Building Code Cities using the Language Server Protocol

Exposé

Falko Galperin

September 10, 2024

Contents

1. Motivation **2**
1.1. SEE . . . 2
1.2. Language Server Protocol . . . 3

2. Goals **5**
2.1. Mandatory goals . . . 6
2.2. Further goals . . . 6

3. Approach **6**
3.1. Implementation . . . 7
3.1.1. Generating code cities using LSP . . . 7
3.1.2. Integrating LSP into code windows . . . 7
3.1.3. Extending code city functionality . . . 11
3.2. Evaluation . . . 11

4. Schedule **12**

A. LSP capabilities **13**

B. References **13**

1. Motivation

Visualization in general often helps facilitate the understanding of complex systems by representing them with a simplified visual model. This can be especially helpful 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 SEE—is being developed at the University of Bremen and will be introduced in the next section.

1.1. SEE

SEE (*Software Engineering Experience*) 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 within a file might correlate to the height of the corresponding building. Relationships[2] between software components, such as where components are referenced, are instead represented by edges drawn between the respective buildings. 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.

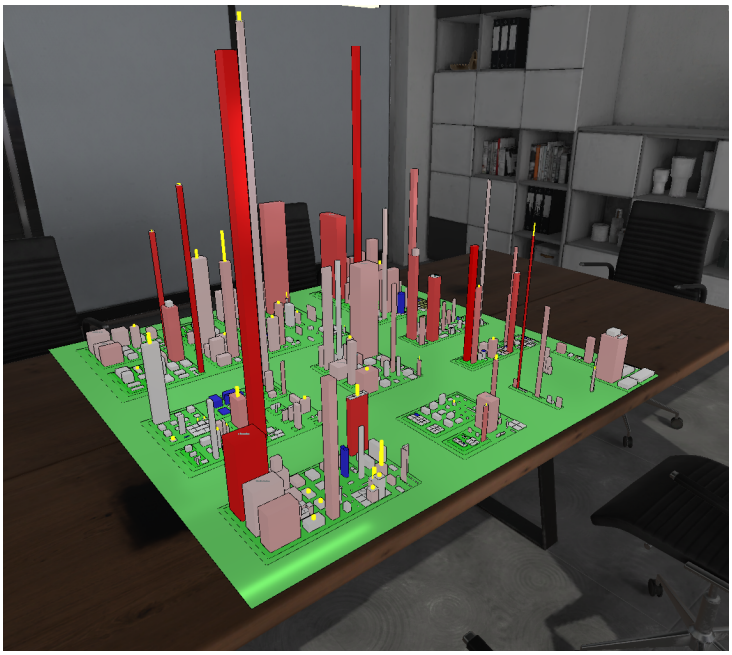


Figure 1: An example of a code city visualized in SEE.

For example, in Figure 1, we can see the source code of SEE itself 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 relative cyclomatic complexity [McC76] readily apparent: the redder a node, the higher its cyclomatic complexity. Figure 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 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,

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

[2] The exception to this are part-of relations, that is, which component belongs to which other component. These are instead visualized in SEE by buildings being nested within their corresponding "parent" building.

