

University of Bremen

TODO: Make sure
this is centered!

figures/unibremen.pdf

Faculty 3 — Mathematics & Computer Science

Master's Thesis

Building Code Cities using the Language Server Protocol — *DRAFT* —

Falko Galperin

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

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

September 18, 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, September 18, 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	4
1.2.3	Integration	7
1.3	Goals & Research Questions	7
1.4	Thesis Structure	9
2	Concepts	11
2.1	SEE	11
2.1.1	Basics	12
2.1.2	Project Graph	12
2.1.3	Other Relevant Features	12
2.1.4	Literature	13
2.2	Language Server Protocol	13
2.2.1	History	13
2.2.2	Basics	13
2.2.3	Capabilities	13
2.2.4	Language Servers	13
2.2.5	Language Clients	13
2.3	Interim Conclusion	14
3	Implementation	15
3.1	Preliminary Changes	15
3.1.1	Specification Cleanup	15
3.1.2	Preparing SEE	15
3.2	Generating Code Cities using LSP	15
3.3	Integrating LSP Functionality into Code Cities	15
3.4	Integrating LSP Functionality into Code Windows	16
3.5	Technical Evaluation	16
3.6	Interim Conclusion	16

4	Evaluation	17
4.1	Plan	17
4.2	Structure	17
4.2.1	Questionnaire	17
4.2.2	Tasks	17
4.3	Results	17
4.4	Threats to Validity	18
4.5	Interim Conclusion	18
5	Conclusion	19
5.1	Limitations	19
5.2	Future Work	19
5.3	The End	19
A	List of TODOs	21
B	Glossary	25
C	Acronyms	27
D	Attached Files	29
E	List of Figures	31
F	List of Tables	33
G	Bibliography	35

1

Introduction



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

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

1.1 Format

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

New Term: LOC (Lines of Code)

The number of lines in a source code file.

- **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 section, 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. 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 platforms¹, 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, relations that describe which component belongs to which other component. These are

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

figures/SEE_screenshot.png

Figure 1.1: A code city visualized in SEE.

New Term: GXL (Graph eXchange Language)

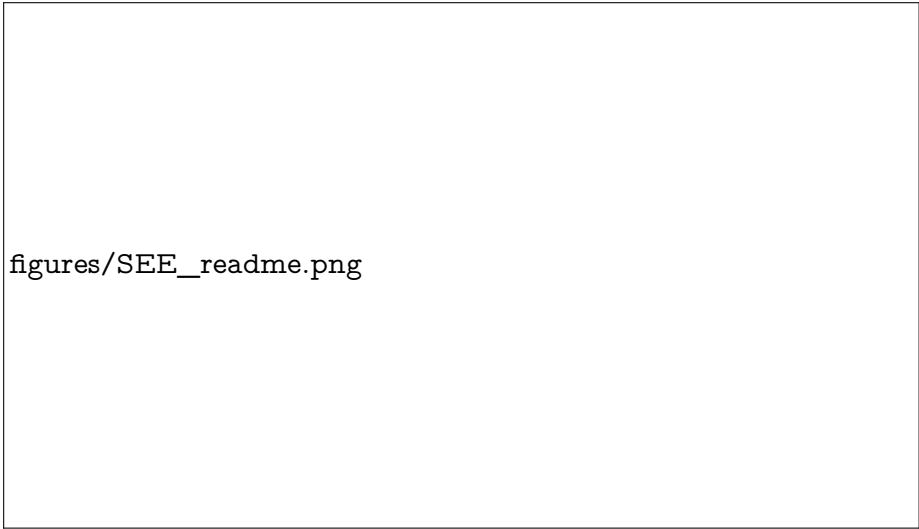
A file format for graphs, used in SEE for representing dependency and hierarchy graphs of software projects.

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 project² 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.

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 *Integrated Development Environment* (IDE)—for

²<https://github.com/spotbugs/spotbugs> (last access: 2024-09-11)



figures/SEE_readme.png

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

New Term: *IDE* (Integrated Development Environment)

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

example, quickly displaying documentation for a given component by hovering over it is not supported. This is where the Language Server Protocol can help.

