University of Bremen

Universität Bremen

Faculty 3 — Mathematics & Computer Science

TODO: Make sure this is centered!

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*

December 7, 2024

Abstract

TODO!

Declaration

I hereby declare that I have completed this master's thesis independently and without any external assistance, unless explicitly stated otherwise. I have not used any sources or aids other than those specified. All passages that have been quoted verbatim or paraphrased from published sources are clearly identified as such.

Bremen, December 7, 2024

Falko Galperin

Acknowledgements

TODO!

Contents

TODO: Listings appear twice in TOC

1

Introduction

TODO: Convert diagrams to TikZ

Linking the Language Server Protocol with code cities, specifically SEE, is the core focus of this master's thesis. 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.

TODO: Maybe rewrite. Could start by going over languages in general, giving spell check as example.

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 B and C—all mentions of such terms also link (in the digital version of this thesis, at least) to the corresponding part of the glossary. 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 D).

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* (IDE). One such software visualization—called Software Engineering Experience (SEE)—is being developed at the University of Bremen and will be introduced in the next section.

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.

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,

[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 *Eclipse* or *JetBrains IntelliJ*.

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

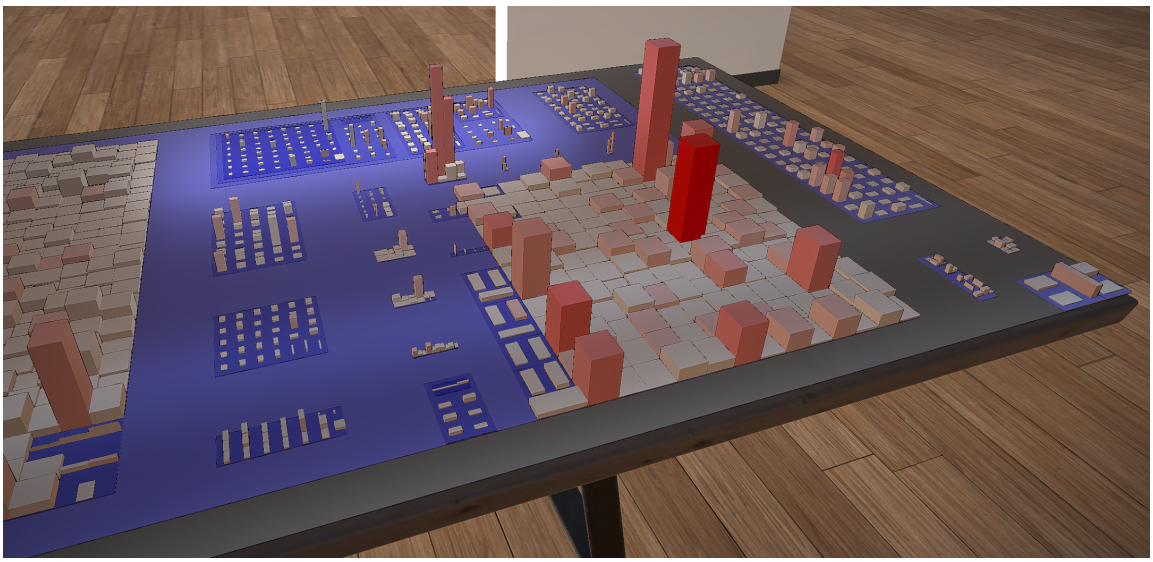


Figure 1.1: A code city visualized in SEE.

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.

For example, in fig. 1.1, we can see the source code of the SpotBugs project[2] 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. fig. 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.

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.

[2] `https://github.com/spotbugs/spotbugs` *(last access: 2024-09-11)*

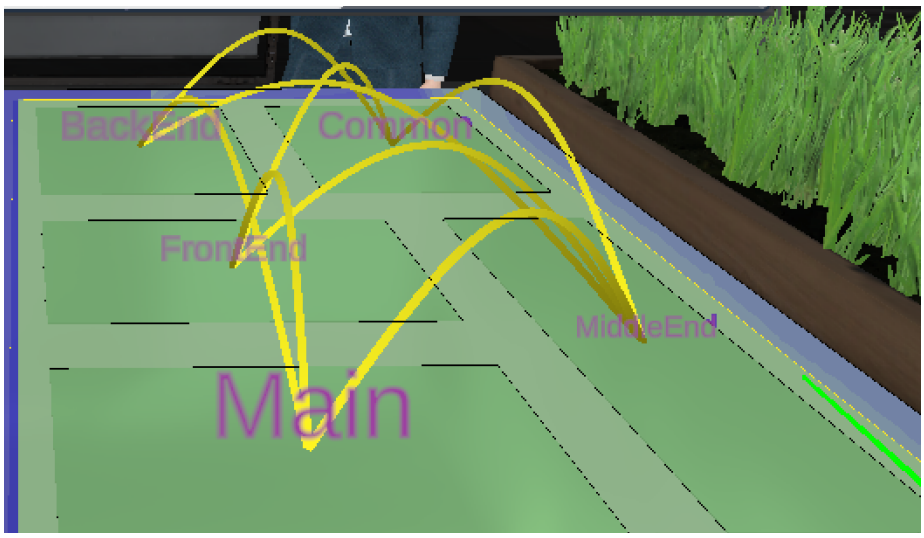


Figure 1.2: An example of what edges can look like in SEE. **TODO:** Use higher quality image.

> 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. ([LspSpec])

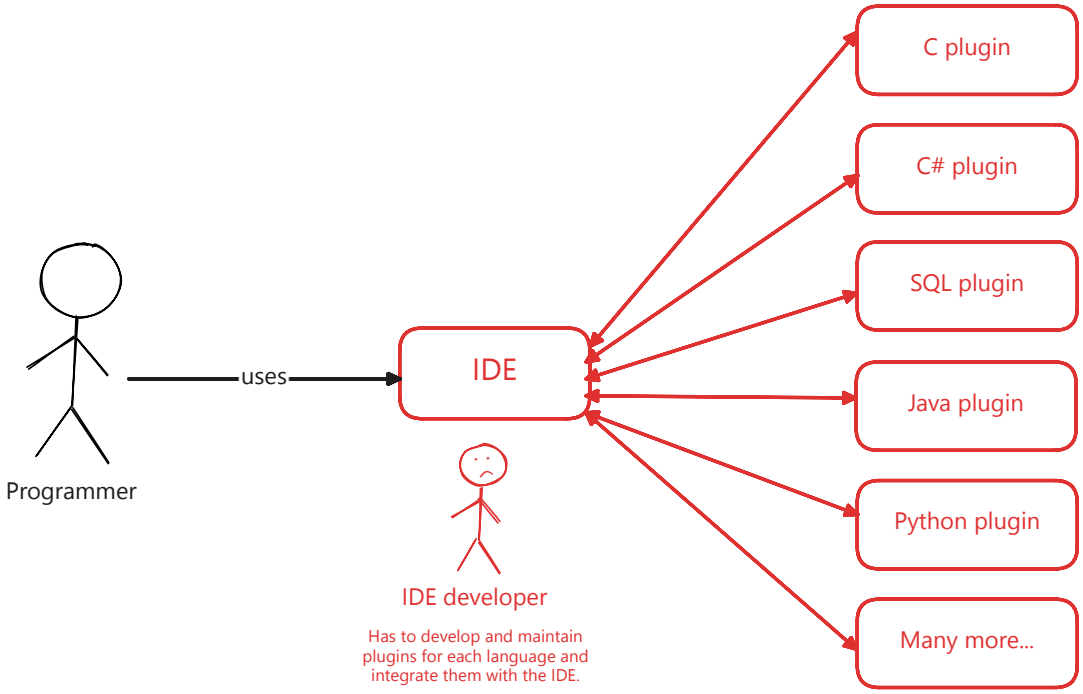
Since LSP[3] is a central component of my master's thesis, I have created a diagram in fig. 1.3 in the hope to strengthen intuitions around the motivation and use of the protocol. While the LSP specification has originally been created by Microsoft, it is by now an open-source project[4], where changes 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 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

[3] 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 [LspSpec] does it as well.
[4] Available at `https://github.com/Microsoft/language-server-protocol` *(last access: 2024-09-11)*.

*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.

(a) IDE development without LSP.



(b) IDE development with LSP.

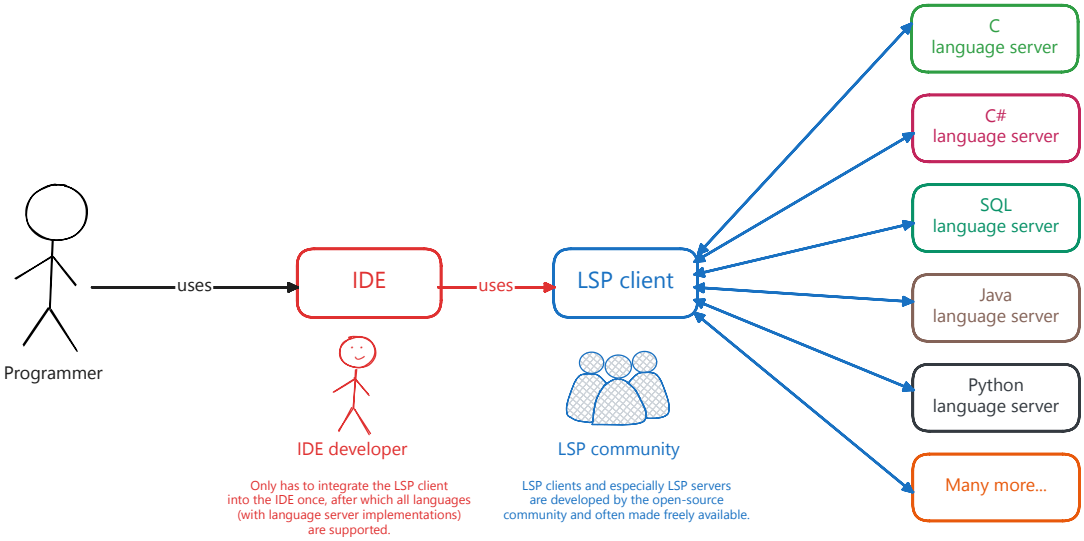


Figure 1.3: An illustration of how LSP can help simplify IDE development.

features, such as autocomplete. To give a specific example of what an LSP capability might look like in practice, the `texlab`[5] 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.

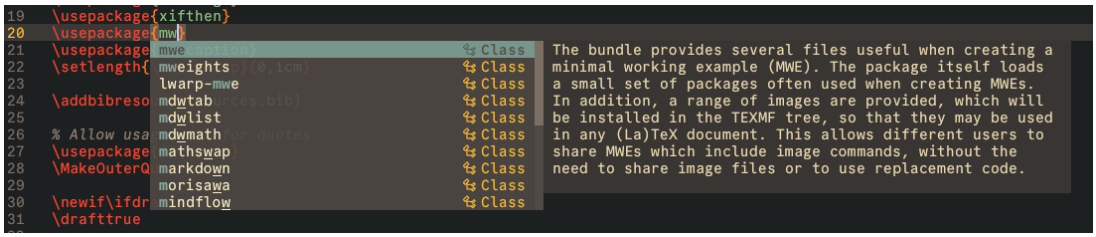


Figure 1.4: An example of the `texlab` Language Server running in Neovim.

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*, JetBrains IntelliJ (Neo)vim, and *Visual Studio Code*.

1.2.3 Integration

Currently, code cities in SEE are rendered by reading in 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 for a given component by hovering over it is not supported. This is where the Language Server Protocol can help.