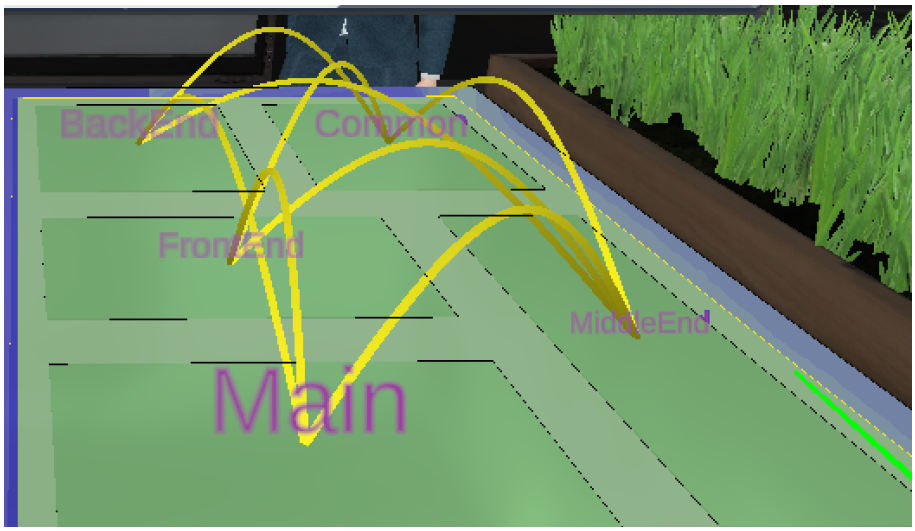


Figure 2: An example of what edges can look like in SEE.

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. Language Server Protocol

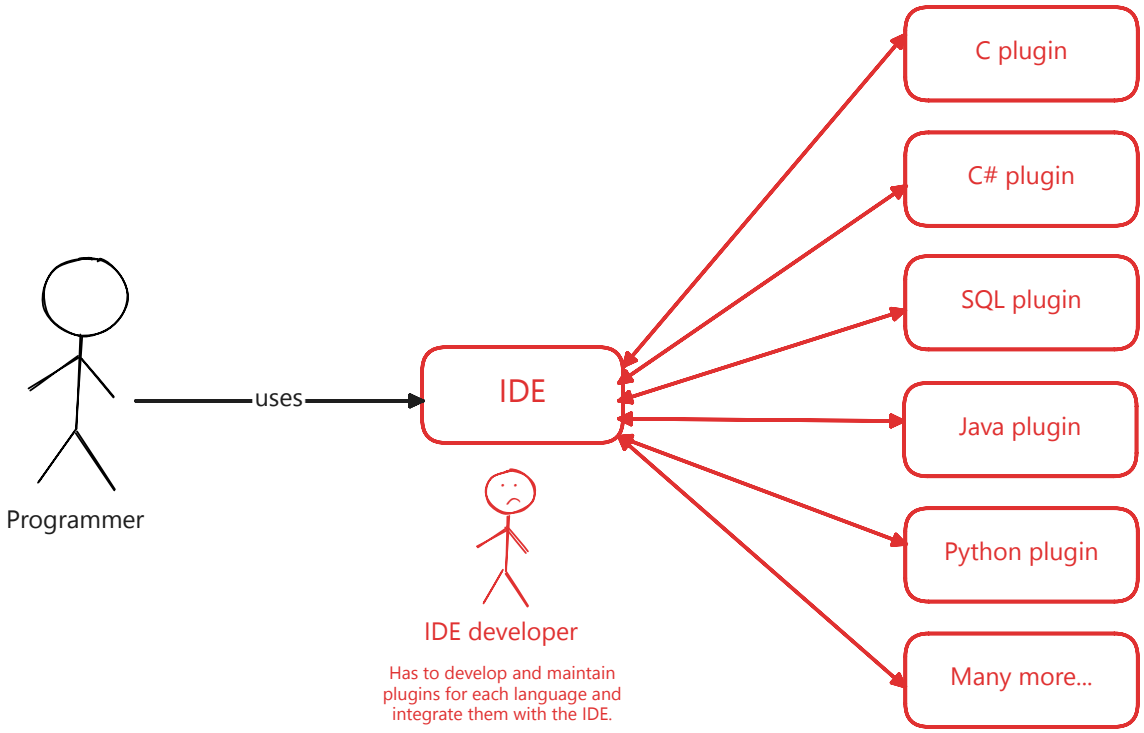
As stated on its website:

