TU WIEN Informatics

Integrating GLSP based Tooling into Visual Studio Code

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Luca Forstner
Matrikelnummer 11807455

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork
Mitwirkung: Dipl.-Ing. Dr.techn. Philip Langer

Wien, 24. Februar 2022

Luca Forstner	Dominik Bork

TU WIEN Informatics

Integrating GLSP based Tooling into Visual Studio Code

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Luca Forstner
Registration Number 11807455

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork
Assistance: Dipl.-Ing. Dr.techn. Philip Langer

Vienna, 24th February, 2022

Luca Forstner	Dominik Bork

Erklärung zur Verfassung der Arbeit

Luca Forstner

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. Februar 2022

Luca Forstner

Kurzfassung

Immer mehr IDEs und Softwareentwicklungswerkzeuge wandern ins Internet. Die Einführung des Language Server Protocol (LSP) spielt hierbei eine große Rolle. Das LSP ermöglicht eine Architektur, in der generische Editoren mehrere Sprachen unterstützen können, indem sie die sprachspezifischen Implementierungen von Sprachservern wiederverwenden. Durch das Entkoppeln der Editoren von sprachspezifischen Details wird der Entwicklungsaufwand bei der Unterstützung neuer Programmiersprachen erheblich reduziert. Mit web- und browserbasierten Modellierungswerkzeugen folgt die modellgetriebene Softwareentwicklung dem Trend ins Internet. Diese grafischen Editoren stehen jedoch vor ähnlichen Problemen wie textuelle Editoren vor der Einführung des LSP. Jedes Modellierungswerkzeug muss neue domänenspezifische Sprachen individuell unterstützen. Die Graphical Language Server Platform (GLSP) möchte dieses Problem lösen, indem sie ein Konzept für generische graphische Clients, graphische Sprachserver und ein neues graphisches Language Server Protocol bereitstellt, das die komplexen Bearbeitungsfunktionen graphischer Modellierungswerkzeuge unterstützt. Für diese Arbeit und um die Migration von Modellierungswerkzeugen ins Web voranzutreiben, haben wir eine quelloffene Bibliothek weiterentwickelt, die die Integration von GLSP-Clients und -Servern in Visual Studio Code, einem hochgradig erweiterbaren browserbasierten Code-Editor, erleichtern soll. Wir haben unseren Beitrag zur Bibliothek erfolgreich bei dem zugehörigen Eclipse GLSP Projekt eingereicht.

Abstract

More and more IDEs and software development tools are moving to the web. The introduction of the Language Server Protocol (LSP) plays a big supporting role in this movement. The LSP allows for an architecture where editors are generic and can support multiple languages by reusing the language-specific implementations from language servers. Decoupling editors from the language-specific details significantly reduces development efforts when introducing new programming languages. Naturally, model-driven software engineering followed the trend of moving to the web with the introduction of web-based modeling clients. However, these graphical editors suffer from the same difficulties textual editors had before the introduction of the LSP. Each graphical editor has to implement the support for new domain-specific languages individually. The Graphical Language Server Platform (GLSP) aims to address this problem by providing a concept for generic graphical clients, graphical language servers, and a new graphical language server protocol that supports the complex editing features of graphical modeling tools. For this thesis and to advance the move of modeling tools to the web, we further developed an open-source library to facilitate integrating GLSP clients and servers into Visual Studio Code, a highly extensible code editor that can run in a browser. We successfully submitted our contribution to the open-source Eclipse GLSP project.

Contents

CHAPTER 1

Introduction

Browser-based Integrated Development Environments (IDEs) and developer tools are rising in popularity. Even large software corporations are starting to move their workflows to the web [1]. Web-based diagram modeling tools are no exception to this hype. Based on the interest in web IDEs and browser-based tooling, we intend to explore integrating web-based modeling software into Visual Studio Code, a lightweight but highly extensible code editor that runs inside a web browser. At the core, the usage of the Graphical Language Server Platform (GLSP), a framework to build custom diagram editors in a browser-based environment, enables this integration.

The textual Language Server Protocol as maintained by Microsoft [2] has shown great success in the realm of textual development tools. Graphical modeling tools tend to suffer from similar problems that textual editing tools had before the introduction of the LSP. Supporting different graphical languages in various editing tools generates a lot of maintenance effort without a shared communication protocol.

The Graphical Language Server Platform [3] aims to fix this problem with a client-server architecture alongside a dedicated protocol for graphical Domain Specific Languages (DSLs). This protocol permits the implementation of clients that are independent of the languages they intend to support. The GLSP Protocol [4] has already been successfully integrated into various tools like Eclipse [5], Eclipse Theia [6] and with limited functionality into Visual Studio Code [7]. As a next step, existing integrations of the Graphical Language Server Platform should be developed further to investigate the protocol's efficacy and to potentially find novel ways of improving it.

For this thesis, we intend to advance the already established "glsp-vscode-integration" library [8], originally created by the maintainers of the Graphical Language Server Platform. This library can be used in the development of VS Code extensions to integrate GLSP software into VS Code extensions and thereby support graphical languages within VS Code. Our contribution aims to improve GLSP's integration into VS Code by making

it possible for users to use native VS Code features in order to interact with GLSP diagrams. We will submit any developed features as open-source contributions to the Eclipse GLSP project [9].

CHAPTER 2

Technology Overview

In this chapter, we will outline and explain the technologies used to achieve the goals of this thesis.

2.1 Visual Studio Code

Visual Studio Code (often referred to as VS Code) is an integrated development environment created by Microsoft. Most of Visual Studio Code's source code was released by Microsoft as open source, and it can be downloaded as freeware. Out of the box, the editor includes usual features for software development like syntax highlighting, debugging, and refactoring tools for a limited range of programming languages. VS Code is currently under active (open source) development and receives monthly updates.

VS Code is built on Electron, a framework to develop cross-platform desktop applications. Electron, in return, uses Chromium, an open source browser, to render the interfaces of applications. Essentially, this set of technology enables VS Code to run entirely inside a browser. The VS Code desktop client application just represents a wrapper that hosts VS Code as a browser application. Its ability to run in the browser is also why VS Code is highly relevant in the realm of web-based development tooling. There are still some differences in what is possible on the VS Code desktop application compared to VS Code on the web, mainly because regular web browsers run web pages in a sandboxed environment. Electron is only limited by what the operating system the application is running on allows. One of the differences being, for example, is file system access. While users can edit files locally at free will on the desktop application, the web client has no such capabilities because it is running in the browser. The web version of VS Code [10] is able to open and edit files in a local file system, yet not all major browsers support the required Application Programming Interface (API) [11].

In order to add additional functionality to VS Code, users can install various extensions for the application through an interface within the editor or through the Visual Studio Marketplace website [12]. VS Code extensions can add a wide selection of powerful features. Currently, the features of extensions can range from editor support for additional programming languages to whole new interfaces within VS Code to interact with remote servers.

Extensions are developed by extension authors. VS Code extensions need to be implemented in Node.js or JavaScript (often transpiled from TypeScript) and can be published by authors on the Visual Studio Marketplace. In order to implement features for their extension, extension authors can utilize the VS Code Extension API [13]. This API enables authors to interact with VS Code directly and add contribution points like menus, commands, and notification feedback.

Using the extension API, creating a diagnostic issue within the editor would look like the following:

```
1  import * as vscode from 'vscode';
2
3  const diagnosticCollection =
4      vscode.languages.createDiagnosticCollection();
5
6  diagnosticCollection.set(
7      vscode.Uri.parse('/path/to/affected/resource.txt'),
8      [
9          new vscode.Diagnostic(
10             new vscode.Range(2, 11, 2, 18), // This is the affected
                   range within the resource
11             'This is a description.',
12             vscode.DiagnosticSeverity.Error
13         )
14     ]
15 );
```

Listing 2.1: Example for creating an entry in the editor diagnostics.

After importing the VS Code API in line 1, we first create a Diagnostic Collection (line 4) which represents a grouping of diagnostics. Within this collection we create a new Diagnostic (lines 6 and 9) for a given resource (line 7). Executing this code block will display a diagnostic error in VS Code as shown in Figure 2.1.

The VS Code Extension API offers many different contribution points, which extension authors can use to influence almost all editor features. However, one particular part of the extension API is of greater interest to our proposed contribution and to the integration of a web-based modeling platform into VS Code: The Webview API.

"Webviews" make it possible for extensions to display any content inside an editor panel (usually the panels where code is edited) that could also be displayed within a web browser. With this feature, the application's extensibility is almost limitless. We will explore the Webview API's capabilities further in the next chapter.

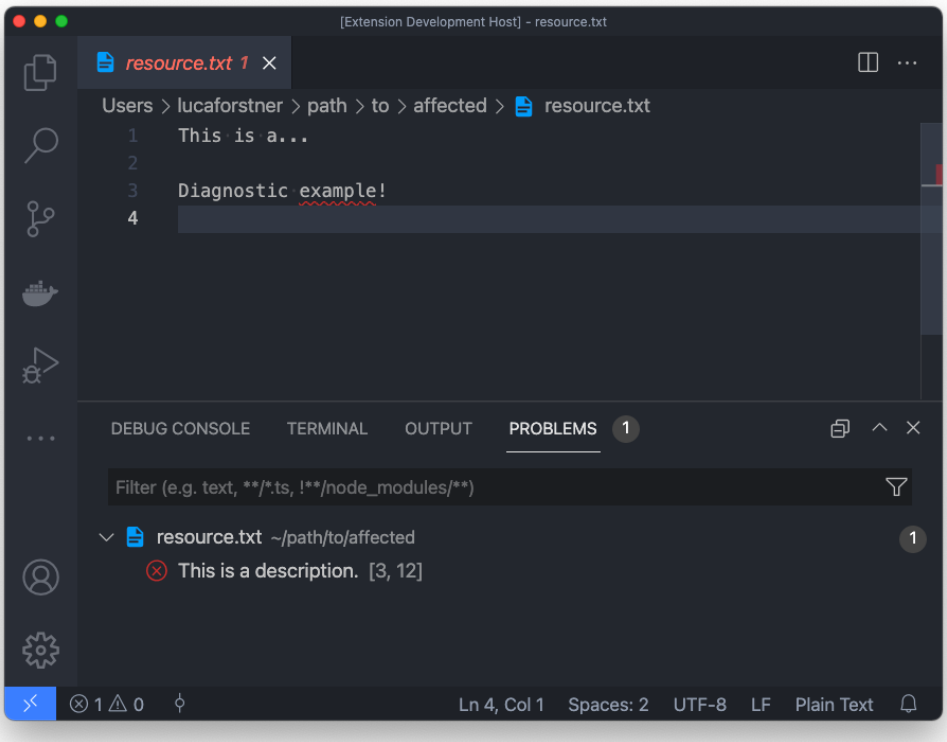


Figure 2.1: Diagnostic being displayed in VS Code.

2.2 Language Server Protocol

IDEs like VS Code aim to provide features to interact with code in novel ways that make programming faster, easier, and less error-prone. These features might include:

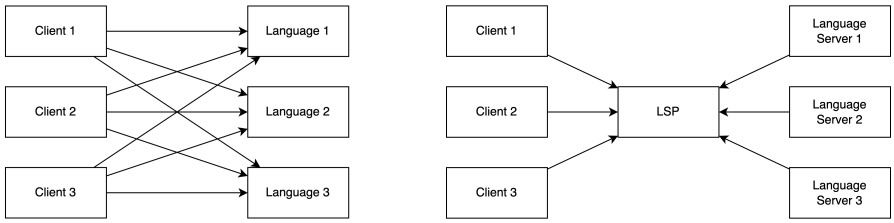
- Highlighting of sections of code that contain syntax errors. This is also known as diagnostics.
- Showing documentation for code when hovering over functions.
- Navigating the user to a variable's declaration when it is clicked.
- Advanced refactoring tools that will restructure code with the press of a button.