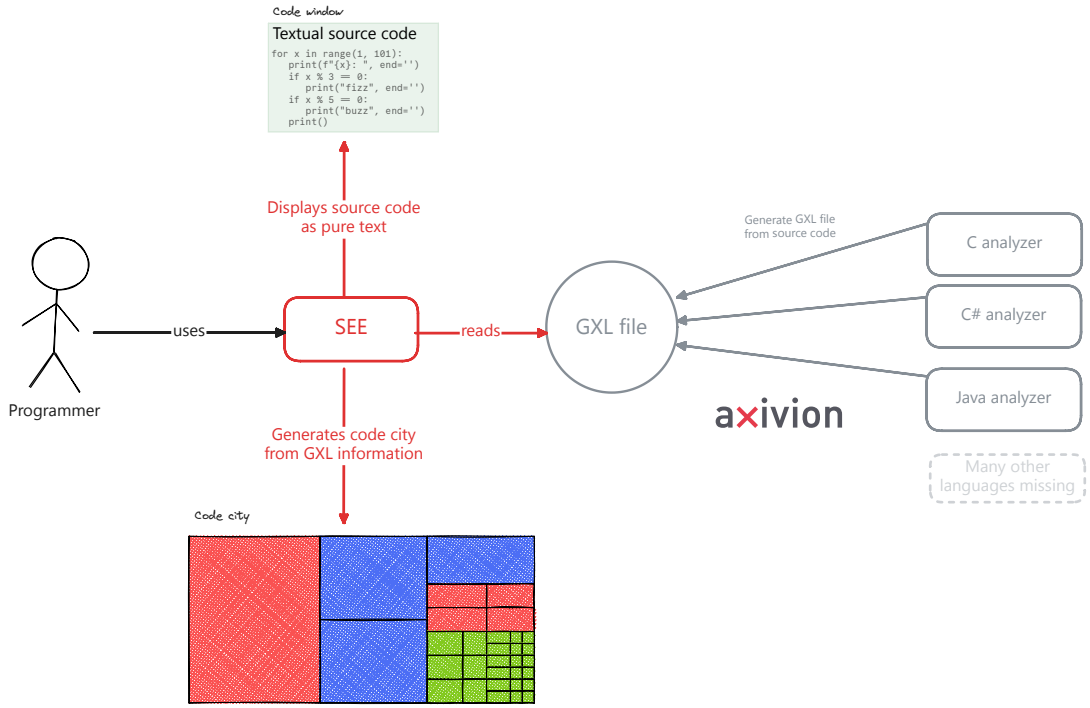
1.3 Goals & Research Questions

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

[5] `https://github.com/latex-lsp/texlab` *(last access: 2024-09-12)*

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

(a) SEE as it exists right now. The analyzers on the right are developed by Axivion.



(b) SEE with implemented LSP integration. The Language Servers on the right are mostly developed by the open-source community.

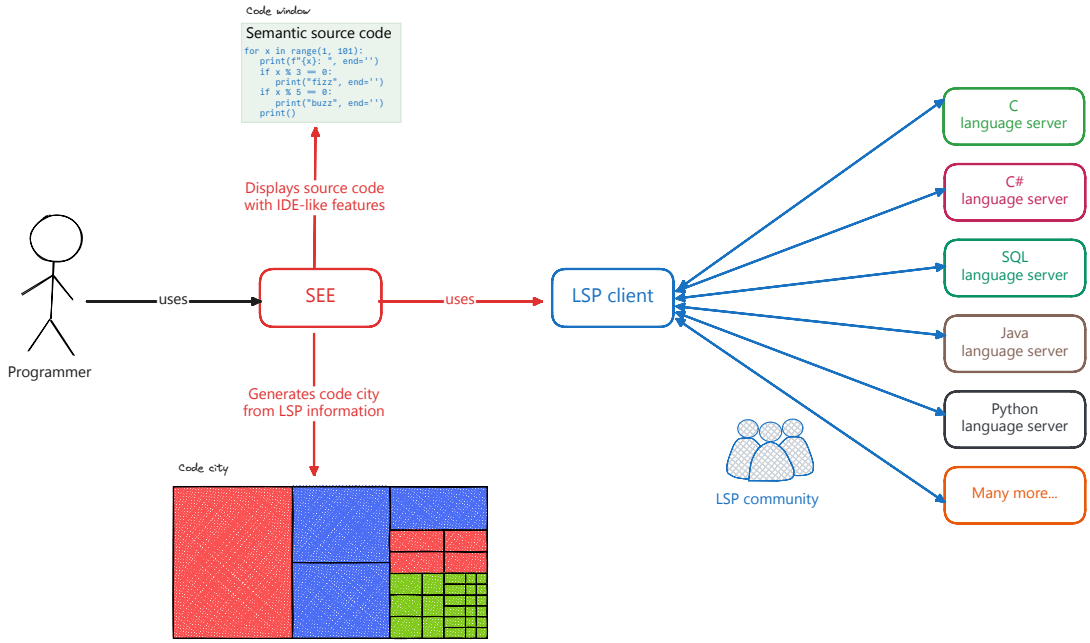


Figure 1.5: A simplified illustration of how SEE currently gains and uses information about software projects, and how the integration of LSP would change that.

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[6] 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.

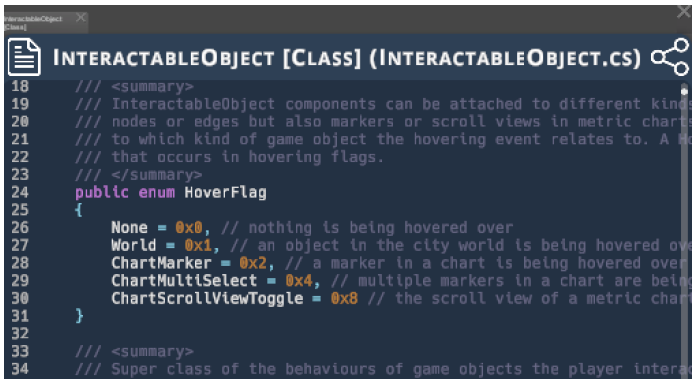


Figure 1.6: A code window.

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.

TODO: Maybe remove the next part if already mentioned before.
TODO: In general, reword some stuff from "planning to do" ⇒ "done"

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 [see also Ble22], 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 considered when planning which features to implement as part of the thesis. Consult section 2.2.2 for a full list of LSP capabilities which I plan to implement.

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-RPC)). There already exist well-made interfaces for this purpose[7], 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.

[6] 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.

[7] Such as OmniSharp's implementation of LSP in C#, which I plan to use: `https://github.com/OmniSharp/csharp-language-server-protocol` *(last access: 2024-28-01)*

*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).

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)?

TODO: This is rather vague. Is that alright or do I need to operationalize here?

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 IDE (specifically, Visual Studio Code* (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.

*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)*

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 some of the existing literature regarding code cities (as this is more of an academic topic than LSP) and describe the essentials that we need to work with in the implementation, such as 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:

TODO: Back this up with sources

- 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 [Gal21]

*Code Smell: Certain structures in source code that suggest that a refactoring is in order, such as duplicated code, or a very long method [FB19, pp. 85–87].

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

SEE is an open-source project, currently hosted at GitHub[8]. It is a research project at the University of Bremen, where it has been in development since 2019, 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.

TODO: Refer back here later, noting how LSP can generate code cities for each of these use cases

TODO: is this correct? And put source here

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 features that are relevant 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). Metrics of the project, such as the McCabe complexity [McC76], can then be mapped onto visual properties of this rendering, such as its color, width, or height.

TODO: Needs a 3D SEE screenshot, esp. for edges

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, as is possible in many video games, 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, as some recent studies have shown. 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.

TODO: add 2D screenshot

TODO: Source for 3D/game/VR being easier to navigate/understand

I heavily recommend readers take a look at this introductory video, as an interactive game-

TODO: link the video

[8] 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.

like project like this comes across much more clearly in a video than in mere texts and disconnected snapshots.

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 relationship it represents. 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.

TODO: A lot of "represents" here

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 undirected cycles.

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.

TODO: Give examples of software projects here to illustrate graph usage?

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).

Context Menu When right-clicking any node or edge, a menu opens with several context-dependent options. 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.

TODO: Show screenshot of context menu (pre-LSP)

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

TODO: Provide screenshot of editor

*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.

input parameters (such as the aforementioned GXL file). 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.

Erosion icons Following my bachelor's thesis [Gal21], 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. 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 [GKS22]. 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.

TODO: Show screenshot of erosion icons

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[9] (though there are still VSCode-specific extensions to LSP that are not in the official specification today). LSP has found widespread use: An overview page by Microsoft lists at least 269 Language Servers[10] (i. e., servers offering support for some programming language) and 61 Language Clients[11] (i. e., IDEs or development tools). The current (as of December 7, 2024) version of the protocol is 3.17, with version 3.18 being under active development.

[9] Available at `https://github.com/microsoft/language-server-protocol` *(last access: 2024-10-05)*.
[10] `https://microsoft.github.io/language-server-protocol/implementors/servers/` *(last access: 2024-10-05)*
[11] `https://microsoft.github.io/language-server-protocol/implementors/tools/` *(last access: 2024-10-05)*

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 server that does not warrant a response [JSO13; Cro06]. 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[12]).

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[13] are represented as a Uniform Resource Identifier (URI) representing the document along with a range (which we have formalized in section 2.1.2). The corresponding method name for this example is `textDocument/hover`.

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 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

[12] `https://nodejs.org` *(last access: 2024-10-05)*
[13] These documents must always be textual—there is no support for binary files.

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 [LspSpec].

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 provides 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 (see section 3.2) .

TODO: More precise reference

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 [Gal21], 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 been added rather recently . 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.

TODO: Inconsistent use of should/will in this section
TODO: When?

- **Hover**: Returns hover information for a given location. The specification does not specify what exactly this "hover text" should be [LspSpec], but most implementations of Language Servers display the documentation of the hovered element, or the signature if it's 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 away.
- **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[14]-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" ([LspSpec]) 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 instead 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 above.

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

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

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 [LsifSpec].

*LSIF: A format which language servers can emit to persist LSP-based information about a software project.

- **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) [DapSpec], which has previously already been integrated into SEE [**rohlfing2022**], 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. 2.1.

- **Notebook-related capabilities**: These are intended for interactive notebook systems such as Jupyter[15], which SEE does not currently support.

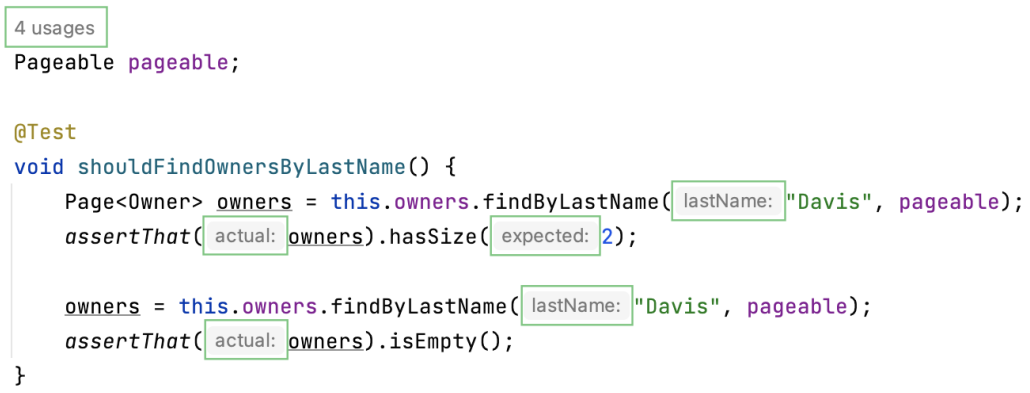


Figure 2.1: An example of inlay hints in the JetBrains IntelliJ IDE. (From `https://www.jetbrains.com/help/idea/inlay-hints.html` *(last access: 2024-10-04)*)

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

*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.

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.

We are now ready to tackle the actual implementation in the next chapter. As a quick overview and recap before then, there is a table of planned capabilities along with their intended use in SEE in table 2.1.

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 [see Gal21]
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 specifically. Finally, we will conduct a brief technical evaluation, with a more thorough user study following in the next chapter.

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 in essence 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.[16] The complexity of the

[16] 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)*.

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.[17]
- 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 [Aho+07, pp. 222–224].

I have submitted these fixes as a pull request[18]. 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 very short:

- I have integrated the OmniSharp LSP C# library[19] 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 Blecker [Ble22] in his bachelor 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 to accomplish.