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

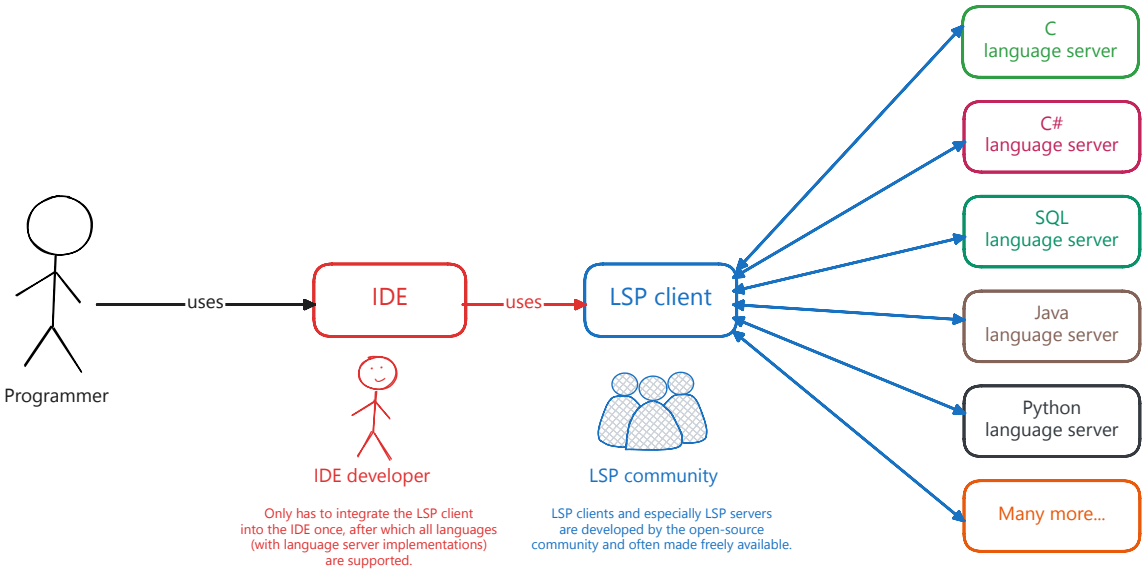
As LSP is a central component of my master's thesis, I have created a diagram in Figure 3 in the hope to strengthen intuitions around the use of the protocol. While the LSP specification has originally been created by Microsoft, it is by now an open-source project[3], where changes can be actively proposed using issues or pull requests. Apart from the specification itself, a great number of open-source implementations of language servers for all kinds of programming languages from Ada to Zig exist. A partial overview of available implementations is listed at `https://microsoft.github.io/language-server-protocol/implementors/servers/` *(last access: 2024-01-29)*.

The protocol introduces the concept of so-called *capabilities*, which define a specific set of features a given language server (and language client) support. These include navigational features, like the ability to jump to a variable's declaration, and editing-related features, such as autocomplete. To give a specific example of what an LSP capability might look like in practice, the `texlab`[4] language server for LaTeX—which I am using while authoring this very document—provides a list of available packages when one starts typing text after "`\usepackage{`". Additionally,

[3] Available at `https://github.com/Microsoft/language-server-protocol` *(last access: 2024-01-28)*.
[4] `https://github.com/latex-lsp/texlab` *(last access: 2024-01-28)*



(a) IDE development without LSP.



(b) IDE development with LSP.

Figure 3: An illustration of how LSP can help simplify IDE development.

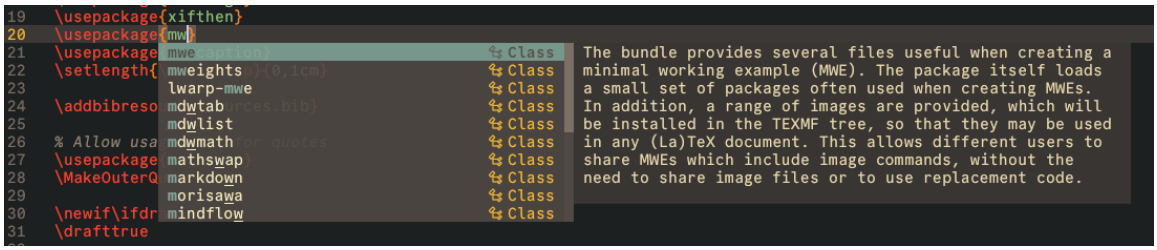


Figure 4: An example of the `texlab` *Language Server running in* Neovim *(on this exposé).*

for the currently hovered package, a short description of it is displayed. A screenshot of this behavior within the Neovim editor is provided in Figure 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, IntelliJ IDEA* (including other JetBrains IDEs), *(Neo)vim*, and *Visual Studio*.

