node belonging to that result should be highlighted (e. g., by glowing and having a line pointed to it).

The other important part in SEE where this should be used is when building a city (i. e., when creating the project graph), as these capabilities provide us with the information we need to create (non-hierarchic) edges. Thus, we can create an edge e for each available navigation relation between two nodes, where $\ell(e)$ becomes the type of capability that was used. This is not as trivial as it sounds, since the ranges these capabilities return do not necessarily exactly match the actual ranges of the referenced nodes, so we need to implement some kind of matching algorithm (which we will do in Section 3.2.2).

The following navigation capabilities are available:

- **Call hierarchy**: Returns the incoming/outgoing calls for the symbol at the given location. Since incoming calls are already covered by the references capability, the context menu will only contain an option for showing outgoing calls.
- **Go to declaration**: Returns the declaration location for the symbol at the given location.
- **Go to definition**: Returns the definition location for the symbol at the given location.
  - For this capability, another feature common in IDEs should be implemented in SEE's code windows: Holding Ctrl, then clicking on a symbol, directly jumps to the definition of that symbol (or opens the corresponding selection menu, if there is more than one result).

- **Go to implementation**: Returns the definition location for the symbol at the given location.
- **Go to type definition**: Returns the location of the type definition for the symbol at the given location.
- **References**: Returns the location of the references to the symbol at the given location.
- **Type hierarchy**: Returns the sub/supertypes for the symbol at the given location. Since subtypes are already covered by the references capability, the context menu will only contain an option for showing supertypes.

Information Using these capabilities, the user can get information about either the project as a whole, or certain components of it. I have grouped the following capabilities into this category:

- **Diagnostics**: Returns diagnostics for a given file (e. g., warnings or errors). These will be integrated in exactly the same way as the Axivion Suite's code smells in my bachelor's thesis were [**galperin2021**], that is:
  - In code cities, we will display erosion icons above affected nodes (see Section 2.1.3).
  - In code windows, the corresponding parts of the source code should be highlighted, while hovering over the highlighted parts should reveal the diagnostic's message and details.

  Note that, instead of this being a proper request followed by a server response, this is only the case for the *pull diagnostics* capability, which has only been added in the most recent version 3.17. The far more commonly used version is the *push diagnostics* capability, where the Language Server sends out diagnostics for the currently opened files at its own discretion as notifications (see Section 2.2.1)—this makes it difficult to collect this information during city construction, a topic we will explore in Section 3.2.
- **Hover**: Returns hover information for a given location. The specification does not specify what exactly this "hover text" should be [**lsp**], but most implementations of Language Servers display the documentation of the hovered element, or the signature if it is a method, or other helpful associated details. We can simply implement this part of the specification as intended: If the user hovers above an element in a code window, or a node in a code city, we should reveal the hover information in some kind of box near the mouse cursor, hiding it again once the cursor is moved.

- **Semantic tokens**: Returns semantic tokens for the given file, which are intended for syntax highlighting. Similar to normal (e. g., lexer-based) syntax highlighting, requesting semantic tokens for a document yields a list of tokens containing their positions and a type, where the type can be one that is specified in the protocol (e. g., `enum`) or one that was previously announced as supported in the client capabilities. IDEs can then render each token type in a different color. An interesting addition to usual syntax highlighting is that each token can also be affected by any number of *token modifiers*, where each modifier may add an additional rendering effect on top of the type-based color. For example, tokens with the `static` modifier might be rendered in *italics*, while ones with the `deprecated` modifier could be rendered in ~~strikethrough~~.

  SEE currently uses Antlr[12]-based syntax highlighting, where we need to manually group each parser's token into some categories to determine colors. The added value of the semantic tokens capability here would be ease of use (i. e., no need to manually configure each Language Server) on the one hand, and support for token modifiers (i. e., "extended" syntax highlighting) on the other hand.

Structure This category actually only comprises a single capability—one that we can use to build the hierarchy of the code city's project graph (outlined in Section 2.1.2), because it gives us information about the structure of the project. The capability I am talking about here is the **document symbols** capability. Given a document, it will return all symbols present within that file, along with some additional information for each symbol, such as its type or range.

There are actually two different possible kinds of symbols that this capability may return:

1. an array of `SymbolInformation`, which is "a flat list of all symbols" ([**lsp**]) that should not be used to infer a hierarchy. Because of this limitation, if a Language Server is only able to return symbols of this data type, we cannot use it to build code cities. This is an older data type, and modern Language Servers should rather return…

2. …an array of `DocumentSymbol`. This contains a field `children`, which stores `DocumentSymbols` that are contained in this one. Using this property, we can establish a hierarchy and build a code city by recursively enumerating all `DocumentSymbols` and their children for each file, then querying for all relevant information by using the other capabilities outlined before.

[12] `https://www.antlr.org` *(last access: 2024-06-10)*

2.2.3 Unplanned Capabilities

There are a number of capabilities that I will not implement into SEE. These can be grouped into roughly three categories, based on the reasoning behind them being unused: The first concerns those capabilities that relate to editing only and provide no features related to simply viewing code—as explained in Section 1.3, editing code is not part of the goals here and requires additional large-scale preparatory changes to SEE. The second concerns the complex capabilities, that is, ones whose implementation would take a lot of time and effort and thus go beyond the scope of this master's thesis. The third concerns the niche capabilities that provide only a very marginal benefit, or do so only in rare situations. For these, I also have not deemed the effort worth it to implement them, at least not as part of this thesis. I will quickly list the contents of all these groups here.

Editing capabilities

- **Code Actions**: Allows the programmer to apply refactoring actions to the code, such as importing a referenced library.
- **Completion**: Computes autocomplete items while the user is typing, and applies them when chosen.
- **Formatting**: Applies automatic formatting to a file, or range, of code.
- **Rename**: Executes a project-wide rename of a symbol, which also renames all references to that symbol.
- **Linked Editing Range**: Returns a list of ranges that will be edited upon executing a rename of a symbol, with the purpose of highlighting those ranges during the rename.
- **Signature Help**: Returns signature information at the given cursor information. This may seem relevant for our purposes, but it is actually intended to be shown while editing (e. g., highlighting the active parameter as one types), and its information is given by the hover capability anyway in almost all cases.

Niche capabilities

- **Document Color**: Lists all color references in the code (e. g., symbolic references like `Colors.red`) along with their color value in the RGB format.
- **Color Presentation**: Allows users to modify color references in the code by using a color picker.
- **Document Link**: Returns the location of links in the document.

- **Code Lens:** Returns commands that can be shown next to source code, such as the number of implementers of an abstract method.
- **Monikers:** This is a description of what symbols a project imports and which ones it exports, and is intended to make relations between multiple projects possible. As LSP usually only deals with a single project at a time, this is more useful in the Language Server Index Format (LSIF)*, whose specification it also originates from [**lsif**].

Complex capabilities

- **Folding Range**: Returns ranges that can be collapsed in the code viewer. For example, the contents of a function could be collapsed, leaving only its signature visible. Due to the way code windows are implemented in SEE, this would increase the complexity of the implementation quite a bit.
- **Inline Value**: In debugging contexts, this supplies the contents of a variable with the purpose of displaying them inline in the Language Client, next to the variable itself. It uses the Debug Adapter Protocol (DAP)** [**dap**], which has previously already been integrated into SEE [**rohlfing2024**], but it would take a lot of refactoring work to make the two implementations compatible with each other.
- **Inlay Hint**: Returns textual hints that can be rendered within the source code. An example would be parameter names that are intended to be shown at the call site, such as in the screenshot in Fig. 2.4.
- **Notebook-related capabilities**: These are intended for interactive notebook systems such as Jupyter[13], which SEE does not currently support.

2.3 Interim Conclusion

In this chapter, we have taken a detailed look at the concepts behind the two topics central to this thesis—namely, the Language Server Protocol and code cities. We have also motivated and laid out the specific ways in which the existing LSP capabilities could be integrated into SEE, including a formalization of the project graph that will become central to Section 3.2.

[13] See `https://jupyter.org/` *(last access: 2024-10-03)*

*LSIF: A format which language servers can emit to persist LSP-based information about a software project.
**DAP: A protocol that can be viewed as the analogue to LSP for debuggers, with the goal to make it easier to integrate debuggers into development tools.

```
4 usages
Pageable pageable;

@Test
void shouldFindOwnersByLastName() {
    Page<Owner> owners = this.owners.findByLastName( lastName: "Davis", pageable);
    assertThat( actual: owners).hasSize( expected: 2);

    owners = this.owners.findByLastName( lastName: "Davis", pageable);
    assertThat( actual: owners).isEmpty();
}
```

Figure 2.4: An example of inlay hints in the IntelliJ IDE. (From https://www.jetbrains.com/help/idea/inlay-hints.html *(last access: 2024-10-04)*)

We are ready to tackle the actual implementation in the next chapter. As a quick overview before then, Table 2.1 lists the planned capabilities along with their intended use in SEE.

Table 2.1: LSP capabilities that will be integrated into SEE as part of this thesis.

Capability	Code Windows	Code Cities
Call hierarchy	Show incoming/outgoing calls and allow jumping to caller	Generate corresponding edges
Diagnostics	Highlight corresponding code ranges and display details on hover	Display code smell icons [**galperin2021**]
References	Show references and allow jumping to usage	Generate corresponding edges
Document symbols	—	Generate corresponding nodes and hierarchy
*Go to location**	Show locations and allow jumping to them	Generate corresponding edges
Hover	Show hover information when hovering above item	Show hover information when hovering above node
Semantic tokens	Extended (“semantic”) syntax highlighting	—
Type hierarchy	Show sub-/supertypes and allow jumping to them	Generate corresponding edges

*This includes the *Go to declaration* / *definition* / *implementation* / *type definition* capabilities.

3

Implementation

Relying on the foundations of SEE and the Language Server Protocol established in the previous chapter, we can now turn to the core part of this thesis: The integration of LSP into SEE, with a special focus on how to build code cities using LSP's capabilities. We will start by briefly going over some preliminary changes to both SEE and the LSP specification. Then, we will spend the majority of this chapter specifying and explaining the algorithm which "converts" LSP information into code cities, before looking into how additional capabilities can be integrated into SEE's code cities and code windows. Finally, we will conduct a brief technical evaluation, with a more thorough user study following in the next chapter.

Before continuing with this chapter, I highly recommend watching the six-minute showcase video at `https://www.youtube.com/watch?v=yAzyv2_q2ng`. It goes over all important user-facing features implemented in this chapter and does a better job of presenting them than can be done in text alone.

TODO: Add retrieval dates for YouTube links

3.1 Preliminary Changes

As promised in the preceding paragraph, we will first quickly list some preparations.

3.1.1 Specification Cleanup

While familiarizing myself with the LSP specification, I noticed and fixed a few small issues along the way. Most of these were of a formal nature (e. g., spelling, grammar, formatting, consistent usage of terms), some were fixing incorrect TypeScript syntax in the definition of LSP's data models. The rest of the changes were related to the so-called snippet grammar.

In the context of LSP, snippets are string templates that are inserted on certain completions (see Section 2.2.3), with some designed parts being filled in by the programmer on insertion.

There are also parts that can be filled in by certain values (e. g., the file name), which can themselves be transformed using regular expressions.[14] The complexity of the combinations of all these features increase the possibility of misunderstandings, which is why the snippet's grammar has been formally specified in Extended Backus–Naur Form (EBNF)*. However, as it was written down in the specification, the grammar had a few problems that I have fixed. Three notable examples are:

- Some alternatives were incorrectly grouped, contradicting the explanatory text above them. Also, the rules on how and when control characters had to be escaped were inconsistent with the surrounding text.[15]
- The grammar contained some string transformations that were unexplained in the text. Since the LSP specification is based on VSCode, I added explanations to the text based on what these transformations did in VSCode's source code.
- Finally, there were ambiguities present in the grammar that led to FIRST/FOLLOW conflicts. I have rewritten the grammar to eliminate these, and it should now be $LL(1)$-parseable [**aho2007**].

I have submitted these fixes as a pull request[16]. After addressing the resulting code review, it has been merged, and the changes will be incorporated in the upcoming 3.18 release of the specification.

3.1.2 Preparing SEE

There was not much I had to do in terms of getting SEE ready, so this section will be short:

- I have integrated the OmniSharp LSP C# library [**csharplanguageserverprotocol2023**] into SEE, which we will leverage in the subsequent sections so that we can use LSP without needing to worry about JSON-RPC encoding, data models, and so on.
- Code windows have previously been made editable by **moritz** in his bachelor's thesis, also enabling collaborative editing over the internet. I unfortunately had to remove these changes because they did not work anymore in the current version of SEE, and additionally caused a lot of complexity overhead in the code window implementation that would have made the LSP integration much harder.

[14] I am skipping over some additional features and details here because this is not that relevant a capability for us—to get the full picture, see https://microsoft.github.io/language-server-protocol/specifications/lsp/3.18/specification/#snippet_syntax *(last access: 2024-10-10)*.

[15] This has lead to confusion in some projects making use of snippets. See, for example, https://github.com/neovim/neovim/issues/30495 *(last access: 2024-10-10)*.

[16] https://github.com/microsoft/language-server-protocol/pull/1886 *(last access: 2024-10-10)*

*EBNF: A syntax in which context-free grammars can be formally expressed.

- Finally, the attribute space $\mathcal{A}$ in SEE did not allow for ranges of the form LSP needs, so I had to replace the existing attributes (which track the line and column, but not a full range) with a proper set of range attributes. In Section 2.1.2, we have introduced this as a single `Source.Range` attribute, but in reality, there are four range attributes—one per member of the decomposed form. We will ignore this reality for the rest of this thesis and act like the range is a single attribute, that is, for all project graphs with nodes V and attributes a, it holds that $\{a(v, \texttt{Source.Range}) \mid v \in V \land (v, \texttt{Source.Range}) \in \text{dom}(a)\} \subseteq \mathcal{R}$.

All of these changes have been made across two pull requests to SEE.[17]

3.2 Generating Code Cities using LSP

In this section, we will examine the centerpiece of this thesis: The algorithm with which code cities can be generated using the Language Server Protocol. While going over how the algorithm works, we will investigate interval trees* and how they relate to the algorithm, before finally taking a look at what the import process looks like in practice in SEE.

3.2.1 Algorithm

We will examine the algorithm in a generalized and programming language independent form here. In this form, the algorithm takes as input a set of source code documents, as well as a family of LSP functions belonging to a specific instantiation of a Language Server. These functions will be used to analyze the documents and extract the required information from them. The output of the algorithm, is a graph representing the given software project.

TODO: Improve coloring of inner boxes and reduce height of boxes.

Overview Before diving into the specifics of *how* the algorithm works, it may help to take a look at the diagram in Fig. 3.1, which gives us a high-level overview of *what* it does. To summarize, the steps can be broken down into three major parts:

I **Node Synthesis:** Here, we create the graph's nodes and combine them together into a hierarchy.

[17] https://github.com/uni-bremen-agst/SEE/pull/687 and https://github.com/uni-bremen-agst/SEE/pull/715 *(last access: 2024-10-11)*.

*Interval tree: A data structure meant to store intervals/ranges in such a way that overlapping or contained intervals can be found efficiently.

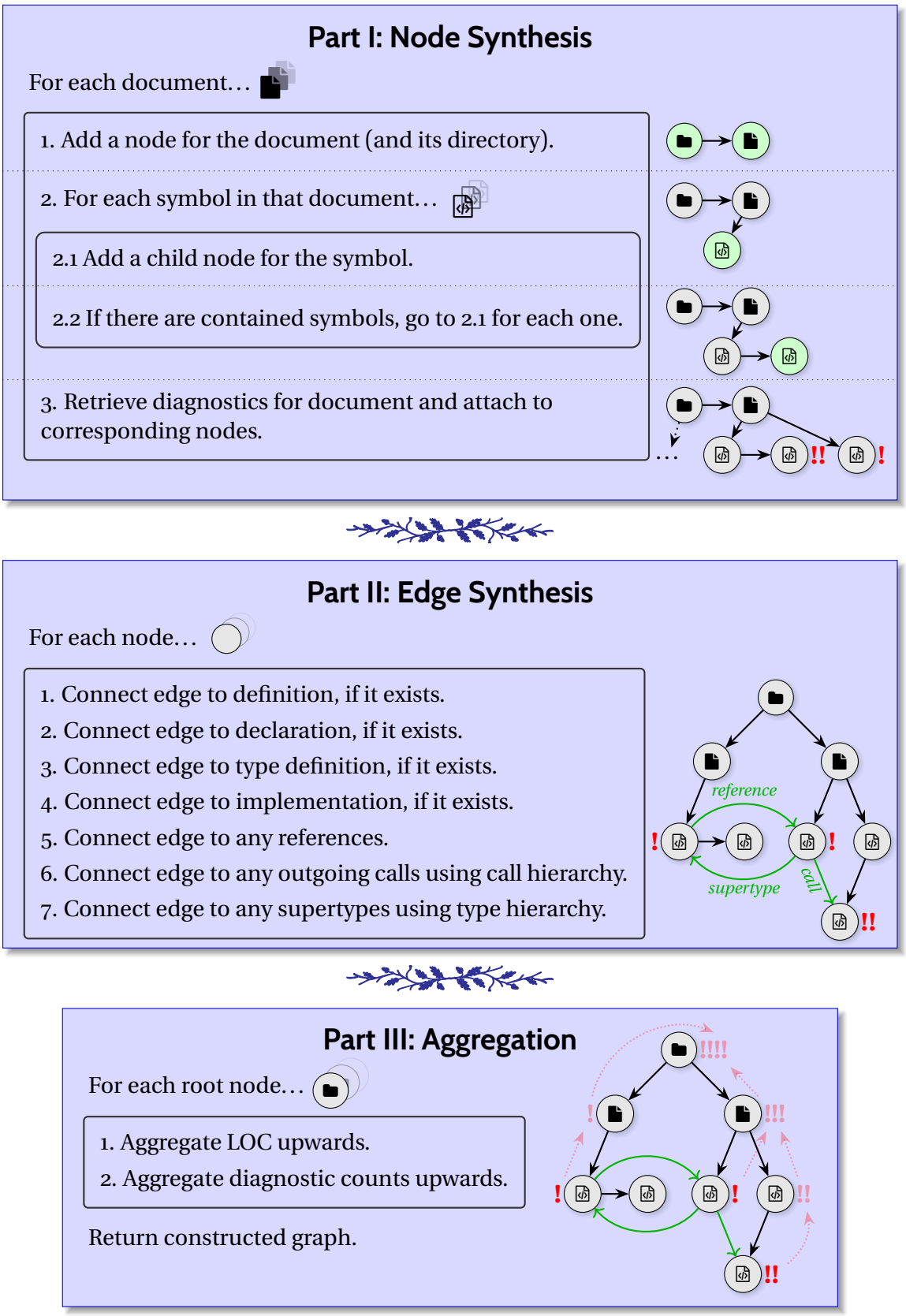


Figure 3.1: A high-level overview of the basic steps of the algorithm. "!" represent diagnostics, while "!" represent aggregated diagnostics.

1. We recreate the parts of the filesystem hierarchy that are relevant to the given documents (i. e., directories the documents are contained in and their relation to each other).

2. For each code symbol within that document, a node will be created as a child to the document. Any symbols contained within that symbol are recursively added in the same manner as a child to their parent nodes.

3. Finally, we will pull diagnostics for the document[18] and attach their counts to the nodes they correspond to.

II Edge Synthesis: Here, we connect the nodes by creating edges between them. To do this, we go over each node, using LSP functions to check for definition locations, references, and so on, and then determine which of our existing nodes best corresponds to that location. This turns out to be the most difficult and complex part of the algorithm to implement efficiently, as we will later see.

III Aggregation: Finally, we want to aggregate LOC and diagnostic count metrics upwards in the hierarchy. This way, we can, for example, see how many diagnostics are contained as a whole in a class, or even a directory.

SPECIFICATION Now that we know what it is the algorithm does, we can take a look at its detailed specification. The specification is given in Algorithm 1 and follows a few special formatting rules that I will briefly list here:

- Text in SMALL CAPS refers to functions. Those starting with LSP specifically refer to functions provisioned by the Language Server.

- Sentences in *gray italics* are comments.

- **Bold** text represents keywords.

- A normal font represents strings (i. e., text that represents itself).

- Finally, parts in a `typewriter font` serve two purposes: They represent attribute keys ($\in \mathcal{A}_K$) as well as properties of LSP-returned objects, the latter of which are prefixed by a dot.

[18] In the actual algorithm, we cannot rely on pulling diagnostics alone, since only few Language Servers support this. Instead, we will collect pushed diagnostics in the background and handle them all at once at the very end.

Page 31 contains the main algorithm. We can map Fig. 3.1 onto it as follows: Lines 1 to 12 contain part I (node synthesis), Lines 12 to 34 contain part II (edge synthesis), and finally, Lines 34 to 38 contain part III (aggregation) as well as handling of pushed diagnostics.

The following Pages 32 to 33 contain the functions referenced in the main algorithm. Here, we represent the "types" of each parameter by specifying the function domain, that is, by noting the sets each parameter must be in. Most of these sets are already defined in Chapter 2 or the algorithm itself, but two exceptions to this are the set of LSP code symbols $\mathcal{S}$ and the set of LSP diagnostics $\mathcal{D}$ (not to be confused with the set of input documents D).

Simplifications As mentioned before, Algorithm 1 is a generalized version of the actual C# algorithm that was implemented into SEE. Hence, a few simplifications[19] were made to not make this section even more technical and longer than it already is. Some noteworthy omissions are:

- **Details on adding nodes,** such as the definition of the NewNode function, or the unique IDs assigned to each node.
- **The configuration of the algorithm.** We present it as only having two input parameters, but in reality, there are many additional explicit and implicit parameters (e. g., importing only certain types of nodes). We will take a look at some of these configuration parameters that are relevant for the user in Section 3.2.3.
  - Actually, even the two input parameters that we did include are simplifications. In truth, the user selects a directory (instead of a set of documents), with the option to exclude some subdirectories, and we then automatically include every file with an extension that is relevant to the selected Language Server.
- **Progress reporting.** A progress bar noting the approximate progress (in percent) appears in both the Unity Editor's UI as well as in-game while the code city is constructed, so the user has an idea of how long the conversion is going to take.
- **Asynchronicity.** Specifically, this refers to the way the algorithm is executed in Unity. If we were to implement it as a normal synchronous function, the UI would become irresponsive to input and freeze until the whole algorithm is done. Instead, we make use of C#'s task-based `async` capabilities [**wagner2023**] in combination with the `UniTask` framework [**kawai2024**]: We yield control to the Unity event loop when progress is suspended (e. g., while we wait for an answer from the Language Server), allowing frames to be rendered while the algorithm is running in the background.

[19] Apart from the obvious simplifications that occur naturally due to the difference between declarative mathematical notation and imperative programming syntax.

Algorithm 1 How code city graphs can be generated from LSP information.