Features of this kind are dependent on the programming language the user uses. Variables are initialized differently from language to language, and refactoring usually cannot be directly translated between languages. Even though it would be possible for Editors to

implement those features for every programming language individually, it would represent a massive maintenance effort.

Assuming all editors would support all languages independently, $N \times M$ different implementations of those features would have to be developed and maintained (where N are the number of different programming languages, and M are the number of different IDEs). The approach of having numerous implementations introduces general development overhead, might cause inconsistencies across languages and is more likely to contain errors.

The LSP was invented to simplify IDE development by eliminating the issue of having to implement language-specific features for each editor individually. The LSP acts as the communication interface between the code editors and language servers. Its goal is to decouple the interface and features of a code editor from the logic of programming languages by having the language-agnostic code editors use the LSP to communicate with language servers. Language servers understand the syntax and the semantic concepts of a programming language and can communicate hints and instructions over the LSP to its clients (clients meaning code editors in this context). IDEs that have implemented a communication interface for the LSP can easily support any language, as long as a language server exists for that language. Having language-agnostic clients that communicate over the LSP with langue servers effectively reduces the number of needed language-specific implementations to $N + M$, as multiple different IDEs may use the same language servers.



(a) $N \times M$ different implementations through individual language support.

(b) $N + M$ implemenations in client-server architecture.

Figure 2.2: Comparison of traditional language support vs. language server architecture.

Currently, the LSP is used by the most prominent code editors on the market [14], and there exist language servers for the most used programming languages [15].

The LSP is governed and maintained by Microsoft. Discussion, research, and development happen on the GitHub Repository of the LSP. The governance by Microsoft inherently leads to the LSP being tailored towards Microsoft products (i.e., VS Code and Visual Studio). LSP features are often developed concurrently on Microsoft's editors, and the limitations and features of these editors seem to give direction to the development of the LSP.

In general, extensions are the primary method of providing support for additional languages for VS Code. VS Code provides a framework to build language servers and

clients in the form of a JavaScript library [16]. The goal of this library is to simplify the implementation of new language servers. Users of this library can implement handlers for LSP messages to build a LSP-conform language server.

VS Code Extension authors can use the "Programmatic Language Features" to implement language features without having to implement a whole language server. The API of this programmatic implementation leans heavily on the API of the library mentioned above. Here, however, extension authors write handlers for events directly triggered by VS Code instead of handlers for method calls by arbitrary clients.

2.2.1 Protocol Messages

Messages of the LSP are transmitted via a structure that conforms to the Hypertext Transfer Protocol (HTTP) semantics by requiring a header part and a content part. The header part has to include a Content-Length header that describes the length of the content part - any additional headers are optional. An extension of the JSON-RPC specification defines the structure of the content part of a message. Programs following the JSON-RPC protocol transmit data exclusively in the JavaScript Object Notation (JSON) format. This protocol can be used by any programming language, meaning that a program written in language A may call a procedure of another program written in language B. A language-agnostic remote procedure protocol is essential for having a system with interchangeable client and server components.

The LSP differentiates between three kinds of messages which the Language Server Protocol directly derives from the JSON-RPC specification:

- **Request Messages** represent a request sent by the client or the server. Whenever the client or server sends a Request Message, they expect to receive a response in the form of a *Response Message*. In return, both client and server must send a response when a Request Message is processed. Request messages must include a unique identifier and the name of the method to be invoked. Senders of a Request Message can optionally include parameters for the invoked methods.

- **Response Messages** are direct responses to Request Messages. Responses must include the identifier of the request message they are responding to. Response Messages must differentiate between whether the invocation succeeded or failed. On success, the Response Message must include a result. If the invocation failed, the Response Message must not include a result. However, it must include an error object containing general information about the error, like an error code, a message, and optional additional information. Some error codes are already reserved by the LSP and JSON-RPC protocol and can be used to indicate errors while executing the protocol.

- **Notification Messages** are messages that must not receive a response. They can be interpreted as asynchronous events that also invoke methods. The LSP also

> defines a special kind of Notification Message denoted by the prefix '$/'. These messages are messages that may not be implementable by some clients or servers. In general, these messages are used for interrupting an ongoing method invocation, which is not always possible, for example, when the server process is single-threaded and can only process one message at a time.

The LSP also defines a variety of utility interfaces and type structures that are used throughout the protocol. Utility types include, for example text document objects, which are structures used to define a range within a text document, or object interfaces which contribute directly to a language server feature, like, for example, an interface for diagnostic issues.

Under the specification of the LSP, clients are responsible for the lifecycle of language servers. It is up to the code editing tool to decide when to start or terminate a language server and how the runtime of the language server looks like.

2.2.2 Initialization Process

The client-server communication should start with a "initialize" request from the client to the server. This initialization request contains general information about the client, the environment of client and server, and the client's capabilities. The server should then answer with its capabilities.

Capabilities are a way for the server and client to advertise to each other what features they provide and what requests and events they understand. This system exists so that the components know what features to expect from each other. A client, for example, might support the functionality of jumping to a variable definition in a document while the language server the client intends to use was not configured to find the locations of variable definitions. If the client did not know of the server's inability to find variable definitions, it would send requests to the language server, asking for the location of variable definitions, without the server being able to respond. However, because the client knows of the server's capabilities, it will not send such requests in the first place.

The number of possible capabilities the client and server can register is large and almost directly relates to the code editing features the LSP intends to provide.

After the client receives the response for its initialization request, it should send an ‘initialized‘ notification. Then, the initialization handshake is complete, client and server know of their capabilities, and subsequent method calls can follow.

2.2.3 Synchronizing Document State between the Server and the Client

Language servers need a mechanism to be informed about changes to documents within a workspace. Even though language servers are allowed to access the file system directly to derive information about a project, they cannot directly access the editing buffer of

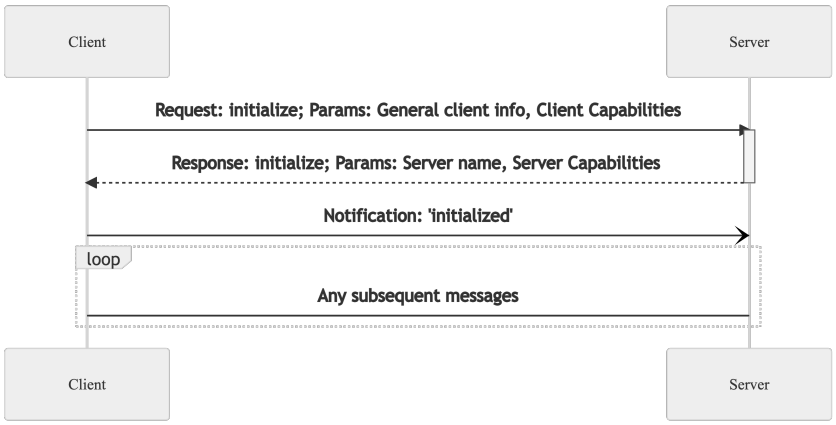


Figure 2.3: LSP Initialization Sequence

the code editor, which is where changes happen that have not been saved yet. The LSP defines methods that synchronize changes with the language server while the user edits a document. The LSP calls this process "Text Document Synchronization".

Figure 2.4: LSP Text Document Synchonization Sequence

In order to signal to the language server that the client loaded a document into its buffer,

the client sends a ‘textDocument/didOpen‘ notification to the language server. This notification must contain data about the opened document: The document’s URI, a language identifier (the programming language of the document), a version number that increases after each change to the document, and finally, the document’s content as text. In most cases, the language server will now load this data (including the document contents) into its memory. The Language Server does not need to read the file contents using the URI, nor is it allowed to after the LSP specification. Keeping a reference to the document is necessary for the following method calls to function.

If the user changes a document, the client must send a ‘textDocument/didChange‘ notification to the server. This notification only contains the Uniform Resource Identifier (URI) of the document for the server to know what document was changed, an increased version number, and a set of changes to be applied to the text document in memory. There are two ways change events can be structured. An incremental change event only defines a range of text within the document that the server should replace. A "full" change event synchronizes the entire content with every change. The language server chooses what synchronization kind to use by setting a capability.

After a change has been applied to the document in memory by the server, it can rerun analysis on the document and provide the user with feedback on the updated document contents - for example, by showing error hints.

The LSP requires version numbers of documents to be sent alongside change events but does not define a specific use case for them. At first glance, they might seem extraneous. However, in case of a transmission error between client and server where the correct order of change events is disturbed, the server can compensate by using the version numbers to reapply the changes in the correct order. Additionally, they can be used for the ‘textDocument/publishDiagnostics‘ event to specify that a set of diagnostics is meant for a specific document version.

When the client closes a document, it sends a ‘textDocument/didClose‘ notification to the server, containing only a URI to a document. This notification indicates that the resource located at the document’s URI is now entirely up to date. The language server is free to clear the document from its memory, and the synchronization process is complete.

The three notifications explained above (‘textDocument/didOpen‘, ‘textDocument/didChange‘ and ‘textDocument/didClose‘) must be implemented by the client. Language servers can either implement compatibility for all three of the notifications or none of them.

2.2.4 Examples and common use-cases of LSP Methods

This section presents examples to illustrate how the methods defined by the LSP contribute to the developer experience.

Hover Information

‘HoverProviders‘ within language servers are used to provide users with information when they hover over a range of text within a document. The code editors usually display the information in a popup above the affected range.

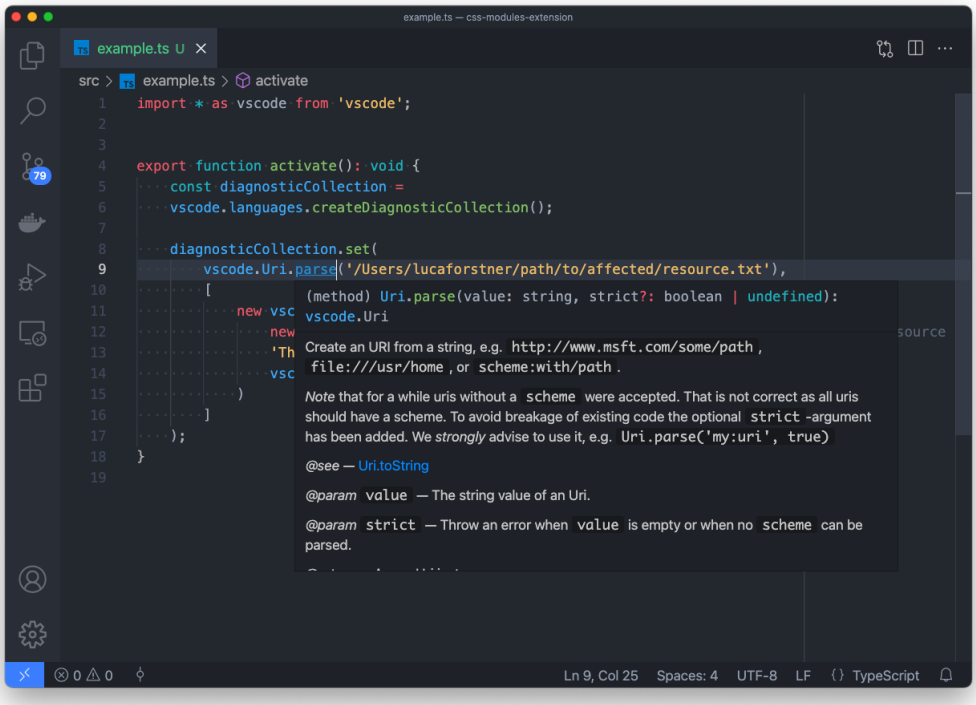


Figure 2.5: Example of Hover Information in VS Code using the LSP.

