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*

November 20, 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, November 20, 2024

Falko Galperin

Acknowledgements

TODO!

Contents

1 Introduction 1
1.1 Format 1
1.2 Motivation 2
1.2.1 Code Cities & SEE 2
1.2.2 Language Server Protocol 3
1.2.3 Integration 6
1.3 Goals & Research Questions 6
1.4 Thesis Structure 9

2 Concepts 11
2.1 SEE 11
2.1.1 Basics 12
2.1.2 Project Graph 13
2.1.3 Other Relevant Features 14
2.2 Language Server Protocol 15
2.2.1 Basics 16
2.2.2 Planned Capabilities 17
2.2.3 Unplanned Capabilities 20
2.3 Interim Conclusion 23

3 Implementation 25
3.1 Preliminary Changes 25
3.1.1 Specification Cleanup 25
3.1.2 Preparing SEE 26
3.2 Generating Code Cities using LSP 27
3.2.1 Algorithm 27
3.2.2 Augmented Interval Trees 35
3.2.3 Usage in Practice 38
3.3 Integrating LSP Functionality into Code Cities 41
3.4 Integrating LSP Functionality into Code Windows 41
3.5 Technical Evaluation 41
3.6 Interim Conclusion 41

4 User Study 43
4.1 Plan 44
4.1.1 Existing Research 45
4.1.2 VSCode 47
4.1.3 Hypotheses 48
4.2 Design 50
4.2.1 Questionnaire 50
4.2.2 Tasks 50
4.3 Results 50
4.3.1 Demographics 50
4.3.2 Correctness 50
4.3.3 Time 50
4.3.4 Usability 50
4.3.5 Comments 51
4.4 Threats to Validity 51
4.5 Interim Conclusion 51

5 Conclusion 53
5.1 Limitations 53
5.2 Future Work 53
5.3 The End 53

A List of TODOs 55

B Glossary 59

C Acronyms 61

D Attached Files 63

E List of Figures 65

F List of Tables 67

List of Listings 69

G List of Listings 69

H Bibliography 71

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.

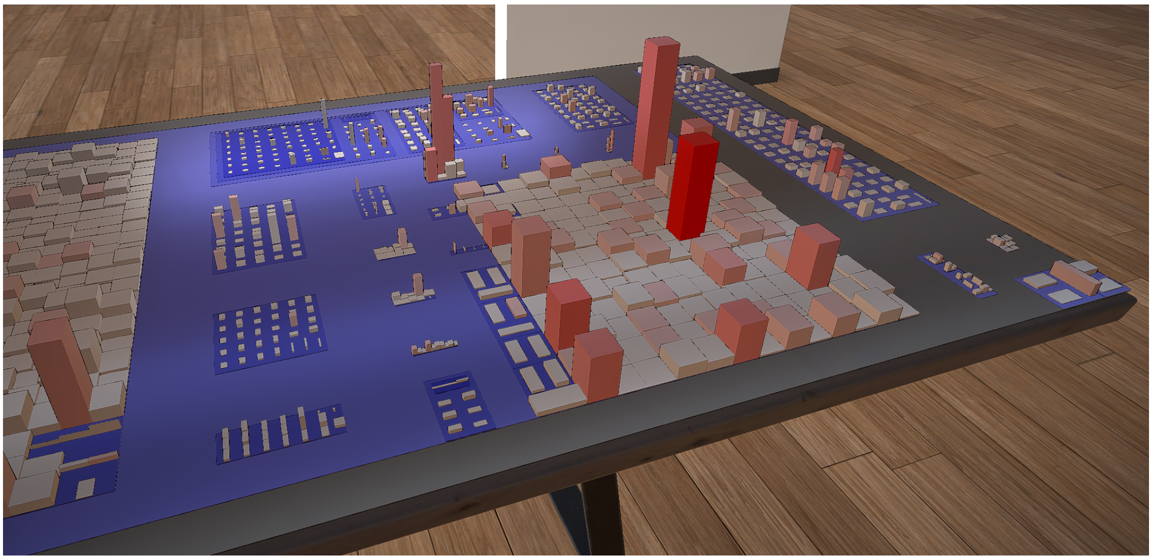


Figure 1.1: A code city visualized in SEE.

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)*

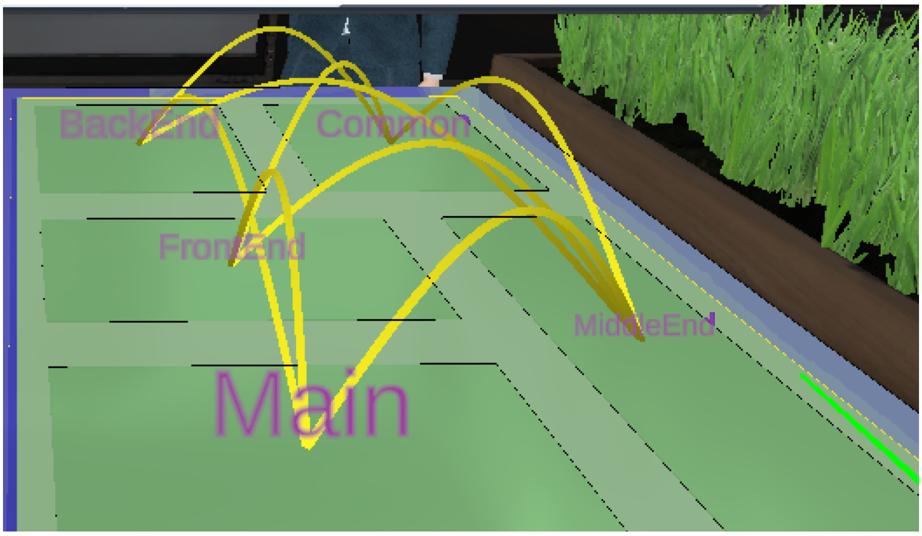


Figure 1.2: An example of what edges can look like in SEE. **TODO:** Use higher quality image.