Input: Family of Lsp functions provided by the Language Server, set of documents D
Output: Graph G representing the underlying software project

```
1  V, E, a, s, t, ℓ, C ← ∅                                   ▷ Initialize empty graph components.
2  for all d ∈ D do
3      LspOpenDocument(d)                                    ▷ Document needs to be opened for all capabilities to work.
4      v_d ← AddDocumentNode(d)                              ▷ Each document becomes a node…
5      for all x ∈ LspDocumentSymbols(d) do
6          MakeChild(AddSymbolNode(x), v_d)                  ▷ …with its symbols as children.
7      if LspLanguageServerSupportsPullDiagnostics() then
8          HandleDiagnostics(LspPullDocumentDiagnostics(d))
9      else
10         ▷ We will save any incoming diagnostics in the background and handle them at the end. ◁
11         LspRegisterPushDiagnosticsCallback(d, (c) ↦ (C ← C ∪ {c}))
12     LspCloseDocument(d)

13 for all v ∈ V : (v, Source.Range) ∈ dom(a) do
14     ▷ First, connect nodes to each other based on LSP relations. ◁
15     ConnectNodeVia(LspGoToDefinition, Definition, v)
16     ConnectNodeVia(LspGoToDeclaration, Declaration, v)
17     ConnectNodeVia(LspGoToTypeDefinition, TypeDefinition, v)
18     ConnectNodeVia(LspGoToImplementation, Implementation, v)
19     ConnectNodeVia(LspReferences, Reference, v)

20     if a(v, Type) = Method then                           ▷ We need to integrate the call hierarchy into the graph.
21         I ← LspPrepareCallHierarchy(a(v, Source.Path), a(v, Source.Range))
22         ▷ GetMatchingItem returns the item in I with the same name and location as v. ◁
23         i ← GetMatchingItem(I, v)
24         R ← LspCallHierarchyOutgoingCalls(i)
25         V′ ← ⋃_{r∈R} FindNodesByLocation(r.path, r.range)
26         for all v′ ∈ V′ do
27             AddEdge(v, v′, Call)
28     else if a(v, Type) = Type then                        ▷ We need to integrate the type hierarchy into the graph.
29         I ← LspPrepareTypeHierarchy(a(v, Source.Path), a(v, Source.Range))
30         i ← GetMatchingItem(I, v)
31         R ← LspTypeHierarchySupertypes(i)
32         V′ ← ⋃_{r∈R} FindNodesByLocation(r.path, r.range)
33         for all v′ ∈ V′ do
34             AddEdge(v, v′, Extend)

35 HandleDiagnostics(C)                                      ▷ Handle diagnostics that were collected in the background.
36 AggregateMetrics({Metric.LOC})
37 AggregateMetrics({ErrorCount, WarningCount, InformationCount, HintCount})
38 return (V, E, a, s, t, ℓ)
```

```
39 function AddDocumentNode(d ∈ D)
40     v_d ← NewNode()
41     a′ ← ∅
42     a′(v_d, Type) ← File
43     a′(v_d, Source.Path) ← d
44     a′(v_d, Metric.LOC) ← |ReadLines(d)|        ▷ ReadLines returns the set of lines in the file.
45     V ← V ∪ {v_d}
46     a ← a ∪ a′
47     MakeChild(v_d, AddDirectoryNode(d.directory))
48     return v_d

49 function AddSymbolNode(x ∈ S)
50     v ← NewNode()
51     a′ ← ∅
52     a′(v, Source.Name) ← x.name
53     a′(v, Source.Path) ← d
54     a′(v, Type) ← x.kind
55     a′(v, Deprecated) ← (deprecated ∈ x.tags)
56     a′(v, Source.Range) ← x.range
57     a′(v, Metric.LOC) ← e_l^{x.range} − b_l^{x.range}
58     ▷ Several other similar attributes omitted here... ◁
59     a′(v, HoverInfo) ← LspHover(d, x.range)
60     if a′ ⊈ a then                              ▷ If an isomorphic node does not already exist...
61         V ← V ∪ {v}                             ▷ ...add it and handle its children.
62         a ← a ∪ a′
63         for all x′ ∈ x.children do
64             MakeChild(AddSymbolNode(x′), v)     ▷ Recurse.
65     return v

66 function MakeChild(v_c ∈ V, v_p ∈ V)
67     ▷ The partOf edges must induce a tree structure. Hence, if a node already is a part of another
         node, we must not add another partOf edge. ◁
68     if ∃e ∈ E : ℓ(e) = partOf ∧ s(e) = v_c then
69         output Warning: Hierarchy is cyclic. Some children will be omitted.
70     else
71         AddEdge(v_c, v_p, partOf)

72 function ConnectNodeVia(LspFun ∈ (D × R)^{D×R}, l ∈ Σ, v ∈ V)
73     ▷ Function LspFun only returns locations, so we need to find the relevant nodes first. ◁
74     for all (d, r) ∈ LspFun(a(v, Source.Path), a(v, Source.Range)) do
75         for all v′ ∈ FindNodesByLocation(d, r) do
76             AddEdge(v, v′, l)

77 function AddEdge(v_s ∈ V, v_t ∈ V, l ∈ Σ)
78     e′ ← NewEdge()
79     E ← E ∪ {e′}
80     s(e′) ← v_s
81     t(e′) ← v_t
82     ℓ(e′) ← l
```

```
1  function FindNodesByLocation(d ∈ D, r ∈ ℛ)
2      ▷ We pick the nodes with the most specific range containing the given location. ◁
3      getR(v) = a(v, Source.Range)
4      N ← {v ∈ V | a(v, Source.Path) = d ∧ b_l^r, e_l^r ∈ [b_l^getR(v), e_l^getR(v)]
                                                  ∧ b_c^r ≥ b_c^getR(v) ∧ e_c^r ≤ e_c^getR(v)}
5      N ← arg min_{v∈N} e_l^getR(v) − b_l^getR(v)
6      return arg min_{v∈N} e_c^getR(v) − b_c^getR(v)

7  function AddDirectoryNode(p ∈ 𝒜_V)
8      if ∃!v ∈ V : a(v, Source.Path) = p then          ▷ Check if node exists already.
9          return v                                     ▷ If so, just pick that one.
10     v_p ← NewNode()
11     a′ ← ∅
12     a′(v_p, Type) ← Directory
13     a′(v_p, Source.Path) ← p
14     V ← V ∪ {v_p}
15     a ← a ∪ a′
16     v_p^* ← AddDirectoryNode(GetParentDirectory(p))   ▷ Recurse to add parent directories.
17     if v_p ≠ v_p^* then
18         MakeChild(v_p, v_p^*)
19     return v_p

20 function HandleDiagnostics(d ⊂ 𝒟)
21     for all c ∈ d do
22         V_c ← FindNodesByLocation(c.path, c.range)
23         for all v ∈ V_c do      ▷ Save diagnostics count (grouped by severity) in all affected nodes.
24             n ← c.severity + Count                   ▷ Concatenate Count to attribute name.
25             if (v, n) ∈ dom(a) then
26                 a(v, n) ← a(v, n) + 1
27             else
28                 a(v, n) ← 1

29 function AggregateMetrics(M ⊂ 𝒜_K)
30     for all v ∈ V : ∄e ∈ E : t(e) = v ∧ ℓ(e) = partOf do   ▷ Aggregate from each root node.
31         for all m ∈ M do
32             AggregateMetricFromRoot(m, v)

33 function AggregateMetricFromRoot(m ∈ 𝒜_K, v_r ∈ V)
34     V_c ← {v ∈ V | ∃e ∈ E : t(e) = v_r ∧ s(e) = v ∧ ℓ(e) = partOf}   ▷ Immediate children.
35     for all v ∈ V_c do
36         AggregateMetricFromRoot(m, v)
37     ▷ After the recursion above, immediate children now definitely have attribute m. ◁
38     if (v_r, m) ∉ dom(a) then          ▷ We don't want to overwrite existing metrics.
39         a(v_r, m) ← Σ_{v∈V_c} a(v, m)
```

This also allows us to implement cancellation support, making it possible for the user to cancel the algorithm at any time[20].

The full C# implementation in SEE is available at https://github.com/uni-bremen-agst/SEE/blob/c4e3de908a022d65723bf82d3b350dade8b5f01a/Assets/SEE/DataModel/DG/IO/LSPImporter.cs *(last access: 2024-10-25)*.

Performance Considerations When we analyze this algorithm in terms of complexity, we can quickly see that the most relevant portions are in matching locations to nodes, that is, FindNodesByLocation[21]. This is only meant to give a quick motivation on why we need the optimization described in Section 3.2.2—a full complexity analysis of the algorithm is outside the scope of this thesis. Part I as a whole (ignoring diagnostics, where FindNodesByLocation is called) can be considered to fall in $\Theta(|V|)$, since there is a constant amount of work per added node. Part III (again ignoring diagnostics), as written, yields a worst-case runtime in $O(|V|^2 \cdot |E|)$, because we potentially need to search through all edges for each node to identify child nodes, and this happens once per node while aggregating metrics upwards. However, in the actual implementation, child nodes are saved alongside their parent and can be accessed in $O(1)$, so this reduces to a runtime of $\Theta(|V|)$.

Part II is where things get interesting: We are calling ConnectNodeVia a few times for each node (ignoring the handling of the type/call hierarchy, where very similar things happen), so let us look at what happens in here. ConnectNodeVia first retrieves all target locations from the given LSP function, and then calls FindNodesByLocation for each of those locations to identify the node in our graph to which the connecting edge should be drawn. This effectively means ConnectNodeVia is called once per added (non-`partOf`) edge. In this function, each node's range is compared to the given location to check whether it is contained therein. We then take the minimum over these nodes twice to determine the nodes with the most specific fit, so in total, the runtime of this function is in $\Theta(|V|)$. This function is called once per added edge, and also once per diagnostics (because diagnostics also need to be associated to nodes), so part II's runtime can be said to be in $\Theta((|E|+n_d)\cdot|V|)$, where n_d refers to the total number of diagnostics for the project.

Hence, the runtime for Algorithm 1 as a whole lies in $\Theta(|V|+|E|\cdot|V|+n_d\cdot|V|+|V|)$. In practice for LSP-built code cities, assuming the default configuration, $n_d < |V|$, but $|E| \gg |V|$ (see, for example, Table 3.1). Hence, part II with $\Theta(|E|\cdot|V|)$ easily dwarfs the rest of the algorithm's runtime due to the high cost of FindNodesByLocation, which searches through every

[20] Internally, we do this by checking every so often if a so-called *cancellation token* has been revoked by the user. If it has, we halt execution.

[21] At least, this is the case when we assume the runtime complexity of externally supplied Lsp functions is constant. This is an oversimplification, but the claim that this function is the most expensive part of the algorithm holds up to analyses of real-world test runs of the C# algorithm, as seen in Section 3.5.

node for every added edge. Section 3.5 confirms this suspicion. For this reason, it would be nice to optimize that function somehow.

3.2.2 Augmented Interval Trees

To restate the problem outlined in the performance considerations from the previous section: The locations returned by LSP functions such as "show references" or "go to location" need to be matched to the nodes in our constructed project graph. We cannot simply create a lookup table from locations to nodes, as the locations are not necessarily equal to the location of the nodes, even when they describe the same logical element. Instead, we need to match the location to the node with the "tightest fitting range", that is: from the nodes whose range completely contains the given location, we pick the one whose range is the smallest (to find the most specific fitting element). The naive solution used in Algorithm 1—simply going over all nodes each time to find the best fit—leads to an unacceptable runtime.

There are a few possible ways to solve this problem (which is in essence a variant of the stabbing problem) more efficiently, such as segment trees or range trees, but we will use augmented interval trees with k-d trees* as a base, as this configuration best fits our circumstances—there is no need to update/re-balance the tree (so the high cost associated with that is fine), the membership query should not be in $\Omega(n)$ (or worse), we need to represent two dimensions (which is possible with a 2d-tree), and so on.

A detailed explanation of augmented interval trees can be found in **cormen2022**, while k-d trees are described in a paper by **bentley1975**. In our case, we can create the tree by constructing a 2d-tree[22] out of all elements (as explained in the previous sources), where the key is the starting position of the range. Afterwards, we save in each node the maximum line number and maximum character offset, respectively, for the subtree rooted by that node, thereby turning this k-d tree into an interval tree.

An updated FindNodesByLocationEfficient is given in Algorithm 2, with these details:

- An augmented interval tree is modeled here as a quintuple $T = (v, l, c, \lambda, \rho)$. The set of all such trees is defined as $\mathcal{T}$, so $T \in \mathcal{T}$. The first element $v \in V$ is the node in the project graph represented by the node in this tree. The second element $l \in \mathbb{N}_0$ refers to the maximum line number across all nodes within this tree, whereas the third element $c \in \mathbb{N}_0$ analogously refers to the maximum character offset. The fourth

[22] Note that, in the actual implementation, the construction of the k-d tree (excluding its augmentation to an interval tree) is handled by the external `Supercluster.KDTree` library [**regina2017**].

* k-D tree: A data structure (using a binary tree as a basis) with which certain spatial data in k dimensions can be efficiently stored and retrieved.

element $\lambda \in \mathcal{T}$ refers to the left subtree rooted by this node, while the fifth element $\rho \in \mathcal{T}$ conversely refers to the right subtree. Taking a single element x from the tuple T is written as x_T.

- We construct such an augmented interval tree for each document (directly after part I in Algorithm 1) and save them all in a hash table H (modeled as a function $H : D \to \mathcal{T}$), where each document maps to its interval tree.

- Checking whether a range r_1 is contained in another range r_2 is written as $r_1 \subseteq r_2$.

- We need a way to compare two ranges against each other to check which one is "more specific". The difficulty here is that the character offsets are hard to compare to each other, since the lines can be of different lengths—handling different line lengths correctly would be both algorithmically more expensive and harder to implement, so we have given up transitivity: We define a homogeneous relation $\lesssim$ on $\mathcal{R}$ that is anti-reflexive and asymmetric (but not necessarily transitive), where $r_1 \lesssim r_2$ implies that r_1 is more specific than r_2. Similarly, $\simeq$ is a homogeneous relation on $\mathcal{R}$ that is reflexive and symmetric (but also not necessarily transitive), where $r_1 \simeq r_2$ if they are both "equally as specific".[23]

Algorithm 2 Efficiently associating LSP-returned ranges to existing elements using an already constructed augmented interval tree.

```
1  function FindNodesByLocationEfficientEfficient(d ∈ D, r ∈ ℛ)
2      return QueryTree(H(d), r, ∅)

3  function QueryTree(T ∈ 𝒯, q ∈ ℛ, R ⊆ V)
4      r ← a(v_T, Source.Range)
5      if b_l^q > l_T ∨ (b_l^q = l_T ∧ b_c^q ≥ c_T) then      ▷ Range is to the right of all nodes in this subtree.
6          return R
7      if q ⊆ r then                                          ▷ Range is contained in this node, but we want only minimal fits.
8          m ← {v ∈ R | r ≲ a(v, Source.Range)}
9          if |m| > 0 then                                    ▷ This range is smaller than other results.
10             R ← (R \ m) ∪ {v_T}
11         else if ∀v ∈ R : a(v, Source.Range) ≃ r then       ▷ Other ranges are equally minimal.
12             R ← R ∪ {v_T}
13         ▷ Otherwise, r is not minimal, so we don't add the node.  ◁
14     R ← QueryTree(λ_T, q, R)
15     if e_l^q ≥ b_l^r ∧ (e_l^q ≠ b_l^r ∨ e_c^q > b_c^r)) then   ▷ Range could be contained in right subtree.
16         R ← QueryTree(ρ_T, q, R)
17     return R
```

[23] My actual implementation of the `CompareTo` function can be found here: https://github.com/uni-bremen-agst/SEE/blob/c4e3de908a022d65723bf82d3b350dade8b5f01a/Assets/SEE/DataModel/DG/Range.cs *(last access: 2024-10-31)*.

Listing 3.1: Example C# source code with demarcated symbol ranges.

```
0 |public class Example {
1   |const char someValue = 'A'|;
2
3   |public static long pow(|int num|) {
4     |long result = num * num|;
5     return result;
6   }|
7 }|
```

As a concrete example on how this works, take a look at the example code in Listing 3.1. Here, we have the following elements:

- The `Example` class with range $(0, 0, 7, 1)$.
- The `someValue` field with range $(1, 2, 1, 28)$.
- The `pow` method with range $(3, 2, 6, 3)$.
- The `num` parameter with range $(3, 25, 3, 32)$.
- The `result` variable with range $(4, 4, 4, 27)$.

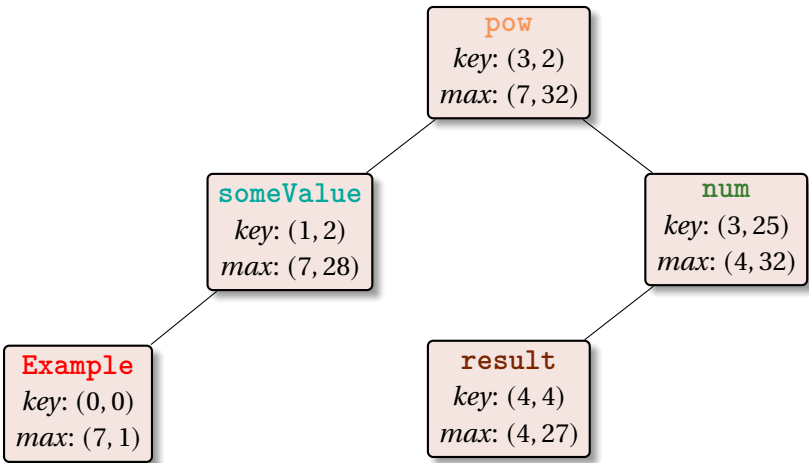


Figure 3.2: An augmented interval tree using a k-d tree, representing the elements in Listing 3.1.

Now, if we were to convert this into an augmented 2-d interval tree, it might look like Fig. 3.2. Here, the key refers to the starting position of each element and the max value refers to the maximum line and maximum character offset in the subtree rooted by each node. Let us say we want to find out what the range $(1, 13, 1, 22)$ (comprising just the name of `someValue`) belongs to. We will start by checking the root node `pow`:

1. $b_l^q < l_{\text{pow}}$ because $1 < 7$, so the range is not to the right of all nodes.

2. It is not contained by `pow`'s range.
3. We query the left subtree $\lambda_{\texttt{pow}} = \texttt{someValue}$:
   a) $b_l^q < l_{\texttt{someValue}}$ because $1 < 7$, so the range is not to the right of all nodes.
   b) It is contained by `someValue`'s range, so now $R = \{v_{\texttt{someValue}}\}$.
   c) We query the left subtree $\lambda_{\texttt{someValue}} = \texttt{Example}$.
      i. $b_l^q < l_{\texttt{Example}}$ because $1 < 7$, so the range is not to the right of all nodes.
      ii. It is contained by `Example`'s range, but $r_{\texttt{Example}} \gtrsim r_{\texttt{someValue}}$, so `someValue` is still the tightest fit and R stays as it is.
      iii. There are no subtrees.
   d) There is no right subtree.
4. $e_l^q \not\geq b_l^r$ because $1 < 3$, so there is no need to check the right subtree.
5. Our final result is $R = \{v_{\texttt{someValue}}\}$, which is intuitively the right answer.

This new and improved FindNodesByLocation implementation in Algorithm 2 has a runtime that is in both $\Omega(1)$ and $O(k + \log|V|)$, where k is the number of ranges that a node is contained in—however, its worst-case runtime is still in $O(|V|)$, because in the worst case, the queried range is contained in every element in the tree, meaning $k = |V|$. Still, this case is almost impossible in real-world generated graphs. In actuality, a given range is only going to be contained by very few elements compared to the number of total elements within a given document tree. Taking into account that in almost all situations, $k \ll |V|$, and especially $k < \log|V|$, our average-case runtime reduces to an upper bound of $O(\log|V|)$. Hence, while the theoretical worst-case runtime complexity stays the same, the average-case runtime of Algorithm 1 reduces from $\Theta(|V| + n_d \cdot |V| + |E| \cdot |V|)$ to $O(|V| + n_d \cdot \log|V| + |E| \cdot \log|V|)$. Since it is easy to make mistakes in the implementation due to the inherent complexity of this part, unit tests have also been implemented to make sure both normal and edge cases[24] are handled correctly.

3.2.3 Usage in Practice

Now, we will have a quick look at how to actually use the algorithm in SEE. As explained before in Section 2.1.3, the LSP importer—referring to the component responsible for using Algorithm 1 to generate a code city—is implemented as a graph provider. The UI of that graph provider in the Unity Editor is shown in Fig. 3.3.

[24] No pun intended.

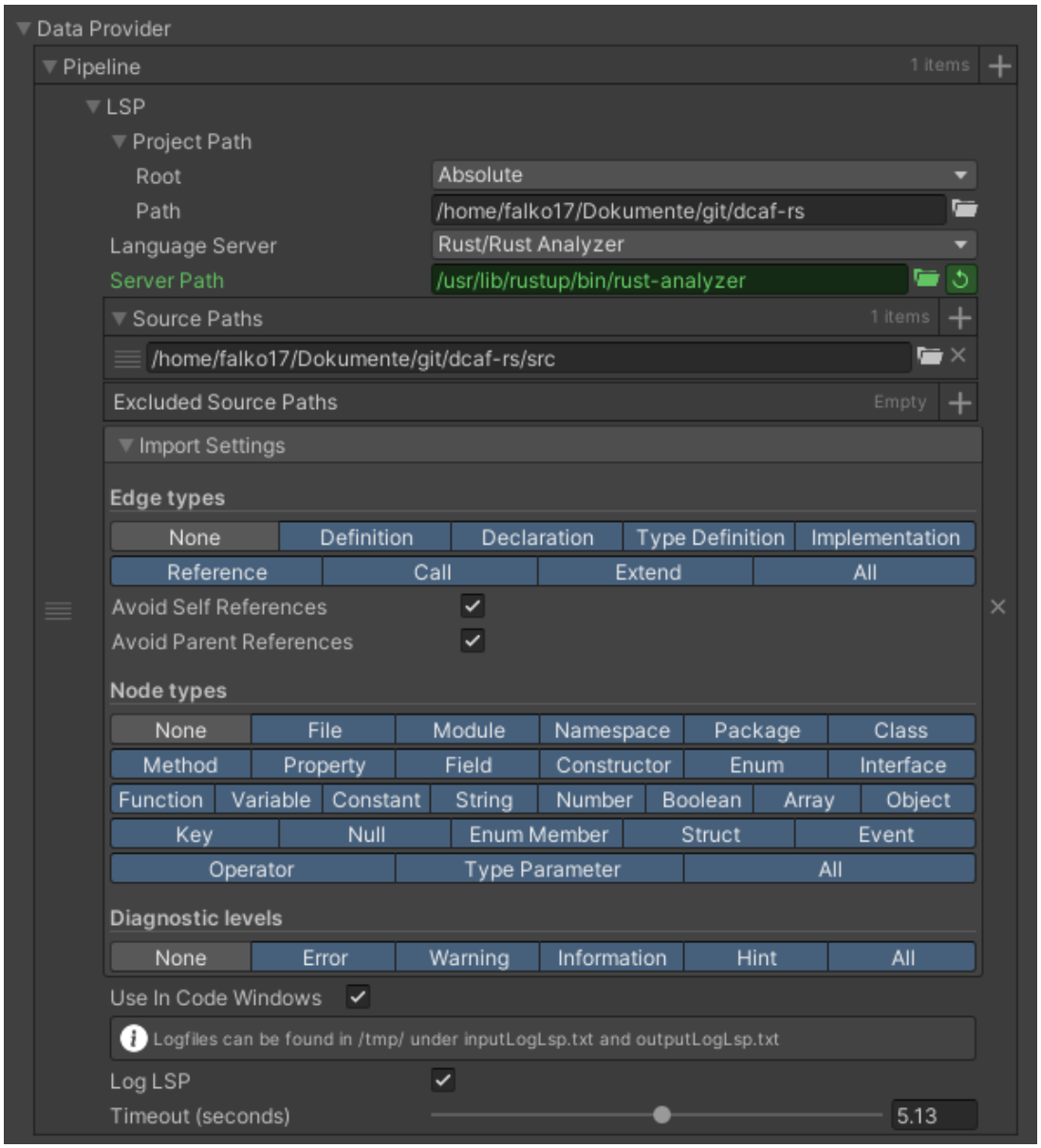


Figure 3.3: Unity Editor UI for the LSP graph provider

In that figure, we can see a configuration for the `dcaf-rs` project[25], where the Language Server is the Rust Analyzer. At the top, we can see that the path to the source project has been set, and that the *Rust Analyzer*[26] has been selected as a Language Server. Below that, there is a field for "Source Paths". This refers to the directories that shall be scanned for relevant documents, whereas a document's relevance is determined by its file extension. This is distinct from the "Project Path" as that is instead used by the Language Server to, for example, scan for project configuration files that describe dependencies. Conversely, "Excluded Source Paths" can be used for those paths within the configured source paths that should be ignored (e. g., generated files, tests).

The selectable Language Servers are only those that I have explicitly tested and confirmed to work with the algorithm—many servers are unusable, for example, due to required capabilities missing (such as the document symbols one). All in all, there are working[27] Language Servers for the programming languages C, C++, C#, Dart, Go, Haskell, Java, JavaScript, Kotlin, Lua, MATLAB, PHP, Python, Ruby, Rust, TypeScript, and Zig. There are also functioning Language Server configurations for the miscellaneous languages JSON, LaTeX, Markdown, and XML. A code city of a markup language like LaTeX, for example, would have nodes for each section, figure, and so on, with edges representing references between these elements.

Next down the list in Fig. 3.3, we have the import settings, which can be used to further refine the LSP-generated data that is used for the project graph. Specifically, we can select what kinds of nodes and edges we want to import, and which diagnostics we want to include. Additionally, for the edges, there is the option to exclude loops and edges (not taking `partOf` into account) to a node's direct parent. The purpose behind this option is that otherwise, there are going to be a lot of definition/declaration edges from elements to their immediate parents, which happens because the locations returned by these LSP functions often extend slightly outside the node's actual locations. For example, a quirk of some Language Servers is that a returned location may begin at `int value`, while the actual node for this variable starts at `int value`—in that case, the returned location would instead resolve to the outer container, such as the function it is contained in.

Finally, there is an option to enable the LSP functions for code windows that are detailed in Section 3.4, a setting to generate log files for the transferred JSON-RPC messages, and a slider to adjust the maximum time we should wait for a Language Server's response to a request. With all of these options configured, the graph can be loaded and drawn (and optionally exported to a GXL file), where a bar displays the approximate progress of the

[25] This is one of the sample projects we use in Section 3.5's technical evaluation.
[26] The menu is grouped by language when opening it, hence the `Rust/` in front in Fig. 3.3.
[27] There may well be more, I could not test some of the available Language Servers either due to setup problems, or (like in the case of Wolfram Mathematica) because I do not have a license for the language in question.

process. At this point, we will once again refer back to the explanatory video[28], which goes over the implemented functionality in a bit more detail and may be easier to follow along than this textual description. Some examples of code cities rendered using this algorithm are given in Fig. 3.11 in Section 3.5.

3.3 Integrating LSP Functionality into Code Cities

After having examined how code cities are generated through the use of LSP, we are now going to take a quick look at how some LSP capabilities are integrated into SEE. For this section, we are going to purely focus on the interaction with the city itself, saving code windows for the following Section 3.4.

3.3.1 Hover information

The first thing I implemented was the functionality that, when hovering above nodes, the LSP hover information should be displayed. We already had an implementation for a tooltip that shows arbitrary text when hovering above nodes in SEE, but I had to change it in two ways to work for this implementation:

1. It is now a singleton* (since there will only be at most one tooltip per screen), making it easier to use as a caller.

2. The tooltip now disappears when the user moves their mouse, and re-appears when the mouse is kept in place for a certain amount of time.

TODO: Crop hover info image better

Displaying the hover info is then relatively straightforward, as Algorithm 1 already saved it in the nodes of the city, so we need not even make any actual LSP requests. However, there is one more feature here that would be nice to support: Some Language Servers return the hover info as Markdown-formatted text instead of just plain text. While Markdown by itself is still readable, we can leverage the XML-like rich text tags in Unity's TextMeshPro package[29] to display the Markdown text (partially, at least) in its proper formatting. For this, I used the Markdig parser [**mutel2024**] and implemented a custom renderer to output the corresponding TextMeshPro tags. An example for a "rich" hover info tooltip like this can be seen in Fig. 3.4.

[28] https://www.youtube.com/watch?v=yAzyv2_q2ng
[29] http://digitalnativestudios.com/textmeshpro/docs/rich-text/ *(last access: 2024-12-15)*

*Singleton: a class for which only one instance exists for the duration of the process.

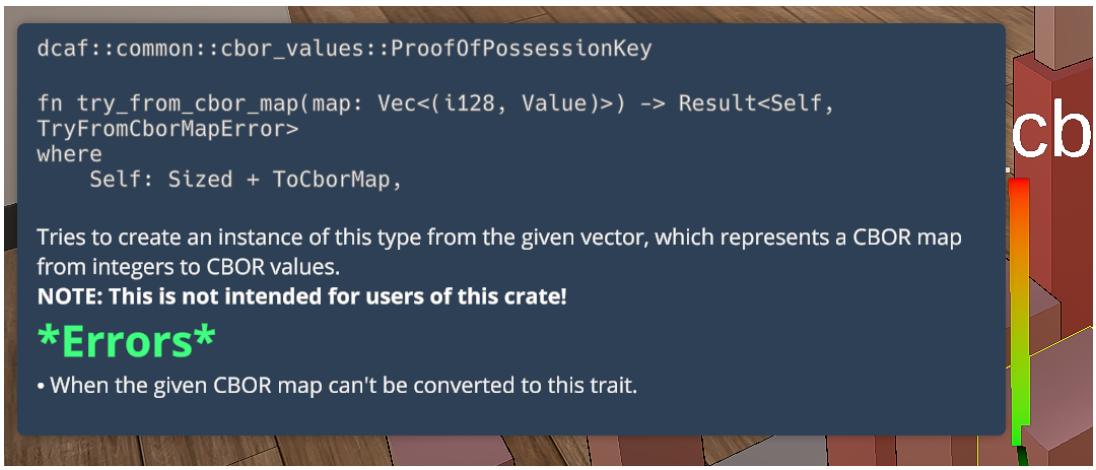


Figure 3.4: A hover info tooltip with Markdown text converted to TextMeshPro rich text tags.