The procedure for showing hover information in VS Code is as follows: After the user leaves the mouse over a position in a document for a certain time, the client will send a ‘textDocument/hover‘ request to the server, containing a document identifier and a position within the document. The server can then determine which section of a document is being hovered over and respond with the correct hover text for the requested position. The server can also respond with ‘null‘ to indicate the absence of information for the requested position.

In many cases, this functionality is used in combination with user-provided documentation. When users write inline documentation for a variable or method, language servers can use that documentation to provide hover functionality for occurrences of these tokens elsewhere in the project.

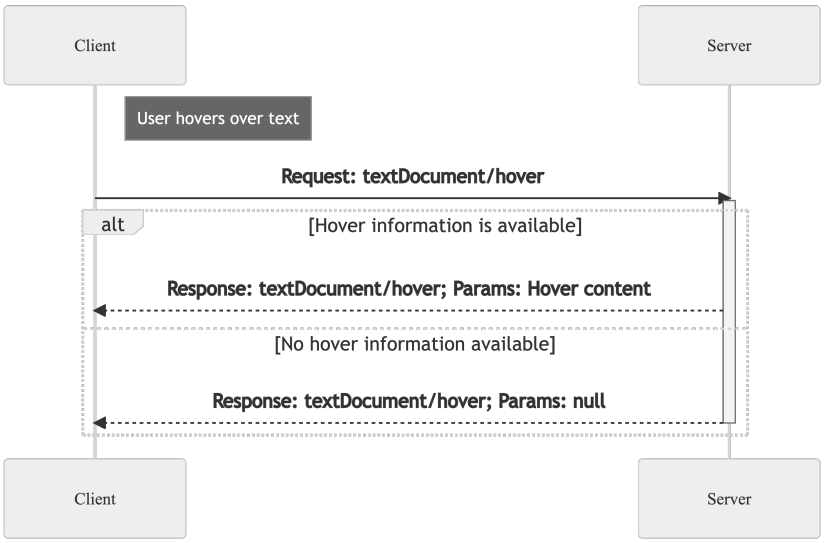


Figure 2.6: LSP Text Hover Sequence

Diagnostics

Diagnostics are a way for a client to draw the user's attention to specific sections of code. Code editors usually do this via colored underlines, where different colors mean different things (red usually means "error", while yellow means "warning"). Diagnostics may be displayed for various reasons, such as highlighting a syntax error. In addition to the underlines, code editors usually have a menu dedicated to diagnostics. In this menu, all diagnostics for one or multiple files are aggregated and displayed.

The server creates diagnostics by sending a 'textDocument/publishDiagnostics' notification to the client. This event contains all the diagnostic information for a particular document. This event overwrites all existing diagnostics on the client for that document (except those owned by other language servers). A usual scenario for language servers is to recompute diagnostics after each 'textDocument/didOpen' or 'textDocument/didChange' event. If the recomputation result indicated changed diagnostics, an updated set of diagnostics would be dispatched to the client.

Diagnostics are fully owned by the language server, meaning that servers also need to clear diagnostics. Language servers can clear diagnostics by not sending any diagnostics within a 'publishDiagnostics' notification. It is up to the implementation of a language server at which exact moment in time it clears existing diagnostics. However, there are two general strategies to follow: For single-file languages (like HTML and JavaScript), diagnostics are cleared when a document is closed in the editor. For project-based languages (like Java), diagnostics are kept, even when a file containing a diagnostic is closed.

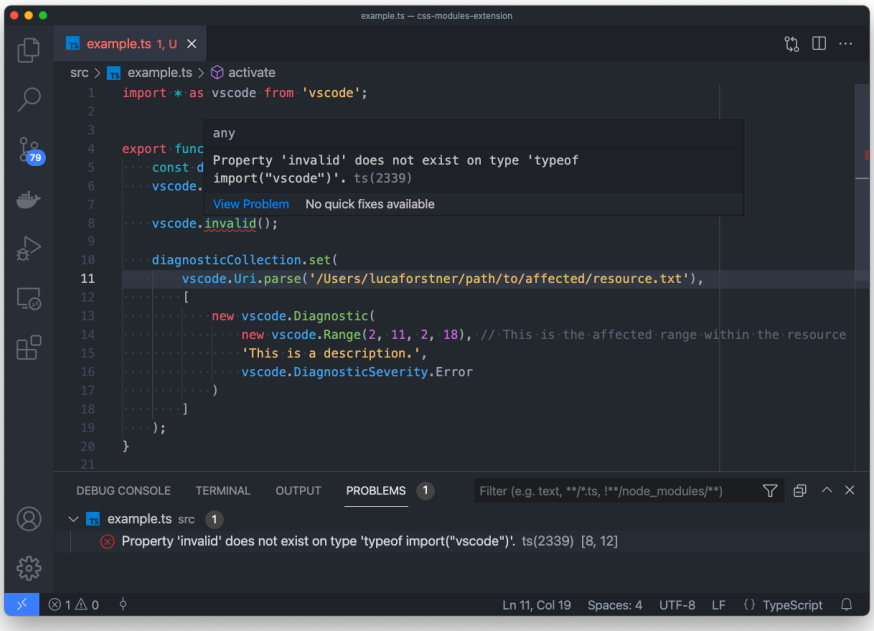


Figure 2.7: Example of Diagnostic Information in VS Code using the LSP.

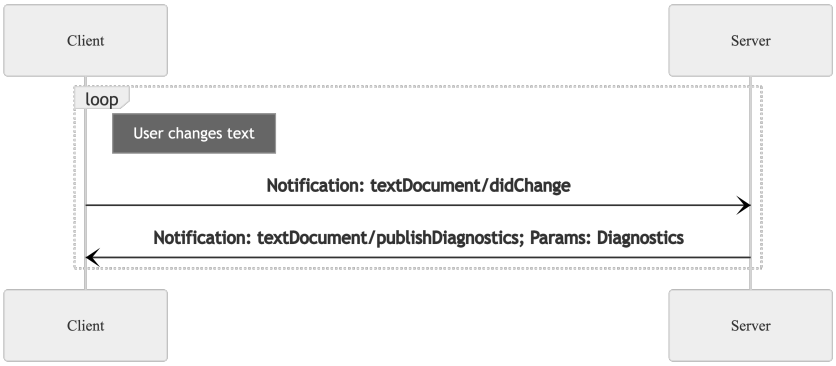


Figure 2.8: LSP Diagnosics Sequence

Code Actions

Code Actions are modifications to the code triggered by the client and computed by the server. The modifications usually entail fixes for various issues, automatic refactorings, or code beautification. If the client has the capability to apply the computed code modifications, it should do so itself. If the client lacks the necessary capabilities, the server can also apply the modifications as a backup strategy.

The client discovers available code actions by sending a ‘textDocument/codeAction‘ request with a relevant document and range to the server. The server responds either with an array of commands that the server can execute or with an array of code action objects containing executable commands. The client can now send a ‘workspace/executeCommand‘ request to the server, along with a command to execute. This request essentially acts as a notification but was defined as a request so that the server can respond with an error message if anything goes wrong during the computation. Once the server calculates the changes the client should apply to the text, it can send a ‘workspace/applyEdit‘ request containing the computed edit instructions. The client can then finally apply those changes and respond to the request indicating whether the edit instructions were executed or not.

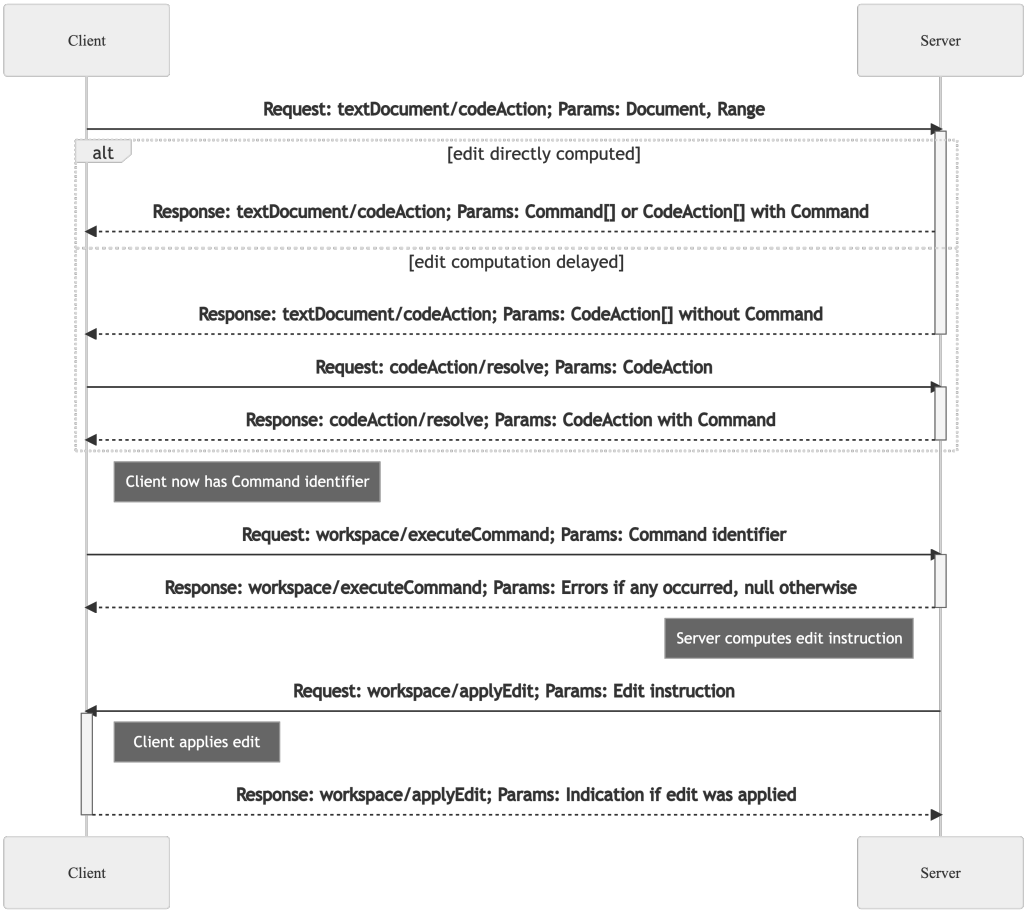


Figure 2.9: LSP Code Action Sequence

Code Navigation

Code navigation entails all methods of navigating in a codebase from a specific text location. The LSP provides different, although very similar, specifications and protocol methods for navigating to declarations, definitions, type definitions, implementations, and references. For each navigation type, the client requests a navigation target for a specific location within the document. The server can then respond with one or multiple locations (an interface, for example, can have multiple implementations) to which the client can jump. If there are no navigation targets, the server should respond with 'null'.

Another type of navigation is navigation through document links. Document links are sections within a text document that usually represent another resource. In client implementations, these sections will often be clickable and have special highlighting that resembles a hyper-link. Within the LSP, the server provides document links all at once for a given document, when requested.

Other Language Features

The LSP is not limited to the functionality and methods outlined above. It includes many additional methods, too many to include all of them in this thesis. The methods explained above are of relevance to this thesis as we will explore their conterparts within a Language Server Protocol designed for graphical languages in the next section. Additional features of the LSP include, but are not limited to:

- Syntax Highlighting: Tells the client to render symbols within code in different colors depending on their meaning.
- Code Completion: A quick-access menu while typing to complete tokens that the user started to type.
- Signature Help: Display of information for callable language elements, like functions.
- Code Lens: Information and actionable hyper-links interspersed within the source code.
- Selection Ranges: Automatic cursor selection based on the syntax of a language.
- Folding Ranges: A method to display or hide blocks of code.
- Linked Editing Ranges: Links two or more text ranges within a document which can be used for globally renaming tokens.
- Workspace Methods: Support for multiple project folders using only one client.