[17] 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)*.
[18] https://github.com/microsoft/language-server-protocol/pull/1886 *(last access: 2024-10-10)*
[19] Available at https://github.com/OmniSharp/csharp-language-server-protocol *(last access: 2024-10-11)*

*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.[20]

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 take a quick look at 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 take a look at 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, then, is a graph representing the given software project.

TODO: Replace diagram sketch with TikZ picture.

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.

[20] 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.

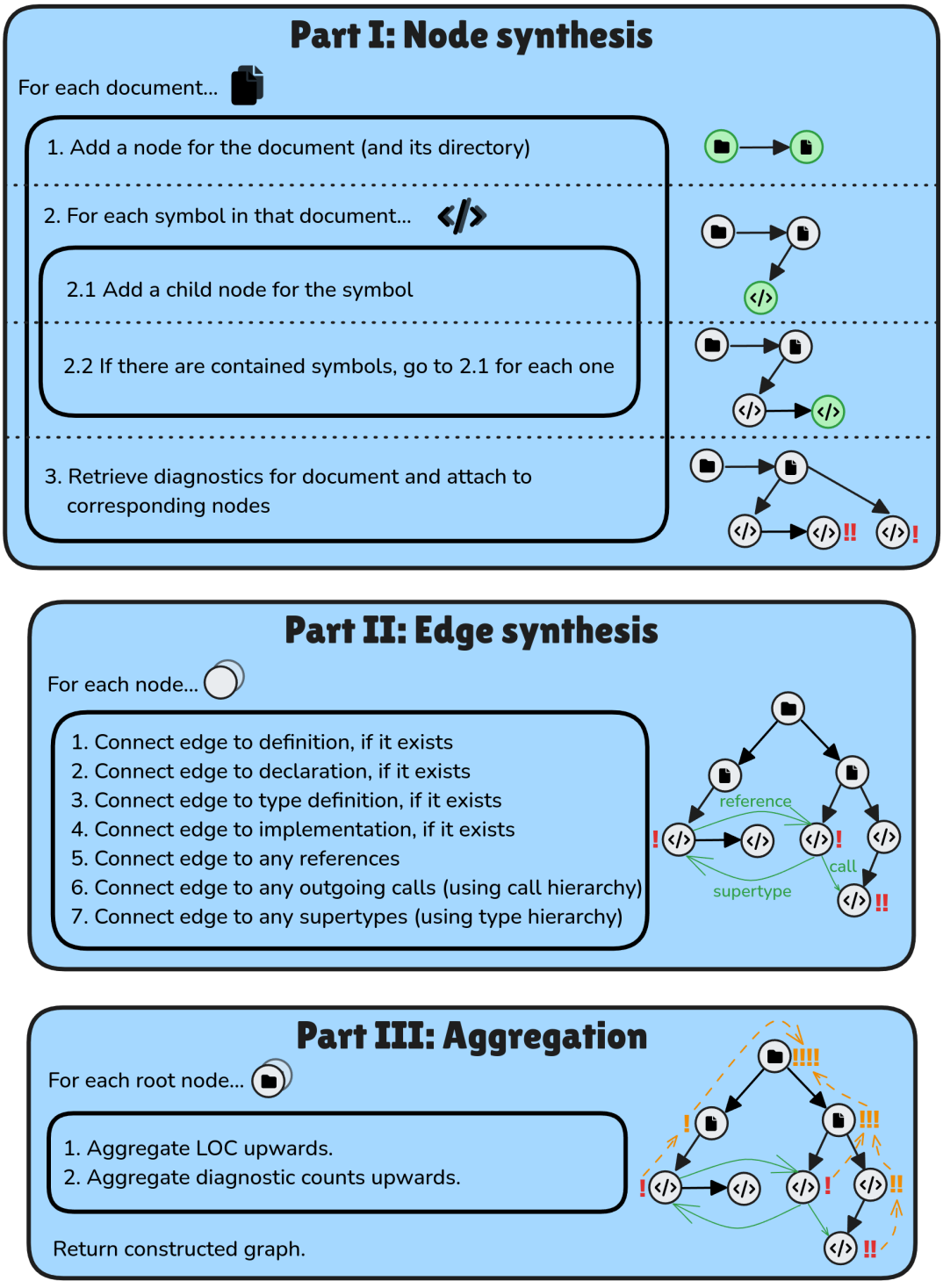


Figure 3.1: A high-level overview of the basic steps of the algorithm.

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[21] 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` have 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.

[21] 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[22] 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 [Wag23] in combination with the `UniTask`[23] framework: 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. This also allows us

[22] Apart from the obvious simplifications that occur naturally due to the difference between declarative mathematical notation and imperative programming syntax.
[23] https://github.com/Cysharp/UniTask *(last access: 2024-10-25)*

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)
```

to implement cancellation support, making it possible for the user to cancel the algorithm at any time[24].

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)*.

TODO: Convert all software mentions to biblatex @software type.

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[25]. 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 $|E| \gg |V|$. Hence, part II with $\Theta(|E| \cdot |V|)$ easily dwarfs the rest of the algorithm's runtime

TODO: Put in examples or more concrete estimation of how much more edges there are than nodes.

[24] 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.

[25] 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.

due to the high cost of FindNodesByLocation, which searches through every node for every added edge. For this reason, it would be nice to optimize that function somehow.

TODO: Do actual comparison between two ways of finding target nodes. Also mention example times from eval section at end.
TODO: Rewrite "the above" to something else, may not be above in printed version

3.2.2 Augmented Interval Trees

To restate the problem outlined in the performance considerations above: 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 $\Omega(n)$, 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 Cormen et al. [Cor+22, section 17.3], while k-d trees are described in a paper by Bentley [Ben75]. In our case, we can create the tree by constructing a 2d-tree[26] 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

[26] 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 (https://github.com/ericreg/Supercluster.KDTree *(last access: 2024-10-31)*).

*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".[27]

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
```

◂ **TODO:** So how much does it help in practice? Reference tech eval here

[27] 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 $(4, 4, 4, 27)$.

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. $b_l^q < l_{\texttt{pow}}$ 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 $\lambda_{\texttt{pow}}$ = `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 $\lambda_{\texttt{someValue}}$ = `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.

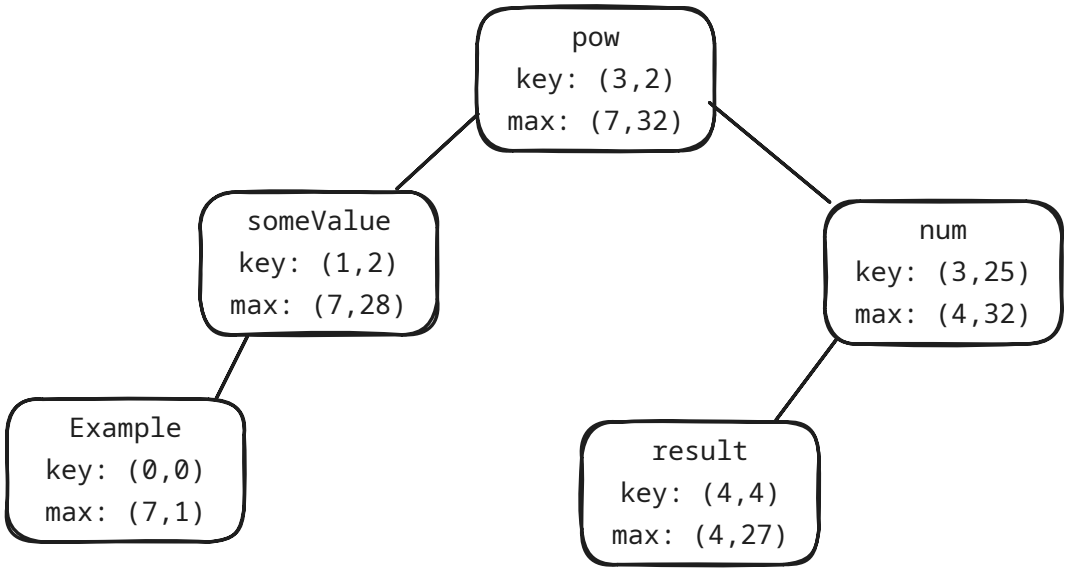


Figure 3.2: An augmented interval tree using a k-d tree, representing the elements in listing 3.1.

TODO: Convert to TikZ diagram and use colors from listing!

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 here due to the inherent complexity of this part, unit tests have also been implemented to make sure both normal and edge cases 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.

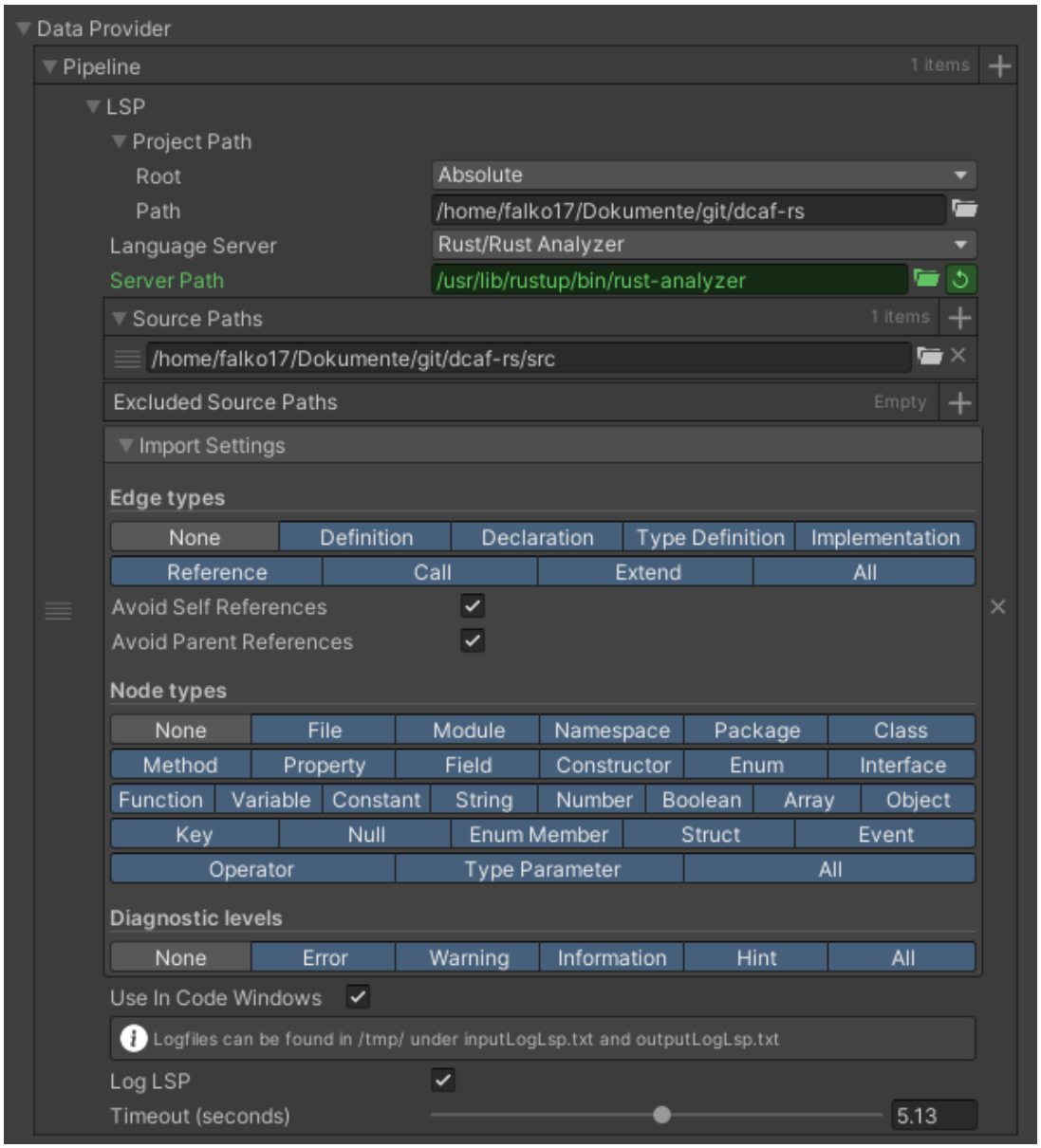


Figure 3.3: Unity Editor UI for the LSP graph provider

In that figure, we can see a configuration for the `dcaf-rs` project[28], 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*[29] 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[30] 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 (excluding `partOf`) 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 many 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 process. At this point, we will once again refer back to the explanatory video TODO, which

TODO: Link to video again here

[28] This is one of the sample projects we use in section 3.5's technical evaluation.
[29] The menu is grouped by language when opening it, hence the `Rust/` in front in fig. 3.3.
[30] 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.

goes over the implemented functionality in a bit more detail and may be easier to follow along than this textual description.