3.3.2 Context Menu Navigation

While the various connections between nodes (i. e., references, definitions, etc.) can already visually be represented as edges in the scene, this is not always the most accessible option. For example, in very large projects, the huge number of edges would make it very hard to discern individual connections, so edges are often excluded from the rendering. Another problem with edges is that, as the evaluation will show (see Section 4.3.7), it is sometimes hard to discern which node a certain edge points to.

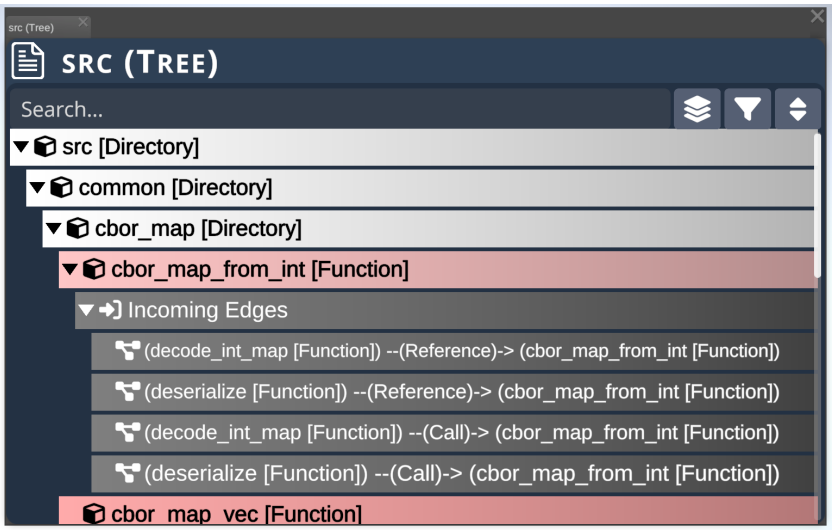


Figure 3.5: The tree view for a code city in SEE.

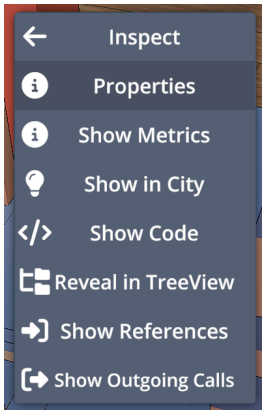


Figure 3.6: Context menu options for a node in an LSP-enabled code city.

For cases like that, users could open the tree view for the city, which is a hierarchical 2D representation of the project graph that also shows the outgoing and incoming edges for

each node (see Fig. 3.5). However, although it is useful, it is still cumbersome to look around in the city and to then have to open this tree view and navigate around in it to find out what a certain node is connected to. The context menu seems like a fitting place to remedy this: Right-clicking a node opens a menu that is shown in Fig. 3.6 with several actions, such as opening the tree view for this node or opening its code window. As part of the implementation here, the context menu will now also display LSP navigation entries such as "Show outgoing calls," given that corresponding edges exist. Clicking on this entry leads to one of two things happening:

- If there is only one target, it will be immediately highlighted with a spear and glow to make it easily identifiable in the scene.[30]
- If there is more than one target, a modified tree view will open, containing only the targets of the edge. The user can click on one of the results here to trigger the highlighting described above.

3.3.3 Diagnostics as Erosion Icons

Finally, we are displaying LSP-sourced diagnostics as erosion icons above each node. As described in Section 3.2 and specifically Pages 31 and 33 of Algorithm 1, we retrieve these diagnostics either by using the pull diagnostics capability if it is available, or we collect pushed diagnostics during the import process. We then associate diagnostics to nodes using the interval tree-based function described in Section 3.2.2. Finally, if the user enabled this in the configuration, we aggregate diagnostics upwards so that collected diagnostics can also be seen on the module they are in, for example.

We are not going to go into the details of the rendering and display of the nodes, as this just re-uses the same mechanisms as the implementation from my bachelor's thesis [**galperin2021**]. There were only minimal changes, like the integration of new icons for the different LSP diagnostic types, and a change that allows leaf erosion icons to be displayed on non-leaf nodes, since the LSP diagnostics are not always present at the lowest level in the hierarchy.

3.4 Integrating LSP Functionality into Code Windows

Now we will go over how the functionalities of Language Servers have been integrated into code windows to make them more IDE-like. Some of these functionalities will seem

[30] Once again, I recommend watching the showcase video to see this effect in action.

quite similar to the way the corresponding code city LSP features were implemented in the previous section. However, a crucial difference here is that, for code cities, the pre-computed LSP attributes from Algorithm 1 are used (i. e., no actual requests are made to the Language Server, it does not even have to be running), while for code windows, all LSP functionality does actually go through the Language Server as a proper JSON-RPC request. Figure 3.7 showcases what a code window with the newly added LSP features looks like.

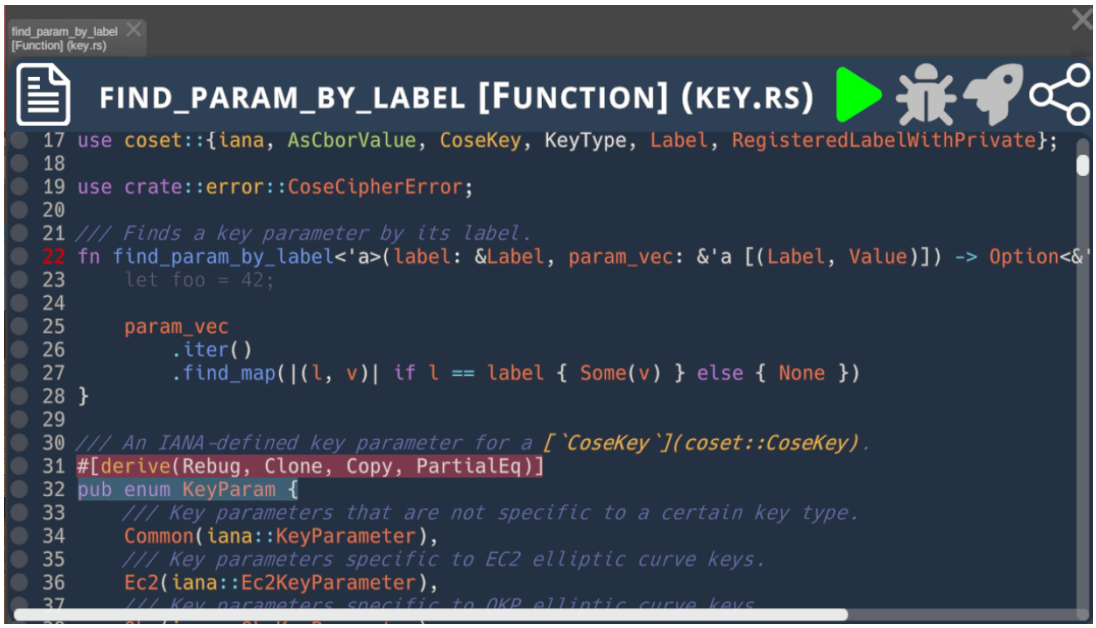


Figure 3.7: A code window with enabled LSP integration.

3.4.1 Syntax Highlighting

SEE already has support for syntax highlighting using Antlr-generated lexers, which works by converting the token stream returned by the lexer into a string of TextMeshPro rich text, as for the hover info in Section 3.3.1. The rich text then contains color tags that render each element in the desired color. This implementation was very heavily coupled to Antlr and the token types it offers, so some larger refactoring was necessary to abstract over both Antlr and LSP as a token source. Retrieving LSP tokens also required a bit more scaffolding code (e. g., semantic tokens are not always immediately available when opening the file, making it necessary to poll the Language Server until the tokens are ready), and I noticed some bugs with the way the existing syntax highlighting was implemented into the code windows. Also, the semantic tokens returned by the Language Server uses a very compact encoding, consisting of only a list of integers that decode to token types along with positions relative to the previous token [**lsp**]—since the code windows need absolute positions to be able to determine the start and end of the rich text tags instead, another decoding and

conversion[31] step was necessary here. Finally, support for token modifiers—a concept exclusive to semantic tokens—had to be added. I decided on the following mapping from modifiers[32] to rich text tags (in accordance to how IDEs usually handle this):

- Static elements and documentation are rendered in *italic text*.
- Deprecated elements are rendered in ~~strikethrough text~~.
- Variables that are modified are rendered in underlined text.

Due to all of this making it quite a bit more complicated to support semantic tokens than I first thought, this is where most of the effort and changed lines referenced in the row for the corresponding pull request in Table 3.2 went, so subsequent subsections shall seem shorter and substantially simpler.

3.4.2 Hover information

Implementing the hover functionality was fairly straightforward, as it has already been implemented for code cities. The only thing I had to work around was that, within Unity, I could only get the byte offset of the clicked character within the text as a whole, while the LSP hover information can only be queried for a two-dimensional range. To fix this without sacrificing too much performance, we have to maintain a mapping from line numbers to the "global" offset of the newline character, which we can then use to convert from a global offset to a range.

3.4.3 Diagnostic Highlighting

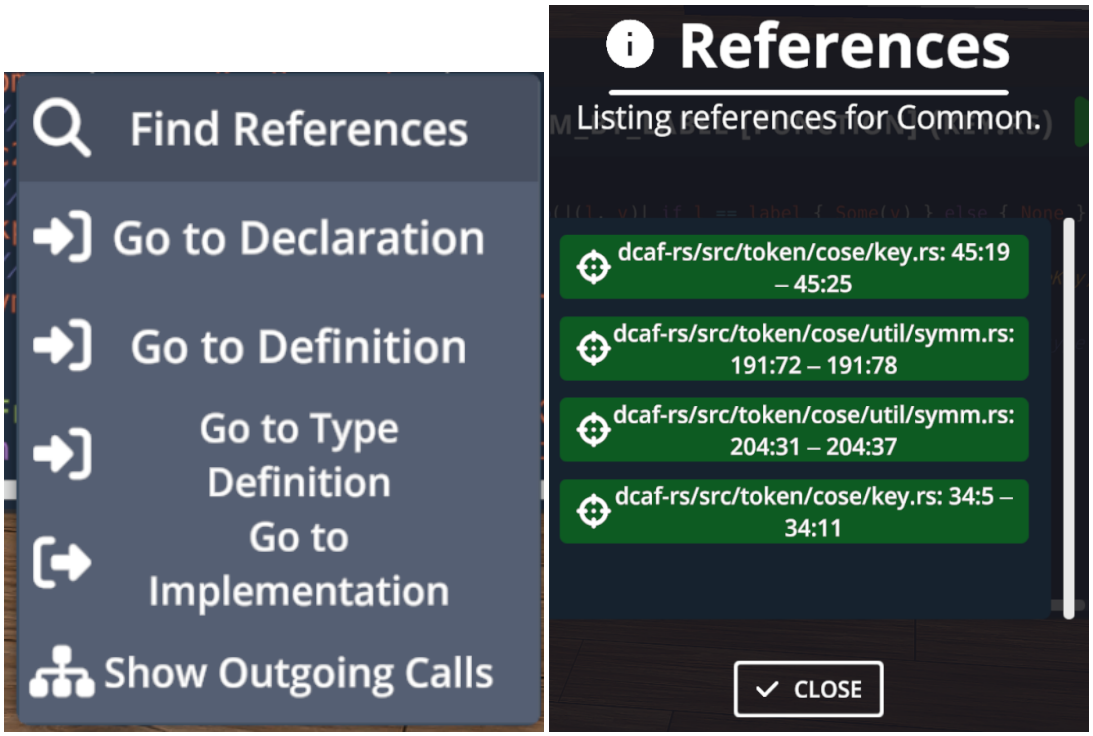
Highlighting diagnostics in the code windows was already implemented with the Axivion Dashboard as a source, so similar to the semantic tokens, I had to implement an abstraction over diagnostics' sources, although this was much easier than abstracting over token sources. Diagnostics show up in two ways in the code windows: By highlighting the corresponding code range in a color corresponding to the type of diagnostic, and by providing the diagnostic's content when hovering over that code range (both of these were already implemented for my bachelor's thesis [**galperin2021**]). I modified the hover text to indicate the source of the diagnostic and added special handling for the "element unused" diagnostic to render the text in grey, as is done in many IDEs. The rest just involved mapping the

[31] Yet *another* issue was the fact that semantic tokens, in contrast to Antlr tokens, do not cover the whole document (e. g., there are no token types for whitespace), so we have to read through the document while iterating over the semantic tokens to construct the final set of tokens that can be used by the code windows.

[32] There are more modifiers than this, such as one for asynchronous code, but at this point I ran out of available (and reasonable) rich text tags.

LSP diagnostics' offset to the global offset in order to insert the rich text tags that highlight the code range at the correct position. We can see three example diagnostics in Fig. 3.7 in lines 23, 31, and 32, respectively.

3.4.4 Navigation



(a) The context-menu that is shown when right-clicking an element.

(b) Menu from which the user can select an LSP reference to navigate to.

Figure 3.8: Navigation features in an LSP-enabled code window.

Similar to Section 3.3.2, I added the ability to right-click code elements and open a context menu with various navigation options, such as "go to definition," or "show references" (see Fig. 3.8a). Taking "show references" as an example, if there are multiple targets, a dialog is shown in which the references are shown as a list of files along with their ranges, as in Fig. 3.8b. The user can then select one of these ranges. This step is skipped if there is only a single target. After selecting a reference, if it belongs to the current file, the range is simply scrolled into view and the corresponding line is highlighted, otherwise a new code window (or an existing one, if the file is already open elsewhere) is opened first. While implementing this, I also noticed a bug in the OmniSharp LSP library that causes the call hierarchy request to fail, so I implemented a workaround and submitted an issue.[33]

[33] https://github.com/OmniSharp/csharp-language-server-protocol/issues/1303 *(last access: 2024-12-27)*

As an additional feature copied from popular IDEs, it is possible to hold down Ctrl and then click on an element in the code window, which has the same effect as right-clicking that word first and then selecting "Go to definition". This is useful because it allows the user to quickly jump to the definition of, for example, a function to find out its content, which is a very common operation.

3.5 Technical Evaluation

Before ending this chapter, we want to briefly evaluate the implementation on a technical level. While there are some unit tests for parts of the algorithm, taking real-world projects and trying to generate code cities out of them using our implementation is important to verify that it is working correctly too. Apart from a quick qualitative glance at each generated city, we are also interested in an analysis of performance here—we made quite a few theoretical statements and analyses in Section 3.2, so we can also use this section to make sure the interval tree-based Algorithm 2 actually does work more efficiently than the original "brute-force" approach of Algorithm 1.

I ran the city generation algorithm on the following projects:

- **JabRef** [**jabref**]: A bibliography manager written in Java.
- **SpotBugs** [**spotbugs**]: A bug-detector using static analysis for Java code.
- `dcaf-rs` [**dcaf**]: A Rust project about authentication in the Internet of Things.
- `aaoffline` [**aaoffline**]: A Rust downloader for a certain kind of online mystery game.
- **LaTeX source of this master's thesis**[34]: Self-explanatory.
- **LaTeX source of my bachelor's thesis**: Self-explanatory.

The algorithm was repeated three times for the first two projects and five times for the others[35], to make sure that any single result was not an outlier. For the generation, we enabled every edge type, node type, and diagnostic, except for JabRef and SpotBugs, where we only enabled some node types (namely File, Module, Namespace, Package, Class, and Interface) and only one edge type (Extend for JabRef and Reference for SpotBugs), otherwise the generation would have taken much too long. Table 3.1 lists all projects along with their node count, edge count, LOC, and average generation time.

[34] I could not use the whole content of the thesis—for example, it would have been difficult to include the results of the technical evaluation while I was still in the process of collecting them.
[35] The difference stems from the fact that the generation for JabRef and SpotBugs took much longer.

Table 3.1: Relevant metadata for all evaluated projects, along with the average total generation time.

Name	Language Server	kLOC	# Nodes	# Edges	Average Time [h:m]	
					Optimized	Brute-force
JabRef	Eclipse JDT	179	5575	919	4:17	4:29
SpotBugs	Eclipse JDT	216	3813	16 262	9:22	43:11
`dcaf-rs`	Rust Analyzer	16.5	1192	9940	4:27	5:59
`aaoffline`	Rust Analyzer	3.3	515	1250	0:46	0:56
Master's thesis	`texlab`	2.5	301	24 048	0:10	0:36
Bachelor's thesis	`texlab`	2	376	46 348	0:13	2:31

The first thing that is worth mentioning here is that generations by different Language Servers are not really comparable to one another. For example, we can see that the generation run for this thesis took on average ten seconds—faster than any other project—even though there were more edges than in almost every other project. This difference cannot be explained by the idea that edges may not play a big role in generation time after all, since a closer look at the data for JabRef and SpotBugs reveals a much longer generation time for SpotBugs, even though SpotBugs has less nodes (but a lot more edges) than JabRef. Hence, the difference there has to come from the different Language Servers. This is also why we have chosen to analyze two projects per Language Server.

We can already make two observations from Table 3.1: The more edges there are, the more time the generation takes (again, only within the same language), and the brute-force version always takes longer than the optimized version. To examine these claims in more detail, I have created some bar graphs comparing the average generation time in Fig. 3.9. There, the optimized version (suffix *O*) is compared against the brute-force version (suffix *B*), and the runtime of the algorithm is broken down by its components:

- **Nodes** refers to part I of the algorithm (Lines 1 to 12),
- **Tree Creation** refers to the construction of the augmented interval tree,
- **Edges** refers to part II of the algorithm (Lines 12 to 34),
- **Diagnostics** refers to collecting and handling the diagnostics (Line 35),
- **Aggregation** refers to the upward aggregation of metrics (Lines 36 to 38), and finally,
- **Miscellaneous** consists of the timespans outside of the measured ranges (e. g., collecting documents).

The generation of edges is by far the most expensive part[36] of the algorithm, partly confirming our theoretical analysis from Section 3.2.1. The one exception to this seems to be

[36] As the distribution of other components is hard to tell due to the edge component taking up so much space, there is another version of the diagram without this component in Appendix B.3.

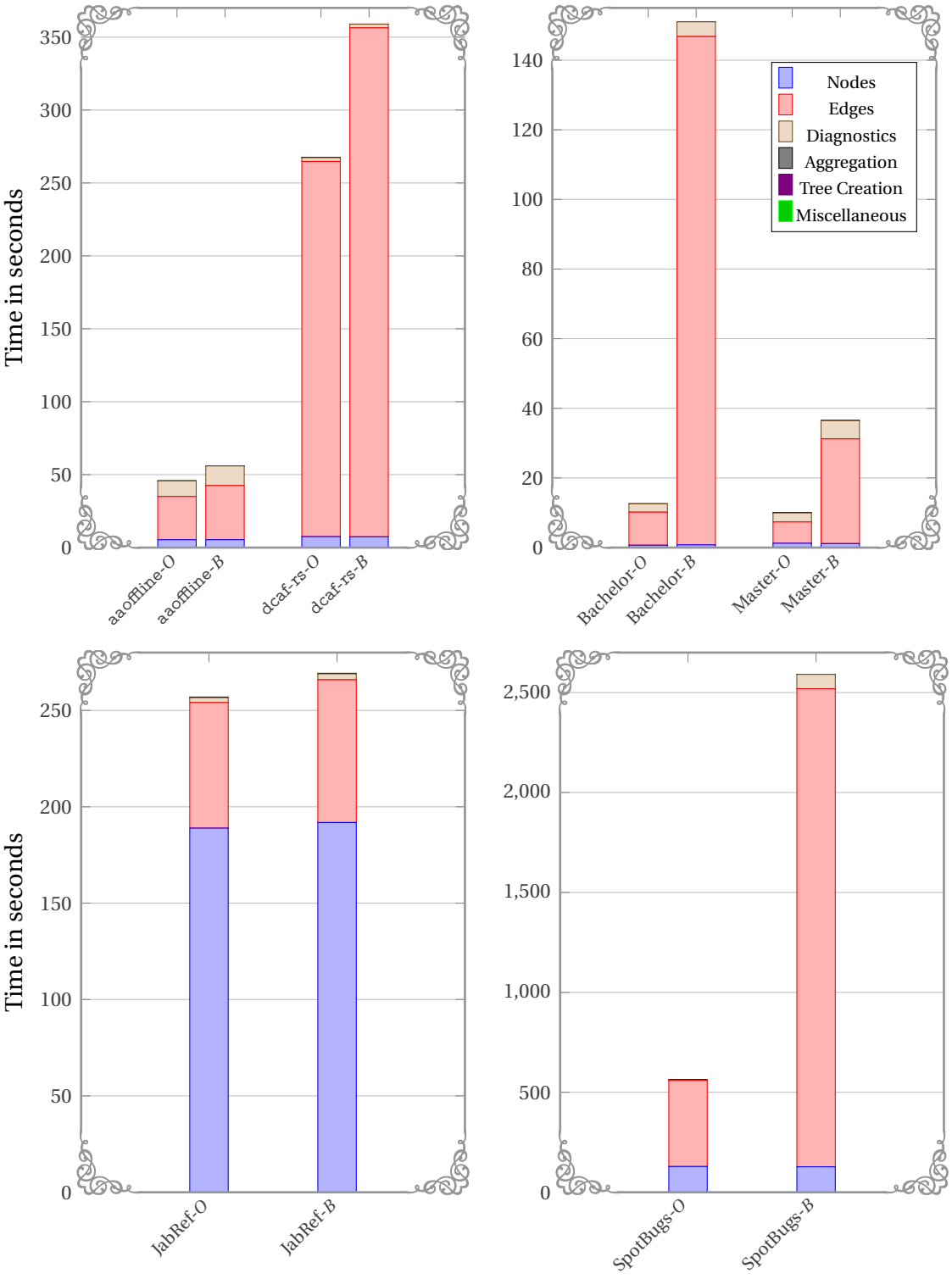


Figure 3.9: Generation time for each project, broken down by parts of the algorithm. The suffix *O* denotes the optimized (interval tree) version of the algorithm, while *B* refers to the brute-force version.

JabRef, where the most expensive component consists of nodes. This can be explained by the respective counts of the graph elements: Because we only selected "Extend" edges for JabRef, it has the fewest edges out of all the projects (919) while having the most nodes out of all the projects (5575). Another thing to note is that the only components big enough to even make out on the diagram are the node synthesis, the edge synthesis, and the collection of diagnostics (but not the aggregation), with the latter probably taking some time because it also makes use of the FindNodesByLocation function.

The optimized version of the algorithm is also always quicker than the brute-force version, and we can hypothesize based on these bar graphs that the relative improvement of the optimized version (i. e., the percentage by which time decreases when switching to the optimized version) over the number of edges is strictly monotonic, that is, the more edges there are, the more the optimized algorithm will shorten the total runtime in relation to the brute-force version. We can check this hypothesis by plotting the relative time improvement as defined here against the number of edges in each project.

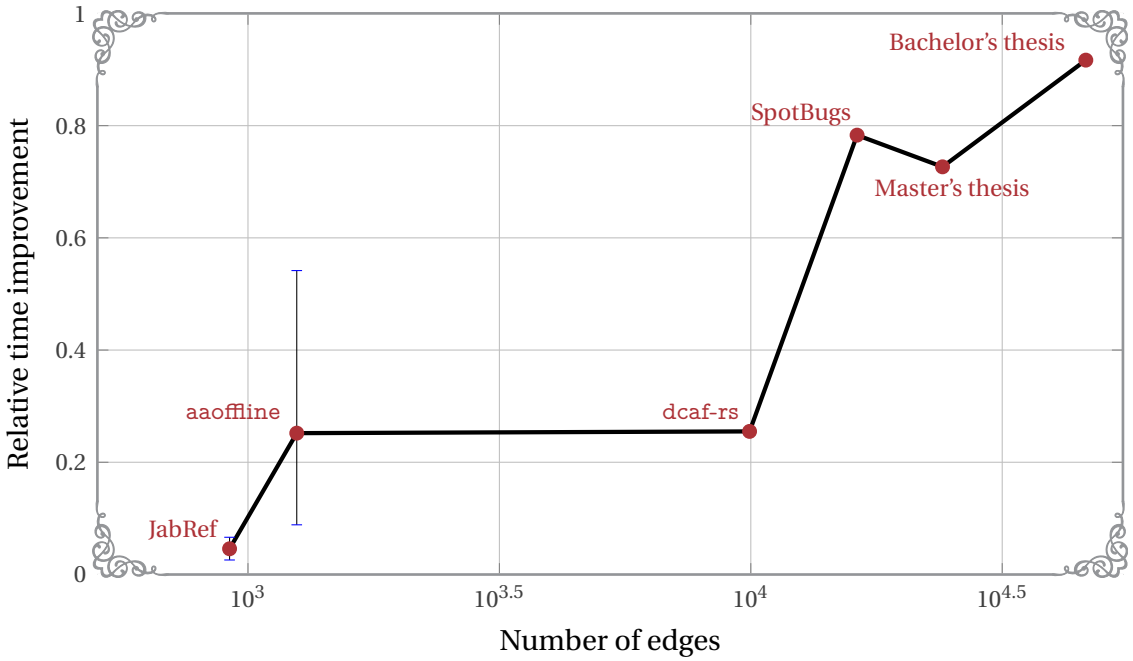
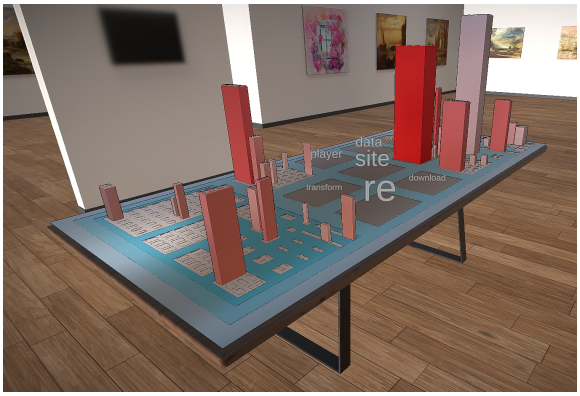


Figure 3.10: The relative improvement of the optimized algorithm by number of edges (i. e., the percentage by which the time is reduced when using Algorithm 2 instead of the base Algorithm 1). Error bars are shown when their extent exceeds the diameter of the point. *Note the logarithmic x-axis.*

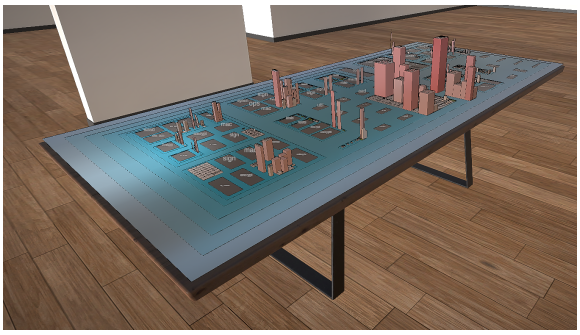
Figure 3.10 shows exactly that plot, but it disproves this hypothesis: While almost every datapoint does seem to follow this rule, the relative time improvement for the master's thesis source code is actually lower than for SpotBugs, even though SpotBugs has fewer edges. I suspect that the hypothesis is still directionally correct (the more edges there are, the more

the optimization helps), but that it does not hold when making comparisons across multiple Language Servers, since the performance differences between their implementation makes too big a dent. The final thing to note from this plot is that, across all our samples, the brute-force version is never better than the optimized version of the algorithm, which we can tell by the fact that the error bars (denoting the minimum and maximum relative time improvement across all samples) never extend into the negatives.