2.2.5 LSP for Graphical Languages

In the realm of graphical languages, we face similar problems as when dealing with textual languages. While designing models through modeling tools, users also want to benefit from features like validation, formatting actions, navigation, and the like. The natural question is whether it is possible to integrate graphical modeling tools into the LSP effectively.

Text editors generally work on raw text buffers. In contrast, graphical editors draw graphical representations using serializable data structures. The LSP methods were designed to operate primarily on text and text ranges and not on structured data. This difference in data format, and thus the mapping from structured data to text and vice versa, represents the main challenge to overcome when applying the LSP to graphical languages.

Langer identifies three key challenges when creating a language server protocol for diagrams [17]. Firstly, a graphical LSP needs to consider that the visual representation of graphical languages may differ from language to language. In a textual editing scenario, text is always represented the same way, even for different languages. Another challenge is imposed by the "impedance mismatch" between the content of a diagram's source and what is shown in the diagram editor. Diagram editors first interpret the sources behind the diagram before showing them. A graphical LSP needs to accommodate for this impedance mismatch. Lastly, while editing a diagram, the possible operations on a diagram are usually restricted in some way. The user may, for example, be constrained where in the diagram they may place a node or what nodes they may connect with an edge. A graphical LSP should consider the need for such constraints.

Textual IDEs already profit from many of the upsides graphical editors would gain if they supported a language server protocol. Currently, most graphical tools have monolithic architectures that only support one if not a few domain-specific languages at most. A language server protocol would permit those tools to support multiple languages while adopting a more decoupled architecture. As it becomes more and more popular to have workflows on the web within browser-based applications, a decoupled architecture would allow these tools to adopt this philosophy. An editor-infrastructure using a standard protocol would also have the upside of introducing a unified user experience where users are used to specific patterns employed by the language server protocol.

Rodriguez-Echeverria et al. [18] took on the task to evaluate whether it is possible and practical to use the LSP for graphical languages. As this application of the LSP presupposes a generalized graphical editor that can support multiple different DSLs, they first grouped the common diagram operations into different classes of generalizability:

- Operations that are dependant on the semantics of a language (e.g., completion or validation).
- Operations that are language-agnostic and might apply to any diagram (e.g., zoom and automatic layout).

- Auxiliary operations like editor initialization and shutdown as well as model storage.

Additionally, they evaluated possible architectures when considering the classes above:

- A monolithic architecture where the client takes care of displaying the model in addition to all operations.
- An architecture where the client manages concrete syntax and the language-agnostic operations of a diagram and the server controls the abstract syntax and auxiliary operations.
- An architecture that builds upon the preceding one where the language server can also execute language-agnostic operations.
- A fully decoupled architecture where the client only displays a diagram and keeps a textual representation while the server manages all syntactic details and operations.

Finally, Rodriguez-Echeverria et al. [18] ask whether the LSP is sufficient to build a language server that can support operations on graphical languages, whether it needs to be extended or whether an entirely new protocol is necessary. Using the standard LSP would have the upside of already being widely supported by tools and libraries alike. However, using the LSP without any extension might introduce complexity because operations on text and text ranges must be mapped to structured diagram data. This implementation requires a textual representation of a language. An extended LSP for graphical languages would open up the possibility of having special graphical operations that the LSP currently does not have. Even though this kind of protocol would also enable non-textual representations of a diagram, some textual representation is still necessary for the default LSP operations - meaning that hybrid representations would be possible. An entirely new protocol gives total freedom over the implementation and operations. In this case, the client would send operations to the server to map these actions into model operations. References and positioning would be much simpler to implement than using the standard LSP because this protocol would not be restricted by using text ranges. Messages could be optimized, reducing communication overhead. This path has the downside of not already being widely supported, meaning it would have to be adopted by the users. Many operations would probably overlap with the existing LSP, implying a reimplementation while future interoperability with the LSP might be impeded entirely.

Rodriguez-Echeverria et al. conclude that the standard LSP is expressive enough to accommodate graphical languages. An appropriate architecture can mitigate most of the drawbacks mentioned above. They even present a proof-of-concept implementation for graphical languages using the default LSP alongside a textual representation.

In another survey, Bünder [19] identifies several advantages of using the LSP for domain-specific languages. However, they also find that graphical notations, which are of great

benefit when examining DSLs, are not well supported by the Language Server Protocol. Bünder suggests that tooling, which supports non-textual notation, may be a threat to the LSP.

In a later paper, Bünder and Hendrik [20] state that multi-editor support for graphical DSLs has not gained much attention. In the next section we will highlight the Graphical Language Server Platform (GLSP), which aims to tackle exactly this problem - providing multi-editor support for graphical languages.

2.3 Graphical Language Server Platform

The Graphical Language Server Platform [3] is a framework to build diagram editors. It is under active open-source development by EclipseSource and the Eclipse Foundation. Its architecture follows the architectural pattern proposed by the Language Server Protocol, where generalized clients (editor views) are mostly decoupled from a language server that holds language-specific information to support the client with additional features. The clients communicate with language servers over a newly defined protocol (the Graphical Language Server Protocol) designed to work with graphical editors and modeling tools. The aim of GLSP is to profit from the same advantages for modeling that the LSP architecture generated for text editing: enabling encapsulated development of clients and domain-specific language servers for improved usability and reusability. An additional goal of this framework is to simplify the transition for modeling tools to move into the web or to web-based IDEs, like VS Code, Eclipse Theia, or Eclipse Che.

2.3.1 Components of the Graphical Language Server Platform

The GLSP includes a client for graphical diagram editing that uses common browser technologies like HTML, CSS, JavaScript, and SVG to guarantee support for web-based operation. This component is responsible for rendering language-specific diagrams. By using the GLSP protocol, the GLSP client can be connected to a domain-specific GLSP language server. The generic version of this client can be used in combination with any language server to support the underlying domain-specific language. The GLSP makes heavy use of Eclipse Sprotty [21] for the client component and even builds the GLSP protocol on top of the client-server architecture defined by Sprotty.

In general, language servers for the GLSP protocol can be implemented using any technology. The GLSP provides multiple frameworks (Java, or Node.js-based), which can be used to speed up the development of graphical language servers.

Current development efforts go into integrating GLSP components into existing IDEs and platforms (Eclipse, Eclipse Theia). This thesis will present a contribution that enables the integration of any GLSP clients and GLSP language servers into VS Code by providing a library for VS Code extensions.

2.3.2 Eclipse Sprotty

As mentioned above, both the GLSP client and the GLSP protocol are based heavily on Eclipse Sprotty [21] and extend it with additional functionality specific to GLSP. We will first examine the architecture of Eclipse Sprotty [22] before analyzing how GLSP expands upon it.

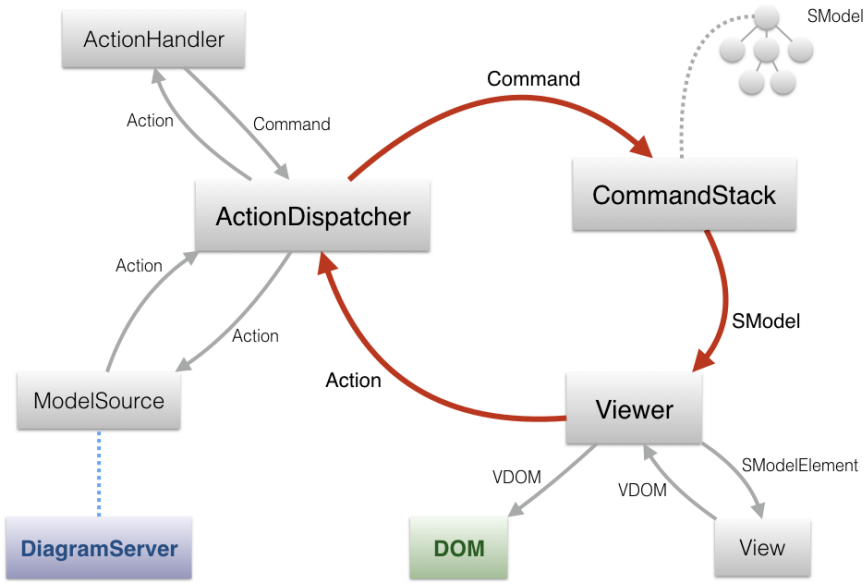


Figure 2.10: Eclipse Sprotty's unidirectional Event Flow. [22]

Sprotty stores diagrams in a data type called *SModel* (SprottyModel). The *SModel* is a tree-like data structure whose nodes all inherit from *SModelElement*. SModels are serializable by design to be transmittable between clients and servers. All *SModelElements* have a unique ID, type, and parent and child nodes references. *SModelElements* always represent an element of a model. They can be decorated with *SModelExtensions*, which provide additional *Features* to *SModelElements*. For example, a "Selectable" extension will make the *SModelElement* selectable, while the "Movable" will make it movable.

The *Viewer* concerns itself with the actual web-based rendering of models. By feeding *SModelElements* into *Views* that handle their type, the *Viewer* creates a virtual Document Object Model (DOM) from the *SModel*, which the *Viewer* uses to modify the currently displayed DOM in the client. The *Viewer* also attaches various event listeners to the DOM Nodes, based on defined *SModelExtensions*.

"Actions" define operations on the model. *Actions* are the actual protocol messages transmitted between client and server - meaning that they also ought to be serializable.

Actions originate from the *Viewer* (through event handlers) or a *ModelSource*, located on a remote diagram server or directly on the client. The *ActionDispatcher* feeds Actions to *ActionHandlers* that convert the *Actions* into *Commands*, defining the *Actions'* actual behavior. The *CommandStack* executes those commands to modify the *SModel*. The *Viewer* will render the updated *SModel*, closing the unidirectional event loop that Sprotty proposes.

The Sprotty Client Server Protocol [23] defines the interactions between a client and its *ModelSource*. Sprotty allows for layout computation on the server as well as the client. In most cases, the client will handle the bounds computation of sub-components within major composite elements like nodes (i.e., "micro layout"). The server will take care of layouting the significant elements like nodes and edges. Depending on what computation mode (client, server, or combined) is chosen, the Sprotty Protocol will slightly differ. The Sprotty Protocol has *Action* sequences for when the client requests models from the server, renders popups and changes element selections. The GLSP Protocol will build on these *Action* sequences.

2.3.3 Contributions of the GLSP Protocol

All commands of the GLSP Protocol [4] are wrapped into an *ActionMessage* before being transmitted between client and server. *ActionMessages* always contain a client ID, determining the receiver client or identifying the sender. *ActionMessages* also hold the *Action*, meaning the executable instruction they intend to transmit. The *Action* interface only defines a *kind* property, which acts as an identifier for various kinds of *Actions*. A *SaveModelAction*, for example, is identifiable by a “saveModel” string as kind.

Actions can furthermore be narrowed down to *RequestActions* and *ResponseActions* that must contain a request ID and a response ID, respectively. *RequestActions* are sent from the client to the server or vice versa in expectance to receive a response in the form of a *ResponseAction*. The request ID of the *RequestAction* and the response ID of the corresponding *ResponseAction* must match.

The GLSP protocol also defines *Operations*. They are the same as *Actions* but denote an actual change on the diagram model. *RejectActions* are a kind of *ResponseAction* that will indicate that a *RequestAction* has been rejected.