> 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])

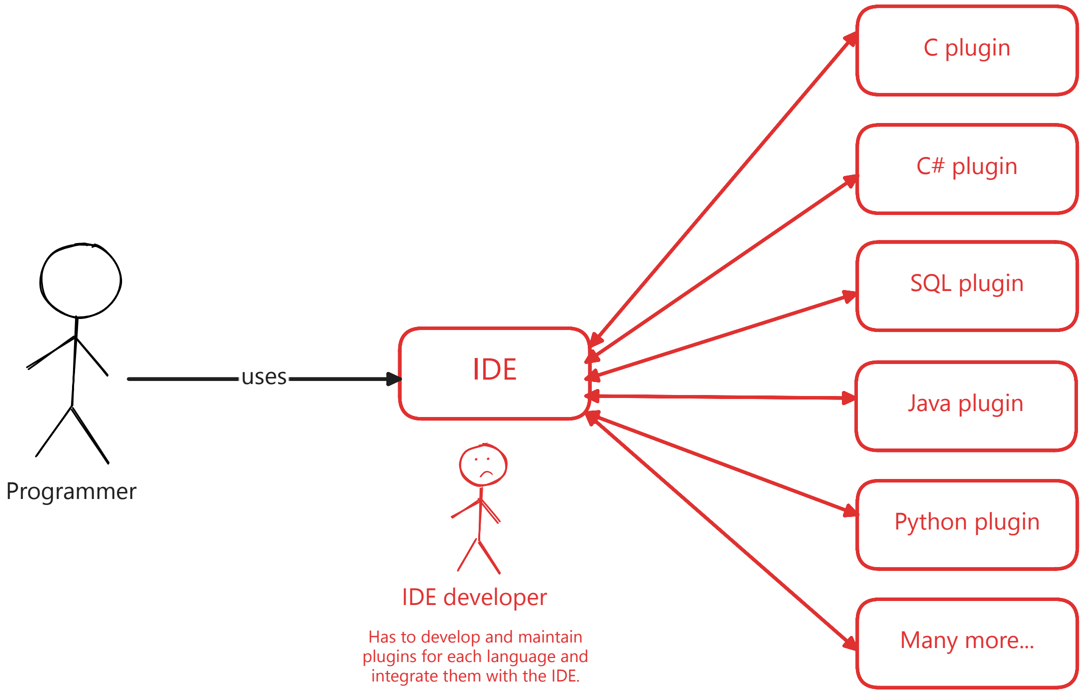
Since LSP[3] is a central component of my master's thesis, I have created a diagram in fig. 1.3 in the hope to strengthen intuitions around the motivation and use of the protocol. While the LSP specification has originally been created by Microsoft, it is by now an open-source project[4], where changes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

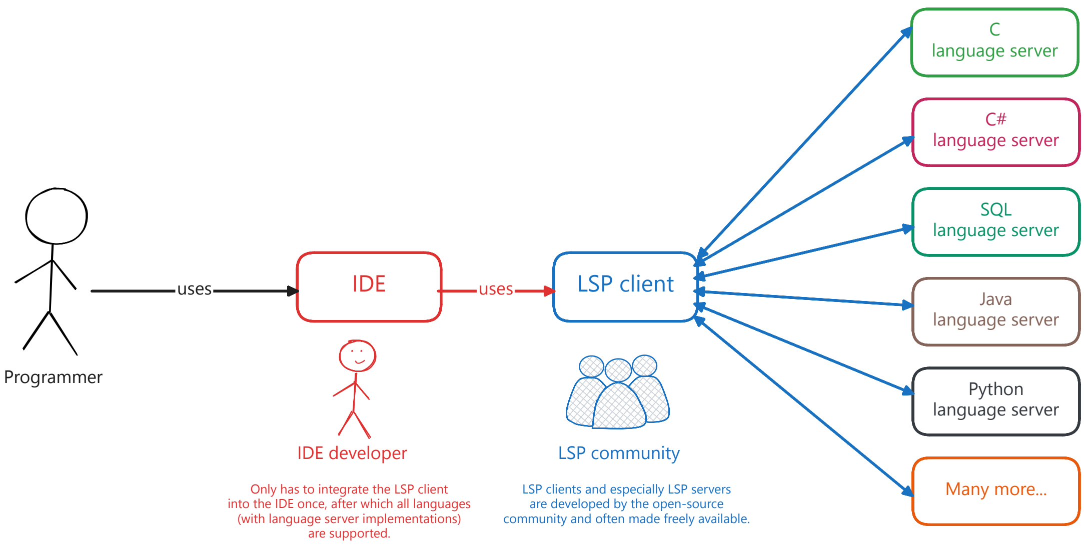
[3] Note that I will often refer to the Language Server Protocol as just "LSP" instead of "the LSP" (e. g., "IDEs use LSP") from now on, as this is how the specification [LspSpec] does it as well.

[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.

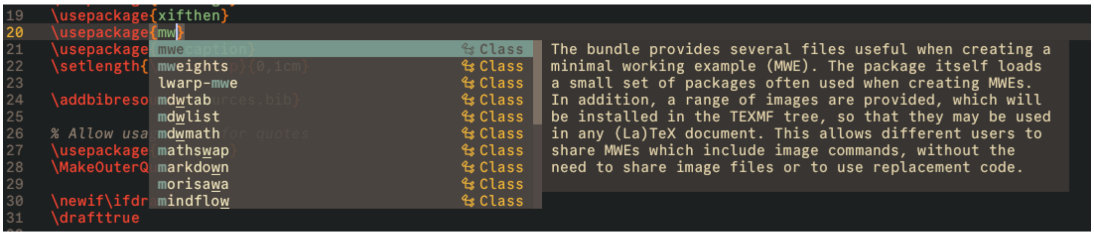


Figure 1.4: An example of the `texlab` Language Server running in NEOVIM.

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.