(a) Code city for the `aaoffline` project.

(b) Code city for my bachelor's thesis.



(c) Code city for the `dcaf-rs` project.

(d) Code city for this master's thesis.

(e) Code city for the JabRef project.

(f) Code city for the SpotBugs project.

Figure 3.11: Sample code cities generated using the LSP algorithm introduced in this chapter.

Finally, we can take a look at the generated code cities shown in Fig. 3.11. No edges are shown in those diagrams—they would clutter up the images too much due to their high

number. While I am not as familiar with the source code of JabRef and SpotBugs, the ones for the first four projects seem correct to me on first glance: Figures 3.11b and 3.11d show the sections of both theses, and the biggest components correspond to the largest parts in the theses themselves (e. g., the evaluation section from my bachelor thesis was by far the longest part of it, which is reflected in that building being the largest in the code city). We can even see some diagnostics for the bachelor's thesis, probably about unused labels. The two Rust projects in Figs. 3.11a and 3.11c also match up with my knowledge of them, such as the biggest function being what I thought it would be.

There is one weird aspect here, though, that is not present in other languages, namely that there seem to be duplicated empty modules. For example, in Fig. 3.11a, we can see an empty grey `re` component, even though the regular expressions contained in the corresponding module are shown elsewhere in the city. My guess is that there is a mismatch between Rust modules and files that is not handled correctly either by the Rust Analyzer or by my algorithm. Still, this should not be a big problem, as the structure of the code city still reflects the original project apart from those empty modules.

3.6 Interim Conclusion

In this chapter, we covered how we implemented the various aspects of the LSP integration, placing a special focus on the generation algorithm for code cities. We closed out the chapter by evaluating the implementation and benchmarking its performance.

The technical evaluation revealed that the implementation does work well for the tested projects and that the implemented optimizations make sense, but the total runtime still is not really at the level where this system can be used productively for larger projects. For example, generating a city for SpotBugs, a project with around 200 000 LOC, took almost ten minutes—and this was with almost all node and edge types being disabled! Actually generating a code city for projects on the order of 1 000 000 LOC from LSP information with all available information being present (i. e., all nodes, all edges, all diagnostics) still seems intractable, but I am unsure whether this is due to additional inefficiencies in the algorithm or if the Language Servers themselves are at fault.[37] Thus, we have to answer our first research question RQ1 with caveats: While it is feasible to generate code cities using the Language Server Protocol, more research and work needs to be done for this to work well with very large software projects.

Table 3.2 contains a summary of all pull requests created for this thesis, along with the number of added and deleted lines. There is one more pull request here that I have not

[37] Of course, this is a slightly unfair assessment—Language Servers were never developed with the intent to be used in high-throughput scenarios like this, only for moderate use in IDEs.

mentioned yet, since it only contains disparate fixes and small additions that were done for the user study that we will talk about in the next chapter. The fixes for this pull request were a mix of things I noticed myself while designing the user study and bugs noticed while conducting the pilot study (see Section 4.3.1). There are two relevant features here that we should quickly go over:

- Edges of a certain type can be hidden, causing them to only be shown when hovering over nodes they are connected to. This mitigates the problem of too many edges obscuring and cluttering the code city.
- *Transitive edge animations:* When an edge is revealed by hovering above it, the "next" connected edge is repeatedly revealed as well. For example, when this is enabled for "extend" edges, hovering over a class node will gradually reveal the path to the base class*. I recommend watching the explanatory video for the user study to get an intuitive understanding about this[38].

Table 3.2: All submitted pull requests done as part of this thesis.

Summary	Pull Request URI	Δ LOC +	Δ LOC −
Cleanup of LSP specification	microsoft/language-server-protocol#1886	475	453
Preparing SEE for LSP integration	uni-bremen-agst/SEE#687	282	2773
Introducing `Source.Range`	uni-bremen-agst/SEE#715	392	313
Generating code cities using LSP	uni-bremen-agst/SEE#727	3475	180
LSP functions in code cities	uni-bremen-agst/SEE#747	1139	432
LSP functions in code windows	uni-bremen-agst/SEE#751	4080	2024
Preparing SEE for user study	uni-bremen-agst/SEE#772	541	150

Only C# line changes have been counted in SEE pull requests.
GitHub pull requests are specified in the format `namespace/repository#PR_number`.

[38] See `https://www.youtube.com/watch?v=WE21naXp_YM` *(last access: 2024-12-28)*.

*Base class: The base class of a class is its superordinate (i. e., above in the inheritance tree) class that itself has no further parent class within the project.

4

User Study

Despite having already done a technical evaluation in Section 3.5, it is usually also a good idea to compare new approaches with existing state-of-the-art tools in a user study. In our case, we want to compare SEE's LSP-generated code cities with the capabilities a normal LSP-enabled IDE offers. For the latter, the Microsoft-developed IDE VSCode is a good fit, given that the Language Server Protocol itself originates here (see Section 2.2) and that it still has a deep integration with LSP.

First, we will outline the general aim of this study by going over some existing related research, explaining important aspects of VSCode, and then enumerating our hypotheses. Next, we will explain the details of the design of the study itself, before analyzing its results with the 20 participants in detail. Finally, we describe some relevant threats to validity.

As a quick aside before we begin, we will frequently use violin plots in this chapter to visualize datasets, especially to visually compare two datasets against one another. A violin plot visualizes the distribution of a collection of data points along with an estimated probability density [**hintze1998**]. The black/white data points are randomly "jittered" along the x-axis to make them more differentiable from one another. A bigger, green point marks the average of the dataset.

To estimate the probability density for the plots, we need to use a non-parametric kernel density estimation (since we do not know which shape the underlying distribution has)—for the kernel itself, we simply use a Gaussian function, but a much more important question is the choice of the bandwidth parameter [**heidenreich2013**]. Here, we use an algorithm by **sheather1991** that has been improved by **botev2010a** to be made more performant and handle multimodal distributions better: The *Improved Sheather–Jones* method, which is a robust choice when one cannot assume normality [**akinshin2020**]. As the algorithm for this method, we use `KDEpy` [**odland2018**], whose implementation of the method is based on **kroese2011**. A drawback here is that this algorithm does not necessarily converge. In those cases, we fall back to using Silverman's rule of thumb [**silverman1986**], the implementation of which is integrated into the library we use to draw these plots [**callil-soares2024**].

4.1 Plan

Our main aim here is to answer our second research question that we defined in Section 1.3:

> Are code cities a suitable means to present LSP information to developers as compared to IDEs + tables (on the dimensions of speed, accuracy, and usability)?

To empirically evaluate this research question, we will devise a series of short software engineering related tasks on the real-world software projects *JabRef* [**jabref**] and *SpotBugs* [**spotbugs**]. Participants then get randomly assigned to either use SEE (along with our implementation from Chapter 3) or VSCode (with an active Language Server). However, evaluating the supported capabilities (see Table 2.1) in this way turns out to be quite difficult—for example, how would one evaluate the *Hover* capability, let alone features like semantic tokens which are almost identically implemented across SEE and VSCode? For this reason, we will abstain from incorporating the code window-related capabilities from Section 3.4. Limiting ourselves, then, to the code city-related changes from Section 3.3, we have:

1. *Diagnostics* being displayed as erosion icons above corresponding nodes.

   ☞ This feature was essentially already evaluated in my bachelor's thesis, albeit with the Axivion Dashboard as a data source instead of LSP [**galperin2021**; **galperin2022**].

2. *Hover* details being displayed when the user hovers the mouse above a node.

   ☞ Since this is used here almost identically as in code windows, it does not make much sense to compare it against VSCode.

3. *Go to location*, *references*, and *call/type hierarchy* being used for rendered edges and context menu actions.

   ☞ The context menu actions are not interesting to evaluate for the same reasons as above, though this does not apply to the generated edges.

It appears that the only capabilities that are worth to be evaluated in a user study of this form are actually the ones used in the generation of the code city in Section 3.2. Besides, the bulk of the implementation pertains to the generation of code cities, so it makes sense to focus on them here. As a result, the user study is now actually of a form (directly comparing code cities against IDEs) that has been researched in previous literature before, so let us take a look at that research first before designing our own study.

4.1.1 Existing Research

Across various bachelor's and master's theses, a number of user studies have been performed about SEE's usability in various aspects [**davidwagner2020**; **felixgaebler2021**; **hannesmasuch2020**; **kevindoehl2020**; **maximilianwick2022**; **michelkrause2024**; **robertbohnsack2020**; **rubensmidt2021**] as well as about its effectiveness compared to traditional tools [**galperin2021**; **lennartkipka2020**; **moritz**; **nicoweiser2021**; **rohlfing2024**; **schramm2022**; **sulanabubakarov2021**; **yannisrohloff2021**]. Especially relevant among the latter kind of studies are the three that compare SEE with traditional IDEs, as that is very close to our own planned evaluation. Two of these evaluate debugging capabilities that have been implemented into SEE: **lennartkipka2020** compares those with Eclipse[39]'s debugger, while **rohlfing2024** uses the debugger of VSCode as a baseline. The final one of these studies has **schramm2022** compare pure IDE usage (in this case, Microsoft's Visual Studio) with a combination of Visual Studio and SEE in which Visual Studio has a plugin setup that integrates it with SEE. The result here was a significant improvement for usability and a partial improvement for efficiency in favor of the combination of Visual Studio and SEE. Almost all of these SEE-related studies also measure its usability in the form of the System Usability Scale (SUS)*, which we will do in our own study, as well. I have collected the existing scores of those studies in a violin plot in Fig. 4.1. As a reminder, the big green point marks the average of the dataset.

Outside of SEE, there are a number of other code city implementations, such as *CodeCity* [**wettel2007**] or *Software World* [**knight2000**] (see also the overview by **jeffery2019**). In their evaluations, these papers often compare different platforms (such as Desktop to VR/AR) [**merino2017**; **fittkau2015**; **merino2018**], but as with the SEE-related theses, we are most interested in those that have controlled experiments comparing a code city tool with a traditional IDE.

One such study was done by **wettel2011** and compares *CodeCity* against the Eclipse IDE, with the caveat that participants using Eclipse can also access an Excel spreadsheet of software metrics, as the code city implementation would otherwise have an unfair advantage. He based his study on an extensive survey of existing empirical work on software visualization, constructing a "wishlist" of desiderata for such studies [**wettel2011**]. We will refer back to this wishlist, and to his experiment design in general, in Section 4.2 for our own study. The study was later replicated by **romano2019** with a subset of tasks.

Similarly, **mortara2024** supplied tabular data for IDE users in a comparison against *VariCity*, although here, participants were allowed to choose whatever IDE they are most comfortable with. **mehra2020** gave participants who were using Eclipse an additional 2D graph

[39] `https://www.eclipse.org/` *(last access: 2024-11-17)*

*SUS: A simple questionnaire by **brooke1996** with ten Likert-scale questions that are supposed to measure the usability of a system.

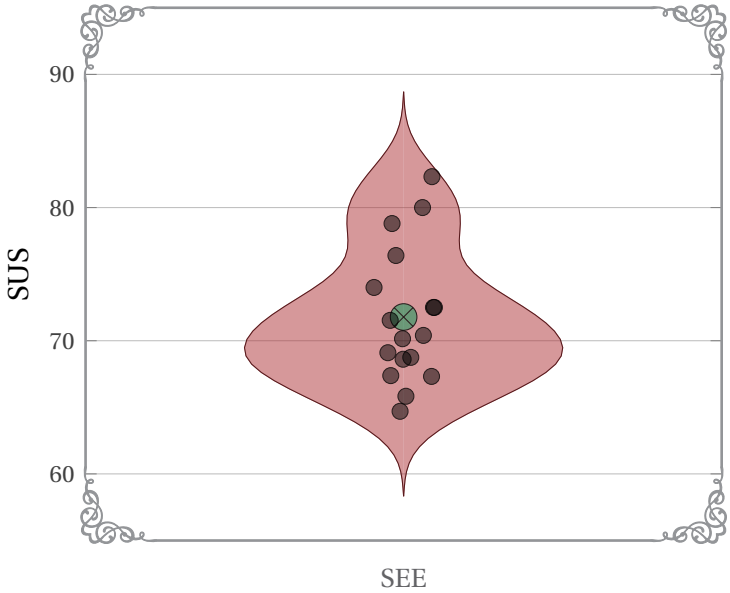


Figure 4.1: SUS results for SEE across sixteen studies [**davidwagner2020**; **felixgaebler2021**; **galperin2021**; **hannesmasuch2020**; **kevindoehl2020**; **lennartkipka2020**; **maximilianwick2022**; **michelkrause2024**; **moritz**; **nicoweiser2021**; **robertbohnsack2020**; **rohlfing2024**; **rubensmidt2021**; **schramm2022**; **sulanabubakarov2021**; **yannisrohloff2021**].

tool when evaluating the augmented reality XRASE code city visualization. On the other hand, the category of comparative user studies between code cities and IDEs without any other helper tools includes ones by **khaloo2017**; **galperin2022**; **lennartkipka2020**. The results of their experiments, and all others cited here, are listed in the color-coded Table 4.1.

4.1.2 VSCode

Here, we will very briefly go over VSCode as the tool that we will compare SEE against. A screenshot of VSCode is provided in Fig. 4.2. On the left side, we can see the filesystem hierarchy of the open project, in the middle is the code itself, and on the right is a minimap as a quick overview of the current file's code. VSCode also has an extension system in place with which Language Servers and other enhancements to the editor can be easily installed—for example, we can see a notification in the bottom right prompting the user to install the C# extension.

It is also possible to quickly find files with the Ctrl + P shortcut, which pops up a menu with a live search through all filenames in the project. The analogue in SEE is the tree view which was showcased in Section 3.3. Also shown in that chapter was a context menu with various options to make use of the LSP "go to location" capabilities. VSCode has a very

Table 4.1: Results of various studies comparing code cities (CC) against IDEs. x/y indicates an advantage in x out of y tasks or questions.

Study	n	Correctness	Time	Usability	IDE	Code city
[wettel2011]	45	$p = 0.001$	$p = 0.043$	*N/A*	Eclipse + metrics	CodeCity
[khaloo2017]	28	*N/A*	3/5 IDE	6/20 CC; 1/20 IDE	Visual Studio (VS)	Code Park
[romano2019]	54	$p = 0.005$	$p < 0.001\%$	No diff.	Eclipse + metrics	Code2City
[lennartkipka2020]	10	No diff.	No diff.	No diff.	Eclipse	SEE
[mehra2020]	20	$p = 0.005$	3/5 CC; 1/5 IDE	Preliminary only	Eclipse + 2D graph	XRASE
[galperin2022]	20	2/6 IDE	4/6 CC	$p = 0.028$	Axivion Dashboard	SEE
[schramm2022]	10	No diff.	1/3 CC	$p = 0.002$	VS + Axivion	VS + SEE
[mortara2024]	49	6/11 CC	4/11 CC; 1/11 IDE	4/11 CC	Various + metrics	VariCity

Legend: ■ CC advantage, ■ Slight CC advantage, ■ IDE advantage, ■ Slight IDE advantage, ■ No significant difference, □ Not measured

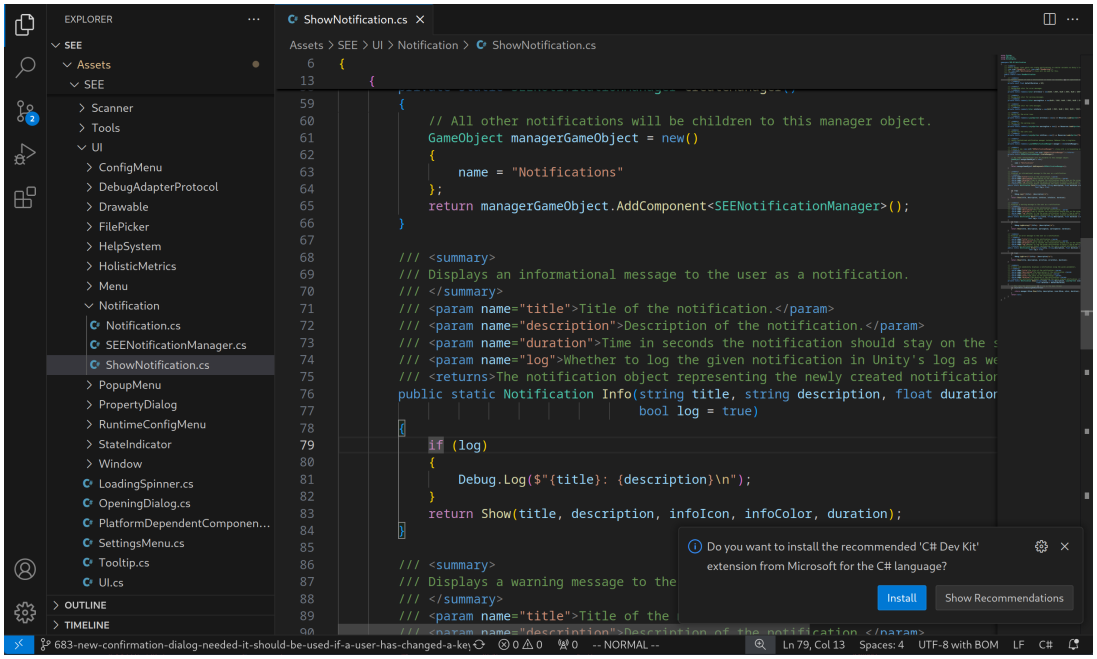


Figure 4.2: Screenshot of the main UI of VSCode.

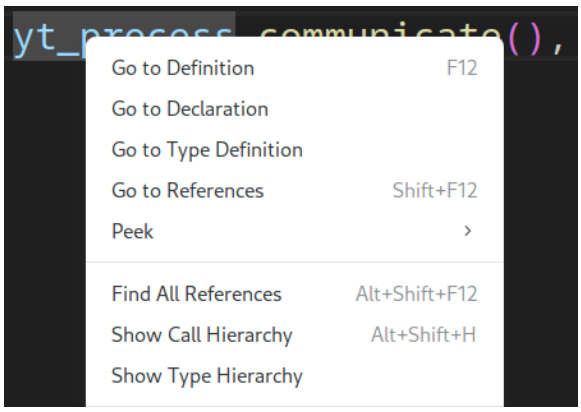


Figure 4.3: Screenshot of (the beginning of) VSCode's context menu for code identifiers.

similar context menu when right-clicking code identifiers, which is displayed in Fig. 4.3. Additionally, VSCode users can also jump to the definition of a symbol by holding down Ctrl and clicking on that symbol, the same as in Section 3.4.

However, a feature of SEE which *does not* have a clear alternative in VSCode is the ability to quickly be able to tell certain metrics, such as the number of methods in a file. In SEE, we could simply visualize that by encoding it as the size of each building, but in VSCode, participants would need to manually count each method, so to remedy this, we offer a table with such metrics, analogously to Wettel's study. The table is hosted on Google Spreadsheets[40] and can be sorted by any column as well as searched. The script with which the metrics were extracted is attached as `extract-metrics.py`.

4.1.3 Hypotheses

To answer RQ2 (see Section 1.3), we would like to know whether there is any significant[41] difference between the approaches on the dimensions of *speed*, *correctness*, and *usability*, similar to most other studies mentioned in Section 4.1.1. We will now create more concrete hypotheses for each of these dimensions.

a) **Correctness**: We call the correctness C_S for tasks done in SEE and C_V for tasks done in VSCode. This will be either a categorical variable with two possible values (i. e., correct or incorrect) or a rational number indicating the percentage of correct answers within a task.

[40] https://docs.google.com/spreadsheets/d/1Z2AQDk2-XeVBB1kAtcpc18mFkPI5ZSsSz_yOwS4pQ88 *(last access: 2024-11-20)* and https://docs.google.com/spreadsheets/d/1erJZTwYtG-CQfZJPT-zt-chX_VHL9jLrVfWAWZMFvEo *(last access: 2024-11-20)*.

[41] Any mention of "significance" in this chapter refers to statistical significance.

- *Null hypothesis* H_{a_0}: The correctness when using SEE is the same as when using VSCode: $C_S = C_V$.
- *Alternative hypothesis* H_{a_1}: The correctness when using SEE is different when using VSCode: $C_S \neq C_V$.

b) **Speed**: We call the time it takes to finish a task t_S for SEE and t_V for VSCode.

- *Null hypothesis* H_{b_0}: The time it takes to solve a task when using SEE is the same as when using VSCode: $t_S = t_V$.
- *Alternative hypothesis* H_{b_1}: The time it takes to solve a task when using SEE is different when using VSCode: $t_S \neq t_V$.

For the **usability**, we need to differentiate between the post-study* SUS we use to evaluate the usability of the system as a whole, and the reduced 2-item post-task** After-Scenario Questionnaire (ASQ)† we use after each task (see Section 4.2.1).

c) **SUS**: We call the SUS score for SEE S_S and the one for VSCode S_V.

- *Null hypothesis* H_{c_0}: The SUS score for SEE is the same as the SUS score for VSCode: $S_S = S_V$.
- *Alternative hypothesis* H_{c_1}: The SUS score for SEE is different from the SUS score for VSCode: $S_S \neq S_V$.

d) **ASQ**: We need to once again differentiate between the two aspects that the ASQ measures.

i) We call the ASQ score for *complexity*[42] A^c_S for SEE and A^c_V for VSCode.

- *Null hypothesis* H_{d_0}: The ASQ score for complexity when using SEE is the same as when using VSCode: $A^c_S = A^c_V$
- *Alternative hypothesis* H_{d_1}: The ASQ score for complexity when using SEE is different when using VSCode: $A^c_S \neq A^c_V$

ii) We call the ASQ score for *effort*[43] A^e_S for SEE and A^e_V for VSCode.

[42] Note that a higher score here means a lower amount of complexity.
[43] Again, a higher score indicates less required effort.

*Post-study: A questionnaire for participants which is answered at the end of the study, after every task has been completed.
**Post-task: A questionnaire for participants which is answered after each task.
†ASQ: A three-item questionnaire asking participants for satisfaction with ease, completion time, and support information [**lewis1991**].

- *Null hypothesis* H_{e_0}: The ASQ score for effort when using SEE is the same as when using VSCode: $A_S^e = A_V^e$
- *Alternative hypothesis* H_{e_1}: The ASQ score for effort when using SEE is different when using VSCode: $A_S^e \neq A_V^e$

We will use a significance level of $\alpha = 0.05$ for all of our tests, though this will get halved to 0.025 since we are using two-sided tests.

4.2 Design

Since all the hypotheses we just listed compare SEE to VSCode, the general form of our study should be to have participants solve representative software engineering related tasks, with one group solving tasks in SEE and the other using VSCode. There are going to be six tasks, each taking no longer than ten minutes, which we will go over in Section 4.2.2. To maximize the number of collected data points, we will have each group use both tools. Specifically, group Ψ will use SEE for the first three tasks and VSCode for the last three tasks, while this will be switched around for group Ω. We will then have $\lfloor \frac{n}{2} \rfloor$ data points per tool for each task, which we can then compare against one another to test for significant differences. This flow is illustrated in Fig. 4.5.

We have several constraints for our study. It should be possible to partake in the study online and asynchronously (i. e., without me needing to be present) to facilitate participation and reduce the Hawthorne effect* (though this will not completely eliminate it [**evans2010**]). To not overexert participants and thus confound any results, the study should also be of a reasonable length—ideally, not longer than an hour (hence the number of six tasks). Finally, we cannot assume familiarity with either SEE or VSCode, so we need to explain the core concepts of each system and verify that the participant understood them before starting the actual tasks.

Additionally, we would like to follow the "wish list" compiled by **wettel2011** (answering each item with yes [✓], maybe/partially [?], or no [✗]):

1. **"Avoid comparing using a technique against not using it."** We are comparing VSCode against SEE. [✓]

*Hawthorne effect: The effect through which participants in a study behave differently when under observation by a researcher. The name is based on experiments conducted at the Hawthorne Plant in the 1930's [**hawthorne1939**], although the effects seen there have later turned out to be likely unrelated to the observation [**jones1992**].

2. **"Involve participants from industry."** Roughly half of our participants have reported working on bigger software projects (either within a company or as open-source contributions) for at least 3 years (see Section 4.3.2), but I have neglected to explicitly ask whether a participant is involved in the industry. [?]

3. **"Provide a not-so-short tutorial of the experimental tool to the participants."** We do give a tutorial of each tool to participants, but it only covers the essentials required for the tasks (since we do not want to drag out the participation longer than necessary.) [?]

4. **"Avoid, whenever possible, to give the tutorial right before the test."** Unfortunately, we cannot really avoid this. Wettel suggests performing the training a few days before the study, but this is intractable for this study. [✗]

5. **"Use the tutorial to cover both the research behind the approach and the implementation."** The motivation here would be to provide another incentive to participate out of interest in furthering the research. We have left out the research both to keep the participation short and to avoid biasing participants (e. g., by claiming beforehand that developers find code cities much nicer to use than IDEs). [✗]

6. **"Find a set of relevant tasks."** Our tasks are representative of real-world activities (see Section 4.2.2), but some may turn out to have been too easy, as we will see in the analysis in Section 4.3.3. [?]