The protocol introduces lifecycle *Actions* to set up the communication basis between a client and a server. First, an *Initialize Request* has to be sent to the server from the client. No other *Actions* can be issued before the server has responded with an *InitializeResult*. In this exchange, the server and the client advertise what version of the GLSP protocol they implement. The client uses the initialization to pass any language-specific configuration parameters to the server. The server will include a list of all the actions it can perform for various graphical languages in its response to the *Initialize Request*.

Displaying a graphical representation of a model is called a *ClientSession*. If a client displays a model (i.e. starts a *ClientSession*), it must send an *InitializeClientSession*

request to the server so the server becomes aware of the session. If the diagram is no longer being edited or displayed, the client should send a *DisposeClientSession Request* to the server to clear up any acquired resources.

2.3.4 GLSP Protocol Actions

We will now briefly outline the most important *Action* definitions of the GLSP Protocol, many of which we will later use in our contribution to the VS Code Extension integration:

Model Synchronization between Client and Server. If a client wants to receive a graphical model from a language server, it should send a *RequestModelAction*. The server should answer with either a *SetModelAction* or a *UpdateModelAction*. The server can send these actions to inform the client that the model should be updated. The *SetModelAction* exists to replace a model entirely or set it if it does not yet exist on the client. The *UpdateModelAction* will only update a model.

Model Saving. The client can send a *SaveModelAction* to the server if it wants the opened model to persist to the model source. It may add a 'fileUri' property to indicate that the model should be saved to a specific destination. The server may send a *SetDirtyStateAction* to the client at any time to announce that the persisted state of the diagram does not currently match the state of the currently opened diagram. The client may use this information to display a hint to the user that the model they are editing is unsaved.

Model Export. The GLSP Protocol reuses the *ExportSvgAction* of the Sprotty Protocol. Both the client or the server may send this action, expecting the other party to convert the current model to an SVG. A response should contain a the diagram *SModel* as serializable SVG.

Model Edit Mode. Within the GLSP Framework, diagrams can either be read-only or editable. If the client or the server wants to change the mode for a diagram, it can send a *SetEditModeAction* to the other party containing the desired editing mode.

Model Layout. The *CenterAction* and *FitToScreenAction* can be sent to the client to center the displayed model and fit it to the screen, respectively. These actions only need to be processable by the client but may originate anywhere. The *LayoutOperation* may be sent from the client to the server. Upon receiving this operation, the server should organize the layout of the current model.

Server Notifications. The server may send a *ServerMessageAction* to notify the client and the user about certain topics (e.g., runtime errors or loading indicators). In a very similar manner, it may also send a *GLSPServerStatusAction* to notify about internal status changes. While both of these message types intend to notify the user about something happening on the server, they may be displayed in different ways - this is up to the implementing client to decide.

Element Selection. The client can send Sprotty's *SelectAction* to inform the server that the user changed the element selection. The server can also send a *SelectAction*

action to the client, signaling that a set of elements should be marked as selected. The GLSP Protocol also defines a *SelectAllAction* which works similar to the *SelectAction* but will always select or deselect all elements.

Hover Information. When a user hovers over an element, the client can send a *RequestPopupModelAction* to the server, including the ID of the element that is being hovered over in the request body. The server responds with a *SetPopupModelAction* containing the popup the client should render.

Model Validation. Element validation in GLSP is based on validation markers. *Markers* are a pointer to an element, annotating it with additional information, similar to underlines and diagnostics in traditional text-editing. The client usually sends a *RequestMarkersAction* to the server to receive validation markers for a model. The server will respond with a *SetMarkersAction* which contains all markers for the requested element. It is then up to the client to render those markers. Markers can have different severities, stored in the 'kind' field.

Navigation. The GLSP Protocol presents a concept for navigation in graphical workspaces. Navigation in GLSP is generic, meaning it is possible for implementing clients and servers to trigger navigation to different kinds of targets. Navigation targets may include, for example, specific elements of a diagram or a section of text in a separate file, containing the documentation for a specific diagram element. Navigation usually starts with the client asking the server for available navigation targets in a given context through the *RequestNavigationTargetsAction*. The server then responds with a *SetNavigationTargetsAction* containing all navigation targets for the queried context. The action that will trigger a navigation is called the *NavigateToTargetAction*. It must contain a navigation target.

Menus. *Contexts* are regions in the client interface. These regions are addressable through a unique identifier. At the time of writing, the GLSP supports three contexts: the *context menu*, the *command palette*, and the *tool palette*. The *context menu* is a menu that appears when the user presses the right mouse button. The *command palette* is intended to work similarly to VS Code's command palette. It is a searchable and "auto-completable" widget that appears when the user presses a key combination. The *tool palette* is a menu that is usually a menu that hosts the tools the user can use to interact with a diagram. The GLSP defines a client-server exchange to fill these menus with executable actions. This works by first querying available actions with the *RequestContextAction* from the server and awaiting a *SetContextAction* as a response. The *SetContextAction* contains all available actions for a particular context, which the client can use to display menus.

Type Hints. The element type hints action *RequestTypeHintsAction* is used by the client to request from the server what modifications are currently possible on a model. The server provides the requested information with a *SetTypeHintsAction* as response. Type hints include, for example, whether an element or an edge is repositionable, deletable, or resizable. Clients typically request type hints during initialization of a diagram.

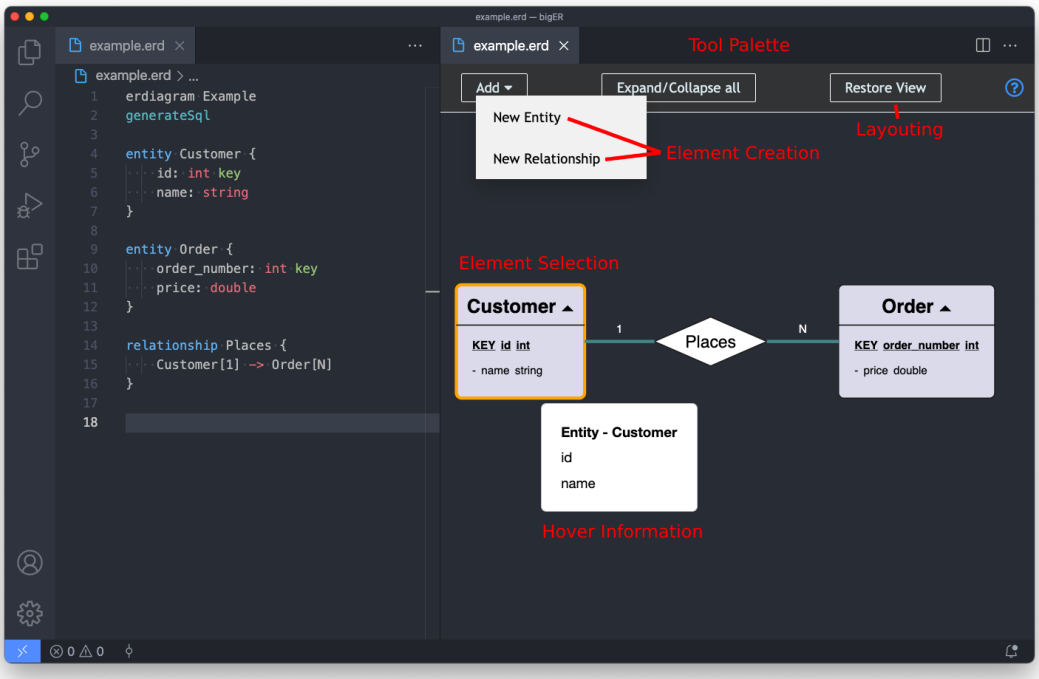


Figure 2.11: A subset of GLSP protocol features within the *bigER* extension [24], a novel modeling tool based on GLSP and Eclipse Sprotty.

Element Creation and Deletion. The client can send a *CreateElementOperation* or *CreateEdgeOperation* to the server to create elements and edges. The *CreateElementOperation* must include information about what kind of element should be created and may contain additional information like the position of the newly created element. The *CreateEdgeOperation* needs to indicate what kind of edge should be created, in addition to the source and target element of the edge. Clients can delete edges and elements via the *DeleteElementOperation*. This *Operation* should contain an array of all the element IDs that the client intends to delete.

Element Modification. A *ChangeBoundsOperation* sent by any party indicates the resizing or repositioning of an element. A *ChangeContainerOperation* sent by the client will move an element into a different container element. Edges can be modified through a *ReconnectEdgeOperation* or a *ChangeRoutingPointsOperation*. The *ReconnectEdgeOperation* changes the source or the target element of an edge. The *ChangeRoutingPointsOperation* will edit the routing points of an edge, along which it runs its path.

Text Editing. When the user is editing text on an element, GLSP provides a *Protocol Action* for the entered text to be validated by the server. The client sends a *RequestEditValidationAction* to the server with properties of the text edit operation. The server will answer with a validation response in the form of a *SetEditValidationResultAction*, either

confirming the edit action or notifying the client of an error. In order to actually persist the text for a label on the model, the client may send an *ApplyLabelEditOperation* to the server.

Clipboard. The GLSP Protocol supports copy, cut & paste operations on models. To copy diagram elements, the client sends a *RequestClipboardDataAction* to the server. This action will ask the server for the clipboard content of the given context (i.e., the currently selected elements). Servers provide the requested information in a *SetClipboardDataAction*. The client should put the data inside this action into the user's clipboard. When the user cuts elements from the model, the client should send a *CutOperation* to the server. This *Operation* will ask the server to delete the currently selected elements, so the client must make sure that the clipboard already contains the elements about to be deleted. When the user pastes content, the client should send a *PasteOperation* to the server containing the user's clipboard content. In response to this operation, the server will apply the clipboard contents to the model.

Undo and Redo. Because the model state is stored on the server and usually keeps an editing history, the client can send an *UndoOperation* or a *RedoOperation* without any arguments to undo and redo changes. The server will persist changes that happen through these operations by sending a *SetModelAction* or an *UpdateModelAction*.

Tool Palette. The *tool palette context* usually hosts a set of *tools*. These *tools* are used to interact with the diagram. *Tools* have the special property that their usage can be enabled and disabled by the server. The GLSP includes three default *tools*: the *selection tool*, a *delete tool*, and a *validation tool*. *Tools* may trigger any *Action*. If a user begins to use a *tool* that creates nodes or edges (i.e., clicks on it), the client should inform the server of this tool activation by sending a *TriggerNodeCreationAction* or a *TriggerEdgeCreationAction*.

2.4 Differences between the GLSP-Protocol and the LSP

Even though the GLSP and the LSP share a client-server architecture, they have a few traits that make them very different from each other.

An apparent difference is that the GLSP is mainly tailored towards graphical languages, while the LSP is used with textual languages. This dissimilarity is especially noticeable in the data types both protocols define - the GLSP declares spatial properties like points, dimensions, and bounds, while the LSP declares types like ranges and textual positions.

Using the LSP, it is necessary to communicate via JSON-RPC. The GLSP does not define a protocol to be used on the communication layer to transmit action messages. It should be noted, however, that the default GLSP implementation uses JSON-RPC as it provides a good basis for inter-process communication.

The protocols share a similarity in the kinds of messages they define. Both protocols include messages that prompt a response, in addition to response messages and messages that can be sent in an asynchronous manner.

The GLSP and the LSP handle file synchronization very differently. The LSP recommends that all persisting changes to a file happen through the client. The LSP server only sends changes it intends to make (e.g., code actions) as suggestions to the client. It is the client's responsibility to apply them to the opened buffer, persist them to a file if desired, and to notify the server as soon as the client edits a buffer. In GLSP, the model source (usually the server) owns the files. This means that the client sends modifications to the server to execute them on the model, introducing the need for protocol messages for all kinds of basic operations, like element and edge deletion/creation/modification, copying and pasting, label editing, and the like. The server then applies the modification. When the user wants to persist changes to a file, the client instructs the server to do so. Because graphical editors need to interpret the source of a model, and those interpretations generally differ from language to language, the editors need to be kept generic. To ensure genericity, GLSP shifts responsibilities related to loading, modifying, and persisting models to the server.