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

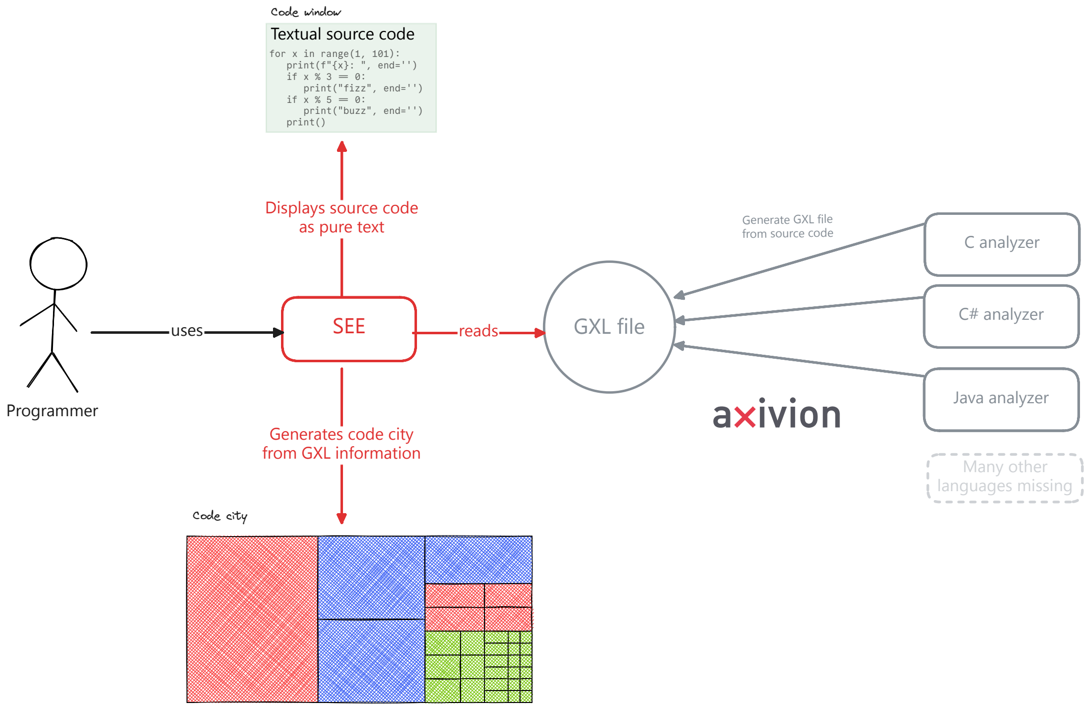
Currently, code cities in SEE are rendered by reading in pre-made Graph eXchange Language* (GXL) files, which can be created by the proprietary Axivion Suite. This approach has the disadvantage of only supporting languages supported by the Axivion Suite, as well as making regenerating cities (e. g., 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

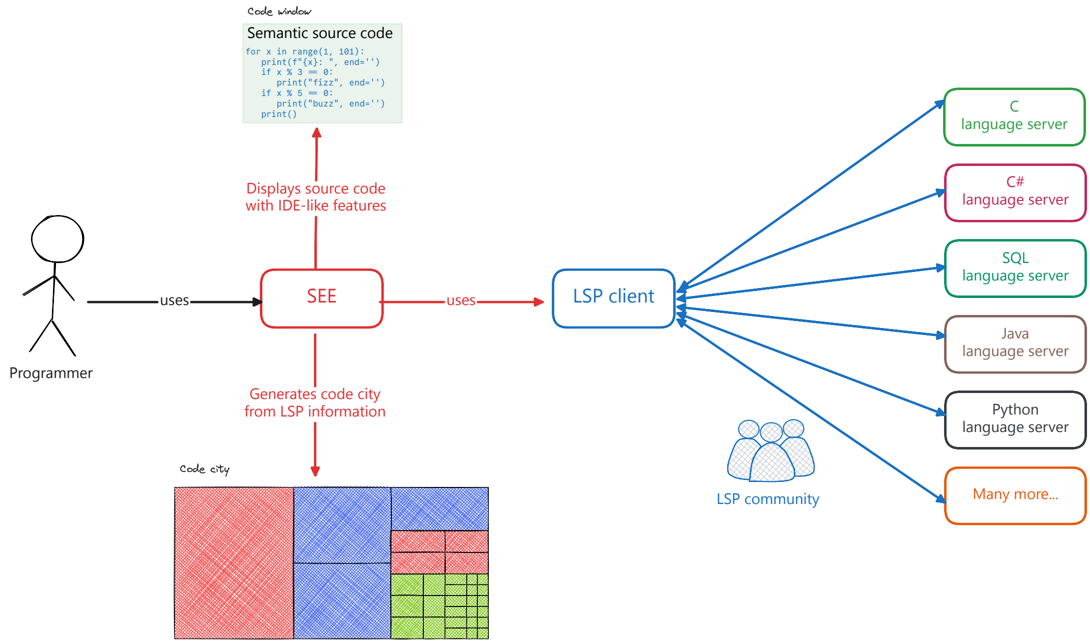
The goal of this master's thesis—as outlined in section 1.2.3—is to integrate the Language Server Protocol into SEE by making it a Language Client, then evaluate this implementation

[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.

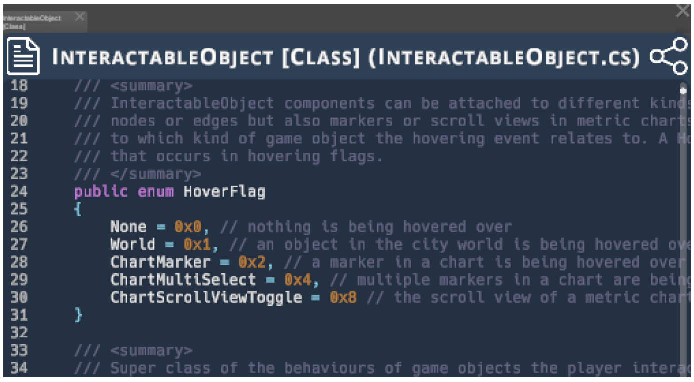


Figure 1.6: A code window.