7. **"Choose real object systems that are relevant for the tasks."** We have chosen *SpotBugs* (3.5k GitHub stars, ≈ 216 kLOC) and *JabRef* (3.6k GitHub stars, ≈ 179 kLOC). The former is a fork of the abandoned static analysis tool *FindBugs* that Wettel used for his study, while the latter is a bibliography manager that is well-known enough that some of our participants have heard about or even used it before (see Section 4.3.2). [✓]

8. **"Include more than one subject system in the [experiment]."** See first item. [✓]

9. **"Provide the same data to all participants."** The tasks for both participants (and accompanying data) are identical, and VSCode participants get access to a table of metrics so that the same data is present. [✓]

10. **"Limit the amount of time allowed for solving each task."** Integrating a time limit like this into the tool we used for the questionnaire would have been difficult, but the tasks were comparatively simple and participants were aware that their time was measured, so we should hopefully have avoided the effects referenced here. [?]

11. **"Provide all the details needed to make the experiment replicable."** In Appendix B.4, I have included (pseudonymized) participation data, the full definition of the questionnaire, and the scripts which performed data cleanup, statistical tests, and generated the data for the graphs shown in this chapter. [✓]

12. **"Report results on individual tasks."** We are going over each task individually when reporting results in Section 4.3. [✓]

13. **"Take into account the possible wide range of experience level of the participants."** We are taking a rigorous look at the effects of experience in Section 4.3.6. [✓]

In total, we were able to fulfill 7/13 items on the wish list, with 4/13 being questionably fulfilled and 2/13 not being implemented. The reasoning behind the unimplemented two were that it would otherwise lengthen the participation time and may strain participants.

4.2.1 Questionnaires

There are three questionnaires that we are going to use for this study: A demographic questionnaire, a post-task questionnaire and a post-study questionnaire, with the last two being used to estimate usability. We are also going to take a look at *KoboToolbox*, the tool we are using to asynchronously conduct our survey. An important general consideration is that we will offer our study in both English and German, so any questionnaire we choose (which is usually given in English) needs to have a validated German translation to make results comparable. This section partially mirrors similar considerations from my bachelor's thesis, which the interested reader can peruse for more details [**galperin2021**]. For a comprehensive overview and evaluation of other post-study and post-task questionnaires not covered here, see the review by **hodrien2021**.

Demographics We ask each participant various demographic questions to (among other reasons) address the threat to validity of selection bias: While participants are randomly assigned into groups Ψ and Ω, it is of course possible due to our moderate sample size of $n = 20$ that there is a significant difference in any independent variable. A difference like this would then matter if it acts as a confounder for one of the dependent variables—for example, age or experience can have a sizable impact on SUS scores [**bangor2008**; **mclellan2012**].

To catch this, we ask for some properties that could be relevant, namely gender, age, programming experience, professional programming experience, knowledge of SEE, knowledge of VSCode, knowledge of JabRef, knowledge of SpotBugs, and experience with 3D video games. We ask the last question since SEE uses some typical video game paradigms

for its controls, so a familiarity with such controls may make SEE more intuitive to use for those people. Additionally, we ask if the participant has ever used IDEs before, and if they are able to use the Java programming language. If the answer to either of those questions is "no," we display a warning telling the participant that this study is not intended for them and may be too difficult (as both JabRef and SpotBugs are Java projects). We will later filter participations with "no" answers out of our dataset to avoid these problems.

POST-TASK: ASQ This questionnaire will be filled out by the participant after each task—since there are six tasks, this will be six times total. For this reason, we should keep this one short, that is, no more than three questions long, which eliminates questionnaires such as NASA's Task Load Index (TLX)*. There are also the rather unconventional Usability Magnitude Estimation (UME)**, which we exclude since it often confuses its users [**sauro2009b**], or Subjective Mental Effort Question (SMEQ)†, which we exclude because this would be hard to implement in our online questionnaire. Instead, we choose the *After-Scenario Questionnaire*, which consists of three statements for which users give answers on a Likert scale‡:

1. "Overall, I am satisfied with the ease of completing the tasks in this scenario."
2. "Overall, I am satisfied with the amount of time it took to complete the tasks in this scenario."
3. "Overall, I am satisfied with the support information (on-line help, messages, documentation) when completing the tasks."

We exclude the last question since we neither offer nor want to measure usage of such support information. However, the rest is ideal for our case: The ASQ is short, easy to understand, and distinguishes between cognitive and temporal ease—we will call "cognitive ease" *complexity* and "temporal ease" *effort* (as in Section 4.1.3). The translation by **roegele2020** will serve as the German version.

POST-STUDY: SUS We will present this questionnaire once after the SEE section and once after the VSCode section, and will use it to collect a general usability score of each system. Since the questionnaire is filled out twice, we set an upper limit of twenty questions. This

*TLX: A post-task questionnaire developed by NASA consisting of six questions asking about mental, physical, and temporal demand, as well as about performance, effort, and frustration level. [**hart1988**]

UME: A post-task questionnaire which does not depend on any pre-defined scale. Instead, numbers given by each user are put in relation to each other, with the resulting ratios serving as the usability measure. [mcgee2003**]

†SMEQ: A single question intended to estimate usability which uses a scale with 150 options. [**zijlstra1985a**]

‡Likert scale: A psychometric scale in which users indicate their agreement on a linear scale from "strongly agree" to "strongly disagree" [**likert1932**].

eliminates both the Software Usability Measurement Inventory (SUMI)* and Questionnaire for User Interaction Satisfaction (QUIS)** from consideration. Because we need a validated German translation, we can also strike out the English-only Post-Study System Usability Questionnaire (PSSUQ)† as an option.

This leaves the *System Usability Scale* as a fitting option: It has a validated German translation [**reinhardt2015**], is the most used post-study questionnaire for this purpose [**sauro2009**; **lewis2018**], and is very well-suited for comparing two systems or interfaces [**peres2013**; **bangor2008**]. It consists of ten Likert scale questions which alternate between positive and negative aspects of the system and returns a score between 0 and 100, representing the system's usability [**brooke1996**]. These questions are collected in Appendix B.1. Existing measured SUS scores for SEE are given in Fig. 4.1.

KoboToolbox The tool with which we manage the survey needs to handle our requirements for it to be feasible to conduct it asynchronously. Specifically, we need to:

- reliably measure the time each task takes to solve,
- provide both an English and a German version of the questionnaire,
- ask questions with numerical, text-based, selection, and Likert scale-based input, and
- conditionally show fields or bar the user from continuing (e. g., until the tutorial question is answered correctly) depending on existing input.

Among the tools that can handle this for free, *KoboToolbox*[44] seems like the most mature and robust option. It accepts form based on the XLSForm‡ specification, which makes forms (relatively) easy to create with a spreadsheet editor. KoboToolbox has also proven itself as a valuable tool in my bachelor's thesis.

Before each task, we tell the user that their time will be measured as soon as they advance to the next page. Then, we keep a timestamp for each editable field in the task, including a checkbox the user can check to indicate they have finished the task. Using the set of those

[44] `https://www.kobotoolbox.org/` *(last access: 2024-11-23)*

*SUMI: A 50-item questionnaire measuring usability on the axes of efficiency, affect, help and support, steerability, and learnability. [**kirakowski1994**]

QUIS: A post-study questionnaire measuring usability across 41 questions in its short version and 122 questions in its long version. [chin1988**]

†PSSUQ: A questionnaire that measures usability in sixteen questions across the three factors *usefulness*, *information quality*, and *interface quality*. [**lewis1992**; **lewis2002**]

‡XLSForm: A form standard based on human-readable Excel spreadsheets, which allows for complex forms (e. g., conditionals, skip logic) to be built [**marder2024**]. The forms can then be converted into the open ODK XForm format, which is compatible with a number of data collection tools [**odkcommunity2019**].

timestamps T and the timestamp t_0 of when they started the task, the time for the task can then be calculated as $\max T - t_0$. This also ensures we catch "cheating" in the form of participants editing fields after they have indicated they are finished with the task. The survey in XLSForm is linked in Appendix B.4.3, and a screenshot of the starting page is shown in Fig. 4.4.

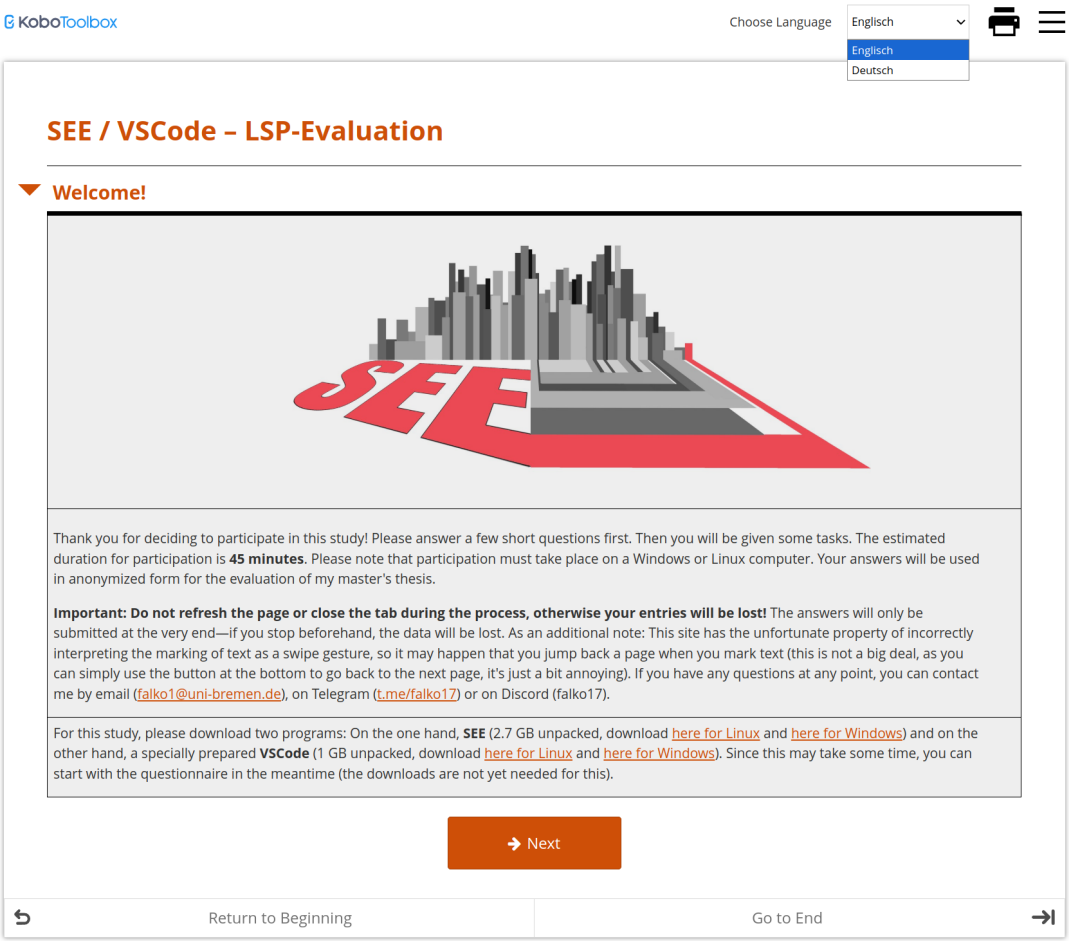


Figure 4.4: Starting page of the survey built with KoboToolbox.

To equally partition participants into two groups, we create two versions of the survey, one for group Ψ and one for group Ω. We then access KoboToolbox's REST API to redirect the participants to version Ψ or Ω—the algorithm we are using is given in Algorithm 3, or can alternatively be seen in `redirect-evaluation.py`. Inviting others to participate then becomes as simple as passing the link to the redirection script around[45].

[45] `https://falko.de/master-evaluation`, but this link may not stay up forever.

Algorithm 3 How participants are redirected to the two versions of the survey.

Input: Participant ID p retrieved from cookie in request, or $\varnothing$ if it does not exist.
Output: Tuple consisting of the participation ID to be set as a cookie and a survey link.

```
▷ Global state: P is an initially empty mapping from IDs to links, i is initially zero. ◁
if p ∉ P then
    if p = ∅ then
        p = NewRandomID()
    n_Ψ, n_Ω ← GetKoboParticipations()
    if n_Ψ = ∅ ∨ n_Ω = ∅ then                ▷ API request failed, so just alternate using index i. ◁
        if i = 0 then
            P(p) ← Ψ
        else
            P(p) ← Ω
        i ← (i + 1) mod 2
    else if n_Ψ ≤ n_Ω then
        P(p) ← Ψ
    else
        P(p) ← Ω
return (p, P(p))
```

4.2.2 Tasks

As explained in Section 4.1, we cannot feasibly evaluate most implemented LSP capabilities except for the generation of the code city itself. This means we are actually comparing a code city against an IDE, with LSP only playing the role of providing us with those code cities, for which there are a number of existing experiments that we went over in Section 4.1.1. Thus, we should base our own study design on the existing literature established in that section. Specifically, out of the similar studies collected in Table 4.1, the study by **wettel2011** (replicated by **romano2019**) seems most fitting for us: It compares a desktop code city implementation against a traditional IDE along with providing tabular metrics to make the comparison fair. In addition, the paper provides a great number of details, including detailed task definitions, and a wish list which we went over at the beginning of Section 4.2.

Wettel uses two kinds of tasks [**wettel2011**]: Those concerned with program comprehension and those concerned with design quality assessment. Of these, the latter requires deeper software engineering knowledge that may not be present in all participants, since a large proportion of them are still going to be students. Thus, we will focus on tasks from the first group:

1. "Locate all the unit test classes of the system and identify the convention (or lack of convention) used by the system's developers to organize the unit tests."

   ☞ This task seems like a nice fit to realistically evaluate usage of both tools, so we include it. We will call it **task *B***.

2. "Look for the term T in the names of the classes and their attributes and methods, and describe the spread of these classes in the system."

   ☞ The concept of "spread" would take time to properly introduce and may be misunderstood by some participants. In addition, quantifying the correctness would be hard here, so we exclude this task.

3. "Evaluate the change impact of class C defined in package P, by considering its caller classes. The assessment is done in terms of both intensity and dispersion."

   ☞ This seems like a complex task that takes time both to properly understand it and then also to execute it, so we exclude it for similar reasons as the previous task.

4. "Find the three classes with the highest Number of Methods (NOM)* in the system."

   ☞ This task also seems simple to understand and probably will not take too long, so we include it. We will call it **task *A***.

Note, however, that none of the tasks we selected here test any edge-related behavior, so we will add one task of our own. This **task *C*** will be to "find the base class for class c in the system." Its drawback is that it requires a short explanation of base classes before the task, but its upsides are that it is both a realistic task and additionally forces participants to have more than a cursory interaction with the edges (specifically, the `Extend` edges[46]) without having any unfair disadvantage in VSCode, where participants can navigate up the inheritance tree by repeatedly Ctrl-clicking on the superclass.

Now we only need to choose a fitting c for both JabRef and SpotBugs, our two object systems. To make sure the task is not too easy, we choose the class which is deepest in the inheritance tree for both systems. For JabRef, this is `GenderEditorViewModel` (five levels deep), while for SpotBugs, this is `OptionalReturnNull` (seven levels deep). This way, we can also evaluate the transitive edge animations we added in Section 3.6.

We should also mention that, for task B, like Wettel, we give multiple choice options for the various kinds of conventions along with brief descriptions to make sure they are understood correctly. If one of the options is then chosen (e. g., "Centralized"), users have to give follow-up information (e. g., "What is the full name of the root package for test classes?") to make sure the correct answer was not just guessed. A full visual overview of our study along with its tasks is given in Fig. 4.5.

[46] We should also note that the code cities for this study *only* contain `Extend` edges—otherwise, there would be too many edges for SEE to handle in a performant manner.

*NOM: The number of methods in a certain class.

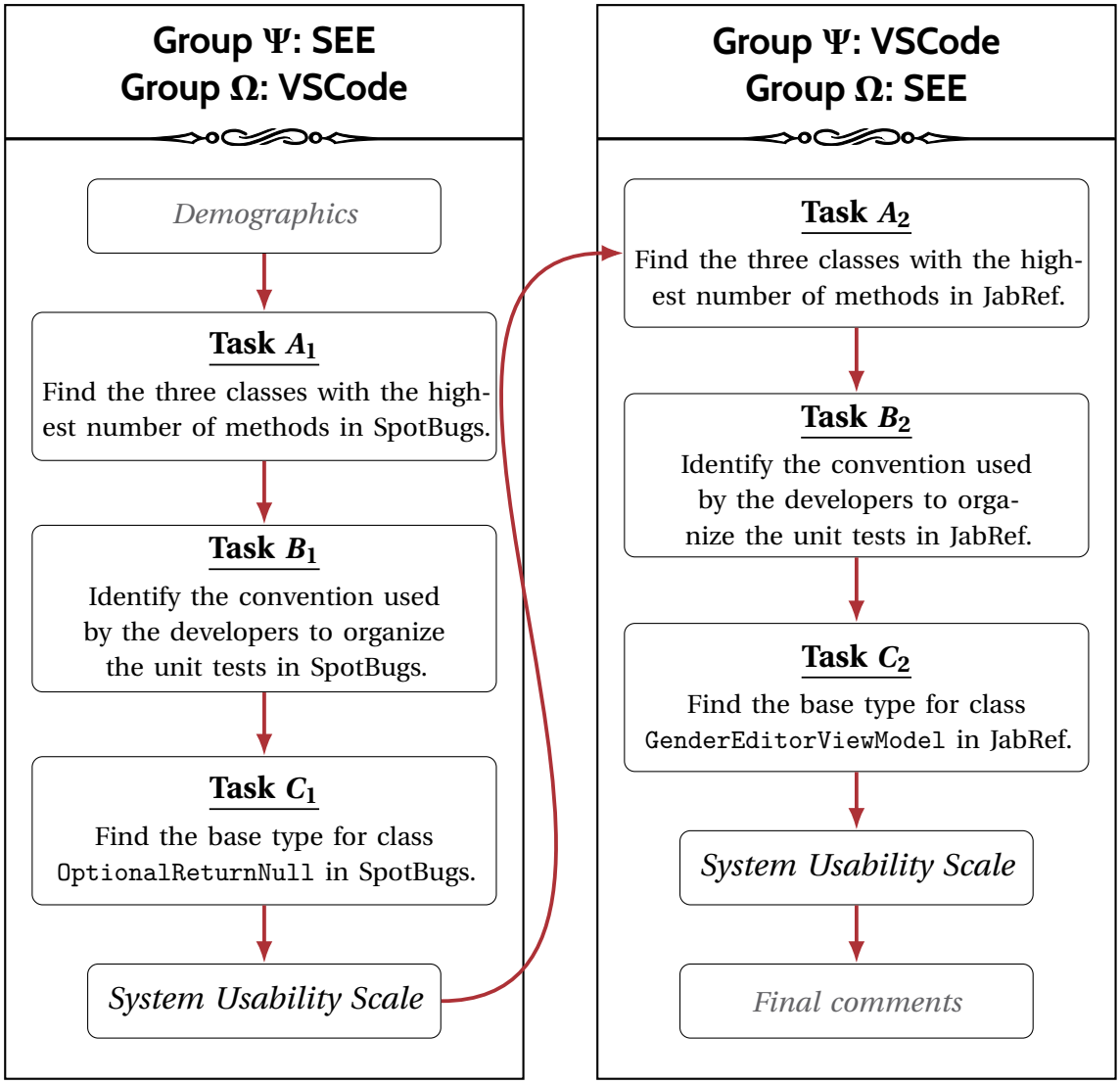


Figure 4.5: Flow of the tasks the participants worked on.
The post-task ASQ questions were asked after each task to gather A^c and A^e.

4.3 Results

In this section, we can finally report on the results of the study. We will first go over the course of how the study was conducted before taking a look at the results themselves. For the latter, we start by analyzing the answers to the demographics questionnaire, and then go on to interpret the dependent variables relevant to our hypotheses, namely, correctness, time, and usability, in that order. We will then also investigate the role of experience (and some other independent variables) and the influence it had on how tasks were handled. Finally, we handle the various comments that were left during the study.

As mentioned before, we use a significance level of $\alpha = 0.05$. We can neither assume normal distributions for our data, nor is it always interval-scaled, so we choose the Mann-Whitney U test* to determine statistically significant differences between either our two groups Ψ and Ω or between the two tools SEE and VSCode. To help the reader quickly identify important results, we put a star (★) in the heading of a paragraph to indicate that it lead to the rejection of a null hypothesis.

4.3.1 Study Procedure

Before we started with the actual study, we did a small-scale pilot study with two participants, one of them being this thesis's first reviewer Prof. Dr. Rainer Koschke. Through their valuable feedback, I was able to see some issues I had not noticed myself before starting the study, such as a few bugs in SEE, confusion regarding the term base class, and errors unpacking the prepared compressed builds of SEE and VSCode[47] when using Windows. I fixed the bugs in SEE, added a tutorial for the base classes (along with a comprehension question), and added detailed instructions on how to correctly unpack the builds. There luckily were no other major issues in the actual study after these problems noticed in the pilot study have been ironed out.

There have been 20 participants in total, though we had to exclude one participant from most analyses because he indicated he had no knowledge of Java. Participants were gained via convenience sampling, that is, I asked fellow students, colleague developers from Axivion, and other acquaintances who had software engineering experience to participate.

[47] As a sidenote, using prepared builds also prevents any potential problems with pre-installed versions of the tools. For example, some plugins of VSCode might otherwise interfere with our study.

*Mann-Whitney U test: A test to check whether two distributions have identical distributions, requiring only ordinal-scaled independent samples drawn from the distributions. [**mann1947**]

4.3.2 Demographics

First, we want to take a quick look at the demographics of our study. We also take this opportunity to find any significant differences between the two groups Ψ and Ω, to make sure any results in the dependent variables later on are not just due to such differences. For this purpose, we define null hypotheses of the form "Variable X is not different between groups Ψ and Ω", which we check with a two-sided Mann-Whitney U test, as we cannot assume normal distributions. This also allows us to compare data that is only ordinal.

All answers to the demographic questions can be viewed as bar charts in Fig. 4.8. Two exceptions are the non-categorical variables, namely age and the total time it took participants to finish the study[48], which we are going to go over first. We have also excluded the question "Have you ever used the program SpotBugs?", as there was only one participant answering "Heard of it," with all others answering "No." We have also excluded gender from analysis, since we only have one female participant.

AGE We have a median age of 29 and an average age of 28.55 in our sample, suggesting few outliers, which can be confirmed by looking at the violin plots in Fig. 4.6. The Mann-Whitney U test also confirms no differences between the groups ($U = 40.5, p \approx 0.4947$).

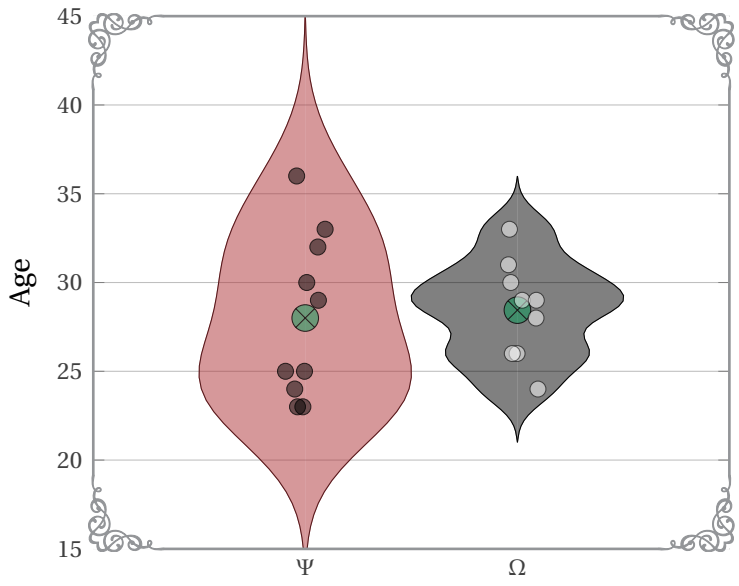


Figure 4.6: Distribution of age between the two groups.

[48] This technically does not belong into the "Demographics" category, but it still seems most fitting to include it here rather than in one of the other sections.

Total Time The median time it took to complete the whole study was an hour and six minutes, which means we slightly overshot our initial aim of an hour to complete the study. In addition, the average being at roughly one and a half hours points to there being some significant outliers: In fact, some participations were longer than 2 hours, with one participant in the Ω group even taking almost five hours—however, as no single task took anyone longer than twenty minutes (see Section 4.3.4), we can conclude that this must be due to breaks in between tasks rather than the participation being actually that long. The same likely applies to the other participations around the three hour range. The various times are visualized in Fig. 4.7, where it also becomes apparent that the two groups had similar total times, which a Mann-Whitney U test confirms for us ($U = 63, p \approx 0.34470$).

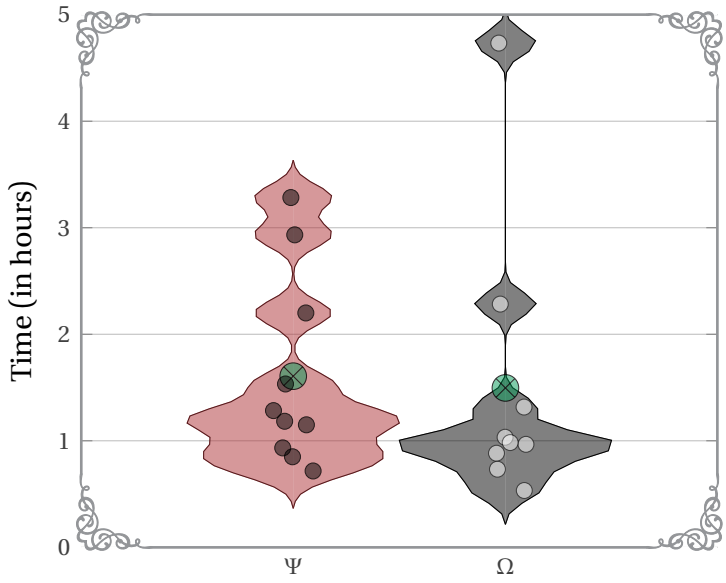


Figure 4.7: Total duration of time in which each participant finished the study. May include time in which participants took breaks.