Another discrepancy between the protocols is the handling of validations. The LSP has a notification model. The server always notifies the client of all current diagnostics. The client does not need to request the diagnostics or keep track of them because the server will always overwrite them at once with a 'textDocument/publishDiagnostics' notification. This means, however, that the client has no way of requesting diagnostics - it is entirely up to the server when it will send the diagnostics. A pull-based model, where the client can request diagnostics, is currently in a proposal stage [25]. The GLSP protocol already has a pull-based model in place. Clients send requests for markers, and the server provides them. This diagnostic model is especially advantageous in cases where markers are expensive to compute, as they are only computed when actually needed. The GLSP server also expects the client to keep state of the current validation markers, as it may also send a *DeleteMarkersAction* - this is not the case in the textual LSP.

In contrast to the GLSP protocol, the LSP does not have edit modes and methods to configure whether files are read-only or editable. While both architectures let clients decide whether a file should be editable or not, the GLSP protocol permits the server to impose editing modes.

Even though we listed a lot of differences above, we can also examine some parallels: The concept of *Code Actions* in the LSP bears many similarities to how *Layout Actions*, *Tools*, and *Contexts* work in the GLSP protocol. In both protocols, the clients ask for possible actions to take, to which the servers respond with available options. The clients then ask the server to execute them. The only difference lies in who persists the changes. In the LSP architecture, it is the client, in GLSP, the server. The same applies to the concepts of navigation and hover information.

Generally, it can be said that GLSP was built with extension in mind. Many GLSP actions can be enhanced with custom properties and arguments to accommodate for different kinds of implementations and language server quirks. Extensibility might very well be a necessity, as the clients of graphical languages tend to be very different from

each other. Apart from some definitions, the textual LSP has a more strict approach. It does not allow for many custom parameters in the protocol messages.

CHAPTER 3

Architecture

In this chapter, we will explore the architecture of the features we intend to contribute to the GLSP VS Code integration.

3.1 Current State of the VS Code Integration Library

Currently, the "glsp-vscode-integration" library provides basic support for using GLSP components within VS Code. It is already able load and display GLSP diagrams in native editor panels. Additionally, the library already handles the saving of diagrams and provides undo & redo functionality. Other than that, the library currently does not provide any functionality that directly integrates with VS Code.

For our contribution, we want to equip the GLSP VS Code integration library with additional native VS Code functionality. We will outline all the features we are planning to implement in the following sections.

3.2 Requirements

By analyzing the requirements of our contribution we hope to gain a clear picture of what components need to be implemented and how they need to work and interact with each other. We need to factor in two user groups when developing for the VS Code integration library. First, we need to consider developers who plan on using our proposed library in their VS Code extensions. Further, we need to consider any users who will interact with the software in a non-technical way - users who will use GLSP components within VS Code to edit diagrams. As a result of our target groups, we can split our requirements into *functional* and *library interface* requirements.

Functional requirements are requirements that define what functionality our software library should provide. They primarily affect end users. We have defined the following functional requirements for the library:

- **File State Management.** The proposed library should handle diagram file states in VS Code. File state management entails opening diagrams, persisting edited diagrams, and marking them as dirty within the editor interface when the user makes changes to a diagram.
- **Diagnostics.** The GLSP VS Code integration should natively display any diagnostic information that comes from the GLSP language server within the diagnostics panel.
- **Clipboard.** The integration should enable users to copy, cut and paste diagram elements.
- **Navigation.** Usage of the library should enable users to navigate to elements and documentation. Navigation should be possible through the native VS Code UI.
- **Diagram Layouting.** Users should be able to use the automatic layouting functionality of GLSP servers directly through menus and commands in the VS Code interface.
- **Viewport Navigation.** Similar to the layouting functionality, users should be able to center and fit diagrams within their viewport through interaction with the native VS Code interface.
- **SVG Export.** The integration should enable users to export diagrams as SVGs via a native file dialog window.

As an additional requirement, the user experience and the feel of the features described above should generally be consistent with the user experience and feel of the VS Code interface. We aim for any introduced interfaces to behave as close as possible to the native application. By considering this usability aspect, we hope to reduce friction while switching between VS Code and GLSP interfaces, and to flatten the learning curve while adopting GLSP tools.

Library Interface Requirements concern all the requirements that define how developers of VS Code extensions should be able to use the integration. Developers will primarily use the library to integrate GLSP components of graphical languages into VS Code. We aim to construct an API that is as straightforward as possible whilst not creating any unnecessary architectural constraints. We define the following library interface requirements:

- **Integration GLSP Components.** The library should make it possible to integrate all GLSP components and provide an interface to interact with them. Integrating GLSP components involves registering a GLSP server and multiple clients.
- **Exposure of Context Variables.** It should be possible for extension authors to retrieve various context variables from diagrams (i.e., how many elements are selected, what elements are selected or what client is currently active). Context variables will enable context-aware viewport navigation.
- **Interfacability.** The integration should provide a simple interface for extension developers to send actions (i.e. GLSP protocol messages) to the GLSP components. This will enable extension authors to connect native VS Code UI elements to GLSP functionality.
- **Extensibility.** Extension authors should have full control over GLSP components and their messages, even though the components are integrated using the proposed library.
- **Default Implementations.** The integration should provide generic default components that enable developers to quickly create VS Code extensions for custom DSLs, albeit with limited functionality.

In addition to the requirements above, we aim for a consistent and accurate implementation that is performant, reliable, and meets basic usability standards. Chapter 6 will assess and evaluate the finished software artifact against its requirements.

3.3 Available Software Components

We need to combine various existing software components for our proposed VS Code integration.

First, we have the *VS Code editor*, which is extensible via the *VS Code Extension API*. This API is only accessible from within VS Code Extensions but behaves like a JavaScript package thanks to a TypeScript definition. A feature of the VS Code Extension API essential to our proposed library is the *Webview API*, which enables the integration of any web-based components into the VS Code editor - including the GLSP client.

The *GLSP client* is responsible for displaying diagrams and enabling interaction between users and the diagrams. Theoretically, a client conforming to the GLSP specifications could be based on any non-web technology. We assume, however, that the majority of developers will aim to integrate web-based tools into VS Code. GLSP's proposed default client is implemented using the web-based Eclipse Sprotty rendering engine, making it ideal to be integrated into VS Code via the Webview API. Similar to the VS Code Extension API, the default GLSP client is distributed as JavaScript package.

The last pre-existing component we need to consider is the *GLSP server*. As with the GLSP client, the GLSP server may be implemented using any technology as long as the communication interface conforms to the GLSP protocol. This requires us not to make any assumptions about the details of this component. We intend to provide a generic interface over which developers may send and receive messages from the server component in their integration. The default server component proposed by GLSP is a separate application implemented in Java. We intend to provide auxiliary components to support this default implementation.

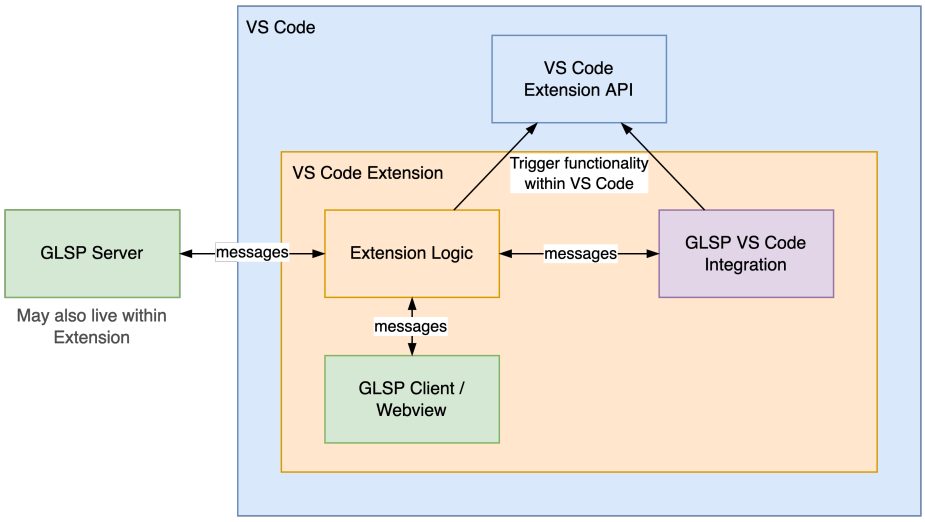


Figure 3.1: Component topology of the proposed architecture.

Given the VS Code Extension API, the GLSP client, and the GLSP server as pre-existing components, we aim for a structure described in Figure 3.1. While the GLSP client will reside in VS Code Extensions directly, the server may exist anywhere (e.g., as a separate process on the machine running VS Code or even on an external server). Extensions are responsible for orchestrating the GLSP components themselves. This assignment of responsibility should guarantee maximum configurability for extension authors. Our integration library is intended to be used and imported directly by extensions. The library will receive all messages from the extensions sent by the client or the server and act upon them by calling the VS Code Extension API.

3.4 Known Challenges and Limitations to overcome

When trying to solve the problems we set out to solve, we need to consider two known issues. The first issue has to do with the fact that VS Code was originally designed for textual documents. If we want users to be able to edit diagrams, we must also consider graphical documents, which VS Code does not directly support. Luckily the VS Code API already provides a system that enables close-to-native editing functionality for

documents that may deviate from a textual nature, called “Custom Documents”. This system, however, is designed to be very generic so that it can accommodate documents for all kinds of use-cases - not just diagrams. As a drawback of this genericity, when using custom documents, it is necessary to provide a significant amount of logic and configuration upfront to achieve even the most basic functionality, like opening and saving a document. Our proposed changes to the GLSP VS Code integration library aim to make this required configuration trivial for extension authors.

The second issue we face is the integration of multiple moving parts into one system. Specifically, the integration must combine the functionality of any GLSP client with any GLSP server and any VS Code extension. This requirement calls for a very generic library that will not make more assumptions than necessary about the components it will interact with.

3.5 Components of this Contribution

The library interface requirements we defined will significantly affect the VS Code integration library’s outward-facing API, and the functional requirements will demand an internal restructuring of the library. We decided, for this reason, to entirely rethink the elements of the library’s current version and introduce a new set of components that will simplify the library’s interface and development for extension authors in the future.

As the main contribution, we want to provide a general component that connects the GLSP components (i.e., server and client) with the VS Code Extension API and is interfacable via conventional VS Code Extensions. We will call this component the “GLSP VS Code Connector”, or ‘GlspVscodeConnector‘ when referenced in code. We intend for this component to be generic and not make any assumptions about the implementation details of the GLSP components or the extension hosting the Connector. The GLSP VS Code Connector will coexist with other components and should not be dependent on any other artifacts except for the VS Code Extension API.

The GLSP VS Code Connector should ideally act as a passthrough for messages sent between the client and the server. Based on the messages sent by the GLSP components, the Connector will react and call the VS Code Extension API to provide native functionality (e.g., editor diagnostics). The GLSP VS Code Connector may also intercept and modify messages before propagating them. The extension’s responsibility is to register all clients and the server on the Connector (see Figure 3.2). By registering those components, the messages are routed through the Connector with its only point of contact being the extension itself.