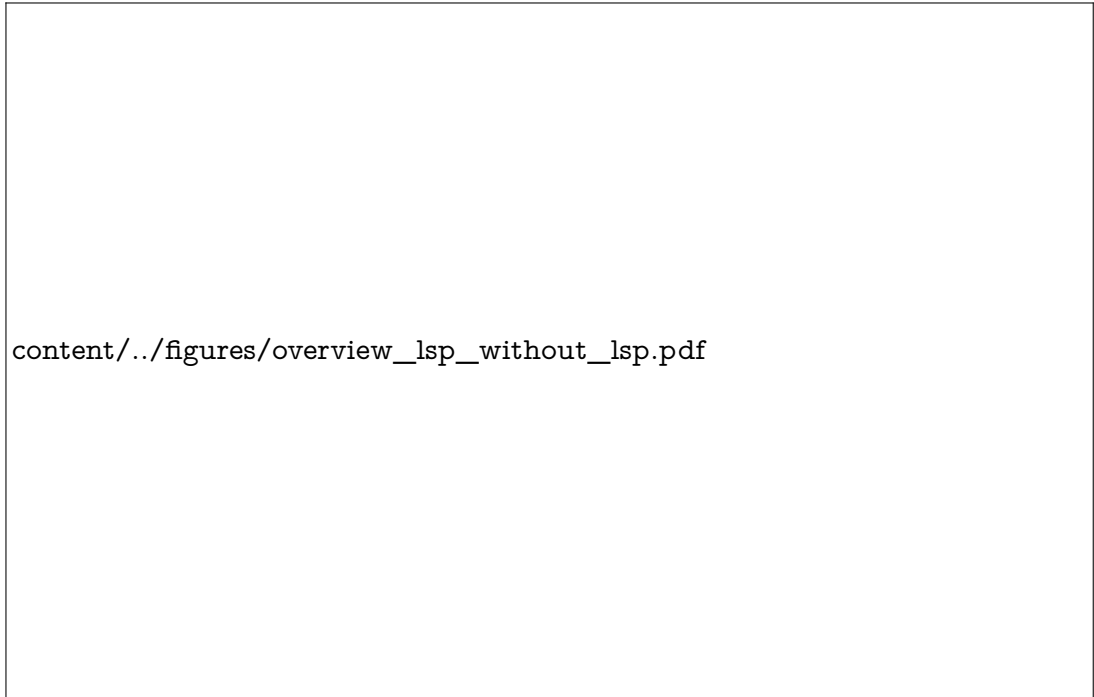
1.2.2 Language Server Protocol

As stated on its website:

Adding features like auto complete, go to definition, or documentation on hover for a programming language takes significant effort. Traditionally[,] this work had to be repeated for each development tool, as each tool provides different APIs for implementing the same feature.

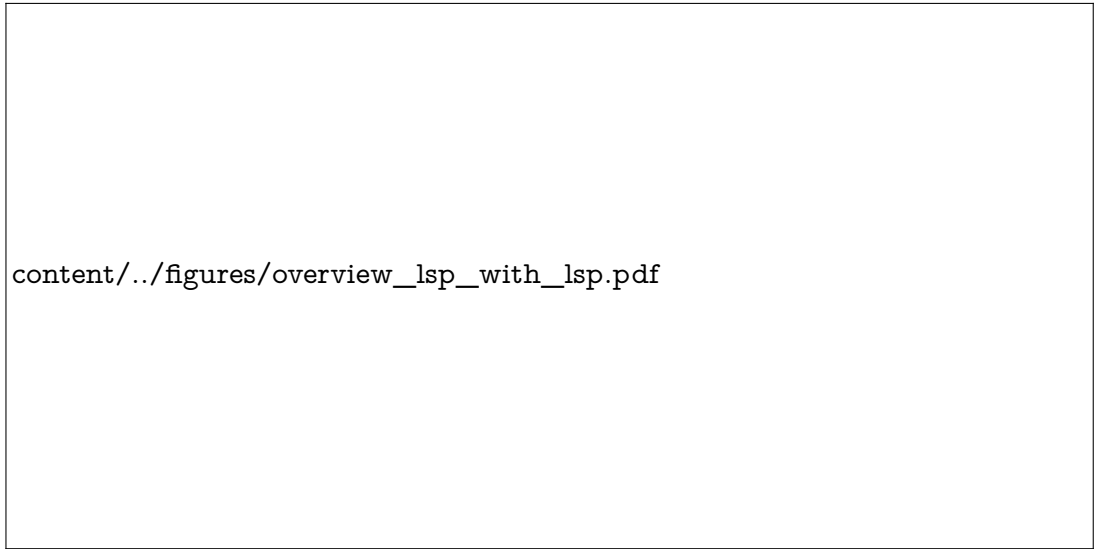
A *Language Server* is meant to provide the language-specific smarts and communicate with development tools over a protocol that enables inter-process communication.