Degree Most participants had a Bachelor's degree (this is also the median), with some Abitur qualifications, Master's degrees, and one person with a doctoral degree (see Fig. 4.8a). The Mann-Whitney U test reveals no significant differences between Ψ and Ω ($U = 57.5, p \approx 0.5693$).

★ **Programming Experience** Both the median and modal programming experience is 3–9 years. However, we can see that the only people with 10–19 years of experience are in group Ψ, and the Mann-Whitney U test indeed confirms a significant difference here ($U = 77, p \approx 0.0214$). It is unclear how this could have happened except coincidentally, as

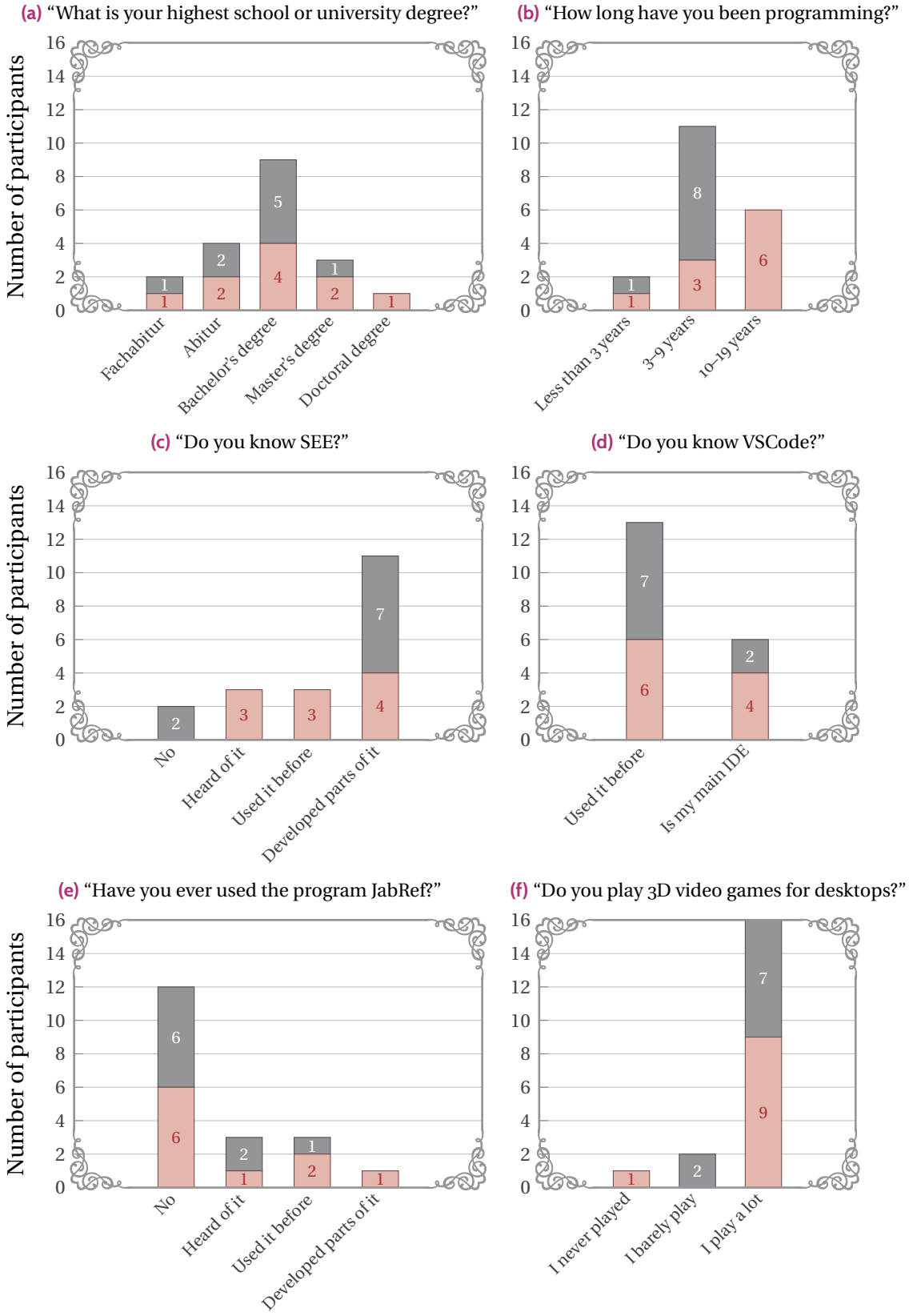


Figure 4.8: Number of respective answers to various demographic questions. Group Ψ is in red, and group Ω is in gray.

participants were always alternatingly assigned to either group Ψ and Ω. Still, we should keep this difference in mind when moving on to the analysis of dependent variables.

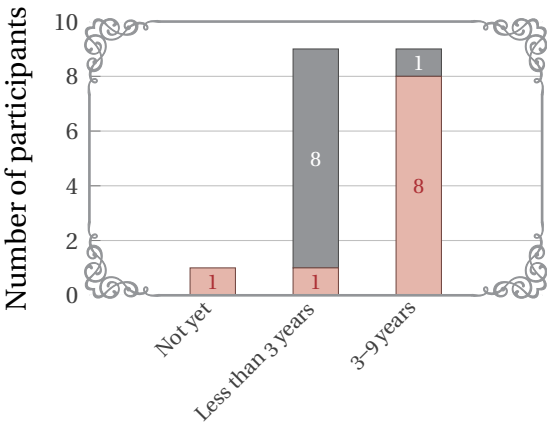


Figure 4.9: "How long have you been programming on bigger software projects (e. g., within a company, or open-source projects)?"

★ Experience on Bigger Software Projects We also asked participants for experience with larger software projects, such as within companies or on open-source projects. Here, all but one participant have been working on bigger software projects, but the split between the "Less than 3 years" and "3–9 years" categories are inverted between groups Ψ and Ω, which can also be seen in Fig. 4.9. The Mann-Whitney U test again confirms a significant difference ($U = 80.5, p \approx 0.0105$).

Experience with SEE The clear majority of participants (eleven people) have developed on SEE before, as can be seen in Fig. 4.8c, which can be attributed to the fact that one major group of people I asked to participate are the students who are working on SEE in a professional or academic capacity. We can also see a qualitative difference between the two groups: Group Ω contains the only people who have answered "No", while group Ψ contains the only people who have answered "I heard of it" and "I used it before". However, this difference is not significant ($U = 42.5, p \approx 0.5598$).

Experience with VSCode Every single participant has used VSCode before, with six people even using it as their main IDE. We can assume that most participants thus have more experience with VSCode than SEE, which may bias our results somewhat—we try to investigate this further in Sections 4.3.6 and 4.4. Figure 4.8d shows the groups to be quite similar, and indeed, there is no significant difference here ($U = 55, p \approx 0.6809$).

Experience with JabRef Most participants in either group have not heard of JabRef before, but in contrast to SpotBugs, some have actually used the bibliography manager before, with one participant in group Ψ even having developed parts of it (see Fig. 4.8e). Again, there is no significant difference between the two groups ($U = 58, p \approx 0.5041$).

Experience with Video Games A clear majority of sixteen participants reported that they play a lot of video games (as can be seen in Fig. 4.8f), with only two responding that they barely play and one stating that they never played video games before. There is no significant difference we need to be aware of ($U = 46, p \approx 0.6701$) here, either.

In summary, the only significant differences between the two groups that could cause problems in our analysis are the programming experience in general and the programming experience for bigger projects, both of which indicate significant differences insofar that group Ψ apparently contains more experienced programmers. We will investigate the effects of this in more detail in Section 4.3.6.

4.3.3 Correctness

To evaluate the correctness of the various answers, I wrote a script which showed me each answer and prompted me to say whether this answer was correct or incorrect, while I used the previously created answer key (see Appendix B.4.4) as a base. This allowed me to catch misspellings or unusual formats[49] and still mark the answers as correct. The results of which answers I marked as correct and the script used to do so are attached as part of *analysis.zip*.

It does not make much sense to use violin plots here, as there are in most cases only two possible values here: The participants were either correct or incorrect in their answer to the task. Even for task *A*, where three answers had to be given, we only have at most three possible values in practice. For this reason, we are using bar graphs again, which are collected in Fig. 4.10 along with the respective results of the statistical tests. I also need to note that, while task *A* can be analyzed with the Mann-Whitney *U* test, tasks *B* and *C* with their binary results are much better suited for Fisher's exact test*, which takes a 2×2 contingency table as input. We chose this test instead of the more frequently used χ^2 test [**pearson1900**] because it is (as its name implies) more exact, and thus works better with smaller sample sizes [**fisher1922**], whereas its downside is a more involved computation, which we do not need to worry about here.

We can make several observations here: First, performance compared across the two object systems JabRef and SpotBugs is remarkably similar within tasks, which tells us that neither of the two systems was harder to analyze than the other under our tasks. We can also see a slight difference across subject systems in task *B*, where participants gave slightly more correct answers when using VSCode for B_1 and when using SEE for B_2, but none of the *p*

[49] For example, some participants also entered the NOM along with the class name for task *A*.

*Fisher's exact test: A test for statistical significance for contingency tables, intended as an alternative to the χ^2 test to be accurate even with smaller sample sizes. [**fisher1922**]

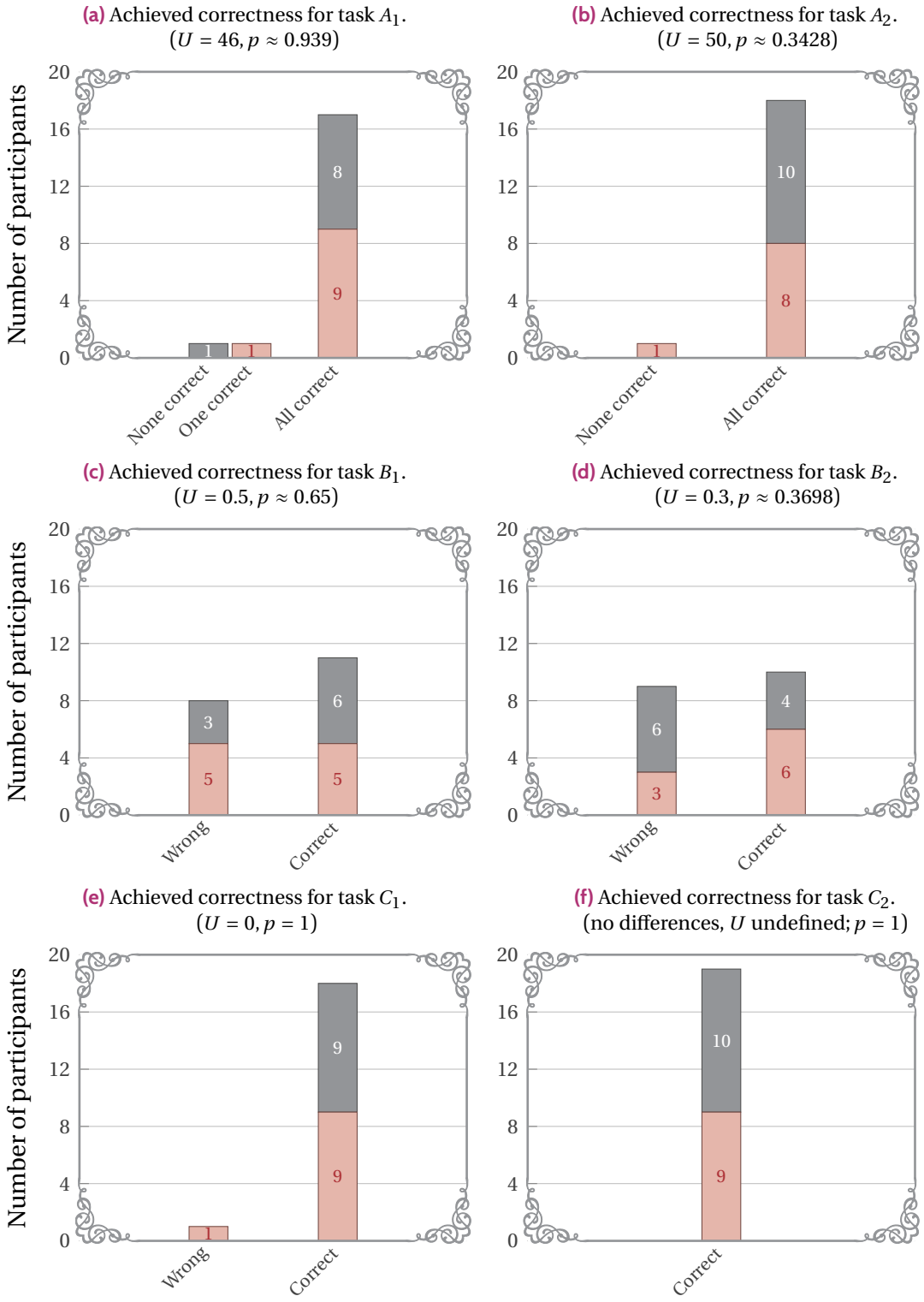


Figure 4.10: Correctness achieved in the various tasks, compared across the two systems. Correctness when using SEE is in red, and when using VSCode is in gray.

values come close to our significance level, meaning that the choice of subject system did not affect the correctness for these tasks. Finally, we can tell that tasks A and C were easy (with only one or two participants per task making mistakes), while task B was seemingly hard (with the wrong/correct divide being closer to 50%).

As a reminder, task B was about determining how unit tests were organized in the system. For SpotBugs, there actually were no unit tests under the root package we included—there were some files with `Test` in their name, but as SpotBugs is a static analysis tool, these were actually about detecting and handling tests for the projects that SpotBugs analyzes. The only class that could be reasonably construed as a test was `TestDataflowAnalysis`, so we allowed both "none" and "dispersed" as answers, though the latter only if `TestDataflowAnalysis` was given as an example. The most common error here was participants mistaking the aforementioned test detectors by SpotBugs as unit test classes *for* SpotBugs (either giving them as a "centralized" or "dispersed" example), while the second-most common type of wrong answer were "other" answers which consisted of confused explanations of the system in the free text field.

For JabRef, unit tests were all centralized under the `src.test` package, so I had expected there to be less issues here for task B than with SpotBugs. Figures 4.10c and 4.10d prove this assumption wrong. The most common (and actually only) type of error was to answer "dispersed" and then give one or more test classes under the centralized test package hierarchy as examples. I am not sure what went wrong here. While I have provided explanations of the "dispersed" and "centralized" conventions, perhaps this was still misunderstood by participants (e. g., maybe because the test package hierarchy mirrors the main package hierarchy, participants thought that the "dispersed" term applies here). Another possible explanation is that participants simply missed the fact that the test file they were looking at (which they likely arrived at by searching for the term "Test") was actually in the `src.test` package instead of `src.main`. Still, it is interesting that this happened with both VSCode and SEE.

In conclusion, we cannot reject H_{a_0} at all, and thus keep assuming that $C_S = C_V$.

4.3.4 ★ Time

As mentioned before, we calculated the time it took to complete a task using the set of timestamps T and the starting timestamp t_0 as $\hat{t} = \max T - t_0$. This ensures that the actual time is no higher than $\hat{t}$, but it does not catch the opposite: It is possible that participants were, for example, distracted and had to stop solving the task to attend some other matter, which could explain outliers like the twenty minute participation time for one participant in task A_1 (see Fig. 4.11a). Still, while there are some such outliers, the measured times were

relatively consistent, so this method should have been accurate enough for our purposes. While the violin plots in Fig. 4.11 include all data points, for the Mann-Whitney U tests we exclude data points belonging to incorrect answers for the respective tasks (as we do not want to count quick but wrong solutions).

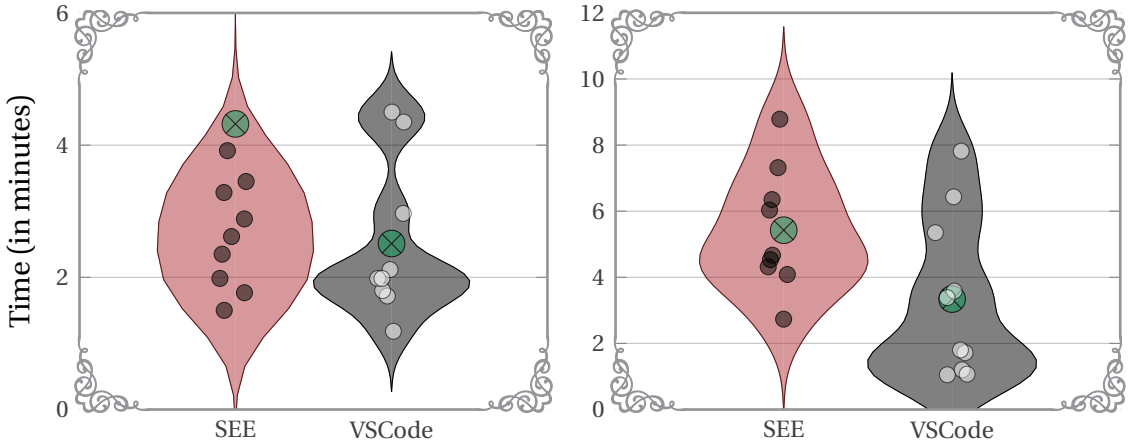
We can see a slight, but non-significant advantage for VSCode for task A (Figs. 4.11a and 4.11b), where the classes with the highest NOM had to be identified. VSCode users were allowed to use a table that could be sorted by exactly this metric, while SEE users had to glean this information by inspecting the size or color of the buildings in the code city. The former made this task very simple and quick to solve, probably even more so than the visual method provided by SEE. In task B, by contrast, we see (Figs. 4.11c and 4.11d) a slight and again non-significant advantage for SEE. This was the task about identifying the way that unit tests were organized. Perhaps SEE made the structure of the package hierarchy more immediately visible, which would have especially helped in task B_2 (where we do see a bigger margin) where all tests were in a separate hierarchy—in VSCode, this would have just been a path at the top of the IDE indicating where the file belongs, while for SEE it would be immediately apparent due to the node's location that the class belongs to a certain package.

The only actual significant difference lies in task C (Figs. 4.11e and 4.11f) for both object systems. Here, participants using VSCode outperformed those using SEE, with an especially clear difference for task C_1, where we have a p-value of ca. 0.1%. This was the task about finding a base class for a given class. In VSCode, this was very simple: All that participants needed to do was to keep Ctrl-clicking on the superclass until they eventually reach a class without any parent. In SEE, this was similarly simple, but required more interaction with the 3D city. The transitive edge animation does pretty quickly point to the base class of the hovered node, but especially for SpotBugs (see Fig. 4.11e for the more pronounced difference) the final edge pointed to a relatively dense and small cluster of nodes, so users would have to zoom in and move the city around further before being able to clearly tell which node the edge targets. This forced participants to become more familiar with SEE's controls compared to VSCode's very simple repeated clicking, so this is my primary guess as to where the time difference comes from. Some of the comments that we got in Section 4.3.7 seem to support this interpretation.

While four out of the six tasks yielded no significant differences, two of them point to VSCode increasing the participants' speed, so we will reject H_{b_0} and accept H_{b_1} instead, specifically, $t_V < t_S$.

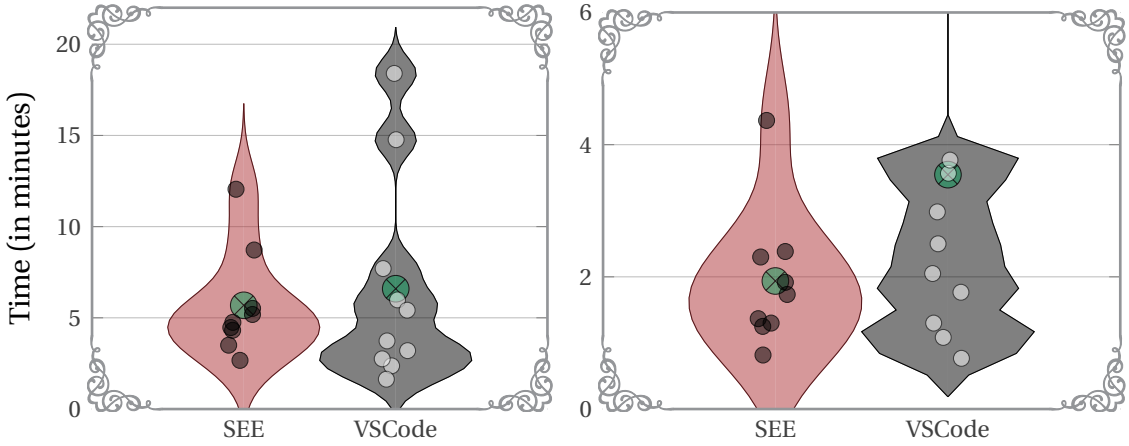
(a) Time taken for task A_1. Excluded a SEE outlier that took 20 minutes. ($U = 47, p \approx 0.3213$)

(b) Time taken for task A_2. ($U = 19, p \approx 0.0676$)



(c) Time taken for task B_1. ($U = 5, p \approx 0.08225$)

(d) Time taken for task B_2. Excluded a VSCode outlier that took 16 minutes. ($U = 15, p \approx 0.6095$)



(e) Time taken for task C_1. ($\bigstar$ $U = 77, p = 0.0015$)

(f) Time taken for task C_2. Excluded a SEE outlier that took 8 minutes. ($\bigstar$ $U = 17, p = 0.0247$)

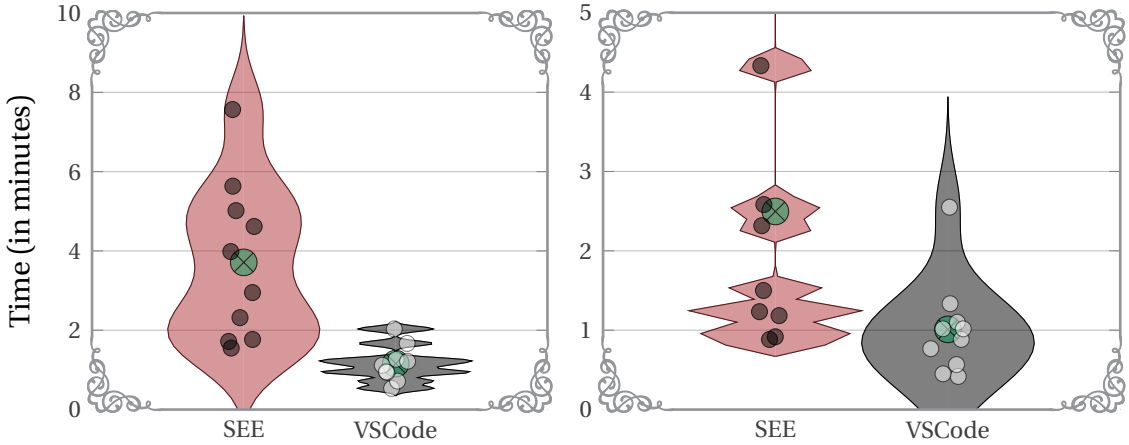


Figure 4.11: Time that it took participants to solve the tasks, compared across the two systems. Outsiders are excluded by cutting off the Y-axis—they are still included in the average.

4.3.5 ★ Usability

We have measured the usability in two ways: With the ASQ after each task (six times total), which we will examine first, and with the SUS after each system (twice total), which we will examine afterwards.

ASQ: Complexity The ASQ consists of two questions, the first of which is about the "cognitive" complexity or difficulty of solving the task—this is the factor we will examine here. Comparisons of this factor between the two systems are visualized in Fig. 4.12. Note that the values are Likert scale responses to the question given in Section 4.2.1, meaning that *lower* scores correspond to *higher* complexity.

Figures 4.12a, 4.12c, 4.12d and 4.12f for tasks A_1, B_1, B_2, and C_2, respectively, show only negligible or even close to no differences between the two systems. The only differences that come at least within an order of magnitude of our significance level of $\frac{\alpha}{2} = 0.025$ are the results for tasks A_2 and C_1, but even those are due to one or two outliers with very high complexity on the side of SEE without which their p-value comes closer to those of the other tasks.

Thus, from both a qualitative view (i. e., by looking at the violin plots) and a quantitative view (i. e., through the Mann-Whitney U tests) we cannot draw any conclusions that are backed by statistically significant differences on the perceived complexity here and keep the null hypothesis H_{d_0}, that is, we keep the assumption that $A_V^c = A_S^c$.

★ ASQ: Effort The second factor measured by the ASQ is the "temporal" effort, that is, whether participants were happy with the amount of time it took them to solve a given task. As with the previous section, the comparisons shown in Fig. 4.13 are of a form where, the *higher* the score is, the *lower* the perceived effort is.

Interestingly, the tasks with the smallest difference in answers to this questions are once again tasks A_1, B_1, B_2, and C_2 in Figs. 4.13a, 4.13c, 4.13d and 4.13f. The remaining two Figs. 4.13b and 4.13e belonging to tasks A_2 and C_1, however, actually produce a statistically significant result here, both times in favor of VSCode, where participants felt they could solve the task faster than those using SEE. A quick comparison against Fig. 4.11 in Section 4.3.4 reveals that they are right: In task C_1, participants were significantly faster when using VSCode, and while there was no significant difference for task A_2, participants were still faster on average with VSCode.