2. Goals

The goal of this master's thesis 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 will be a way of generating code cities using the Language Server, where all the information obtainable by relevant LSP capabilities should be manifested[5] 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.

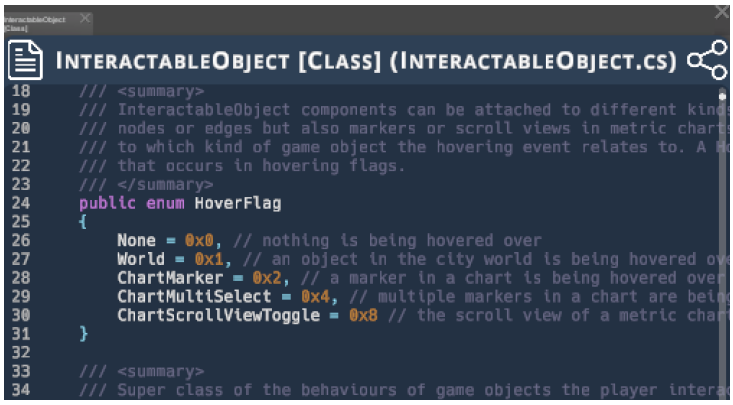


Figure 5: A code window.

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 Figure 5. 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 LSP capabilities that would then become useful to implement. As such, only LSP capabilities that are "read-only" (i.e., 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 Appendix A for a full list of LSP capabilities which I plan to implement.

[5] 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 UI.

Additionally, I will not implement the C# interface to the Language Server Protocol (i.e., translating between C# method calls and JSON-RPC calls). There already exist well-made interfaces for this purpose[6], 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.

2.1. Mandatory goals

In short, I plan to fulfill all of the following goals as part of this thesis:

- Integrating an LSP SDK into SEE and allowing users to manage Language Servers from within SEE
- Making SEE an LSP 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 Appendix A, and
  3. Code cities gain similar functionality (where applicable), such as displaying relevant documentation when hovering over a node.
- Evaluating the above empirically via a user study.

2.2. Further goals

These are ideas for further goals, which probably won't be implemented as part of the master's thesis, but build on the parts described above and introduce additional features.

- Regenerating a code city automatically when the underlying source code is modified
- Offering to download and install Language Servers from within SEE
- Introducing LSP-assisted edit functionalities for both code cities and code windows
- Using the Language Server Index Format [LSIF] to build code cities for remote projects (e.g., ones hosted in remote Git repositories)
- Supporting additional LSP capabilities not included in Appendix A, such as folding ranges or inlay hints
- Integrating SEE's LSP implementation with the Debug Adapter Protocol [DAP] (e.g., to provide inline values in debugging contexts)
- Implementing a Language Server for the Axivion Suite

3. Approach

In this section, I will describe how to fulfill the above goals in terms of the implementation and subsequent evaluation.

[6]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)*

3.1. Implementation

As outlined in section 2, the implementation can be divided into three parts: Generating code cities using LSP, Integrating LSP into code windows, and Extending code city functionality. A general visualization of the approach and the concrete changes in SEE for the implementation is given in Figure 6.

3.1.1. Generating code cities using LSP

Before working on the actual code city generation, a required preliminary step is to add an LSP client component to SEE. As per the LSP specification [LSP], this component would then be responsible for both connecting to the Language Server and for managing its lifecycle, that is, starting it and shutting it down. Additionally, there should be a graphical user interface in SEE (either in-game or in the Unity editor) with which LSP servers can be added, removed, and configured. Note that it is still the user's responsibility to setup language servers by, for example, installing them via their system's package manager.