The GLSP project includes an example for the standalone, Eclipse and Theia integrations, enabling developers to pick up the GLSP stack and create a working prototype with reduced effort. We want to provide a set of components that help set up a GLSP VS Code Extension with these GLSP example implementations. We will label those components

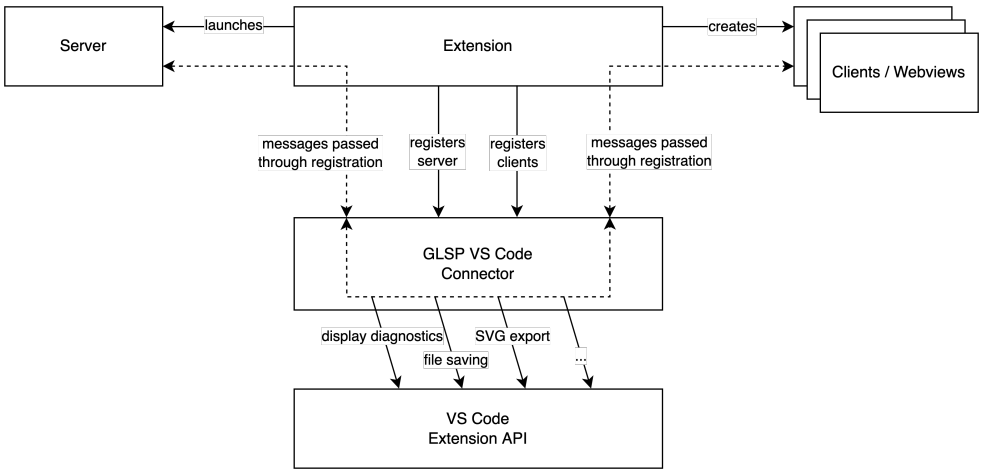


Figure 3.2: Component orchestration within the proposed architecture.

"Quickstart Components". Section 4.3 discusses the exact implementation details of these components.

The "GlspServerLauncher" Quickstart Component should be a simple way to start and stop the example GLSP server process. This process can then be wrapped by the "SocketGlspVscodeServer" Quickstart Component to provide a simple interface to send and receive GLSP messages from and to the server. The wrapper also implements the correct interface to be directly used in conjunction with the main GLSP VS Code Connector component.

It is necessary to register a "CustomEditorProvider" from an extension in VS Code to use webviews. Editor Providers are responsible for creating an editor panel (i.e., a tab in VS Code). We will provide an example Editor Provider Quickstart Component to bootstrap the initialization process of the default GLSP client.

In the next chapter, we will outline the development of the GLSP VS Code integration by providing the implementation details of this chapter's proposed architecture.

CHAPTER 4

Technical Implementation

In this chapter, we will explore the implementation details of our contribution to the GLSP VS Code integration library.

4.1 GLSP VS Code Connector

We will start with the most crucial component, the ‘GlspVscodeConnector‘. As described in the previous chapter, this component is the glue between the GLSP server, the GLSP clients, and the VS Code Extension API. Extensions using the Connector should pass all messages from the GLSP components through it.

```
1  interface IGlspVscodeConnector<D extends vscode.CustomDocument> {
2
3      /**
4       * A subscribable event which fires with an array containing
5       * the IDs of all selected elements when the selection of the
6       * editor changes.
7       */
8      onSelectionUpdate: vscode.Event<string[]>;
9
10     /**
11      * A subscribable event which fires when a document changed.
12      * The event body will contain that document. Use this event
13      * for the onDidChangeCustomDocument on your implementation
14      * of the 'CustomEditorProvider'.
15      */
16     onDidChangeCustomDocument: vscode.Event<vscode.
           CustomDocumentEditEvent<D>>;
17
18     /**
19      * Register a client on the GLSP-VSCode connector. All
20      * communication will subsequently run through the VSCode
```

```
21        * integration. Clients do not need to be unregistered as
22        * they are automatically disposed of when the panel they
23        * belong to is closed.
24        *
25        * @param client The client to register.
26        */
27       registerClient(client: GlspVscodeClient<D>): void;
28
29       /**
30        * Send an action to the client/panel that is currently
31        * focused. If no registered panel is focused, the message
32        * will not be sent.
33        *
34        * @param action The action to send to the active client.
35        */
36       sendActionToActiveClient(action: Action): void;
37
38       /**
39        * Saves a document. Make sure to call this function in the
40        * `saveCustomDocument` and `saveCustomDocumentAs` functions
41        * of your `CustomEditorProvider` implementation.
42        *
43        * @param document The document to save.
44        * @param destination Optional parameter. When this parameter
45        * is provided the file will instead be saved at this location.
46        * @returns A promise that resolves when the file has been
47        * successfully saved.
48        */
49       saveDocument(document: D, destination?: vscode.Uri):
50           Promise<void>;
51
52       /**
53        * Reverts a document.
54        *
55        * @param document Document to revert.
56        * @param diagramType Diagram type as it is configured on the
57        * server.
58        * @returns A promise that resolves when the file has been
59        * successfully reverted.
60        */
61       revertDocument(document: D, diagramType: string): Promise<void>;
62
63       dispose(): void;
64   }
```

Listing 4.1: TypeScript interface of the GLSP VS Code Connector component.

4.1.1 Class Signature

The GLSP VS Code Connector is implemented as a TypeScript class called ‘GlspVscode-Connector‘. It takes a generic type parameter ‘D‘ that must extend the ‘CustomDocument‘ interface, defined by the VS Code Extension API. ‘CustomDocuments‘ represent docu-

ments that can be viewed and edited in VS Code through webviews - they do not have to be of textual nature. Their creation and lifecycles are managed by custom ‘CustomEditorProvider‘ objects, registered by extensions. We will examine ‘CustomEditorProviders‘ further later on in this chapter. The type parameter ‘D‘ will control the documents the Connector accepts when registering clients.

```
1 class GlspVscodeConnector<D extends vscode.CustomDocument>
2     implements vscode.Disposable {
3
4     // ...
5
6     public dispose(): void {
7         // ...
8     }
9 }
```

Listing 4.2: GLSP VS Code Connector class.

The Connector implements the ‘Disposable‘ interface defined by the VS Code API. The VS Code documentation describes Disposables as “[...] a type which can release resources, [...]” [26]. Disposables only need to expose a ‘dispose‘ function, which should release any acquired resources when called. Disposables are used frequently throughout the VS Code Extension API and also throughout this contribution. The GLSP VS Code Connector’s ‘dispose‘ function will, for instance, clear any diagnostics it created.

4.1.2 Constructor

The Connector’s constructor only takes one “options” argument. Its options must contain a server object which conforms to the ‘GlspVscodeServer‘ interface. This interface requires two fields; one is intended to be used to send messages to the GLSP server, and the other to receive messages. The ‘onServerMessage‘ field is of type ‘Event‘. The VS Code API defines events as functions that one can subscribe to by calling them with a callback function as argument. The callback function will then be executed when the event is fired. By subscribing to the ‘onServerMessage‘ field, the GLSP VS Code Connector will receive messages from the GLSP server. To propagate events to the server it will fire an Event on the ‘onSendToServerEmitter‘. This field is of type ‘EventEmitter‘. Event Emitters lie one layer above Events and expose a ‘fire‘ function to trigger their underlying event.

```
1  interface GlspVscodeServer {
2      /**
3       * An event emitter used by the VSCode extension to send messages
4       * to the server.
5       */
6      onSendToServerEmitter: vscode.EventEmitter<unknown>;
7
8      /**
9       * An event the VSCode integration uses to receive messages from
10      * the server. The messages are then propagated to the client
```

```
11        * or processed by the VSCode integration to provide
12        * functionality.
13        */
14       onServerMessage: vscode.Event<unknown>;
15   }
```

Listing 4.3: GLSP VS Code Server interface.

Optionally, the Connector's constructor takes additional "interceptor" arguments. We introduce the concept of interceptors so extension authors can control what messages the Connector sends and processes. Interceptors can be defined at four different locations within the message flow between GLSP components and the Connector. Two interceptors intercept messages before the Connector receives a message from the client or the server, respectively. These interceptors are called 'onBeforeReceiveMessageFromClient' and 'onBeforeReceiveMessageFromServer'. They can be used to modify incoming messages before they enter the realm of the Connector. Interceptors may also control whether the messages should even be processed by the Connector at all.

```
1   interface GlspVscodeConnectorOptions {
2
3       server: GlspVscodeServer;
4
5       logging?: boolean;
6
7       onBeforeReceiveMessageFromClient?:
8           (message: unknown, callback: InterceptorCallback) => void;
9
10      onBeforeReceiveMessageFromServer?:
11          (message: unknown, callback: InterceptorCallback) => void;
12
13      onBeforePropagateMessageToServer?(
14          originalMessage: unknown,
15          processedMessage: unknown,
16          messageChanged: boolean
17      ): unknown | undefined;
18
19      onBeforePropagateMessageToClient?(
20          originalMessage: unknown,
21          processedMessage: unknown,
22          messageChanged: boolean
23      ): unknown | undefined;
24
25  }
26
27  interface InterceptorCallback {
28      (
29          newMessage: unknown | undefined,
30          shouldBeProcessedByConnector?: boolean
31      ): void;
32  }
```

Listing 4.4: GLSP VS Code Connector Options interface.

The two interceptors ‘onBeforePropagateToServer‘ and ‘onBeforePropagateToClient‘ intercept outgoing messages coming from the GLSP VS Code Connector. The arguments passed to these interceptors will contain

- the original message the Connector received,
- the message the Connector intends to propagate, and
- a boolean flag to indicate whether the message it intends to propagate differs from the original message.

The latter exists to make expensive comparisons between original and propagated messages obsolete.

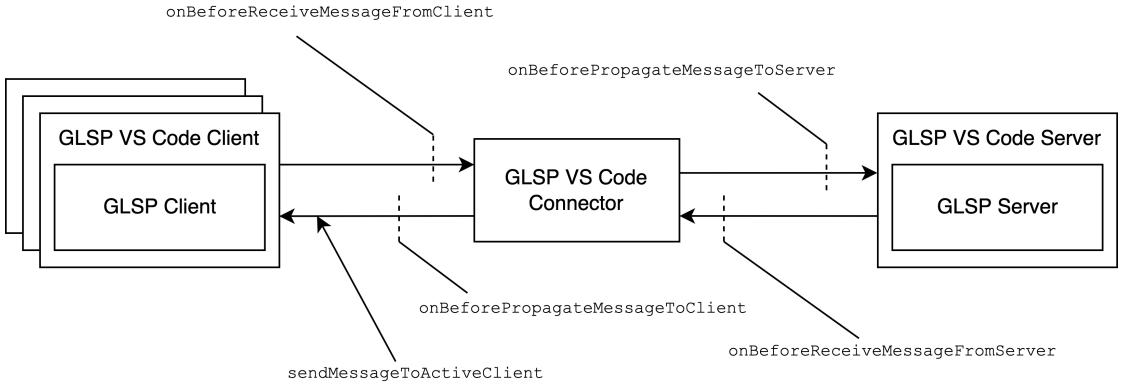


Figure 4.1: GLSP message flow through VS Code integration interceptors.

The constructor itself will attach a listener to the GlspVscodeServer’s ‘onServerMessage‘ event to listen for incoming messages from the server. Those messages are directly passed into the ‘onBeforeReceiveMessageFromServer‘ interceptor before the Connector processes the message internally. A private ‘processMessage‘ function is responsible for reacting to any incoming messages, including messages from the server and messages from the client. This function takes two arguments. One argument contains the message to process. The other describes its origin via a string (i.e., either “client” or “server”).

4.1.3 Server Message Processing

The Connector’s private ‘processMessage‘ function can be seen as a collection of hooks for different kinds of messages. It contains five hooks in total:

- The ‘handleSetDirtyStateAction‘ hook will trigger on received ‘SetDirtyStateAction‘s. Based on their content, it will set the editor’s saved state. This hook also

handles *undo* and *redo* functionality by firing the Connector's public 'onDidChangeCustomDocument' event with undo and redo operations. When VS Code calls either of these history operations, for instance when the user pressed the "undo" shortcut, it will send a GLSP 'UndoOperation' to the GLSP client.