It is slightly puzzling, though, that no such difference in the perceived effort exists for task C_2, where Fig. 4.11f *does* show a significant difference. Looking again at the result of

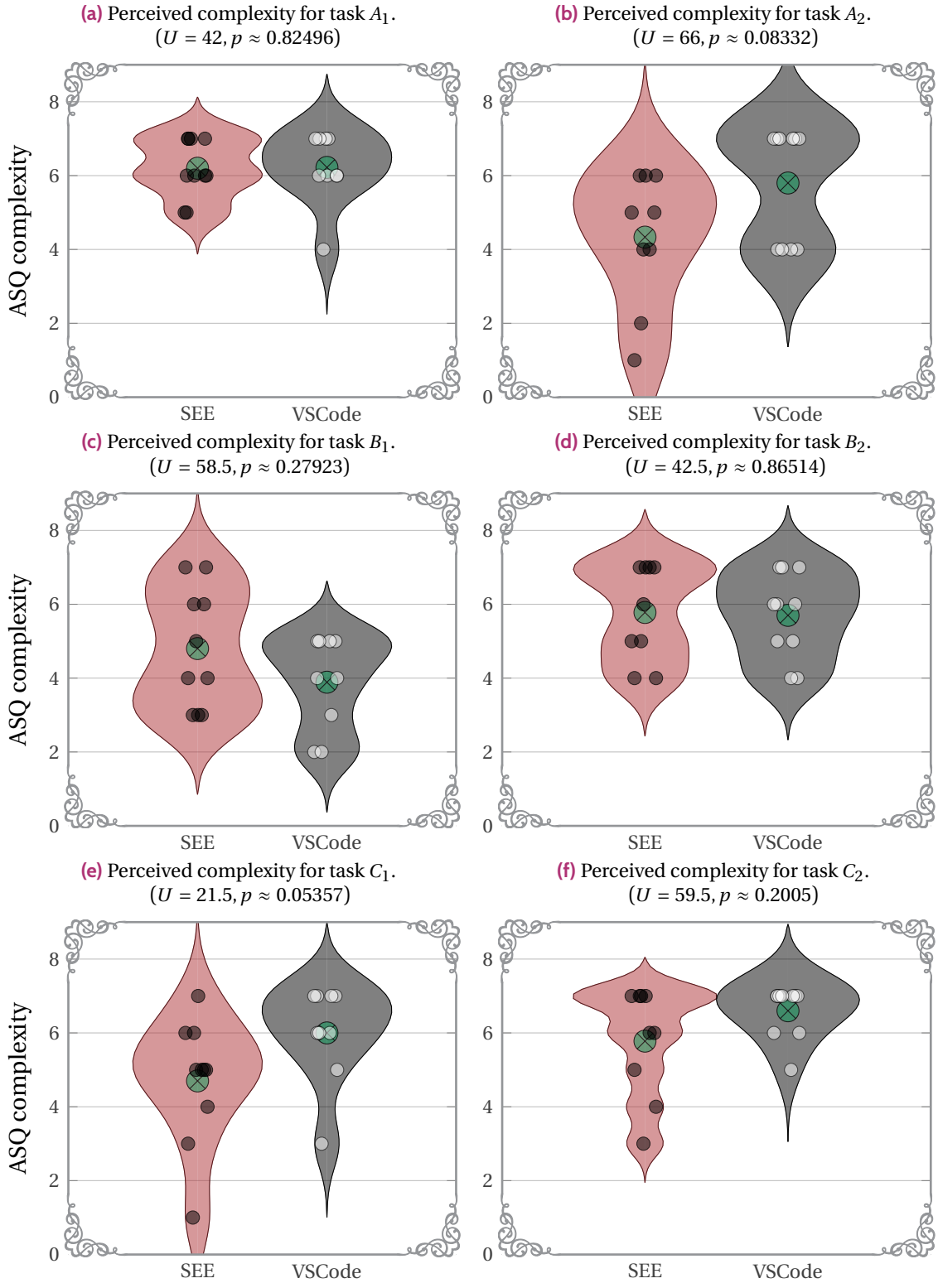


Figure 4.12: Perceived complexity for each system as measured by the ASQ for each task. **A higher number means an easier perception of the task, i. e., *lower* complexity.**

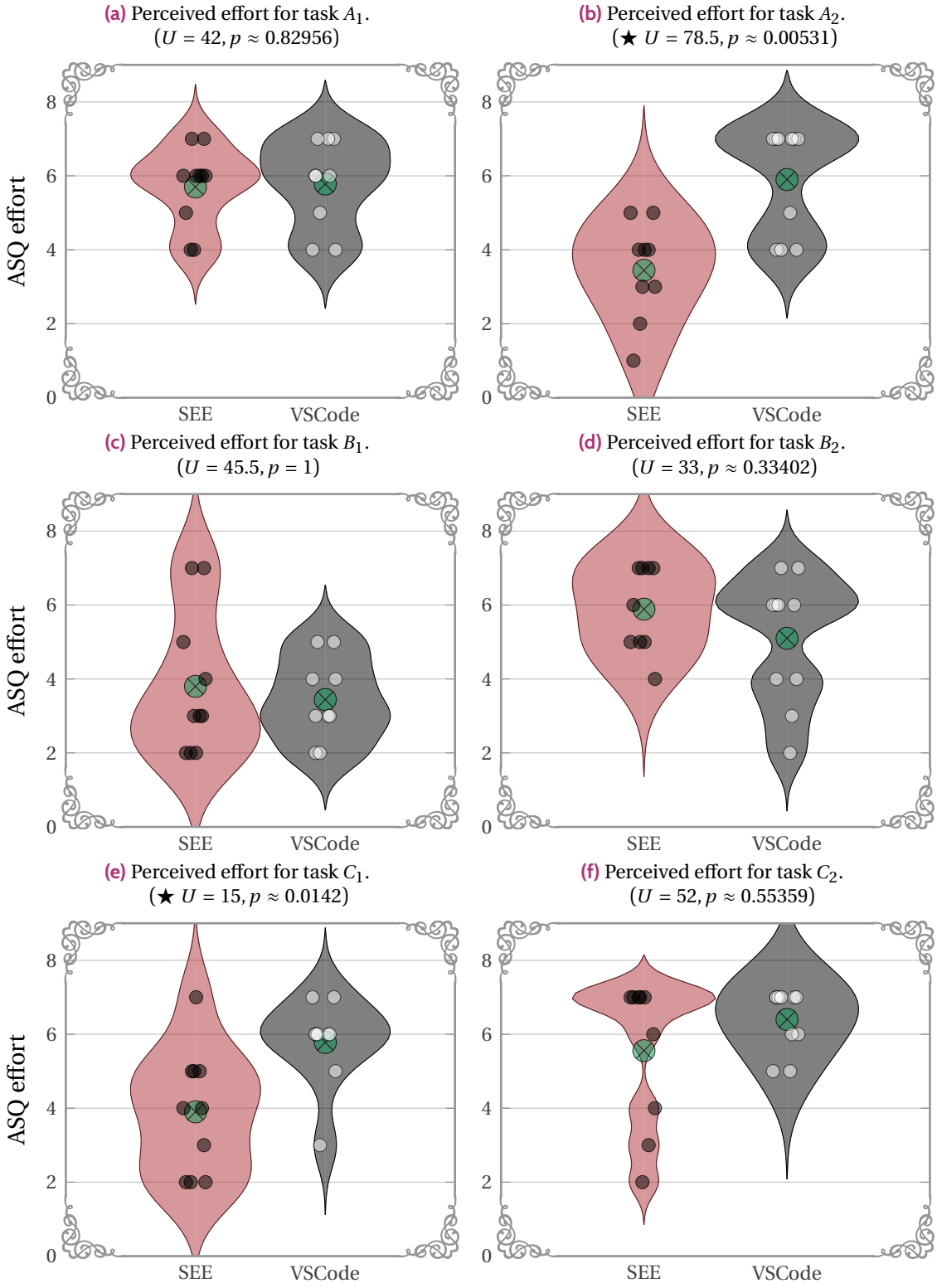


Figure 4.13: Perceived effort for each system as measured by the ASQ for each task. **A higher number means an easier perception of the task, i. e., *lower* effort.**

the significance test, the p-value of 0.0247 is extremely close to our significance level, so perhaps the difference is not as pronounced, although in that case I would have expected the ASQ effort for task A_2 to also have a noticeable difference. Maybe there is a difference between perceived versus actual time taken for the A task, in that it feels more strenuous or at least slower than the C task (e. g., maybe it is more interesting to follow the transitive hops of the edge around the city than to look for big buildings and enter them into a table).

Similar to Section 4.3.4, four of the six tests did not have significant differences, but two of them did in favor of VSCode, so we will reject H^{e_0} and accept H^{e_1} instead with $A^e_V > A^e_S$.

★ SUS Finally, the System Usability Scale measures the usability of a system as a whole with ten questions. To map a set of answers onto a score from 0–100, we apply the formula by **brooke1996**, that is, $2.5 \cdot (20 + \sum_{i=1}^{10} (-1)^{i+1} \cdot s_i)$, where s_i is the answer to the i-th SUS question on a Likert scale from 0–4. The alternation introduced by the factor $(-1)^{i+1}$ exists due to the fact that even-numbered questions are inverted phrasings of the odd-numbered questions. Figure 4.14 exhibits the difference in measured SUS scores between SEE and VSCode. We have 38 data points here because participants were asked to rate both systems.

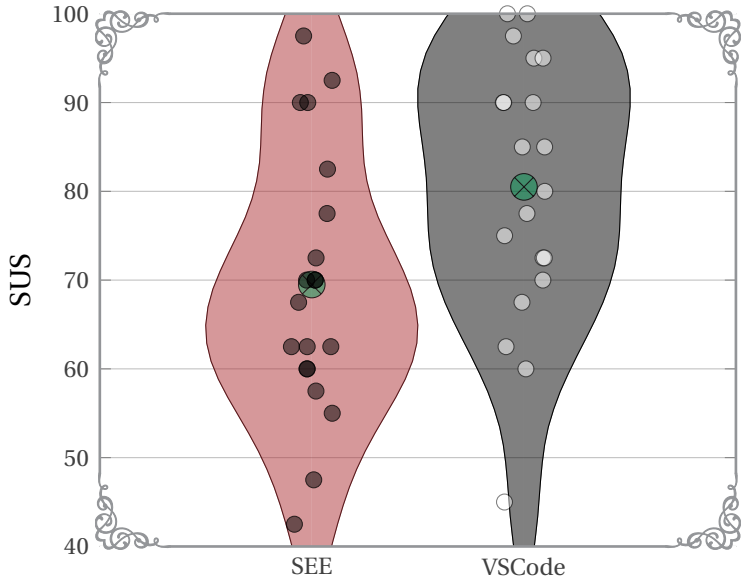


Figure 4.14: Comparisons between measured SUS scores for SEE and VSCode.

As one might guess from the star in the heading, the difference here is significant when we apply the Mann-Whitney U test, specifically in favor of VSCode ($U = 114.5, p \approx 0.02116$). The average SUS score for SEE is 69.5, with a median of 68.75, while the SUS score for VSCode was on average 80.5, with a median of 82.5. Our measured score for SEE falls in line with the average for SEE established from other experiments (see Fig. 4.1), and its difference to a traditional tool also replicates the results of the study for my bachelor's

thesis [**galperin2022**]. There may be multiple reasons why SEE does worse than VSCode here: As a research project, it is less polished than the professionally developed VSCode, it uses paradigms and controls that users may be less familiar with than those in a traditional IDE, or there may be specific grievances users had with SEE. We can at least cast light on the last of these explanations by examining the feedback participants have left about SEE, which we will do in Section 4.3.7.

In summary, we can reject H_{c_0} and accept the alternative hypothesis H_{c_1}, specifically, that $S_V > S_S$.

4.3.6 The Effects of Experience

As noted before, we now want to see if there is a correlation between any independent variables (especially ones related to experience) and any dependent variables (such as correctness). One of the reasons we do this is simple scientific interest, but another is to check whether any significant differences between groups Ψ and Ω that we uncovered in Section 4.3.2 actually have a non-negligible bearing on the results of the study.

There is one problem we need to handle first. Assuming that we create a null hypothesis of the form "independent variable X is not correlated to dependent variable Y for task Z_i," for each possible X, Y, Z, and i, we would create $9 \cdot (4 \cdot 3 \cdot 2 + 2) = 234$[50] null hypotheses and perform a corresponding number of statistical tests. At a significance level of $\alpha = 0.05$, we have a 5% chance of a false positive. Hence, for this many tests, we are virtually certain to hit at least one—in fact, we should expect around 12 (i. e., $234 \cdot 0.05$) of them. This is know as the multiple comparisons problem*. Since we do not really care that much about the correlation for each individual task, we can at least reduce this problem somewhat by aggregating along the two subject systems, that is, by averaging results for the first three tasks and then for the last three tasks, to get a single dependent variable per system instead of three. Still, this leaves us with 90 tests, where we can expect roughly 5 false positives.

There are a few ways to solve this, one of the most famous being the Bonferroni correction, which simply divides the significance level by the number of hypotheses [**miller1981**]. However, this method of controlling the Family-Wise Error Rate (FWER)** is known to be very conservative—with our moderate sample size of $n = 20$ we would prefer a method less likely to result in false negatives. Luckily, such a tool exists in the procedure by **benjamini1995** to

[50] The +2 is due to the SUS, which is asked twice.

*Multiple comparisons problem: A problem in statistics that happens when the same dataset is tested for significant results multiple times, thereby increasing the chance of a false positive beyond what is acceptable. [**tukey1953**]

FWER: The probability of getting at least one false positive in a family of statistical tests. [tukey1953**]

control the False Discovery Rate (FDR)*. Since our data points here might not be fully independent, we choose the variation by **benjamini2001** and fix the FDR at our usual α of 0.05. We designate the "original" p-value (i. e., the one before fixing the multiple comparisons problem) as $\not{p}$.

For the correlations themselves, we follow the recommendations by **khamis2008** and use Kendall's coefficient of rank correlation τ_b for comparisons with ordinal independent variables [**kendall1938**; **kendall1945**], and Spearman's rank correlation coefficient r[51] for comparisons with continuous independent variables [**spearman1904**] (since all our dependent variables, at least in aggregated form, are continuous).

Figure 4.15: Correlations between independent and dependent variables. Dependent variables for SEE are marked in red and those for VSCode in gray.

A matrix of correlations between the independent and dependent variables is given in Fig. 4.15. The size of the circles corresponds to the strength of the correlation, while the color indicates its direction in addition to strength. We can see that no particularly dark colors, and thus no particularly strong correlations are present. Indeed, the only correlations that would even be normally classified as "moderate" [**akoglu2018**] (these also turn out to be the only statistically significant results *before* applying the FDR correction) are the following two negative correlations:

[51] We could have also used Pearson's correlation coefficient r [**bravais1844**; **pearson1895**], but Spearman's r seems more fitting due to our dataset including some outliers.

*FDR: The expected proportion of false positives within all positives. [**benjamini1995**]

- **Experience with bigger software projects** and the **SUS for VSCode** correlate negatively with $\tau_b \approx -0.4225$ ($\not{p} \approx 0.0313$, $p \approx 1$).
  - This is a very surprising result to me. I would have expected more experience on bigger software projects to have led to more familiarity with tools like VSCode and hence to higher usability ratings. If we want to speculate, maybe very experienced developers have more specific "tastes", such as a heavily customized editor, leading to less enjoyment when they have to use the normal VSCode, but this does not really ring true, personally.
- **Experience with VSCode** and the **SUS for SEE** correlate negatively with $\tau_b \approx -0.4523$ ($\not{p} \approx 0.0247$, $p \approx 1$).
  - This result at least seems to make intuitive sense—developers used to VSCode may feel less productive when switching to a very new environment like SEE.

However, we cannot put too much stock in the speculation given here, since neither of the two FDR-adjusted p-values are anywhere near below our significance level. Thus, in total, we cannot make out any significant correlation between independent and dependent variables here, which has the upside that we need not worry as much about any of the differences in independent variables from Section 4.3.6 biasing our results.

4.3.7 Comments

Here, we shall go over the comments that participants have left in the free text fields. There has been one such field for feedback about SEE, one for feedback about VSCode, and one for general comments. I will group the comments together and rephrase them in a more general and direct way to eliminate duplicates, make this easier to read, and to preserve the anonymity of the participants. The number of times a type of comment has been left is indicated by a number in brackets after the description, if this number is greater than one. My responses to comments are indicated by the "☞" symbol, while simple agreement without any further elaboration is indicated by the "👍" symbol[52].

SEE Since this is where most of the comments were left, I will group these comments into three categories.

- **Feature/change requests**:

[52] A missing "👍" symbol does not necessarily indicate disagreement on my part, some comments (e. g., "VSCode was great to use") are just of a form where agreement does not make much sense as a response.

- A copy-paste feature for text in SEE (including in code windows) would have been useful. **[3×]** 👍

- It should be possible to sort all classes in the tree view independent of the tree structure. **[2×]**

  ☞ This use-case is already planned as issue #680.

- The Field of view (FOV)* should be increased.

  ☞ This is easy to implement, and should probably even be adjustable in-game instead of being a fixed value.

- After zooming in and zooming out again, it is hard to move the code city back to its original spot. The code city should ideally be fixed in place when zooming. 👍

- There should be a more obvious function to highlight the superclass instead of just the edges.

  ☞ This function already exists via the context menu, but was not included in the explanatory video.

- It would be useful to have the feature to directly navigate to the base class instead of having to go from one parent to the next.

  ☞ I agree when talking about SEE as it is normally used, but for this study, the intent was to have a non-trivial task that required moderate interaction with the code city environment.

- File paths should have been shown more prominently.

  ☞ There should at least be an option to include this information in the hover popup. Showing paths for every node in the city would overcrowd it a bit.

- The "show in city" effect should stop more quickly. 👍

- A key combination to trigger edges for a given node would be useful to quickly navigate using edges.

  ☞ As I write this, we are working on a new keyboard shortcut system that would accommodate this.

- A gradient may have been better on the nodes. The single color makes it harder to distinguish nodes whose NOM is similar. 👍

*FOV: The area of the world that can be seen by a camera or eye.

  - A less three-dimensional view may have been useful to navigate this city more easily.

    ☞ This may be true here, since we have not used all three dimensions for metrics, but at its core, SEE is a 3D code city visualization.

- **Bug reports**:

  - There were some smaller bugs that only became apparent after longer usage.

  - The code window could not be opened for one of the files in task *C*.

  - Using "show code" made SEE freeze.

    ☞ The above two problems are likely related, and this happened to me once too, but it also seems to occur non-deterministically, making it hard to debug.

  - SEE had to be restarted at one point because a node went missing.

    ☞ The most likely explanation here is that the node was accidentally moved. A "reset" button to completely undo any changes to a city's layout might be a good idea.

  - Under Linux, the search menu is flickering, making it much harder to use.

    ☞ I cannot replicate this, even though I am a Linux user—perhaps there are more specific sets of circumstances under which this bug occurs.

  - Clicks in the tree view are sometimes transferred to the code city behind it.

    ☞ We are aware of this problem and tracking it in issue #702.

- **Miscellaneous comments and criticism**:

  - Edges drawn between clusters of nodes that are close to one another are hard to visually distinguish. **[2×]**

    ☞ This is indeed a problem for denser clusters of nodes, but it is not immediately clear how to reliably solve this.

  - A first-person camera combined with mouse controls is a very unfamiliar combination.

  - Quickly estimating which node has the highest NOM works well, but it is harder to gain confidence over numeric exactness.

  - It is cumbersome to check the NOM using the properties window.

  - SEE is getting very visually appealing, and the transitive edge animations remind me of stones skipping across water, which is very cool.

VSCode

- It was great to use. **[2×]**
- An extension in VSCode with which metrics can be viewed directly in the IDE would have been helpful. **[2×]**

  ☞ A related work is the bachelor's thesis by **herrmann2024**, which explores how visualizations of such metrics can be integrated into VSCode via an extension.
- The feature set is huge.
- I was unsure if we were allowed to use the table for task *A*, which is why I slightly hesitated at first.
- The task seemed to only be solvable using the provided table.

  ☞ This was intentional—the table was added to make the comparison to SEE fair.
- I would have rather used the command-line on Linux to solve tasks more quickly.

  ☞ I agree that this would have been faster for an experienced developer, but the aim of the study was to compare *IDEs* with code cities.
- The symbols in VSCode were too small to reliably hit with the mouse.
- I prefer Emacs and Vim over VSCode. 👍

General Comments

- I found the question about the structure of the unit tests misleading.

  ☞ Unfortunately, there were no further details in this comment, so I am not sure which part was misleading. I have tried to give a detailed description of each type of unit test organization, but perhaps I should have added another tutorial question like I did for the base classes.
- Telemetry functions in VSCode were active—this should never be the case in a study.

  ☞ I fully agree and apologize for this oversight.
- Without the table, solving the tasks would have been very difficult in VSCode.
- The study was well done.

4.4 Threats to Validity

Before finishing this chapter, we want to take a look at some possible ways that the results from our study may be incorrect or misleading in some manner. We group these potential problems into two categories: *Internal* threats to validity, which are about whether our study really measures the causal relationship between the independent and dependent variables, and *external* threats to validity, which are about whether our results are generalizable. We use the relevant threats listed by **campbell1963** as a basis here.

4.4.1 Internal Validity

We are controlling for *selection* effects that would otherwise become an issue by randomly assigning participants to groups Ψ and Ω, both of which answer both task versions with opposite tools (see Fig. 4.5). This way, there should be no systematic differences (such as *History*, *Experimental Mortality*) between the two groups[53] that would bias the comparison between VSCode and SEE.

By keeping the study relatively short, the threat of *maturation* should also be adequately addressed. There were some outlier participants that needed long enough for maturation effects like tiredness to become a concern, but as discussed in Section 4.3.2, they were almost certainly related to breaks rather than an actually long-winded participation time. Additionally, the correctness results of the final JabRef tasks are proportionally very similar to the first SpotBugs tasks, which also suggests that participants did not get noticeably more tired as the study went on.

There may be some other interfering factors that caused type I or type II errors. For example, as several participants noted, a copy-paste feature in SEE (as it exists in VSCode) would have been helpful—while I do not personally think it would have mattered enough to cause any change in the results, it is possible that copy-pasting might have increased SEE's usability or decreased the time spent on tasks enough to cause a significant difference. Another factor we may need to be aware of is that some participants know me personally. Those participants could have (intentionally or unintentionally) tried to achieve a favorable result for me, such as SEE having a higher usability rating than VSCode. However, given that all significant differences point towards VSCode as the better tool, I am only left with the disturbing possibility that participants who know me personally tried to achieve a worse result for me, which is hopefully unlikely.

[53] Seemingly by coincidence, there still were some significant differences between Ψ and Ω, but as Section 4.3.6 revealed, this should not have significantly influenced any of the dependent variables.

4.4.2 External Validity

While we tried to make this study as realistic as we could (e. g., by following the checklist by **wettel2011**), there are still several reasons why it might not have been completely representative of real-world scenarios. For example, one concern is that our set of participants contains only one woman, and that a majority of participants are students who have helped develop SEE (see Section 4.3.2), which is not necessarily representative.

Another potential problem is that the tasks or the setup could have been too unusual to be meaningfully realistic. Two of the three tasks have been based on existing literature, and all seem reasonably typical to me in the context of trying to understand large software projects: Identifying the biggest components in the codebase is a good starting point for finding code smells; when adding new tests to a project, it is a good idea to first identifying the convention with which existing tests are organized; and when trying to understand a certain class, it is often necessary to navigate up its inheritance hierarchy to investigate parts specified in superordinate classes. However, in retrospect, some of the tasks (specifically *A* and *C*) may have been too easy, given that almost no participants made any mistake, as seen in Fig. 4.10. This makes analysis of this aspect a bit harder insofar that we may miss relevant answers to our research question RQ2 here (e. g., maybe there *are* significant differences between LSP-enabled code cities and IDEs, and our tasks were just too easy to catch them), but it should not be a big problem for the study as a whole, because time and usability were varied enough to cause meaningful results.

As for the setup, it consists of the subject systems (SEE and VSCode) and the object systems (JabRef and SpotBugs). JabRef and SpotBugs should be reasonably representative (see Section 4.2) but the combination of VSCode together with external tables might be somewhat unusual. It is probably more common to encounter a setup where such information is integrated into the IDE, for example, in the form of a code smell tool that tells the developer when components exceed certain metrics. Still, this should not be too big a problem, as on the one hand, the same setup was used in Wettel's study, and on the other hand, setups like this do exist in the real world (e. g., the Axivion Dashboard, used in a similar comparative study [**galperin2021**], is such a tabular external tool). However, even though we tried to reduce it (as discussed in Section 4.2), our results might be biased due to the Hawthorne effect or related effects. For example, I was not physically present for the tasks, but just knowing that they were being timed could have caused participants to perform worse than they would have without any perceived time pressure [**sussman2022a**].

4.5 Interim Conclusion

In this chapter, we evaluated our second research question RQ2 by conducting a user study. The study consisted of six program understanding tasks and compared the LSP-integrated version of SEE from Chapter 3 to the LSP-enabled IDE VSCode together with tables listing code metrics. Using the data of 20 participants, it turned out that some tasks took less perceived effort and were faster to solve when using VSCode instead of SEE, with no other significant results. Additionally, participants found VSCode significantly more usable than SEE. Table 4.2 summarizes these results.

Table 4.2: Significant differences between the variables, all in favor of VSCode.

Variable	A_1	B_1	C_1	A_2	B_2	C_2
Correctness	—	—	—	—	—	—
Time	—	—	$p \approx 0.0015$	—	—	$p \approx 0.0247$
ASQ: Complexity	—	—	—	—	—	—
ASQ: Effort	—	—	$p \approx 0.0142$	$p \approx 0.00531$	—	—
SUS				$p \approx 0.02116$		

Based on this data, we can answer RQ2 as follows: Code cities are a suitable means to present LSP information, but developers work faster and report both lower perceived effort and higher usability ratings when using traditional IDEs, at least in the case of SEE versus VSCode. It is unclear what the exact cause of this difference is. The SUS score of SEE is very close to average SUS scores [**sauro2013**], while VSCode reaches a very high score, so one possible reason is that VSCode is a much more polished and "tried-and-tested" product than SEE, which made it easier to use. Another possible cause is that software developers have much more day-to-day experience with traditional IDEs than with approaches like SEE, making VSCode more intuitive and thus quicker and easier to use. There are other possible reasons, such as our tasks possibly favoring VSCode's interface: For example, task *C* is extremely easy and quick to do in VSCode: After having found the initial file the task asks for, the participant merely needs to hold down Ctrl and repeatedly click the name of the parent class, which is always right near the mouse cursor. In SEE, the participant instead has to visually follow the edges into what may be a pretty dense cluster of nodes, zoom in, and move the city around until the target of the edge is identified.

Looking at the existing research, we cannot replicate the results of **wettel2011**; **romano2019**, even though these use very similar tasks. Both studies have as a result that code cities perform significantly better than IDEs under the factors of correctness and time, while we have the opposite result for time and no significant difference for correctness (though this latter part may have been because our tasks were too easy). While **wettel2011**'s study did not measure usability, **romano2019**'s did, and found no difference when comparing

the two tools, which is also different from our result where the IDE does better. These differences could come from the fact that more tasks were used (twelve for **wettel2011** and ten for **romano2019**), making their experiments more representative, but it could also have been due to the choice of Eclipse as the IDE, which in one study by **morales2019** received an average SUS of 60.9, lower than our average SEE SUS score of 69.5.