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.

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.

TODO: In general, reword some stuff from "planning to do" ⇒ "done"

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?

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

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

1.4 Thesis Structure

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

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

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: 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, `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 input parameters (such as the aforementioned GXL file). After implementing the city-generation

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.

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 November 20, 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.

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)*)

Complex capabilities

- **Folding Range**: Returns ranges that can be collapsed in the code viewer. For example, the contents of a function could be collapsed, leaving only its signature visible. Due to the way code windows are implemented in SEE, this would increase the complexity of the implementation quite a bit.

- **Inline Value**: In debugging contexts, this supplies the contents of a variable with the purpose of displaying them inline in the Language Client, next to the variable itself. It uses the Debug Adapter Protocol* (DAP) [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.

[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 $\{a(v, \texttt{Source.Range}) \mid v \in V \land (v, \texttt{Source.Range}) \in \operatorname{dom}(a)\} \subseteq \mathcal{R}$.

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.

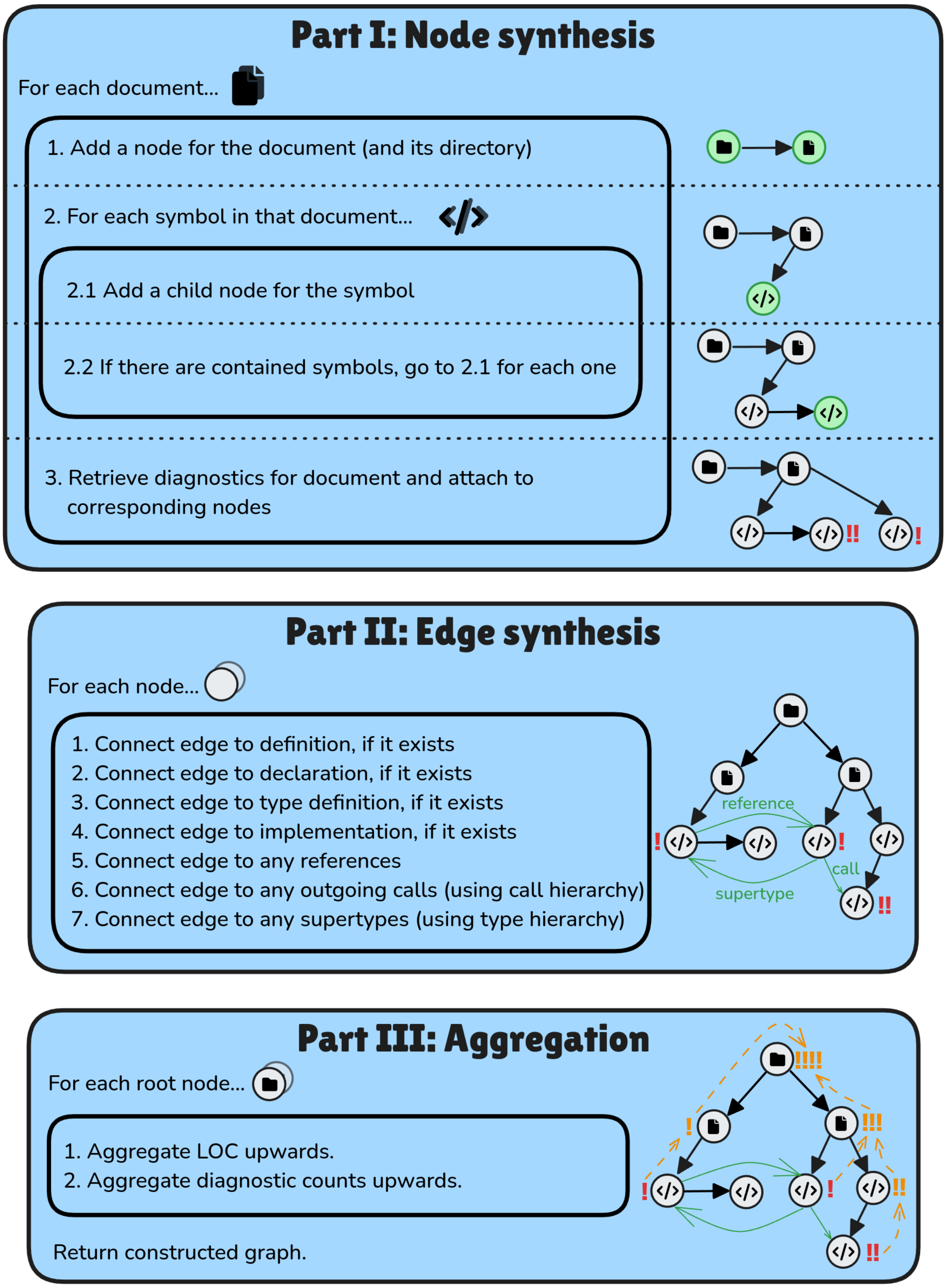


Figure 3.1: A high-level overview of the basic steps of the algorithm.