The idea behind the *Language Server Protocol* (LSP) is to standardize the protocol for how such servers and development tools communicate. This way, a single Language Server can be re-used in multiple development tools, which in turn can support multiple languages with minimal effort. ([LSPSpec])



```
content/../figures/overview_lsp_without_lsp.pdf
```

(a) IDE development without LSP.



```
content/../figures/overview_lsp_with_lsp.pdf
```

(b) IDE development with LSP.

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

New Term: *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.

content/../../figures/mwe.png

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

As LSP³ 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 use of the protocol. While the LSP specification has originally been created by Microsoft, it is by now an open-source project⁴, 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 features, such as autocomplete. To give a specific example of what an LSP capability might look like in practice, the texlab⁵ 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*.

³Note 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.

⁴Available at <https://github.com/Microsoft/language-server-protocol> (last access: 2024-09-11).

⁵<https://github.com/latex-lsp/texlab> (last access: 2024-09-12)

1.2.3 Integration

TODO: Motivate integration

1.3 Goals & Research Questions

The goal of this master’s thesis—as outlined in section 1.2.3—is to integrate the Language Server Protocol into SEE by making it a Language Client, then evaluate this implementation by comparing it with traditional IDEs in a user study. To this end, the main contribution is a way of generating code cities using the Language Server, where all the information obtainable by relevant LSP capabilities should be manifested⁶ in the city in a suitable way. This is an unintended (or at the very least, unusual) use of LSP and may require some experimentation.

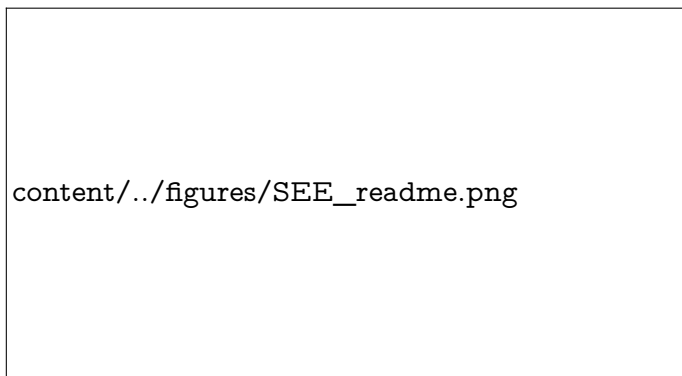


Figure 1.6: A code window.

Apart from the code cities, SEE also provides the so-called *code windows*, in which the source code of a specific component can be viewed in a similar way as in an IDE. This can be seen in fig. 1.6. An additional goal of this master’s thesis is to enhance the functionality of the code windows by implementing more IDE-like behavior into it (e. g., allowing users to go to

a variable’s declaration, or displaying diagnostics inline) using the Language Server.


It should be noted that any LSP capabilities which involve modifying the underlying software project is out of scope for this master’s thesis. Rewriting the code windows to be editable (in a way that distributes the edits over the network) is complex enough to warrant its own thesis [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 ?? for a full list of LSP capabilities which I plan to implement.