The general algorithm of how code cities—or more specifically, their underlying dependency graphs—can be generated from information obtained via LSP is specified[7] in detail in Algorithm 1. In that algorithm, we assume that as part of the input, we have a family of LSP functions which are run on the underlying software project. Accessing an attribute y on an object x returned by an LSP function has the notation `x.y`. The graphs are formalized as $G = (V, E, \alpha, s, t, \ell)$, with:

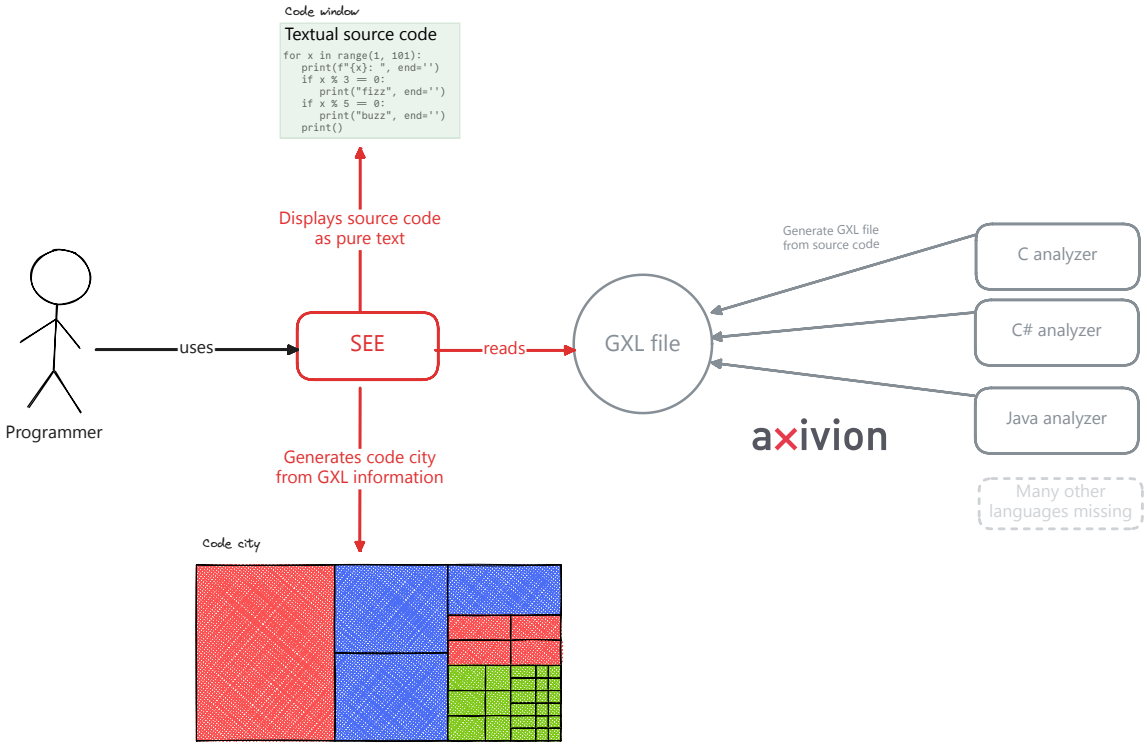
- V being a set of nodes and E being a set of edges.
- $\alpha : (V \times \mathcal{A}_K) \rightharpoonup \mathcal{A}_V$ associating nodes and an attribute name ($\in \mathcal{A}_K$) with a value ($\in \mathcal{A}_V$). Note that this is a partial function.
- $s : E \to V$ denoting the source node of 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 (e.g., to denote its type). Also, `partOf` $\in \Sigma$ such that the subgraph $(V, \{e \in E \mid \ell(e) = \texttt{partOf}\}, \alpha, s, t, \ell)$ is a polytree.