We can also make a comparison with the study for my bachelor's thesis, whose tasks *A* and *B* are very similar to our task *A* [**galperin2022**]. There, I compared SEE against the Axivion Dashboard, a web-based tabular code smell explorer, with the result being that using the Dashboard leads to higher correctness, but using SEE leads to higher speed and a better ASQ effort rating. While this also goes against our results, we can at least replicate the difference between the SUS, as the Dashboard (analogously to VSCode) received a significantly higher score than SEE.

In summary, our results here go against most of the existing literature covered in Section 4.1.1, especially when looking at Table 4.1. As said before, this may be due to the choice of IDE (the most-used IDE by other studies has been Eclipse, while we used VSCode), choice of code city implementation, or choice of tasks. Further investigation into the exact causes here—perhaps by rigorously comparing various IDEs, code city tools, tasks, and so on, against each other in further user studies—could be a good avenue for future research.

5

Conclusion

Ultimately, having now successfully completed both the implementation and the user study, we are ready to conclude this thesis. In this final chapter, we do so by first pointing out some limitations of the work presented here, mentioning some possibilities of future expansions, and finally, by offering some closing remarks.

5.1 Limitations

We will first go over a few ways in which the implementation and user study are limited.

5.1.1 Implementation

Possible Bugs For some Language Server, there sometimes seem to be slight inconsistencies between the generated code cities and the actual projects. For example, I noticed that the generated city in Fig. 3.11a contains some empty modules, even though the module's contents themselves exist and are grouped together, which seems to be a quirk of the Rust Analyzer. Similarly, while preparing SEE for the evaluation, I noticed that Java package hierarchies were not collected properly by the Language Server, so I had to add some workaround code that infers the structure from the file paths. To make the LSP import reliable and production-ready, it would be a good idea to do detailed tests for each Language Server and add more special-purpose fixes, if necessary.

Performance As we have seen in Section 3.5, generation times get less bearable as we increase the number of edges. The generation time for SpotBugs, under realistic settings (only reference edges and node types for classes and bigger components enabled), was on average at about nine minutes. This is not ideal, and generation is currently infeasible for

projects on the order of a million LOC. To make this truly usable for very large projects—which is where code cities also have arguably the biggest use—more investigation into how to increase performance further might be necessary.

City Editor So far, LSP code cities cannot be created within actual builds of SEE, because the editor UI for the graph provider is only available within the Unity Editor. SEE does have an in-game code city creation menu which uses the editor UI as a base, but the LSP import menu would still need to be configured to work with it correctly. Additionally, the implementation for this thesis added a progress bar to the graph provider UI that should also be incorporated into the in-game dialog.

Language Server Differences There are still a few suboptimal aspects for the code city generation that are due to differences between the various Language Servers. For example, not all of them support the hierarchic `DocumentSymbol` format that we need to actually construct the graph—since this is the modern format, though, Language Servers should over time be updated to output it instead of the outdated `SymbolInformation`. Another possible point of concern is that, due to those differences, code cities generated by different Language Servers are not really comparable to one another. An example of this is that the Jedi Python Language Server outputs different code cities than the Pyright Language Server[54], even though both are written for the same programming language. Finally, only very few Language Servers support the *pull diagnostics* capability. For all other ones, we need to collect the pushed diagnostics in the background, meaning we could easily miss some if the import algorithm is done before all diagnostics have been pushed.

5.1.2 User Study

We already went over some important threats to validity in Section 4.4. Apart from those, another major limitation may be that we have not actually evaluated most of the implementation: as explained in Section 4.1, we basically only evaluated the changes from Section 3.2, leaving out almost all features implemented in Sections 3.3 and 3.4. Even under those constraints, what we really evaluated in Chapter 4 was *generated* code city rather than the *generation of the* code city, although we did at least evaluate the latter in the technical evaluation in Section 3.5. Sections 3.3 and 3.4, however, remain very under-evaluated—a further user study could make sure that LSP-enabled code cities and code windows are a usable combination, maybe again by comparing those to VSCode.

[54] This may be due to a variety of factors, such as slight differences in the ranges of the code elements (which causes our matching algorithm to return different nodes for edges), or differences in how a given Language Server maps a node onto the (very TypeScript-oriented) LSP symbol types.

5.2 Future Work

I see many opportunities to build on this thesis and its implementation. Since there is no real unifying thread here, I will just list these ideas for future work individually:

- In this thesis, we constrained ourselves to "read-only" capabilities. Adding support for editing-related capabilities can have two different manifestations:
  - Editable code windows with the full LSP feature set would basically turn them into "proper" IDEs, and having a full-fledged IDE within SEE would reduce friction when working with code cities (as compared to having to switch over to, e. g., VSCode).
  - There are various ways to integrate editing capabilities into code cities. As an example, LSP offers so-called *Code Actions* (see Section 2.2.3), which often provide automatic fixes for code smells, so one could extend the erosion features to offer fixing all fixable code smells. Another example would be the ability to rename nodes in the code city, with those renames applied across the actual source code via the *Rename* capability.
- Additionally, we excluded two other sets of capabilities: Ones that are too niche, and ones that are too complex. The former ones may be useful in some contexts, but especially the latter set contains some very handy tools, such as the *Folding Range* capability, which allows the user to collapse certain ranges (e. g., every function, allowing them to see all function signatures in collapsed form).
- Currently, Language Servers need to be installed manually and must lie in the `PATH` for SEE to detect them. A possible future addition could be to implement something like *Mason* [**boman2025**] that contains a registry of available Language Servers and offers to install[55] them automatically so that they can subsequently be used in SEE.
- While the focus of this thesis was on SEE, Algorithms 1 and 2 are general enough to port them over to other code city implementations, or—even better—into a separate tool or library that generates a GXL file out of the input project.
- Instead of LSP, one could utilize LSIF to construct code cities fully remotely, that is, without having a local copy of the analyzed project. In the most advanced form, the user would just need to enter, for example, a GitHub link, and then SEE would utilize LSIF information in combination with GitHub's REST API to construct the city. If the user tries to open a code window, the corresponding file could be retrieved using the same API, or perhaps using git's *sparse checkout* feature.

[55] Where "install" does not have to refer to a system-wide installation, but could just involve placing the binary into a SEE-local folder.

5.3 Closing Remarks

We started this thesis by motivating the use of LSP, code cities, and the integration of the former into the latter in Chapter 1. There, we also pinned down our main two research questions, which are about:

1. whether using LSP to generate code cities is a feasible approach (RQ1), and
2. whether code cities are a suitable means to present LSP information as compared to IDEs + tables (RQ2).

After introducing these concepts in more detail in Chapter 2, we answered the first research question in Chapter 3: the approach is feasible, with the caveat that our implementation does not perform well on larger projects (roughly above 100 000 LOC). We continued with a controlled experiment in the form of a user study in Chapter 4, which gave us an answer to our second research question: code cities seem like a suitable representation for LSP information, but developers work faster and report higher usability when using the traditional IDE VSCode.

What has been implemented in this thesis seems promising. It is now possible to generate code cities without needing to setup and configure a separate tool to generate the GXL files first, instead using Language Servers that are easy to install and often already installed due to their use in other IDEs. Additionally, many more languages are now supported by SEE than before. However, as discussed in the current Chapter 5 and as apparent from the aforementioned results, there are still limitations and further possibilities to extend this work in the future, be it performance improvements to the generation algorithm, support for more capabilities, or further user studies.

A

List of TODOs

Any open tasks/notes/mistakes for this master's thesis are collected within this appendix. If you see any mistake or empty section not covered by such a note, please tell me. Note that this appendix will only appear in draft versions.

B

Additional Information

This appendix collects a variety of additional information that I did not want to put within the main text of the thesis, in the fear that it breaks up the reading flow too much. Hence, there are various disparate sections in here that bear no direct relation to one another.

B.1 All SUS Questions

Here are all ten questions of the System Usability Scale:

1. “I think that I would like to use the system frequently.”
2. “I found the system unnecessarily complex.”
3. “I thought the system was easy to use.”
4. “I think that I would need the support of a technical person to be able to use the system.”
5. “I found the various functions in the system were well integrated.”
6. “I thought there was too much inconsistency in the system.”
7. “I would imagine that most people would learn to use the system very quickly.”
8. “I found the system very cumbersome to use.”
9. “I felt very confident using the system.”
10. “I needed to learn a lot of things before I could get going with the system.”

B.2 Technical Evaluation Data & Scripts

The data measured by the technical evaluation is provided in the ZIP file *benchmark.zip* as CSV files, one for each run. The filename for each of these follows the scheme `perf-<project>-<type><number>.csv`, where `<project>` is one of:

- `aao` for `aaoffline`,
- `bachelor` for my bachelor's thesis,
- `dcaf` for `dcaf-rs`,
- `jab` for JabRef,
- `master` for this master's thesis,
- or `spot` for SpotBugs.

On the other hand, `<type>` is either `norm` for Algorithm 2 or `x` for the brute-force Algorithm 1, and `<number>` is just an ascending number assigned to each benchmark run. The code which has been used to generate these CSV files is available on branch `performance-changes-falko` on the SEE repository.

In addition to the CSV files, the ZIP archive contains a `benchmark.py` Python script, which reads the provided CSV files and generates the DAT files used by LaTeX for the plots in Section 3.5. For convenience, the generated DAT files have also been included in the archive.

B.3 Generation Time Without Edges

The breakdown of the generation algorithm components for the sample projects in the technical evaluation in Section 3.5 made it hard to tell the distribution of other components, since the edge components took up so much space. For this reason, we present another version of the diagram in Fig. B.1 with the edge component removed.

B.4 User Study Data & Scripts

Here, we give all data that is necessary[56] to replicate the study (either as a new study, or by re-analyzing the data).

[56] With the exception of the full result data, which I have redacted for privacy reasons.

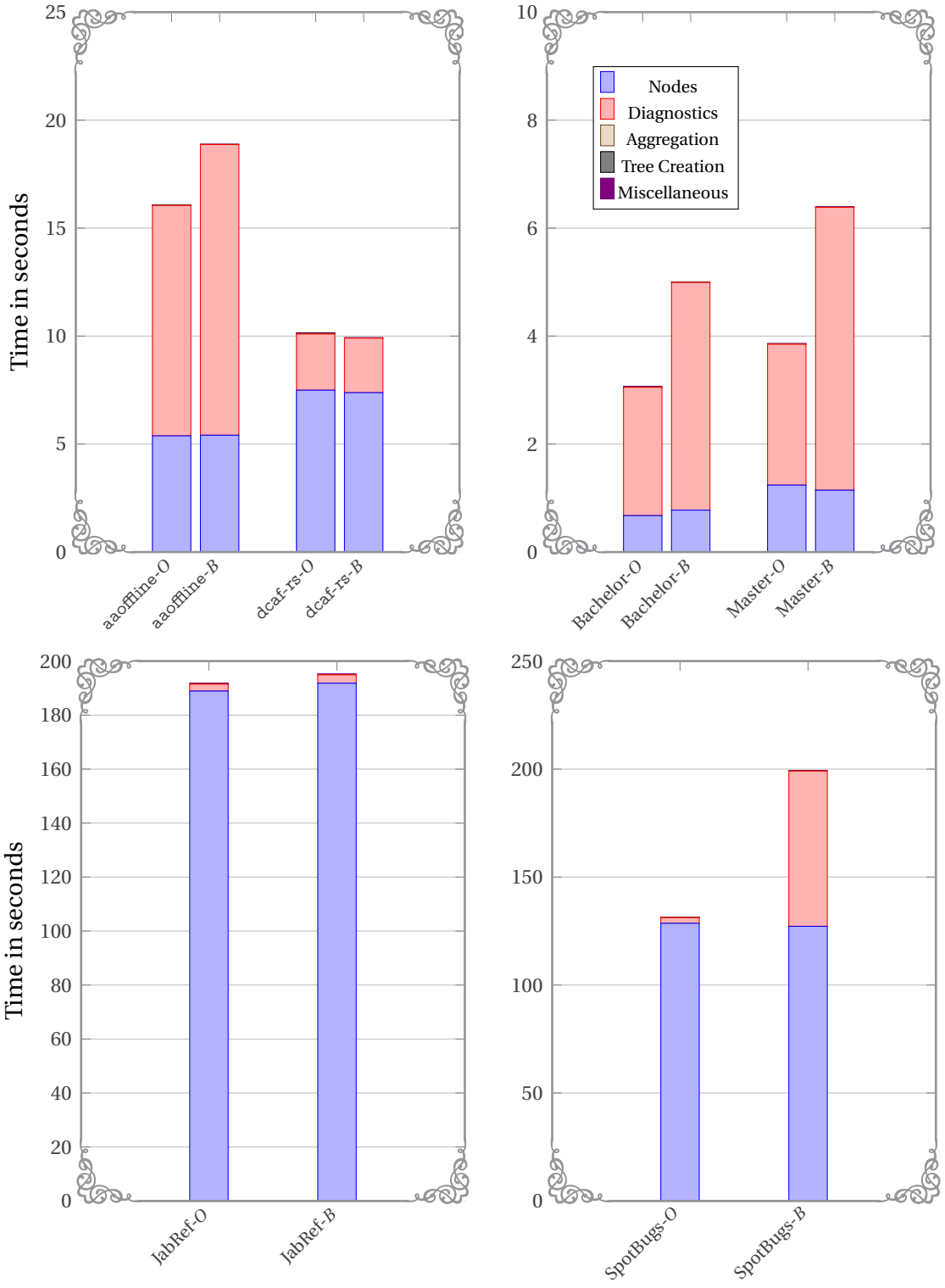


Figure B.1: Generation time for each project, broken down by parts of the algorithm, excluding the edge generation component. The suffix *O* denotes the optimized (interval tree) version of the algorithm, while *B* refers to the brute-force version.

B.4.1 Participation Data

The user data, in pseudonymized form (i. e., with all potentially identifying information scrubbed and replaced with the string `[REDACTED]`), is attached as *SEE-VSCode_redacted.csv* for group Ψ and *VSCode-SEE_redacted.csv* for group Ω.

B.4.2 Analysis Scripts

The analysis scripts are attached as *analysis.zip*, with the following points to bear in mind:

- `analyze.py` acts as the entry-point of the script—this is the only script that actually needs to be run.
- The script expects a file `SEE-VSCode.csv` and `VSCode-SEE.csv` to be in the same directory as itself. This script should be the direct result of downloading the participation data from KoboToolbox. You could use the redacted versions mentioned above, but will likely run into errors since we expect the redacted fields to be filled out properly, so you would either have to manually modify the code to not analyze those potentially identifying parts of the questionnaire, or add in some dummy data into the CSVs.
- The ZIP file also contains two files by the name of `correctness_values_sv.json` and `correctness_values_vs.json`. These are my "cached" answers to which participant responses were correct or incorrect. If you delete them, the script will prompt you for the information instead.
- The script creates a bunch of DAT files under the `dat/` directory, which is what LaTeX uses to generate the plots for Chapter 4.

B.4.3 Full Questionnaire

Finally, the full questionnaire in XLSForm standard that can be directly imported into KoboToolbox is given at *Eval-See-Vscode.xlsx* for group Ψ and at *Eval-Vscode-See.xlsx* for group Ω.

B.4.4 Answer Key

Here are the accepted answers to each of the six tasks:

***A*$_1$** 1. `AbstractFrameModelingVisitor` (198 methods)

2. `MainFrame` (165 methods)

3. `BugInstance` *or* `TypeFrameModelingVisitor` (126 methods each)

***B*$_1$** We accept *either* of:

- "Centralized" with `spotbugs-test` as the root[57]
- "Dispersed" with `TestDataflowAnalysis` (which may reasonably be construed as a class that tests something) as an example
- "None"

***C*$_1$** `BetterVisitor` *or* `Visitor`[58]

***A*$_2$** 1. `BibTexParserTest` (143 methods)

2. `JabRefPreferences` (136 methods)

3. `AuthorListTest` (127 methods)

***B*$_2$** "Centralized" with root `src.test.java.org.jabref` *or* `src.test` *or* `test`

***C*$_2$** `AbstractViewModel`

[57] This would only be visible on VSCode if participants incorrectly opened the project, which is why this is not the canonical answer.

[58] We accept either because the former is the base class, and the latter is the base interface, which is close enough.

C

Glossary

base class The base class of a class is its superordinate (i. e., above in the inheritance tree) class that itself has no further parent class within the project. 53, 69, 71, 80, 87, 90

capability A specific set of features a given Language Server (and Language Client) support. 5, 7, 11, 16–26, 40, 41, 43, 55, 56, 60, 68, 106, 115

code city In the code city metaphor, software components are visualized as buildings within a city, where various metrics of the software are represented visually (e. g., the height of a building could represent the lines of code of the component). 2, 3, 5, 7, 9–12, 14, 15, 18–21, 23, 25, 27, 30, 31, 34, 38, 40–42, 44, 45, 47, 51–53, 55, 56, 58, 63, 68, 69, 78, 87–89, 91–93, 113–116

code smell certain structures in source code that suggest that a refactoring is in order, such as duplicated code, or a very long method [**fowler2019**]. 11, 16, 20, 91–93, 113

code window A source code viewer in SEE which supports very basic IDE-like functionality. 7, 9, 14, 15, 19, 20, 23, 25, 26, 40, 41, 43–47, 53, 56, 87, 88, 114

Fisher's exact test A test for statistical significance for contingency tables, intended as an alternative to the χ^2 test to be accurate even with smaller sample sizes. [**fisher1922**] 76

graph provider A component in SEE that produces or transforms a project graph by some means. Can optionally take configuration parameters by users. 15, 38, 39, 114

Hawthorne effect The effect through which participants in a study behave differently when under observation by a researcher. The name is based on experiments conducted at the Hawthorne Plant in the 1930's [**hawthorne1939**], although the effects seen there have later turned out to be likely unrelated to the observation [**jones1992**]. 62, 92

interval tree A data structure meant to store intervals/ranges in such a way that overlapping or contained intervals can be found efficiently. 27, 35–37, 43, 47–49, 101, 114–116

***k*-d tree** A data structure (using a binary tree as a basis) with which certain spatial data in k dimensions can be efficiently stored and retrieved. 35, 37, 114

Language Client A development tool, such as an IDE, that supports LSP and can hence integrate language-specific features into itself using compatible Language Servers. 5, 9, 11, 17, 18, 23, 105

Language Server A locally-running JSON-RPC-based application following the Language Server Protocol that provides language-specific features and aids to the Language Client. 4, 5, 7–9, 11, 17–21, 27, 29–31, 40, 41, 43, 44, 48, 51, 52, 56, 59, 95, 97, 105, 106

Likert scale A psychometric scale in which users indicate their agreement on a linear scale from "strongly agree" to "strongly disagree" [**likert1932**]. 65, 66, 80, 82

Mann-Whitney U test A test to check whether two distributions have identical distributions, requiring only ordinal-scaled independent samples drawn from the distributions. [**mann1947**] 69, 71–73, 76, 78, 80, 82

multiple comparisons problem A problem in statistics that happens when the same dataset is tested for significant results multiple times, thereby increasing the chance of a false positive beyond what is acceptable. [**tukey1953**] 85

polytree A directed acylic graph which also has no cycles in undirected form. 13, 14

post-study A questionnaire for participants which is answered at the end of the study, after every task has been completed. 61, 64, 65, 110

post-task A questionnaire for participants which is answered after each task. 61, 64

reflexion analysis The process of comparing the architecture and implementation of a software project and finding incongruencies between the two. 12

semantic token An LSP capability that returns tokens for a document, with the intention that those tokens can be used to apply syntax highlighting to the file. Apart from token types, it also offers token *modifiers*, which can be used to apply additional formatting on top of the colors from the types. 21, 24, 56

singleton a class for which only one instance exists for the duration of the process. 41

Source Range the contiguous portion within a source code file that a certain element occupies. In this thesis, we will express it as an ordered pair of a start and end position (s, e), or, alternatively, as a 4-tuple (s_l, s_c, e_l, e_c). In the latter representation, the first two elements describe the zero-indexed start line and start character offset, respectively, and the last two describe the corresponding (exclusive) end line and end character offset. 14, 18, 19, 21–24, 27, 35, 36, 38, 45, 53, 116

Unity Editor The main UI of the Unity game engine, in which scenes can be set up, components can be configured, the game itself can be run, etc. Note that it is only used for development purposes, and hence not included within generated builds of a game. 15, 30, 38, 39, 114

violin plot A plot visualizing the distribution of a collection of data points along with an estimated probability density [**hintze1998**]. The black/white data points are

randomly "jittered" along the x-axis to make them more differentiable from one another. A bigger, green point marks the average of the dataset. 55, 57, 76, 78, 80

XLSForm A form standard based on human-readable Excel spreadsheets, which allows for complex forms (e. g., conditionals, skip logic) to be built [**marder2024**]. The forms can then be converted into the open ODK XForm format, which is compatible with a number of data collection tools [**odkcommunity2019**]. 66, 102, 111

D

Acronyms

After-Scenario Questionnaire (ASQ) A three-item questionnaire asking participants for satisfaction with ease, completion time, and support information [**lewis1991**]. 61, 64, 65, 70, 80–83, 92, 93, 114, 115

Debug Adapter Protocol (DAP) A protocol that can be viewed as the analogue to LSP for debuggers, with the goal to make it easier to integrate debuggers into development tools. 23

Extended Backus–Naur Form (EBNF) A syntax in which context-free grammars can be formally expressed. 26

False Discovery Rate (FDR) The expected proportion of false positives within all positives. [**benjamini1995**] 85, 86

Family-Wise Error Rate (FWER) The probability of getting at least one false positive in a family of statistical tests. [**tukey1953**] 85

Field of view (FOV) The area of the world that can be seen by a camera or eye. 87

Graph eXchange Language (GXL) A file format for graphs, used in SEE for representing dependency and hierarchy graphs of software projects. 5, 13, 15, 40, 111

Integrated Development Environment (IDE) Editor for source code with features that are useful for development (e. g., highlighting errors). Examples are VSCode or IntelliJ. 2, 4–7, 9–11, 15–17, 19, 21, 24, 43, 45, 47, 52, 55–58, 63, 64, 68, 75, 78, 84, 89, 91–93, 105, 106, 110, 113, 115

JavaScript Object Notation—Remote Procedure Call (JSON-RPC) A remote procedure call protocol that uses JSON as its encoding, supporting (among other features) asynchronous calls and notifications. It is used as the base for LSP (even though LSP is technically not a remote protocol). 9, 17, 18, 26, 40, 44

Language Server Index Format (LSIF) A format which language servers can emit to persist LSP-based information about a software project. 23

Language Server Protocol (LSP) A protocol which specifies how language servers can provide language-specific features to IDEs, such as hover information, go to definition, or diagnostics. 4–18, 23–27, 29–31, 34–36, 38–46, 51–53, 55, 56, 60, 68, 91, 92, 106, 109, 113–116

Lines of Code (LOC) The number of lines in a source code file. 3, 29, 47, 48, 52, 63

Number of Methods (NOM) The number of methods in a certain class. 68, 75, 78, 88, 89

Post-Study System Usability Questionnaire (PSSUQ) A questionnaire that measures usability in sixteen questions across the three factors *usefulness*, *information quality*, and *interface quality*. [**lewis1992**; **lewis2002**] 65

Questionnaire for User Interaction Satisfaction (QUIS) A post-study questionnaire measuring usability across 41 questions in its short version and 122 questions in its long version. [**chin1988**] 65

Software Engineering Experience (SEE) An interactive software visualization tool using the code city metaphor in 3D, developed in the Unity game engine at the University of Bremen. 2–5, 7–16, 18, 19, 21–27, 30, 34, 38, 41, 42, 44, 53, 55–62, 64–66, 69, 71, 73–76, 78–80, 82, 84, 86–93, 100, 105, 109, 113–115

Software Usability Measurement Inventory (SUMI) A 50-item questionnaire measuring usability on the axes of efficiency, affect, help and support, steerability, and learnability. [**kirakowski1994**] 65

Subjective Mental Effort Question (SMEQ) A single question intended to estimate usability which uses a scale with 150 options. [**zijlstra1985a**] 65

System Usability Scale (SUS) A simple questionnaire by **brooke1996** with ten Likert-scale questions that are supposed to measure the usability of a system. 57, 61, 64–66, 80, 82, 84–86, 92, 93, 114, 115

Task Load Index (TLX) A post-task questionnaire developed by NASA consisting of six questions asking about mental, physical, and temporal demand, as well as about performance, effort, and frustration level. [**hart1988**] 64

Uniform Resource Identifier (URI) A string that uniquely identifies some resource. 17

Usability Magnitude Estimation (UME) A post-task questionnaire which does not depend on any pre-defined scale. Instead, numbers given by each user are put in relation to each other, with the resulting ratios serving as the usability measure. [**mcgee2003**] 64

User Interface (UI) 12, 15, 18, 30, 38, 39, 59, 106, 114

Visual Studio Code (VSCode) A proprietary, but free IDE developed by Microsoft with a plugin system from which LSP originated. See `https://code.visualstudio.co`

E

Attached Files

Notation	Description
`benchmark.zip`	A ZIP archive containing both the script and all measured data of the technical evaluation. Refer to Appendix B.2 for more information.
`extract-metrics.py`	A Python script which extracts metrics from a GXL file and outputs them into a CSV file.
`SEE-VSCode_redacted.csv`	The pseudonymized participation data for group Ψ, collected from KoboToolbox during the user study.
`VSCode-SEE_redacted.csv`	The pseudonymized participation data for group Ω, collected from KoboToolbox during the user study.
`Eval-See-Vscode.xlsx`	The XLSForm questionnaire for the Ψ version of the user study.
`Eval-Vscode-See.xlsx`	The XLSForm questionnaire for the Ω version of the user study.
`analysis.zip`	A ZIP archive containing all scripts for the analysis of the user study. Refer to Appendix B.4 for more information.
`redirect-evaluation.py`	A Python script which acts as a web server that redirects users to either the Ψ or Ω version of the study, as described by Algorithm 3.

TODO: Make sure all digital appendix items are actually attached (and that all are present in the table)!

F

Lists

HERE we have a collection of all lists of things from this thesis. Specifically, we are enumerating all figures, all tables, all code listings, and finally, all algorithms. Each of these is numbered by chapter and ordered by occurrence.

F.1 List of Figures

F.2 List of Tables

F.3 List of Listings

F.4 List of Algorithms