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, text 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

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

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

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

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

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

function AddSymbolNode(x ∈ S)
    v ← NewNode()
    a′ ← ∅
    a′(v, Source.Name) ← x.name
    a′(v, Source.Path) ← d
    a′(v, Type) ← x.kind
    a′(v, Deprecated) ← (deprecated ∈ x.tags)
    a′(v, Source.Range) ← x.range
    a′(v, Metric.LOC) ← e_l^{x.range} − b_l^{x.range}
    ▷ Several other similar attributes omitted here... ◁
    a′(v, HoverInfo) ← LspHover(d, x.range)
    if a′ ∉ a then                               ▷ If an isomorphic node does not already exist...
        V ← V ∪ {v}                              ▷ ...add it and handle its children.
        a ← a ∪ a′
        for all x′ ∈ x.children do
            MakeChild(AddSymbolNode(x′), v)      ▷ Recurse.
    return v

function MakeChild(v_c ∈ V, v_p ∈ V)
    ▷ 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. ◁
    if ∃e ∈ E : ℓ(e) = partOf ∧ s(e) = v_c then
        output Warning: Hierarchy is cyclic. Some children will be omitted.
    else
        AddEdge(v_c, v_p, partOf)

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

function AddEdge(v_s ∈ V, v_t ∈ V, l ∈ Σ)
    e′ ← NewEdge()
    E ← E ∪ {e′}
    s(e′) ← v_s
    t(e′) ← v_t
    ℓ(e′) ← l
```

function FindNodesByLocation($d \in D, r \in \mathcal{R}$)
  ▷ *We pick the nodes with the most specific range containing the given location.* ◁
  $\mathrm{getR}(v) = a(v, \texttt{Source.Range})$
  $N \leftarrow \{v \in V \mid a(v, \texttt{Source.Path}) = d \land b_l^r, e_l^r \in \left[b_l^{\mathrm{getR}(v)}, e_l^{\mathrm{getR}(v)}\right] \land\ b_c^r \geq b_c^{\mathrm{getR}(v)} \land e_c^r \leq e_c^{\mathrm{getR}(v)}\}$
  $N \leftarrow \arg\min_{v \in N} e_l^{\mathrm{getR}(v)} - b_l^{\mathrm{getR}(v)}$
  return $\arg\min_{v \in N} e_c^{\mathrm{getR}(v)} - b_c^{\mathrm{getR}(v)}$

function AddDirectoryNode($p \in \mathcal{A}_V$)
  if $\exists! v \in V : a(v, \texttt{Source.Path}) = p$ **then** ▷ *Check if node exists already.*
    return v ▷ *If so, just pick that one.*
  $v_p \leftarrow$ NewNode()
  $a' \leftarrow \emptyset$
  $a'(v_p, \texttt{Type}) \leftarrow$ Directory
  $a'(v_p, \texttt{Source.Path}) \leftarrow p$
  $V \leftarrow V \cup \{v_p\}$
  $a \leftarrow a \cup a'$
  $v_p^* \leftarrow$ AddDirectoryNode(GetParentDirectory(p)) ▷ *Recurse to add parent directories.*
  if $v_p \neq v_p^*$ **then**
    MakeChild(v_p, v_p^*)
  return v_p

function HandleDiagnostics($d \subset \mathcal{D}$)
  for all $c \in d$ **do**
    $V_c \leftarrow$ FindNodesByLocation($c.\texttt{path}, c.\texttt{range}$)
    for all $v \in V_c$ **do** ▷ *Save diagnostics count (grouped by severity) in all affected nodes.*
      $n \leftarrow c.\texttt{severity} + \mathrm{Count}$ ▷ *Concatenate* Count *to attribute name.*
      if $(v, n) \in \mathrm{dom}(a)$ **then**
        $a(v, n) \leftarrow a(v, n) + 1$
      else
        $a(v, n) \leftarrow 1$

function AggregateMetrics($M \subset \mathcal{A}_K$)
  for all $v \in V : \nexists e \in E : t(e) = v \land \ell(e) = \mathrm{partOf}$ **do** ▷ *Aggregate from each root node.*
    for all $m \in M$ **do**
      AggregateMetricFromRoot(m, v)

function AggregateMetricFromRoot($m \in \mathcal{A}_K, v_r \in V$)
  $V_c \leftarrow \{v \in V \mid \exists e \in E : t(e) = v_r \land s(e) = v \land \ell(e) = \mathrm{partOf}\}$ ▷ *Immediate children.*
  for all $v \in V_c$ **do**
    AggregateMetricFromRoot(m, v)
  ▷ *After the recursion above, immediate children now definitely have attribute m.* ◁
  if $(v_r, m) \notin \mathrm{dom}(a)$ **then** ▷ *We don't want to overwrite existing metrics.*
    $a(v_r, m) \leftarrow \sum_{v \in V_c} a(v, m)$

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.

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

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 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.

[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.

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.

TODO: Rewrite "the above" to something else, may not be above in printed version

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 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.

[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.

- 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.

function FindNodesByLocationEfficientEfficient($d \in D, r \in \mathcal{R}$)
  return QueryTree($H(d), r, \emptyset$)

function QueryTree($T \in \mathcal{T}, q \in \mathcal{R}, R \subseteq V$)
  $r \leftarrow a(v_T, \texttt{Source.Range})$
  if $b_l^q > l_T \lor (b_l^q = l_T \land b_c^q \geq c_T)$ **then** ▹ *Range is to the right of all nodes in this subtree.*
    return R
  if $q \subseteq r$ **then** ▹ *Range is contained in this node, but we want only minimal fits.* **TODO:** So how much does it help in practice? Reference tech eval here
    $m \leftarrow \{v \in R \mid r \lesssim a(v, \texttt{Source.Range})\}$
    if $|m| > 0$ **then** ▹ *This range is smaller than other results.*
      $R \leftarrow (R \setminus m) \cup \{v_T\}$
    else if $\forall v \in R : a(v, \texttt{Source.Range}) \simeq r$ **then** ▹ *Other ranges are equally minimal.*
      $R \leftarrow R \cup \{v_T\}$
    ▹ *Otherwise, r is not minimal, so we don't add the node.* ◃
  $R \leftarrow$ QueryTree(λ_T, q, R)
  if $e_l^q \geq b_l^r \land (e_l^q \neq b_l^r \lor e_c^q > b_c^r)$ **then** ▹ *Range could be contained in right subtree.*
    $R \leftarrow$ QueryTree(ρ_T, q, R)
  return R

[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.