3.3 Integrating LSP Functionality into Code Cities

TODO!

3.4 Integrating LSP Functionality into Code Windows

TODO!

3.5 Technical Evaluation

TODO!

3.6 Interim Conclusion

TODO!

Table 3.1: 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`.

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 it.

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. 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 [HSS13]. Here, we use an algorithm by Sheather and Jones [SJ91] that has been improved to be made more performant and handle multimodal distributions better by Botev et al. [BGK10]: The *Improved Sheather–Jones* method, which is a robust choice when one cannot assume normality [Aki20]. As the algorithm for this method, we use `KDEpy` [Odl18], whose implementation of the method is based on Kroese et al. [KTB11, pp. 326–328]. A drawback here is that this algorithm does not necessarily converge. In those cases, we fall back to using Silverman's rule of thumb [Sil86], the implementation of which is integrated into the library we use to draw these plots [Cal24].

*Violin plot: A plot visualizing the distribution of a collection of data points along with an estimated probability density [HN98]. 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.

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*[31] and *SpotBugs*[32]. 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 [Gal21; GKS22].

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 reasonably evaluable 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.

[31] https://github.com/JabRef/jabref *(last access: 2024-11-22)*
[32] https://github.com/spotbugs/spotbugs *(last access: 2024-11-22)*

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 [Wag20; Gae21; Mas20; Döh20; Wic22; Kra24; Boh20; Smi21] as well as about its effectiveness compared to traditional tools [Gal21; Kip20; Ble22; Wei21; Roh24; Sch22; Abu21; Roh21]. 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: Kipka [Kip20] compares those with Eclipse[33]'s debugger, while Rohlfing [Roh24] uses the debugger of VSCode as a baseline. The final of these studies has Schramm [Sch22] 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 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.

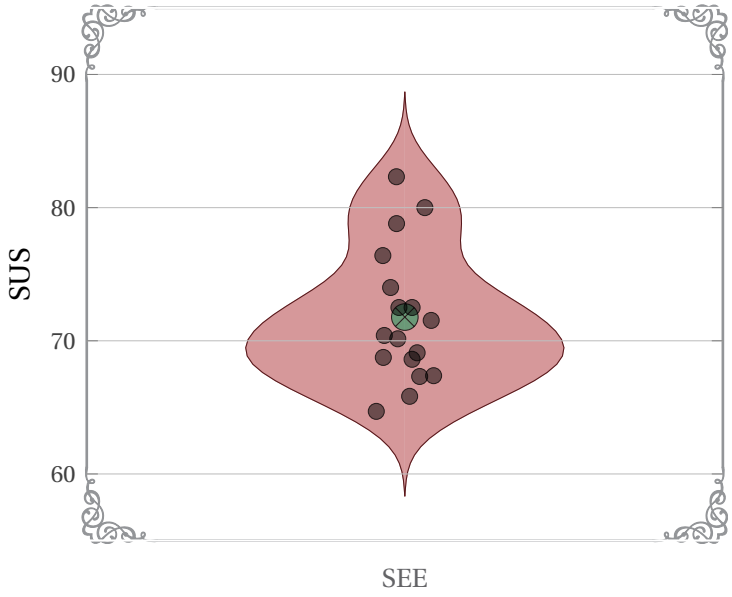


Figure 4.1: SUS results for SEE across sixteen studies [Wag20; Gae21; Gal21; Mas20; Döh20; Kip20; Wic22; Kra24; Ble22; Wei21; Boh20; Roh24; Smi21; Sch22; Abu21; Roh21].

Outside of SEE, there are a number of other code city implementations, such as *CodeCity* [WL07] or *Software World* [KM00] (see also the overview by Jeffery [Jef19]). In their evaluations,

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

*SUS: A simple questionnaire by Brooke [Bro96] with ten Likert-scale questions that are supposed to measure the usability of a system.

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
[WLR11]	45	$p = 0.001$	$p = 0.043$	*N/A*	Eclipse + metrics	CodeCity
[Kha+17]	28	*N/A*	3/5 IDE	6/20 CC; 1/20 IDE	Visual Studio (VS)	Code Park
[Rom+19]	54	$p = 0.005$	$p < 0.001\%$	No diff.	Eclipse + metrics	Code2City
[Kip20]	10	No diff.	No diff.	No diff.	Eclipse	SEE
[Meh+20]	20	$p = 0.005$	3/5 CC; 1/5 IDE	Preliminary only	Eclipse + 2D graph	XRASE
[GKS22]	20	2/6 IDE	4/6 CC	$p = 0.028$	Axivion Dashboard	SEE
[Sch22]	10	No diff.	1/3 CC	$p = 0.002$	VS + Axivion	VS + SEE
[MCD24]	49	6/11 CC	4/11 CC; 1/11 IDE	4/11 CC	Various + metrics	VariCity

Legend: ■ Code city advantage, ■ Slight code city advantage, ■ IDE advantage, ■ Slight IDE advantage, ■ No significant difference, □ Not measured

these papers often compare different platforms (such as Desktop to VR/AR) [e. g., Mer+17; FKH15; MBN18], but as with the SEE-related theses above, we are most interested in those that have controlled experiments comparing a code city implementation with a traditional IDE.

One such study was done by Wettel et al. [WLR11] 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 [WLR11, chapter 7]. 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 Romano et al. [Rom+19] with a subset of tasks.

Similarly, Mortara et al. [MCD24] supplied Comma-Separated Values* (CSV) files for IDE users in a comparison against *VariCity*, although here, participants were allowed to choose whatever IDE they are most comfortable with. Mehra et al. [Meh+20] gave participants who were using Eclipse an additional 2D graph 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 Khaloo et al. [Kha+17], Galperin et al. [GKS22], and Kipka [Kip20]. The results of their experiments, and all others cited here, are listed in the color-coded table 4.1.

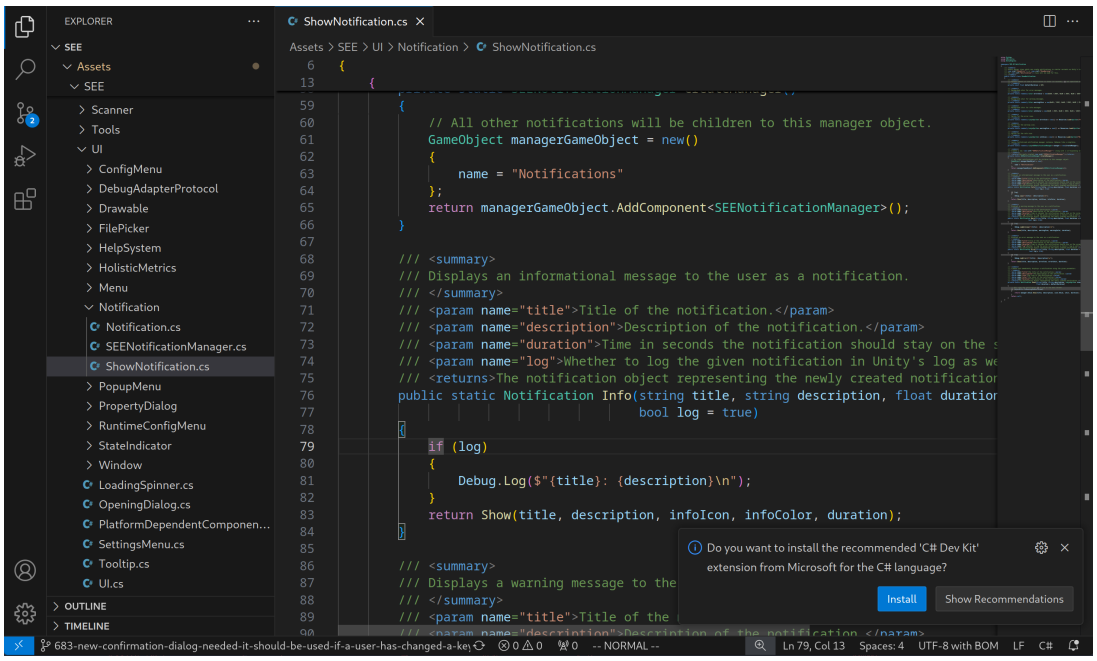


Figure 4.2: Screenshot of the main UI of VSCode.

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 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,

*CSV: A file format in which tabular data is stored as comma-separated columns.

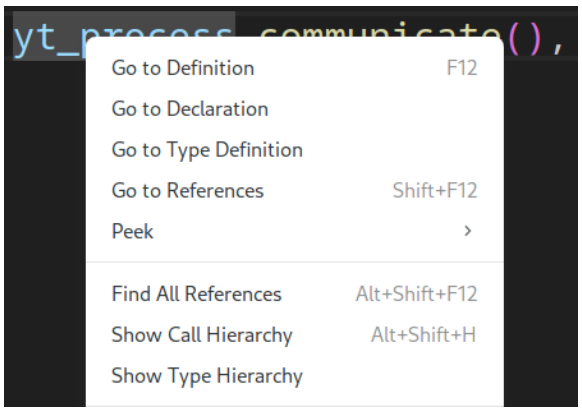


Figure 4.3: Screenshot of (the beginning of) VSCode's context menu for code identifiers.

participants would need to manually count each method, so to remedy this, we offer a table with such metrics. The table is hosted on Google Spreadsheets[34] and can be sorted by any column as well as searched. The script with which the metrics were extracted is attached as TODO.

TODO: Attach script in appendix.

4.1.3 Hypotheses

To answer RQ2 (see section 1.3), we would like to know whether there is any significant[35] 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.

- *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.

[34] 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)*.

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

- *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_D.
   - *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*[36] A_S^c for SEE and A_V^c for VSCode.
      - *Null hypothesis* H_{d_0}: The ASQ score for complexity when using SEE is the same as when using VSCode: $A_S^c = A_V^c$
      - *Alternative hypothesis* H_{d_1}: The ASQ score for complexity when using SEE is different when using VSCode: $A_S^c \neq A_V^c$
   ii) We call the ASQ score for *effort*[37] A_S^e for SEE and A_V^e for VSCode.
      - *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.

[36] Note that a higher score here means a lower amount of complexity.
[37] 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 [Lew91].

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 datapoints, 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 $\frac{n}{2}$ datapoints per tool for each task, which we can then compare against each other 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 [e. g. Eva+10]). 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 and verify that the participant understood them before starting the actual tasks.

Additionally, we would like to follow the "wish list" compiled by Wettel et al. [WLR11, pp. 122–124]:

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

*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 [Roe+39], although the effects observed have later turned out to be likely unrelated to the observation [Jon92].

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). [✓]

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 limit like this into the tool we used for the questionnaire would have proved difficult, but the tasks were comparatively simple and participants were aware that their completion time was measured, so we should hopefully have avoided the effects Wettel references here. [?]

11. **"Provide all the details needed to make the experiment replicable."** I have included (pseudonymized) participation data, the full definition of the questionnaire, and the analysis script which performed data cleanup and statistical tests. [✓]

TODO: Link to appendix here.

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 detailed 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 3/13 being questionably/partially fulfilled and 2/12 not being implemented, with the reasoning for the latter two being 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, where we will use the latter two to get an estimation on 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 [Gal21]. For a comprehensive overview and evaluation of other post-study and post-task questionnaires covered neither here nor in my bachelor's thesis, see the review by Hodrien and Fernando [HF21].