3.1.2. Integrating LSP into code windows

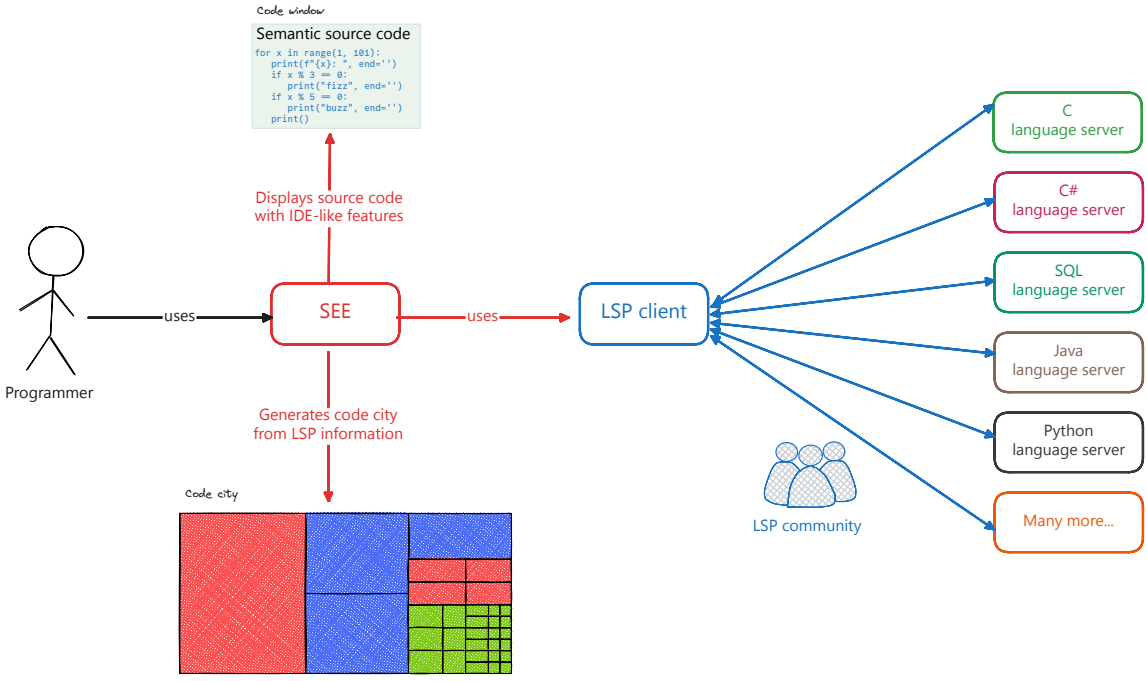
As mentioned before, code windows, which currently only display the source code without any additional navigational features, should gain "read-only" IDE-like functionality. Specifically, as listed in Appendix A, the following should be supported:

- By right clicking on a method and selecting "call hierarchy", a list of incoming and outgoing calls for this method should be displayed. Clicking on a call should open the location of that call in the code window.
  - Right clicking on a type and selecting "type hierarchy" should work similarly.
- Code ranges mentioned by diagnostics should be highlighted, while hovering above such a code range should display details, similar to the already implemented highlighting of code smells detected by the Axivion Suite [Gal21].
- Right-clicking on a suitable element and clicking on "Go to declaration" should open a list of declarations, whereas clicking on a declaration should lead to that location being opened in the code window. If there is only one declaration, the selection step should be skipped.

[7] This part is described in such detail because the approach of generating a dependency graph using only LSP information is a new and unorthodox way of using LSP, as it is primarily intended for IDEs and code editors.