```
|public class Example {
  |const char someValue = 'A'|;

  |public static long pow(|int num|) {
    |long result = num * num|;
    return result;
  }|
}|
```

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

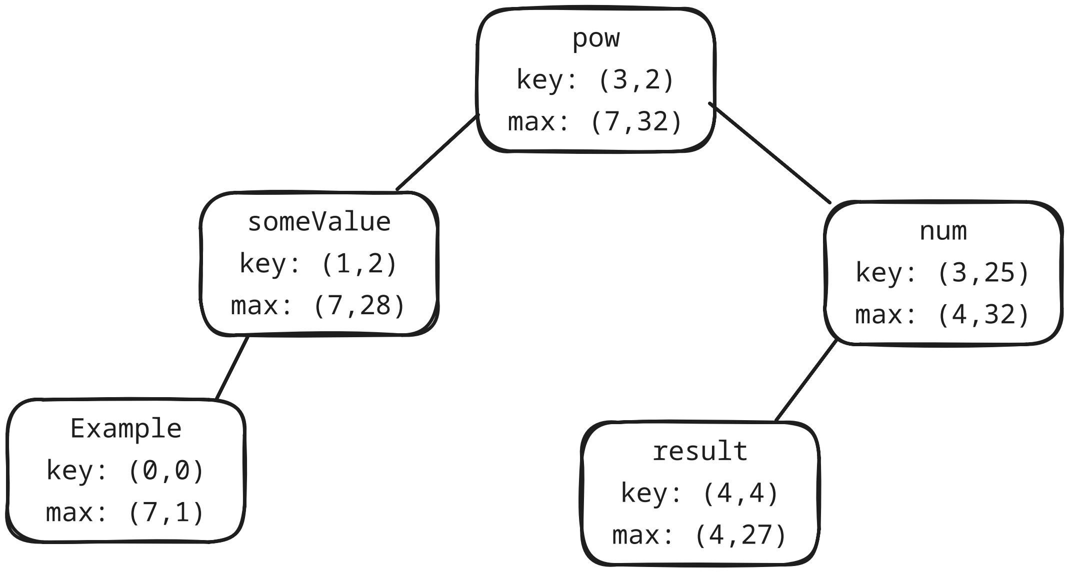


Figure 3.2: An augmented interval tree using a k-d tree, representing the elements in listing 3.1.

4. $e_l^q \ngeq b_l^r$ 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.

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.

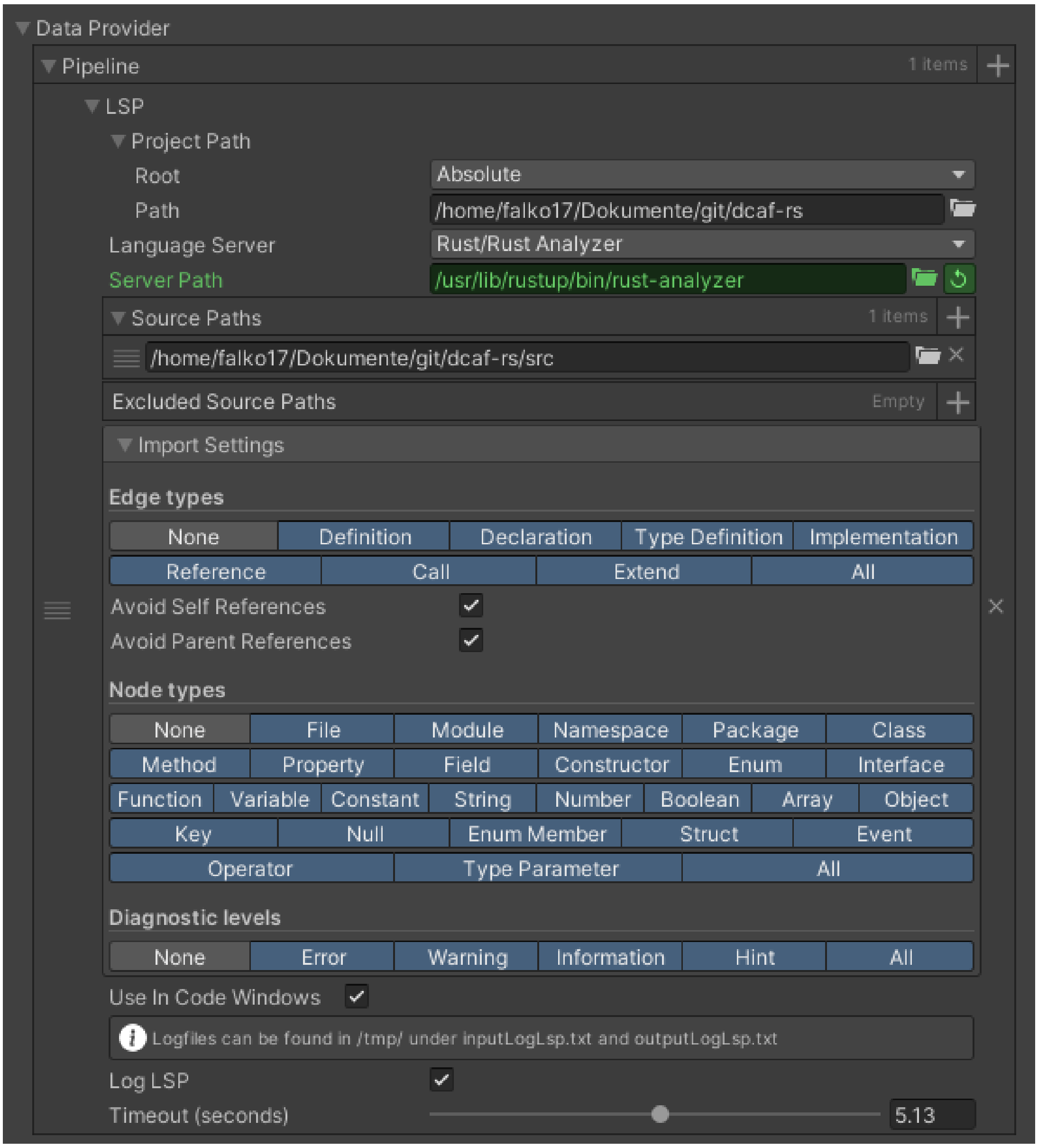


Figure 3.3: Unity Editor UI for the LSP graph provider

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 in 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. 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.

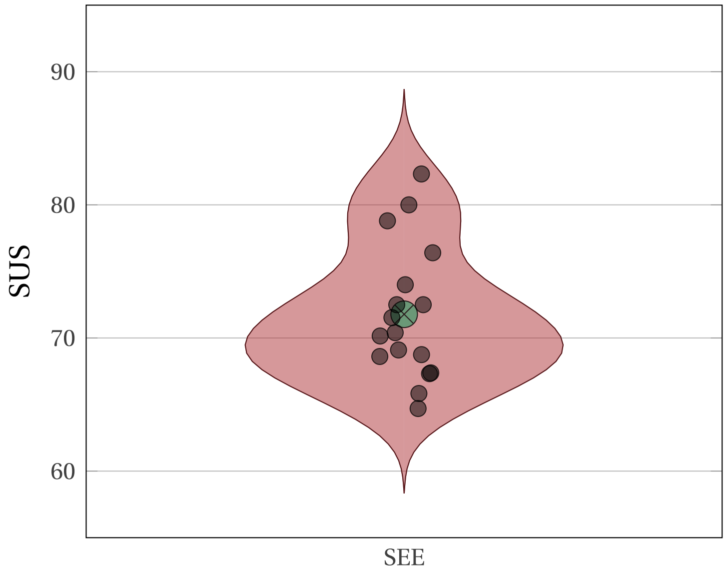


Figure 4.1: SUS results for SEE across various studies [Wag20; Gae21; Gal21; Mas20; Döh20; Kip20; Wic22; Kra24; Ble22; Wei21; Boh20; Roh24; Smi21; Sch22; Abu21; Roh21].

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[31]'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.

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,

[31] 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 (which our own study will also fall into) includes ones by Khaloo et al. [Kha+17], Rohlfing [Roh24], and Kipka [Kip20]. The results of their experiments, and all others cited here, are listed in the color-coded table 4.1.

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

4.1.2 VSCode

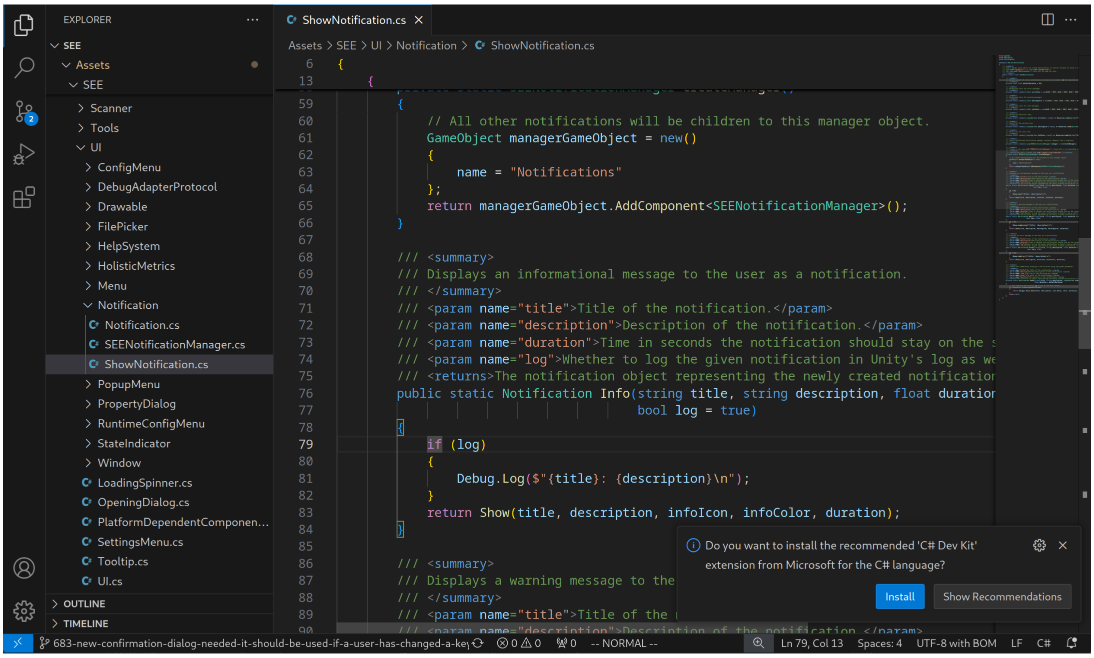


Figure 4.2: Screenshot of the main UI of VSCode.

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.

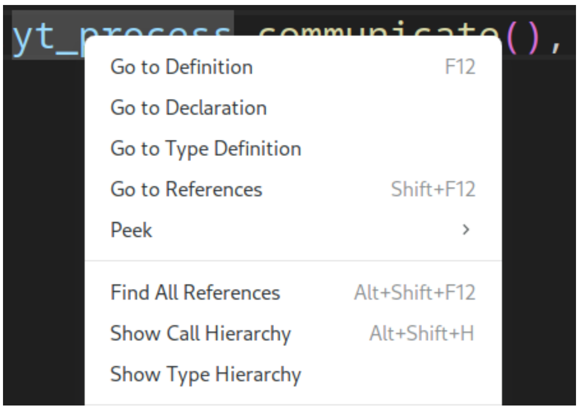


Figure 4.3: Screenshot of (the beginning of) VSCode's context menu for code identifiers.

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, 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[32] and can be sorted by any column as well as searched.

4.1.3 Hypotheses

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

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

[32] https://docs.google.com/spreadsheets/d/1Z2AQDk2-XeVBB1kAtcpc18mFkPI5ZSsSz_y0wS4pQ88 *(last access: 2024-11-20)* and https://docs.google.com/spreadsheets/d/1erJZTwYtG-CQfZJPT-zt-chX_VHL9jLrVfWAWZMFvEo *(last access: 2024-11-20)*.

- *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*[33] 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.

      - *Alternative hypothesis* H_{d_1}: The ASQ score for complexity when using SEE is different when using VSCode.

   ii) We call the ASQ score for *effort*[34] 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.

      - *Alternative hypothesis* H_{e_1}: The ASQ score for effort when using SEE is different when using VSCode.

TODO: Except maybe effects!

We will use a significance level of $\alpha = 0.05$ for all of our tests.

[33] Note that a higher score here means a lower amount of complexity.
[34] 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