Additionally, I will not implement the C# interface to the LSP (i. e., translating between C# method calls and *JavaScript Object Notation—Remote Procedure Call* (JSON-RPC) calls).

⁶Since 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 UI.

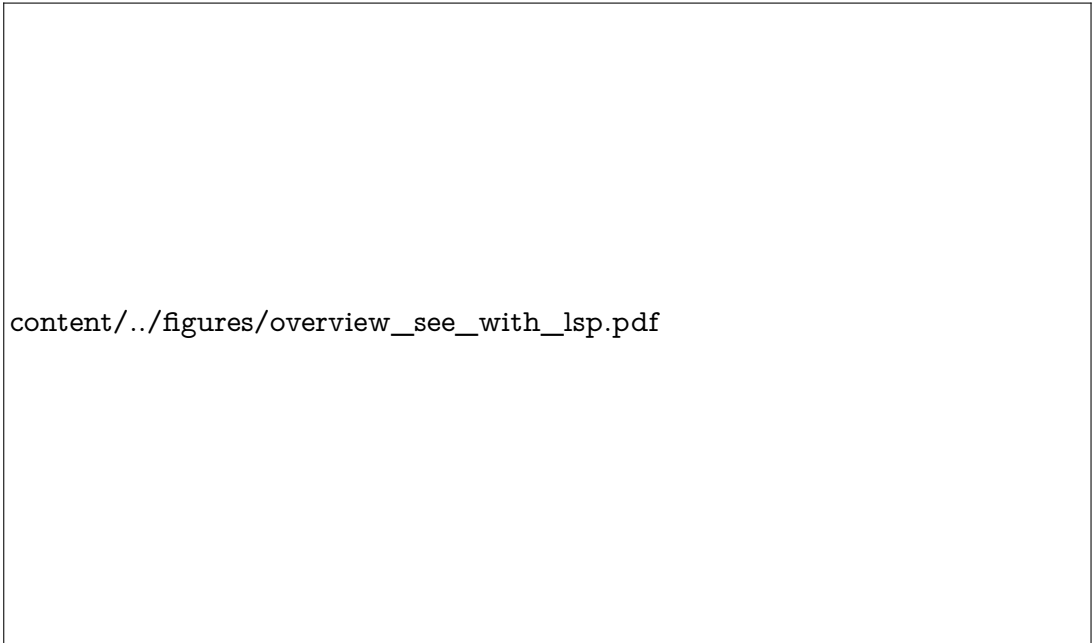
TODO: Maybe remove the next part if already mentioned before.

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



content/../figures/overview__see__without__lsp.pdf

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



content/../figures/overview__see__with__lsp.pdf

(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.

New Term: *JSON-RPC* (JavaScript Object Notation—Remote Procedure Call)

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).

There already exist well-made interfaces for this purpose⁷, and the focus of the thesis will be on the integration of the protocol's *data* into SEE's code cities and code windows, not the integration of the protocol itself.

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

- Integrating an LSP framework into SEE and allowing users to manage Language Servers from within SEE
- Making SEE a Language Client, such that:
 1. Code cities can be generated directly from source code directories, using Language Servers that SEE interfaces with,
 2. code windows gain “read-only” IDE-like functionality, covering behavior of capabilities listed in ??, 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 to generate code cities using the Language Server Protocol?

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

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

1.4 Thesis Structure

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

⁷Such 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)

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

Concepts



BEFORE tackling the implementation, we need to take a closer look at the concepts central to this thesis, 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:

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

TODO: Back this up with sources

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

New Term: *Code Smell*

Certain structures in source code that make it apparent that a refactoring is in order [Fow20].

New Term: *Reflexion Analysis*

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

SEE is an open-source project, currently hosted at GitHub¹. 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. Finally, we will take a look at some of the literature regarding code cities in general.

2.1.1 Basics

TODO!

TODO: Also provide screenshots here

2.1.2 Project Graph

TODO!

2.1.3 Other Relevant Features

TODO!

¹<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.)

2.1.4 Literature

TODO!

2.2 Language Server Protocol

TODO!