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

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

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

```
1  V, E, α, s, t, ℓ ← ∅                                   ▷ Initialize empty graph components.
2  for all d ∈ D do
3      V ← V ∪ {d}                                        ▷ Each document becomes a node, with its symbols as children.
4      for all s ∈ LspDocumentSymbols(d) do
5          MakeChild(AddSymbolNode(s), d)
6  for all v ∈ V do
7      ConnectNodeVia(LspGoToDeclaration, Declaration, v)
8      ConnectNodeVia(LspGoToDefinition, Definition, v)
9      ConnectNodeVia(LspGoToTypeDefinition, TypeDefinition, v)
10     ConnectNodeVia(LspGoToImplementation, Implementation, v)
11     ConnectNodeVia(LspReferences, Reference, v)
12     if α(v, Type) = "Method" then                      ▷ We need to integrate the call hierarchy into the graph.
13         I ← LspPrepareCallHierarchy(α(v, SourcePath), α(v, SourceRange))
14         ▷ GetMatchingItem just returns the item in I with the same name, kind, and location as v. ◁
15         i ← GetMatchingItem(I, v)
16         R ← LspCallHierarchyOutgoingCalls(i)
17         ▷ GetMatchingNode returns the node in V with the same name, kind, and location as r. ◁
18         V′ ← {GetMatchingNode(r) | r ∈ R}
19         for all v′ ∈ V′ do
20             AddEdge(v, v′, OutgoingCall)
21     else if α(v, Type) = "Type" then                   ▷ We need to integrate the type hierarchy into the graph.
22         I ← LspPrepareTypeHierarchy(α(v, SourcePath), α(v, SourceRange))
23         i ← GetMatchingItem(I, v)
24         R ← LspTypeHierarchySupertypes(i)
25         V′ ← {GetMatchingNode(r) | r ∈ R}
26         for all v′ ∈ V′ do
27             AddEdge(v, v′, Supertype)
28 return (V, E, α, s, t, ℓ)
```

```
29 function AddSymbolNode(s)
30     v ← NewNode()
31     a′ ← ∅
32     a′(v, SourceName) ← s.name
33     a′(v, Type) ← s.kind
34     a′(v, SourcePath) ← d
35     a′(v, Deprecated) ← (deprecated ∈ s.tags)
36     ▷ Note that range is composed of a start and end, each consisting of a line and a column. ◁
37     a′(v, SourceRange) ← s.range
38     ▷ Several other similar attributes omitted here... ◁
39     a′(v, HoverInfo) ← LspHover(d, s.range)
40     if a′ ⊈ a then                                  ▷ If node does not already exist...
41         V ← V ∪ {v}                                 ▷ ...add it and handle its children.
42         a ← a ∪ a′
43         for all s′ ∈ s.children do
44             MakeChild(AddSymbolNode(s′), v)         ▷ Recurse.
45     return v

46 function MakeChild(v_c, v_p)
47     ▷ 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. ◁
48     if ∃e ∈ E : ℓ(e) = partOf ∧ s(e) = v_c then
49         output Warning: Hierarchy is cyclic. Some children will be omitted.
50     else
51         AddEdge(v_c, v_p, partOf)

52 function ConnectNodeVia(LspFun, type, v)
53     ▷ Function LspFun only returns locations, so we need to locate the relevant nodes first. ◁
54     for all (d, r) ∈ LspFun(a(v, SourcePath), a(v, SourceRange)) do
55         for all v′ ∈ FindNodesByLocation(d, r) do
56             AddEdge(v, v′, type)

57 function AddEdge(v_s, v_t, l)
58     e′ ← NewEdge()
59     E ← E ∪ {e′}
60     s(e′) ← v_s
61     t(e′) ← v_t
62     ℓ(e′) ← l

63 function FindNodesByLocation(doc, range)
64     ▷ We pick the nodes with the most specific range containing the given location. ◁
65     B ← {v ∈ V | a(v, SourcePath) = doc ∧ range.start.line, range.end.line ∈
       [a(v, SourceRange).start.line, a(v, SourceRange).end.line] ∧
       range.start.column ⩾ a(v, SourceRange).start.column ∧ range.end.column ⩽
       a(v, SourceRange).end.column}
66     B ← arg min_{v∈B} a(v, SourceRange).end.line − a(v, SourceRange).start.line
67     return arg min_{v∈B} a(v, SourceRange).end.column − a(v, SourceRange).start.column
```

  - This should work equivalently for "Go to definition/implementation/type definition" and "show references".

- Hovering over an item with associated hover information should lead to that information being displayed next to the cursor.
- Finally, if the Semantic Tokens capability is supported by the Language Server, syntax highlighting should be implemented using this information.

3.1.3. Extending code city functionality

There is an important difference between the way LSP functionality is integrated into code windows and the way it is integrated into code cities. For the former, actual JSON-RPC queries are sent to the Language Server whenever an LSP capability is used (as intended by the specification). However, for code cities, we need not actually do this: The relevant information for all nodes and edges has already been retrieved as part of Algorithm 1 when the code city was generated, and is stored in the form of attributes within the graph elements. As such—and importantly, because the code city is static—there is no need to introduce the additional overhead of communication with the Language Server here.