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 [BKM08, p. 585; MMP12].

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

*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. [HS88, p. 169]

Magnitude Estimation* (UME) (which we exclude since it often confuses its users [SD09, pp. 1607–1608]) 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 Roegele and Funk [RF20, p. 32] 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 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 [Rei15], is the most used post-study questionnaire for this purpose [SL09, p. 1615; Lew18, p. 577], and is very well-suited for comparing two systems or interfaces [PPP13, p. 195; BKM08, pp. 590–591]. It consists of ten likert scale questions which alternate between

*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. [McG03]

**SMEQ: A single question intended to estimate usability which uses a scale with 150 options. [ZD85, pp. 53 sqq.]

†Likert scale: A psychometric scale in which users indicate their agreement on a linear scale from "strongly agree" to "strongly disagree" [Lik32].

‡SUMI: A 50-item questionnaire measuring usability on the axes of efficiency, affect, help and support, steerability, and learnability. [Kir94]

§QUIS: A post-study questionnaire measuring usability across 41 questions in its short version and 122 questions in its long version. [CDN88]

¶PSSUQ: A questionnaire that measures usability in sixteen questions across the three factors *usefulness*, *information quality*, and *interface quality*. [Lew92; Lew02]

positive and negative aspects of the system and returns a score between 0 and 100, representing the system's usability [Bro96]. 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*[38] seems like the most mature, robust option. It accepts form based on the XLSForm* specification, which makes forms (relatively) easy to create with a spreadsheet editor.

Before each task, we tell the user beforehand 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 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 TODO, and a screenshot of the starting page is shown in fig. 4.4.

TODO: Link to questionnaire in appendix

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 TODO. Inviting others to participate then becomes as simple as passing the link to the redirection script around[39].

TODO: Link to script in appendix

[38] https://www.kobotoolbox.org/ *(last access: 2024-11-23)*
[39] https://falko.de/master-evaluation, but this link may not stay up forever.

*XLSForm: A form standard based on human-readable Excel spreadsheets, which allows for complex forms (e. g., conditionals, skip logic) to be built [MDO24]. The forms can then be converted into the open ODK XForm format, which is compatible with a number of data collection tools [OO19].

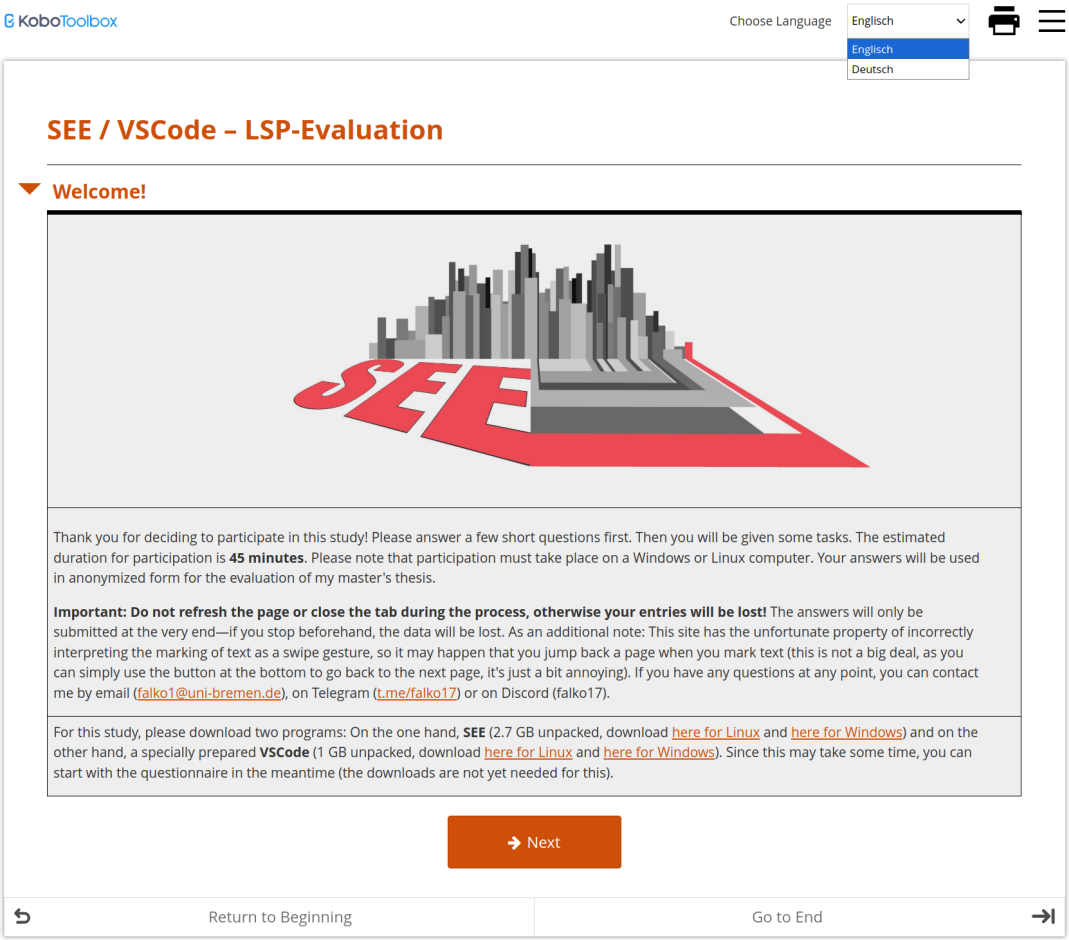


Figure 4.4: Starting page of the survey built with KoboToolbox.

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 $p \notin P$ **then**
  if $p = \varnothing$ **then**
    $p = \text{NewRandomID}()$
  $n_\Psi, n_\Omega \leftarrow \text{GetKoboParticipations}()$
  if $n_\Psi = \varnothing \vee n_\Omega = \varnothing$ **then** ▷ *API request failed, so just alternate using index i.*
    if $i = 0$ **then**
      $P(p) \leftarrow \Psi$
    else
      $P(p) \leftarrow \Omega$
    $i \leftarrow (i + 1) \bmod 2$
  else if $n_\Psi \leq n_\Omega$ **then**
    $P(p) \leftarrow \Psi$
  else
    $P(p) \leftarrow \Omega$
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 Wettel et al. [WLR11] (replicated by Romano et al. [Rom+19]) seems most fitting for us: It compares a Desktop code city implementation against a traditional IDE along with providing CSV metrics to make the comparison fair. In addition, it 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 [WLR11, pp. 128–130]: Those concerened 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, 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.

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***.

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

Note, however, that none of the tasks we selected above 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[40]) 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 system. 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 some section.

TODO: Where?

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.

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

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

*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.

**Mann-Whitney *U* test: A test to check whether two distributions have identical distributions, requiring only ordinal-scaled independent samples drawn from the distributions. [MW47]

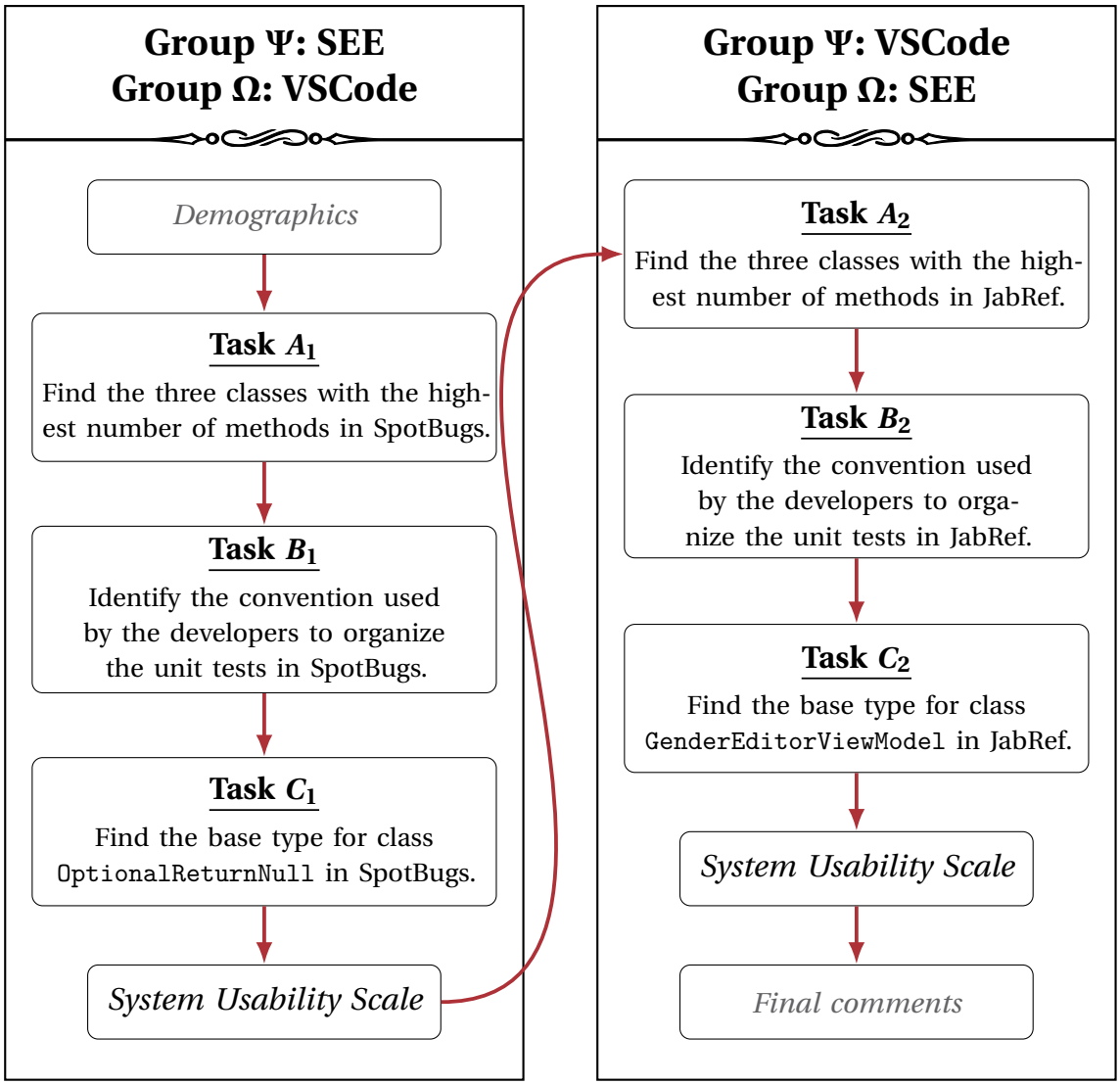


Figure 4.5: Flow of the tasks the participants worked on. A^c and A^e were gathered after each task.

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[41] 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.

TODO: In Appendix, give answer key.

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 merely ordinal data.

All answers[42] 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, 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."

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

[42] The question "Do you know SpotBugs?" has also been excluded from the bar charts, as only one participant answered "I've heard of it", with all others answering "no".

Age We have a median age of $\widetilde{A} = 29$ and an average age of $\overline{A} = 28.55$ 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 ($U = 40.5$, $p \approx 0.4947$).

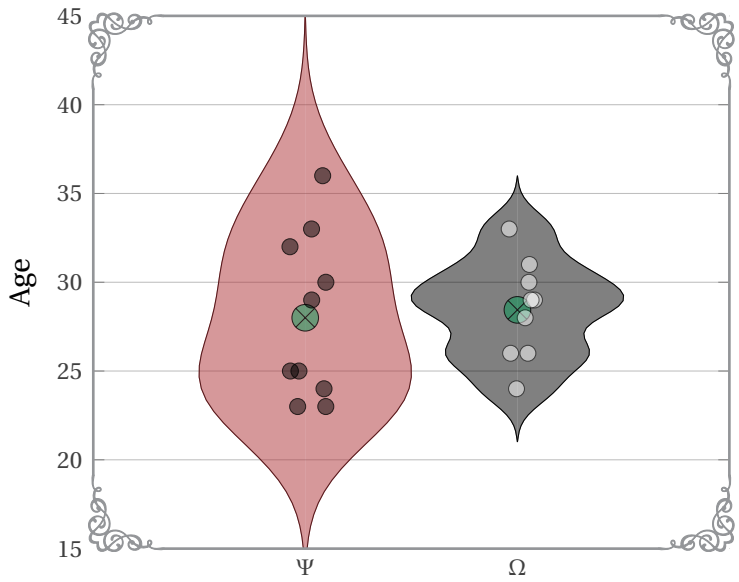


Figure 4.6: Distribution of age between the two groups.