2.2.1 History

TODO!

2.2.2 Basics

TODO!

JSON-RPC **TODO!**

Lifecycle **TODO!**

2.2.3 Capabilities

TODO!

2.2.4 Language Servers

TODO!

2.2.5 Language Clients

TODO!

2.3 Interim Conclusion

TODO!

3

Implementation



TODO!

3.1 Preliminary Changes

TODO!

3.1.1 Specification Cleanup

TODO!

3.1.2 Preparing SEE

TODO!

3.2 Generating Code Cities using LSP

TODO!

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!

4

Evaluation



TODO!

4.1 Plan

TODO!

4.2 Structure

TODO!

4.2.1 Questionnaire

TODO!

4.2.2 Tasks

TODO!

4.3 Results

TODO!

4.4 Threats to Validity

TODO!

4.5 Interim Conclusion

TODO!

5

Conclusion



TODO!

5.1 Limitations

TODO!

5.2 Future Work

TODO!

5.3 The End

TODO!

TODO: Find
better title here

A

List of TODOs



PEN 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
Convert diagrams to TikZ	1
Use higher quality image.	4
Motivate integration	7
Maybe remove the next part if already mentioned before.	7
In general, reword some stuff from “planning to do” \Rightarrow “done”	7
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
Unfinished section: Basics (2.1.1)	12
Also provide screenshots here	12
Unfinished section: Project Graph (2.1.2)	12
Unfinished section: Other Relevant Features (2.1.3)	12

Unfinished section: Literature (2.1.4)	13
Unfinished section: Language Server Protocol (2.2)	13
Unfinished section: History (2.2.1)	13
Unfinished section: Basics (2.2.2)	13
Unfinished section: JSON-RPC (2.2.2)	13
Unfinished section: Lifecycle (2.2.2)	13
Unfinished section: Capabilities (2.2.3)	13
Unfinished section: Language Servers (2.2.4)	13
Unfinished section: Language Clients (2.2.5)	13
Unfinished section: Interim Conclusion (2.3)	14
Unfinished section: Implementation (3)	15
Unfinished section: Preliminary Changes (3.1)	15
Unfinished section: Specification Cleanup (3.1.1)	15
Unfinished section: Preparing SEE (3.1.2)	15
Unfinished section: Generating Code Cities using LSP (3.2)	15
Unfinished section: Integrating LSP Functionality into Code Cities (3.3)	15
Unfinished section: Integrating LSP Functionality into Code Windows (3.4)	16
Unfinished section: Technical Evaluation (3.5)	16
Unfinished section: Interim Conclusion (3.6)	16
Unfinished section: Evaluation (4)	17
Unfinished section: Plan (4.1)	17
Unfinished section: Structure (4.2)	17
Unfinished section: Questionnaire (4.2.1)	17
Unfinished section: Tasks (4.2.2)	17
Unfinished section: Results (4.3)	17
Unfinished section: Threats to Validity (4.4)	18
Unfinished section: Interim Conclusion (4.5)	18

Unfinished section: Conclusion (5)	19
Unfinished section: Limitations (5.1)	19
Unfinished section: Future Work (5.2)	19
Unfinished section: The End (5.3)	19
Find better title here	19
Use higher quality image.	31

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.gls`. 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.gls-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.

E

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
1.6	A code window.	7
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. .	8

List of Tables

Bibliography

- [Ble22] Moritz Blecker. “Developing a Multi-user Source Code Editor for SEE”. Bachelor’s Thesis. University of Bremen, Jan. 31, 2022 (cit. on p. 7).
- [Fow20] Martin Fowler. *Refactoring – Wie Sie das Design bestehender Software verbessern*. 2nd edition. 472. mitp Verlag, 2020 (cit. on p. 12).
- [Gal21] Falko Galperin. “Visualizing Code Smells in Code Cities”. Bachelor’s Thesis. University of Bremen, Sept. 27, 2021 (cit. on p. 11).
- [LSPSpec] *Language Server Protocol*. Microsoft. Oct. 2022 (cit. on pp. 4, 6).