Since most of the LSP-provided "read-only" functionality has already been used as part of the code city generation, only the following three features need to be added:

- Similar context menu options as for the code windows should be added, such as "show references" or "go to declaration". Rather than opening the corresponding location in a code window, the node corresponding to that location should be highlighted in the scene instead.
- Code smell icons [Gal21] should be displayed above a node for diagnostics scoped to that node. Icons should be partitioned based on their severity or tags.
- LSP-provided hover information should be displayed when hovering above a node.

3.2. Evaluation

Finally, the integration of LSP into code cities shall be evaluated via a user study. Since the integration into the code windows is not a novel approach, this aspect of the implementation will not be evaluated, and I will purely focus on the generated code cities.

My goal is to have at least 10 participants (but ideally, at least 20) take part in the experiment to have a meaningfully representative sample. In the study, participants will attempt to solve a software engineering task (e.g., understanding a piece of software, or finding out certain information about it) either in SEE or in an LSP-enabled IDE, such as *Visual Studio Code*, or *JetBrains IntelliJ*. Ideally, participants would be able to do this remotely (both to make it easier to participate and to avoid bias from the Hawthorne effect) by downloading a copy of SEE and the IDE, then following a remote questionnaire. Aside from measuring task accuracy and time to complete the task, participants will also be given a post-task and/or post-study questionnaire to measure the subjective user experience of SEE versus the IDE.

The research question I seek to answer here is whether code cities are a suitable means to present LSP information to developers as compared to IDEs—specifically, whether code cities are better than, as good as, or worse than IDEs for this purpose on the dimensions of speed, accuracy, and usability.

4. Schedule

As of today, September 10, 2024, the implementation has been completed, while the evaluation (and master's thesis itself) still remains to be done. Below is a rough schedule outlining the remaining steps to be taken, as well as completion dates for finished milestones.

1. **May 1, 2024**: Finished code city generation via LSP as described in subsubsection 3.1.1.
2. **June 18, 2024**: Finished extending code city functionality via LSP information as described in subsubsection 3.1.3.
3. **August 1, 2024**: Finished integrating LSP into code windows as described in subsubsection 3.1.2 (and thereby finished the implementation part as a whole).
4. **August 14, 2024**: Finished preparation of the evaluation (i.e., setting up the user study).
5. **September, 2024**: Finished the evaluation as described in subsection 3.2.
6. **October, 2024**: Completed the master's thesis. After this date, only smaller details (e.g., formatting improvements) and fixes (e.g., for spelling mistakes) should be applied.
7. **November, 2024**: Finished and submitted the master's thesis.

A. LSP capabilities

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[8]	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

B. References

[Ble22] Moritz Blecker. "Developing a Multi-user Source Code Editor for SEE". Bachelor's Thesis. University of Bremen, Jan. 31, 2022. URL: https://www.szi.uni-bremen.de/wp-content/uploads/2022/07/Bachelorarbeit_-Moritz-Blecker_redfs-1.pdf (visited on 02/02/2024).

[DAP] *Debug Adapter Protocol*. Microsoft. URL: https://microsoft.github.io/debug-adapter-protocol/overview (visited on 01/31/2024).

[Gal21] Falko Galperin. "Visualizing Code Smells in Code Cities". Bachelor's Thesis. University of Bremen, Sept. 27, 2021. URL: https://github.com/uni-bremen-agst/VISSOFT2022/blob/main/BachelorThesis.pdf (visited on 02/02/2024).

[LSIF] *Language Server Index Format*. Microsoft. Sept. 2021. URL: https://microsoft.github.io/language-server-protocol/overviews/lsif/overview/ (visited on 01/31/2024).

[LSP] *Language Server Protocol*. Microsoft. Oct. 2022. URL: https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/ (visited on 01/28/2024).

[McC76] T.J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[8] This includes the *Go to Declaration*, *Go to Definition*, *Go to Implementation*, and *Go to Type Definition* capabilities.