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 participation took 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 ($U = 63$, $p \approx 0.34470$).

Degree Most participants had a Bachelor's degree (this is also the median), with some Abitur degrees, 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 ($U = 57.5$, $p \approx 0.5693$).

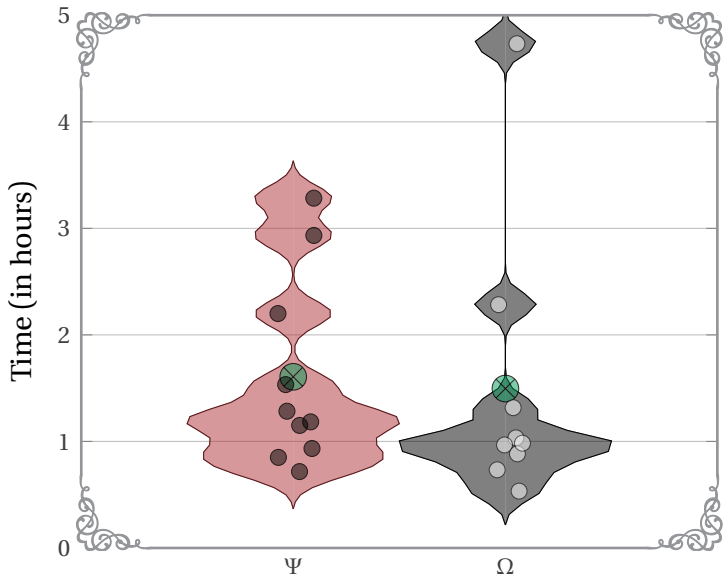


Figure 4.7: Total duration of time in which each participant finished the study. May include time in which participants took breaks.

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

★ **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

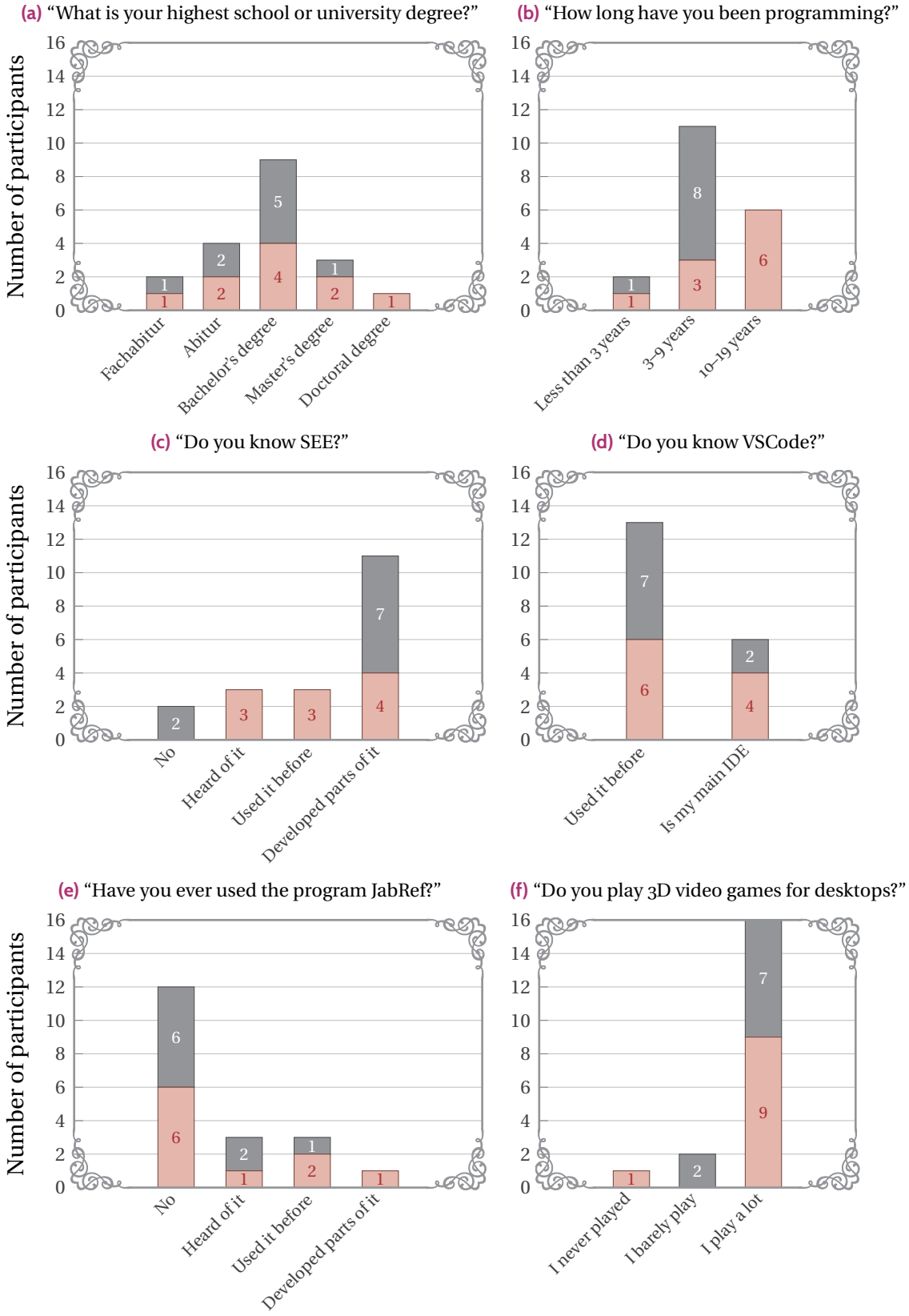


Figure 4.8: Number of respective answers to various demographic questions. Group Ψ is in red, and group Ω is in gray.

contains the only people who have answered “I heard of it” and “I used it before”. However, this difference is not significant ($U = 42.5, p \approx 0.5598$).

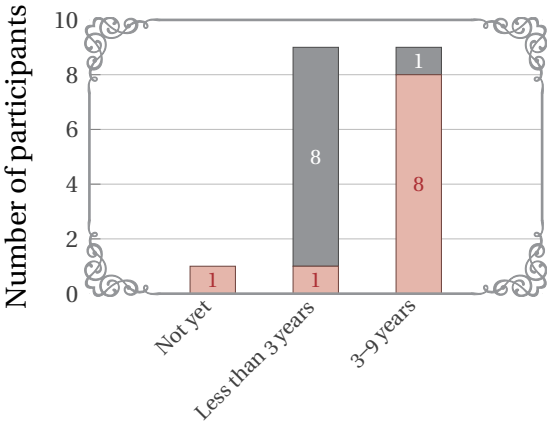


Figure 4.9: “How long have you been programming on bigger software projects (e. g., within a company, or open-source projects)?”

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. Here as well there is no significant difference we need to be aware of ($U = 46, p \approx 0.6701$).

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 TODO) as a base. This allowed me to catch misspellings or unusual formats[43] and still mark the answers as correct. The results of which answers I marked as correct are attached as TODO, while the script used to do so is part of the general analysis script in TODO.

TODO: Attach and link answer key, correctness answers, script in appendix

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 Mann-Whitney *U* test.

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* 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 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 answers "none" and "dispersed", 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. Looking at figs. 4.10c and 4.10d proves 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

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

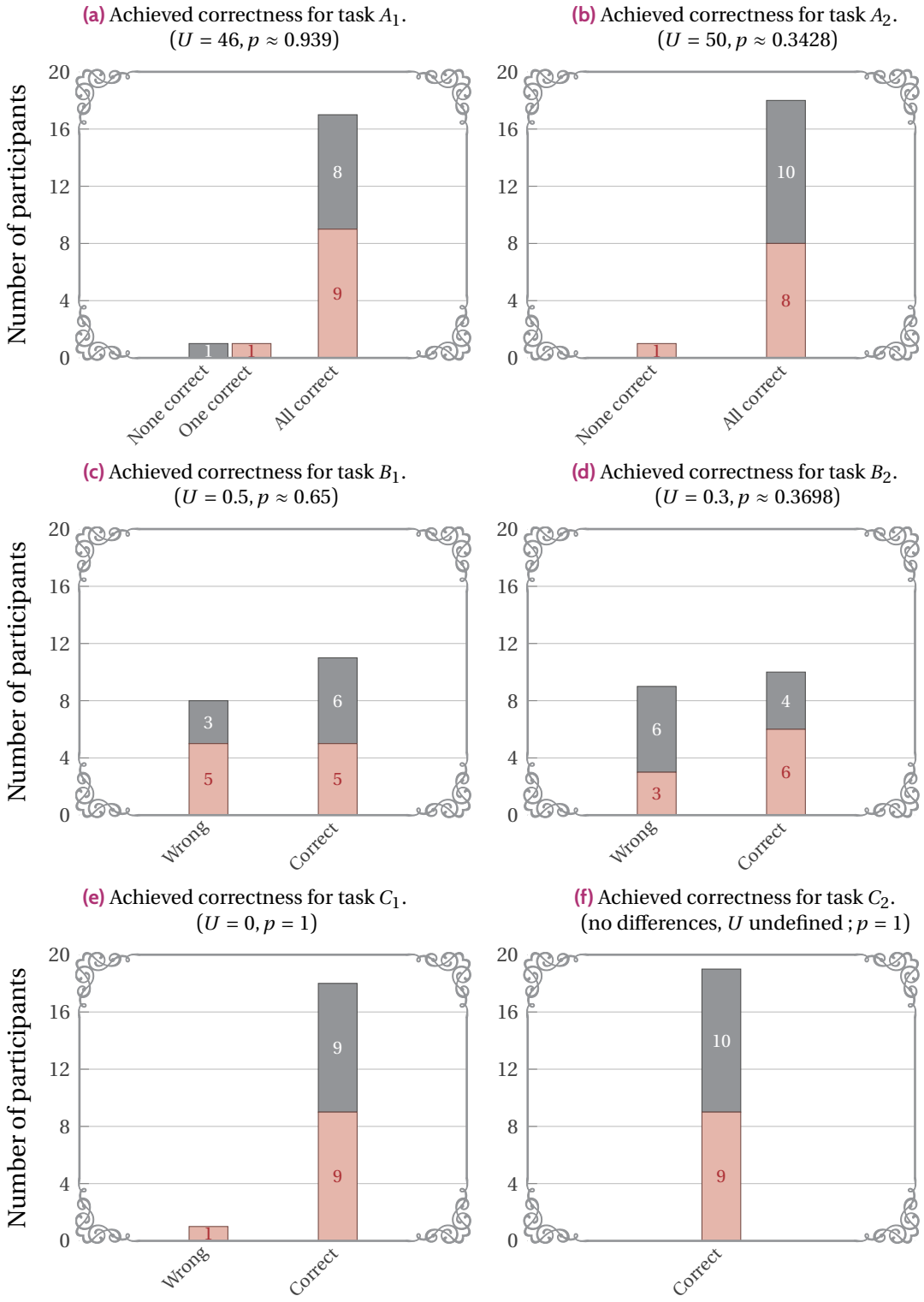


Figure 4.10: Correctness achieved in the various tasks, compared across the two systems. Correctness when using SEE is in red, and when using VSCode is in gray.

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.