TODO!

TODO: Refer back to Wettel's wish list

4.2.1 Questionnaire

TODO!

4.2.2 Tasks

TODO!

4.3 Results

TODO!

4.3.1 Demographics

TODO!

4.3.2 Correctness

TODO!

4.3.3 Time

TODO!

TODO: Also a section on the effect of experience (and others)

4.3.4 Usability

TODO!

4.3.5 Comments

TODO!

4.4 Threats to Validity

TODO!

4.5 Interim Conclusion

TODO!

5

Conclusion

UPON... **TODO!**

5.1 Limitations

TODO!

5.2 Future Work

TODO!

5.3 The End

TODO!

TODO: Find better title here

A

List of TODOs

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

Make sure this is centered! i

Unfinished section: () iii

Unfinished section: () v

Listings appear twice in TOC viii

Convert diagrams to TikZ 1

Maybe rewrite. Could start by going over languages in general, giving spell check as example. 1

Use higher quality image. 4

Maybe remove the next part if already mentioned before. 8

In general, reword some stuff from "planning to do" ⇒ "done" 8

This is rather vague. Is that alright or do I need to operationalize here? 9

Back this up with sources 11

Refer back here later, noting how LSP can generate code cities for each of these use cases 12

is this correct? And put source here 12

Needs a 3D SEE screenshot, esp. for edges 12

add 2D screenshot 12

Source for 3D/game/VR being easier to navigate/understand 12

link the video 12
A lot of "represents" here 13
Give examples of software projects here to illustrate graph usage? 14
Show screenshot of context menu (pre-LSP) 14
Provide screenshot of editor 14
Show screenshot of erosion icons 15
More precise reference 18
Inconsistent use of should/will in this section 19
When? 19
Replace diagram sketch with TikZ picture. 27
Convert all software mentions to biblatex @software type. 34
Put in examples or more concrete estimation of how much more edges there are than nodes. 34
Do actual comparison between two ways of finding target nodes. Also mention example times from eval section at end. 34
Rewrite "the above" to something else, may not be above in printed version . . . 35
So how much does it help in practice? Reference tech eval here 36
Convert to TikZ diagram and use colors from listing! 38
Link to video again here 40
Unfinished section: Integrating LSP Functionality into Code Cities (3.3) 41
Unfinished section: Integrating LSP Functionality into Code Windows (3.4) . . . 41
Unfinished section: Technical Evaluation (3.5) 41
Unfinished section: Interim Conclusion (3.6) 41
Except maybe effects! 49
Unfinished section: Design (4.2) 50
Refer back to Wettel's wish list 50
Unfinished section: Questionnaire (4.2.1) 50

Unfinished section: Tasks (4.2.2) 50
Unfinished section: Results (4.3) 50
Unfinished section: Demographics (4.3.1) 50
Unfinished section: Correctness (4.3.2) 50
Unfinished section: Time (4.3.3) 50
Unfinished section: Usability (4.3.4) 50
Also a section on the effect of experience (and others) 50
Unfinished section: Comments (4.3.5) 51