4.3.4 Time ★

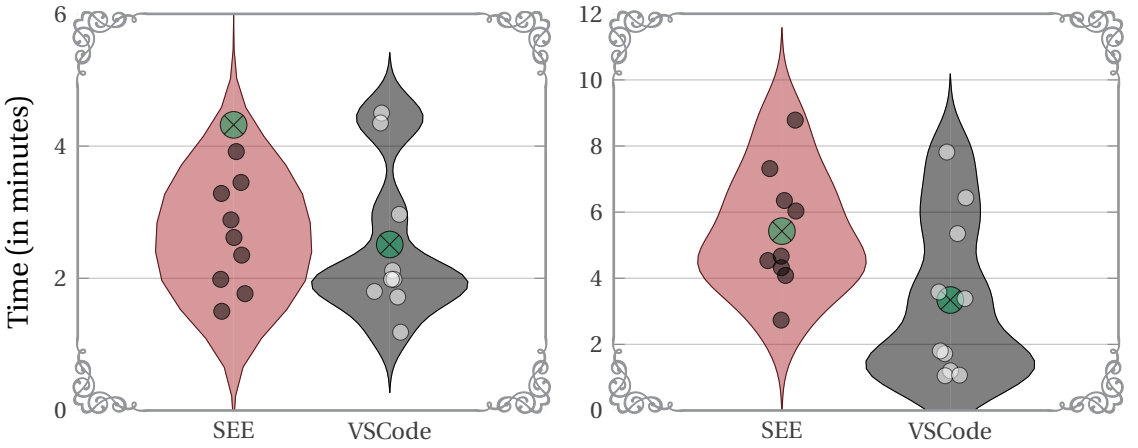
As mentioned before, we calculated the time it took to complete a task using the set of timestamps T and the starting timestamp t_0 as $\hat{t} = \max T - t_0$. While the violin plots in fig. 4.11 include all datapoints, for the Mann-Whitney U tests we exclude datapoints belonging to incorrect answers for the respective tasks (as we do not want to count quick but wrong solutions). 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.

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 moreso 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 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

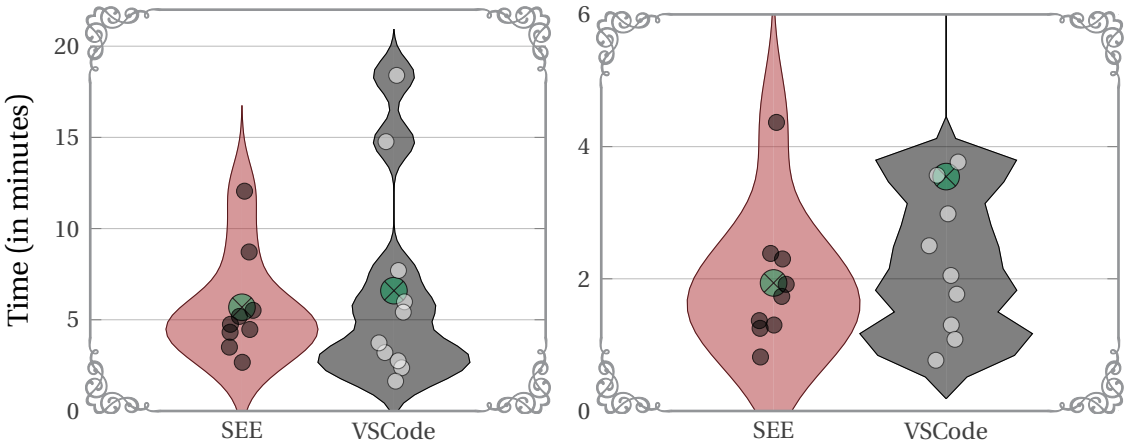
(a) Time taken for task A_1. Excluded a SEE outlier that took 20 minutes. ($U = 47, p \approx 0.3213$)

(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. ($\bigstar$ $U = 17, p = 0.0247$)

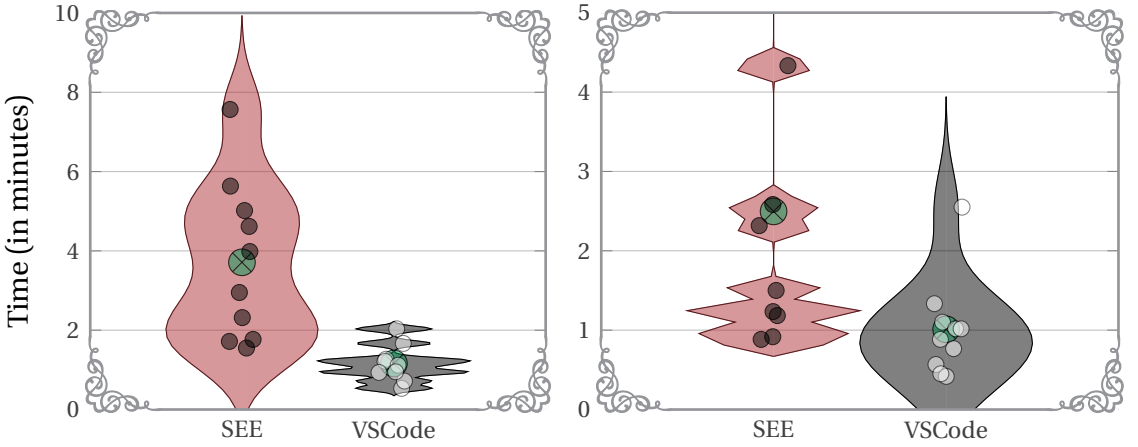


Figure 4.11: Time that it took participants to solve the tasks, compared across the two systems.

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.

4.3.5 Usability

TODO!

4.3.6 The Effects of Experience

TODO!

4.3.7 Comments

TODO!

4.4 Threats to Validity

TODO!

4.5 Interim Conclusion

TODO!

5

Conclusion

 PON... **TODO!**

5.1 Limitations

TODO!

5.2 Future Work

TODO!

5.3 The End

TODO!

TODO: Find better title here

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.

Glossary

This document is incomplete. The external file associated with the glossary 'main' (which should be called `thesis.gls`) hasn't been created.

Check the contents of the file `thesis.glo`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

If you don't want this glossary, add `nomain` to your package option list when you load `glossaries-extra.sty`. For example:

```
\usepackage[nomain]{glossaries-extra}
```

Try one of the following:

- Add `automake` to your package option list when you load `glossaries-extra.sty`. For example:

  ```
  \usepackage[automake]{glossaries-extra}
  ```

- Run the external (Lua) application:

  ```
  makeglossaries-lite.lua "thesis"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "thesis"
  ```

Then rerun LaTeX on this document.

This message will be removed once the problem has been fixed.

Acronyms

This document is incomplete. The external file associated with the glossary 'abbreviations' (which should be called `thesis.gls-abr`) hasn't been created.

Check the contents of the file `thesis.glo-abr`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

Try one of the following:

- Add `automake` to your package option list when you load `glossaries-extra.sty`. For example:

  ```
  \usepackage[automake]{glossaries-extra}
  ```

- Run the external (Lua) application:

  ```
  makeglossaries-lite.lua "thesis"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "thesis"
  ```

Then rerun LaTeX on this document.

This message will be removed once the problem has been fixed.

Attached Files

Notation	Description
`SEE.zip`	The build of SEE used for the evaluation.

List of Figures

List of Tables

List of Listings

Bibliography

[Abu21] Sulan Abubakarov. “Recording and visualization of software runtime data with SEE”. BA thesis. University of Bremen, Dec. 6, 2021 (cit. on p. 45).

[Aho+07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, eds. *Compilers: Principles, Techniques, & Tools*. 2. ed., Pearson internat. ed. Boston Munich: Pearson Addison-Wesley, 2007. 1009 pp. (cit. on p. 26).

[Aki20] Andrey Akinshin. *The Importance of Kernel Density Estimation Bandwidth*. Oct. 13, 2020. URL: https://aakinshin.net/posts/kde-bw/ (visited on Nov. 20, 2024) (cit. on p. 43).

[BKM08] Aaron Bangor, Philip T. Kortum, and James T. Miller. “An Empirical Evaluation of the System Usability Scale”. In: *International Journal of Human-Computer Interaction* 24.6 (July 29, 2008), pp. 574–594 (cit. on pp. 52, 53).

[Ben75] Jon Louis Bentley. “Multidimensional Binary Search Trees Used for Associative Searching”. In: *Communications of the ACM* 18.9 (Sept. 1975), pp. 509–517 (cit. on p. 35).

[Ble22] Moritz Blecker. “Developing a Multi-user Source Code Editor for SEE”. BA thesis. University of Bremen, Jan. 31, 2022 (cit. on pp. 8, 26, 45).

[Boh20] Robert Bohnsack. “Grafische Darstellung der Metriken des SEE City Projekts”. BA thesis. University of Bremen, June 8, 2020 (cit. on p. 45).

[BGK10] Z. I. Botev, J. F. Grotowski, and D. P. Kroese. “Kernel Density Estimation via Diffusion”. In: *The Annals of Statistics* 38.5 (Oct. 1, 2010) (cit. on p. 43).

[Bro96] John Brooke. “SUS: A ’Quick and Dirty’ Usability Scale”. In: *Usability Evaluation In Industry*. CRC Press, 1996 (cit. on pp. 45, 54).

[Cal24] Pedro Henrique Callil-Soares. *Tikzviolinplots*. Oct. 31, 2024 (cit. on p. 43).

[CDN88] J. P. Chin, V. A. Diehl, and L. K. Norman. “Development of an Instrument Measuring User Satisfaction of the Human-Computer Interface”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI ’88*. The SIGCHI Conference. Washington, D.C., United States: ACM Press, 1988, pp. 213–218 (cit. on p. 53).

[Cor+22] Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest, and Clifford Stein. *Introduction to Algorithms*. Fourth edition. Cambridge, Massachusetts London: The MIT Press, 2022. 1 p. (cit. on p. 35).

[Cro06] Douglas Crockford. *The Application/Json Media Type for JavaScript Object Notation (JSON)*. Request for Comments RFC 4627. Internet Engineering Task Force, July 2006. 10 pp. (cit. on p. 16).

[Döh20] Kevin Döhl. “Modellierung einer Softwarearchitektur in Virtual Reality mithilfe der Leap Motion”. BA thesis. University of Bremen, Sept. 15, 2020 (cit. on p. 45).

[Eva+10] Rhodri Evans, Natalie Joseph-Williams, Adrian Edwards, et al. “Supporting Informed Decision Making for Prostate Specific Antigen (PSA) Testing on the Web: An Online Randomized Controlled Trial”. In: *Journal of Medical Internet Research* 12.3 (Aug. 6, 2010), e27 (cit. on p. 50).

[FKH15] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. “Exploring Software Cities in Virtual Reality”. In: *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. 2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT). Bremen, Germany: IEEE, Sept. 2015, pp. 130–134 (cit. on p. 46).

[FB19] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Second edition. The Addison-Wesley Signature Series. Boston Columbus New York San Francisco Amsterdam Cape Town Dubai London Munich: Addison-Wesley, 2019. 418 pp. (cit. on p. 11).

[Gae21] Felix Gaebler. “Extraktion von Daten aus Versionskontrollsystemen zur anschließenden Visualisierung in einer CodeCity”. BA thesis. University of Bremen, Dec. 30, 2021 (cit. on p. 45).

[Gal21] Falko Galperin. “Visualizing Code Smells in Code Cities”. BA thesis. University of Bremen, Sept. 27, 2021 (cit. on pp. 11, 15, 19, 23, 44, 45, 52).

[GKS22] Falko Galperin, Rainer Koschke, and Marcel Steinbeck. “Visualizing Code Smells: Tables or Code Cities? A Controlled Experiment”. In: *2022 Working Conference on Software Visualization (VISSOFT)*. 2022 Working Conference on Software Visualization (VISSOFT). Limassol, Cyprus: IEEE, Oct. 2022, pp. 51–62 (cit. on pp. 15, 44, 46).

[HS88] Sandra G. Hart and Lowell E. Staveland. “Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research”. In: *Advances in Psychology*. Vol. 52. Elsevier, 1988, pp. 139–183 (cit. on p. 52).

[HSS13] Nils-Bastian Heidenreich, Anja Schindler, and Stefan Sperlich. “Bandwidth Selection for Kernel Density Estimation: A Review of Fully Automatic Selectors”. In: *AStA Advances in Statistical Analysis* 97.4 (Oct. 1, 2013), pp. 403–433 (cit. on p. 43).

[HN98] Jerry L. Hintze and Ray D. Nelson. “Violin Plots: A Box Plot-Density Trace Synergism”. In: *The American Statistician* 52.2 (May 1998), pp. 181–184 (cit. on p. 43).

[HF21] Andrew Hodrien and Terrence Fernando. “A Review of Post-Study and Post-Task Subjective Questionnaires to Guide Assessment of System Usability”. In: *J. Usability Studies* 16.3 (May 1, 2021), pp. 203–232 (cit. on p. 52).

[Jef19] Clinton L. Jeffery. “The City Metaphor in Software Visualization”. In: *Computer Science Research Notes*. WSCG’2019 - 27. International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision’2019. Západočeská univerzita, 2019 (cit. on p. 45).

[Jon92] Stephen R. G. Jones. “Was There a Hawthorne Effect?” In: *American Journal of Sociology* 98.3 (Nov. 1992), pp. 451–468 (cit. on p. 50).

[JSO13] JSON-RPC Working Group. *JSON-RPC 2.0 Specification*. Version 2.0. Jan. 4, 2013 (cit. on p. 16).

[Kha+17] Pooya Khaloo, Mehran Maghoumi, Eugene Taranta, David Bettner, and Joseph Laviola. “Code Park: A New 3D Code Visualization Tool”. In: *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. 2017 IEEE Working Conference on Software Visualization (VISSOFT). Shanghai: IEEE, Sept. 2017, pp. 43–53 (cit. on p. 46).

[Kip20] Lennart Kipka. “Software-Debugging with Code Cities”. BA thesis. University of Bremen, Oct. 13, 2020 (cit. on pp. 45, 46).

[Kir94] Jurek Kirakowski. “The Use of Questionnaire Methods for Usability Assessment”. 1994 (cit. on p. 53).

[KM00] C. Knight and M. Munro. “Virtual but Visible Software”. In: *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*. IEEE International Conference on Information Visualization. London, UK: IEEE Comput. Soc, 2000, pp. 198–205 (cit. on p. 45).

[Kra24] Michel Krause. “Integration of an interactive whiteboard (drawable) for SEE”. MA thesis. University of Bremen, Mar. 26, 2024 (cit. on p. 45).

[KTB11] Dirk P. Kroese, Thomas Taimre, and Zdravko I. Botev. *Handbook of Monte Carlo Methods*. 1st ed. Wiley Series in Probability and Statistics. Wiley, Feb. 28, 2011 (cit. on p. 43).

[Lew91] James R. Lewis. "Psychometric Evaluation of an After-Scenario Questionnaire for Computer Usability Studies: The ASQ". In: *SIGCHI Bull.* 23.1 (Jan. 1, 1991), pp. 78–81 (cit. on p. 49).

[Lew92] James R. Lewis. "Psychometric Evaluation of the Post-Study System Usability Questionnaire: The PSSUQ". In: *Proceedings of the Human Factors Society Annual Meeting* 36.16 (Oct. 1992), pp. 1259–1260 (cit. on p. 53).

[Lew02] James R. Lewis. "Psychometric Evaluation of the PSSUQ Using Data from Five Years of Usability Studies". In: *International Journal of Human-Computer Interaction* 14.3-4 (Sept. 2002), pp. 463–488 (cit. on p. 53).

[Lew18] James R. Lewis. "The System Usability Scale: Past, Present, and Future". In: *International Journal of Human–Computer Interaction* 34.7 (July 3, 2018), pp. 577–590 (cit. on p. 53).

[Lik32] R. Likert. "A Technique for the Measurement of Attitudes". In: *Archives of Psychology* 22 140 (1932), pp. 55–55 (cit. on p. 53).

[MW47] H. B. Mann and D. R. Whitney. "On a Test of Whether One of Two Random Variables Is Stochastically Larger than the Other". In: *The Annals of Mathematical Statistics* 18.1 (Mar. 1947), pp. 50–60 (cit. on p. 57).

[MDO24] Andrew Marder, Alex Dorey, and ODK Community. *XLSForm*. Mar. 9, 2024 (cit. on p. 54).

[Mas20] Hannes Masuch. "Virtuelle Software-Städte annotieren und umgestalten". BA thesis. University of Bremen, Dec. 16, 2020 (cit. on p. 45).

[McC76] T.J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.SE-2 (4 Dec. 1976), pp. 308–320 (cit. on p. 12).

[McG03] Mick McGee. "Usability Magnitude Estimation". In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 47.4 (Oct. 2003), pp. 691–695 (cit. on p. 53).

[MMP12] Sam McLellan, Andrew Muddimer, and S. Camille Peres. "The Effect of Experience on System Usability Scale Ratings". In: *J. Usability Studies* 7.2 (Feb. 1, 2012), pp. 56–67 (cit. on p. 52).

[Meh+20] Rohit Mehra, Vibhu Saujanya Sharma, Vikrant Kaulgud, Sanjay Podder, and Adam P. Burden. “Towards Immersive Comprehension of Software Systems Using Augmented Reality: An Empirical Evaluation”. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’20: 35th IEEE/ACM International Conference on Automated Software Engineering. Virtual Event Australia: ACM, Dec. 21, 2020, pp. 1267–1269 (cit. on p. 46).

[MBN18] Leonel Merino, Alexandre Bergel, and Oscar Nierstrasz. “Overcoming Issues of 3D Software Visualization through Immersive Augmented Reality”. In: *2018 IEEE Working Conference on Software Visualization (VISSOFT)*. 2018 IEEE Working Conference on Software Visualization (VISSOFT). Madrid: IEEE, Sept. 2018, pp. 54–64 (cit. on p. 46).

[Mer+17] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. “CityVR: Gameful Software Visualization”. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). Shanghai: IEEE, Sept. 2017, pp. 633–637 (cit. on p. 46).

[DapSpec] Microsoft. *Debug Adapter Protocol*. May 2022. URL: https://microsoft.github.io/debug-adapter-protocol/overview (visited on Jan. 31, 2024) (cit. on p. 22).

[LsifSpec] Microsoft. *Language Server Index Format*. Sept. 2021. URL: https://microsoft.github.io/language-server-protocol/overviews/lsif/overview/ (visited on Jan. 31, 2024) (cit. on p. 21).

[LspSpec] Microsoft. *Language Server Protocol*. Oct. 2022. URL: https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/ (visited on Jan. 28, 2024) (cit. on pp. 4, 17, 19, 20).

[MCD24] Johann Mortara, Philippe Collet, and Anne-Marie Dery-Pinna. “Visualization of Object-Oriented Software in a City Metaphor: Comprehending the Implemented Variability and Its Technical Debt”. In: *Journal of Systems and Software* 208 (Feb. 2024), p. 111876 (cit. on p. 46).

[OO19] ODK Community and OpenRosa Consortium. *ODK XForms Specification*. Version 1.0.0. Nov. 11, 2019 (cit. on p. 54).

[Odl18] Tommy Odland. *Tommyod/KDEpy: Kernel Density Estimation in Python*. Version v0.9.10. Zenodo, Dec. 18, 2018 (cit. on p. 43).

[PPP13] S. Camille Peres, Tri Pham, and Ronald Phillips. “Validation of the System Usability Scale (SUS): SUS in the Wild”. In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 57.1 (Sept. 2013), pp. 192–196 (cit. on p. 53).

[Rei15] Wolfgang Reinhardt. *System Usability Scale – jetzt auch auf Deutsch*. SAP User Experience Community. Jan. 13, 2015. URL: https://web.archive.org/web/20200607035254/https://experience.sap.com/skillup/system-usability-scale-jetzt-auch-auf-deutsch/ (visited on Nov. 22, 2024) (cit. on p. 53).

[RF20] Barbara Roegele and Walter Funk. *Safety4Bikes. Arbeitspaket 1: Usability-evaluation Des Gesamtsystems Aus Der Anforderungsperspektive von Kindern*. Sept. 2020 (cit. on p. 53).

[Roe+39] F. J Roethlisberger, W. J Dickson, Harold A Wright, Carl H Pforzheimer, and Western Electric Company. *Management and the Worker: An Account of a Research Program Conducted by the Western Electric Company, Hawthorne Works, Chicago*. Cambridge, Mass.: Harvard University Press, 1939 (cit. on p. 50).

[Roh24] Ferdinand Rohlfing. "Debug Adapter Protocol in SEE". BA thesis. University of Bremen, Aug. 2024 (cit. on p. 45).

[Roh21] Yannis Rohloff. "Performance Profiling and Visualization in Software Cities". MA thesis. University of Bremen, Aug. 23, 2021 (cit. on p. 45).

[Rom+19] Simone Romano, Nicola Capece, Ugo Erra, Giuseppe Scanniello, and Michele Lanza. "On the Use of Virtual Reality in Software Visualization: The Case of the City Metaphor". In: *Information and Software Technology* 114 (Oct. 2019), pp. 92–106 (cit. on pp. 46, 56).

[SD09] Jeff Sauro and Joseph S. Dumas. "Comparison of Three One-Question, Post-Task Usability Questionnaires". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '09: CHI Conference on Human Factors in Computing Systems. Boston MA USA: ACM, Apr. 4, 2009, pp. 1599–1608 (cit. on p. 53).

[SL09] Jeff Sauro and James R. Lewis. "Correlations among Prototypical Usability Metrics: Evidence for the Construct of Usability". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '09: CHI Conference on Human Factors in Computing Systems. Boston MA USA: ACM, Apr. 4, 2009, pp. 1609–1618 (cit. on p. 53).

[Sch22] Jan-Phillipp Schramm. "Bidirektionale Integrierte Entwicklungsumgebung". BA thesis. University of Bremen, Feb. 2022 (cit. on pp. 45, 46).

[SJ91] S. J. Sheather and M. C. Jones. "A Reliable Data-Based Bandwidth Selection Method for Kernel Density Estimation". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 53.3 (Jan. 1, 1991), pp. 683–690 (cit. on p. 43).

[Sil86] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall/CRC Monographs on Statistics and Applied Probability v.26. Boca Raton: Routledge, Apr. 1, 1986 (cit. on p. 43).

[Smi21] Ruben Smidt. "Laufzeitkonfiguration von SEE-Cities". BA thesis. University of Bremen, June 14, 2021 (cit. on p. 45).

[Wag23] Bill Wagner. *The Task Asynchronous Programming (TAP) Model with Async and Await' - C#*. Feb. 13, 2023. URL: https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/task-asynchronous-programming-model (visited on Oct. 25, 2024) (cit. on p. 30).

[Wag20] David Wagner. "Visualisierung für eine Software-Architekturprüfung in Virtual Reality". BA thesis. University of Bremen, Nov. 9, 2020 (cit. on p. 45).

[Wei21] Nico Weiser. "Modelling Software Architecture Using Two-Dimensional Stroke Gestures". BA thesis. University of Bremen, Nov. 2, 2021 (cit. on p. 45).

[WL07] Richard Wettel and Michele Lanza. "Visualizing Software Systems as Cities". In: *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis. Banff, AB, Canada: IEEE, June 2007, pp. 92–99 (cit. on p. 45).

[WLR11] Richard Wettel, Michele Lanza, and Romain Robbes. "Software Systems as Cities: A Controlled Experiment". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE11: International Conference on Software Engineering. Waikiki, Honolulu HI USA: ACM, May 21, 2011, pp. 551–560 (cit. on pp. 46, 50, 56).

[Wic22] Maximilian Wick. "Visualisierung holistischer Code-Metriken von Code Cities in SEE". BA thesis. University of Bremen, Dec. 29, 2022 (cit. on p. 45).

[ZD85] Fred Zijlstra and L. Doorn. "The Construction of a Scale to Measure Perceived Effort". In: *Department of Philosophy and Social Sciences* (1985) (cit. on p. 53).

TODO: Bibliography doesn't work