Unfinished section: Threats to Validity (4.4) 51
Unfinished section: Interim Conclusion (4.5) 51
Unfinished section: Conclusion (5) 53
Unfinished section: Limitations (5.1) 53
Unfinished section: Future Work (5.2) 53
Unfinished section: The End (5.3) 53
Find better title here 53
Use higher quality image. 65
Bibliography doesn't work 75

B

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.

D Attached Files

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

List of Figures

1.1 A code City visualized in SEE. . . . 3

1.2 An example of what edges can look like in SEE. **TODO:** Use higher quality image. . . . 4

1.3 An illustration of how LSP can help simplify IDE development. . . . 5

1.4 An example of the `texlab` language server running in NEOVIM. . . . 6

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. . 7

1.6 A code window. . . . 8

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)*) . 22

3.1 A high-level overview of the basic steps of the algorithm. . . . 28

3.2 An augmented interval tree using a k-d tree, representing the elements in listing 3.1. . . . 38

3.3 Unity Editor UI for the LSP Graph provider 39

4.1 SUS results for SEE across various studies [Wag20; Gae21; Gal21; Mas20; Döh20; Kip20; Wic22; Kra24; Ble22; Wei21; Boh20; Roh24; Smi21; Sch22; Abu21; Roh21]. 45

4.2 Screenshot of the main UI of VSCode. . . . 47

4.3 Screenshot of (the beginning of) VSCode's context menu for code identifiers. 47

List of Tables

2.1 LSP capabilities that will be integrated into SEE as part of this thesis. 23

3.1 All submitted pull requests done as part of this thesis. 41

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. 46

List of Listings

2.1. Example specification of request and response objects for the Hover capability. 16

3.1. Example C# source code with demarcated symbol ranges. 37

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).

[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 p. 45).

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

[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).

[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).

[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).

[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).

[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).

[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).

[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).

[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 (4 Dec. 1976), pp. 308–320 (cit. on p. 12).

[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).

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

[Roh24] Ferdinand Rohlfing. "Debug Adapter Protocol in SEE". BA thesis. University of Bremen, Aug. 2024 (cit. on pp. 45, 46).

[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 p. 46).

[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 p. 46).

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

TODO: Bibliography doesn't work