- The 'handleSetMarkersAction' hook will react upon 'SetMarkersAction's and display GLSP Markers within the native diagnostics panel. The GLSP Marker kind maps to VS Code's severity grades ("info", "warning" and "error").
- The 'handleNavigateToExternalTargetAction' hook will navigate to an external resource when it intercepts a 'NavigateToExternalTargetAction'. This hook will set the message to propagate to 'undefined', meaning it will not be sent to the server when received from the client or vice versa. Extension authors may still choose to propagate the Action via an interceptor.
- The 'handleSelectAction' hook will fire the event behind the public 'onSelectionUpdate' field on the GLSP VS Code Connector with an array of selected elements. Extension authors can subscribe to this event to receive updates on any selected elements.
- The 'handleExportSvgAction' hook triggers a native save dialog when it receives an 'ExportSvgAction'. This hook will also prevent the message from being propagated.

After the hooks process a message, the Connector passes it to the 'onBeforePropagateMessageToClient' interceptor. The interceptor's result will then be sent to the client.

4.1.4 Public Fields and Methods

The interface of the GLSP VS Code Connector is the main interaction point for extension authors who want to integrate GLSP components into VS Code. Extension authors should use the Connector to register GLSP Clients, to handle model lifecycle and it may also be used to send GLSP actions to active (i.e., visible) clients. The Connector's complete interface can be found in Listing 4.1.

Registering Clients

Extension authors should use the public 'registerClient' method to notify the Connector about incoming messages. This method takes one 'client' argument, implementing the 'GlspVscodeClient' interface. Objects of this type resemble containers for all data relevant to a client. Relevant data includes a web-based webview panel (see section 4.2) and a document related to a client, which, in essence, resembles the edited or displayed diagram.

```
1 interface GlspVscodeClient<D extends vscode.CustomDocument> {
2     readonly clientId: string;
3     readonly webviewPanel: vscode.WebviewPanel;
```

```
4       readonly document: D;
5
6       /**
7        * This event emitter is used by the VSCode integration to pass
8        * messages/actions to the client. These messages can come from
9        * the server or the VSCode integration
10       * itself.
11       */
12      readonly onSendToClientEmitter: vscode.EventEmitter<unknown>;
13
14      /**
15       * The VSCode integration will subscribe to this event to listen
16       * to messages from the client.
17       */
18      readonly onClientMessage: vscode.Event<unknown>;
19  }
```

Listing 4.5: GLSP VS Code Client interface.

A ‘GlspVscodeClient‘ must contain a ‘clientId‘ field. The GLSP server will identify clients based on this ID. The webview panel belonging to the client needs to reside in the ‘webviewPanel‘ field and the responding document in the ‘document‘ field. The webview and document objects are created in the lifecycle methods of a ‘CustomEditorProvider‘. It is, therefore, a good place to call the ‘registerClient‘ function.

Similar to the ‘GlspVscodeServer‘ interface, ‘GlspVscodeClient‘ objects must expose an ‘EventEmitter‘, through which the Connector sends messages to a client and an ‘Event‘ through which it can receive messages from a client. These fields are called ‘onSendToClientEmitter‘ and ‘onClientMessage‘ respectively.

The ‘registerClient‘ function will direct messages received from the client through the ‘onBeforeReceiveMessageFromClient‘ interceptor before sending them to the ‘processMessage‘ method described in section 4.1.3. After the message passes through the hooks, it will run through the ‘onBeforePropagateToServer‘ interceptor before being sent to the GLSP server.

Additionally, when the webview panel belonging to the client is disposed of (i.e., closed by the user), the ‘registerClient‘ function will free any resources attached to the client. This includes diagnostics, message listeners, and a dedicated listener that will listen to an event triggered by the user when they switch editor tabs. Switching tabs must fire the ‘onSelectionUpdate‘ event, as different tabs usually have different elements selected.

Sending Actions to the active Client

The Connector exposes a method to transmit any GLSP message to the currently active client - “active” means focused by the user. This method, called ‘sendActionToActiveClient‘, will iterate through all GLSP webviews, select the active one if it exists, and pass its GLSP action argument. The VS Code API maintains the active state of all webviews, meaning that no additional state needs to be kept for this operation. The

‘sendActionToActiveClient‘ method can be used for various valuable functionalities, for example, to center the current diagram within the view, for navigation purposes, or to automatically layout the diagram, exclusively using VS Code native UI elements.

Document Lifecycle Methods

We provide methods on the Connector to facilitate managing the lifecycle of VS Code documents. Lifecycle methods currently include saving a document, saving a document to a new location (“save as”), and reverting it to its saved state. The lifecycle methods are intended to be used in a ‘CustomEditorProvider‘.

The ‘saveDocument‘ method will take a document, find the ‘clientId‘ which belongs to the document, and send a ‘SaveModelAction‘ to the corresponding client. The client will return this Action to the server, which persists the current document to the disk. A ‘DirtyStateChangeAction‘ from the server will enable the Connector to update the document to display a “clean” (unchanged / saved) state. Documents may be “saved as” (i.e., to a new location) by providing a URI as additional argument to the ‘saveDocument‘ method.

VS Code permits users to revert an opened text document to its last save state. Users can trigger this functionality by calling the native “File: Revert File” command via the command menu. This functionality is supported in our GLSP integration by calling the Connector’s ‘revertDocument‘ method. It will instruct the client to send a ‘RequestModelAction‘ to the server, the response of which (a ‘SetModelAction‘) will display the currently saved model state on the client. The examined behavior is equivalent to the native behavior for text files.

4.2 Webview

Webviews in VS Code are very similar to traditional web browsers’ `<iframe>`-tags [27] - to no surprise, as VS code renders web views as Inline Frames. VS Code, however, exposes a slightly different API to control them when compared to web browsers.

Webviews are initially created in a ‘CustomEditorProvider‘ and they are of type ‘Webview‘ as described by the VS Code API. We can set a webview’s content by binding its ‘html‘ field to arbitrary HTML code as string. To communicate with a webview’s content, we can use the VS Code API’s ‘postMessage‘ function to send any value or object. A webview’s content, on the other side, may call ‘postMessage‘ itself, after acquiring the VS Code API, to send messages. To receive messages from a webview, we can subscribe to its ‘onDidReceiveMessage‘ event.

The GLSP VS Code integration uses a pre-existing glue-code library called “sprotty-vscode” [28] to display the Eclipse Sprotty client within the webview. Using this library, we instruct Sprotty to receive and send messages through the VS Code Webview API - in essence, “mocking” the GLSP Server with our VS Code integration.

```
1  class ... {
2      // Injected component comes from @eclipse/glsp-client library
3      @inject(GLSP_TYPES.ICopyPasteHandler) copyPasteHandler;
4
5      initialize(): void {
6          // ...
7
8          window.addEventListener('copy', (e: ClipboardEvent) => {
9              this.copyPasteHandler.handleCopy(e);
10         });
11
12         window.addEventListener('cut', (e: ClipboardEvent) => {
13             this.copyPasteHandler.handleCut(e);
14         });
15
16         window.addEventListener('paste', (e: ClipboardEvent) => {
17             this.copyPasteHandler.handlePaste(e);
18         });
19     }
20 }
```

Listing 4.6: Registration of Cut, Copy & Paste listeners.

During the initialization process of Sprotty, we attach event listeners to the webview to handle cut, copy & paste events triggered by the user. The imported '@eclipse-glsp/client' library and the dependency-injection library 'InversifyJS' handle the internal processing of these events and propagate the necessary events to the GLSP server.

4.3 Quickstart Components

As described in the previous chapter, our contribution contains a set of "quickstart components" intended to help bootstrap new GLSP projects within VS Code.

4.3.1 GLSP Server Launcher

The 'GlspServerLauncher' is a small component that will start up a given Java-based GLSP JSON-RPC server process. Its constructor takes several arguments:

- `jarPath`: The path to the location of a Java Archive (JAR) file intended to be launched as process.
- `serverPort`: The port on which the server should listen for incoming client connections.
- `logging`: Whether the launcher should log the server's standard and standard error output to the VS Code Extension Host.
- `additionalArgs`: Arguments to pass when starting the server process.

When called, the launcher's ‘start‘ method will spawn a child process, running the defined JAR file using the locally installed Java runtime. The ‘stop‘ function can be invoked to kill the child process entirely. The ‘GlspServerLauncher‘ implements the ‘Disposable‘ interface. When disposed of, the launcher will call the ‘stop‘ function to stop the server.

4.3.2 GLSP Server Wrapper

The ‘SocketGlspVscodeServer‘ component wraps the server launcher component and implements the ‘GlspVscodeServer‘ interface to be directly used in combination with the GLSP VS Code Connector. The constructor takes a ‘serverPort‘ argument, which should point to the port of a GLSP server, in addition to a ‘clientId‘ and a ‘clientName‘ argument that identifies the JSON-RPC client.

During construction, the wrapper will start listening to incoming client messages on the ‘onSendToServerEmitter‘ field. It will propagate messages it receives from the server by firing the event behind the ‘onServerSendEmitter‘ field, consequently closing the message circle with the GLSP VS Code Connector.

4.3.3 GLSP Editor Provider

As mentioned previously, VS Code Extensions must register a ‘CustomEditorProvider‘ to use webviews. The VS Code API requires registered Custom Editor Providers to implement a set of methods:

- `saveCustomDocument` to save a custom document.
- `saveCustomDocumentAs` to save a custom document to a different location.
- `revertCustomDocument` to revert a document to its last saved state.
- `backupCustomDocument` to persist a “dirty” document to the disc when the user edits it without saving.
- `openCustomDocument` to create a document for a given resource. This function is called when a resource is opened that does not already have an editor assigned. This function must return a ‘Document‘ type that is passed to the ‘resolveCustomEditor‘ function.
- `resolveCustomEditor` to fill a webview with content and attach event listeners.

Additionally, an Editor Provider must expose an ‘onDidChangeCustomDocument‘ event which signals to VS Code that a document was edited and is now “dirty”.

We provide an abstract bootstrap component called ‘GlspEditorProvider‘ that implements the ‘CustomEditorProvider‘ interface and is compatible with the default GLSP components. The GLSP Editor Provider uses the GLSP Vscode Connector lifecycle

methods for saving and reverting documents. The Provider also reuses the ‘onDidChange-CustomDocument‘ field from the Connector to comply with the ‘CustomEditorProvider‘ interface. During the execution of the ‘resolveCustomEitor‘ method, the GLSP Editor Provider will attach listeners to the webview and the GLSP VS Code Connector to receive messages from the client and the server. Furthermore, it will send the necessary initialization messages to the GLSP components for a diagram to be displayed.

The abstract field ‘diagramType‘ on the GLSP Editor Provider can be set by extension authors to tell the GLSP server what diagram type to serve. Extension authors may use the abstract ‘setUpWebview‘ function to define how the editor webview is filled with content. Usually, it should be filled with the necessary scripts to run GLSP Sprotty.

4.4 Constructing an Example Extension

Using the components described above, we can construct an extension that supports the default GLSP component implementations.

We start by defining the ‘activate‘ function. This function is called when VS Code “activates” the extension based on an event we define in the extension manifest - we will choose the “*” event, which is equivalent to a “startup” event.

```
1  import * as vscode from 'vscode';
2  import { GlspServerLauncher } from '@eclipse-glsp/vscode-integration/
       lib/quickstart-components';
3  import MyCustomEditorProvider from './my-custom-editor-provider.ts';
4
5  export async function activate(context: vscode.ExtensionContext) {
6      const serverProcess = new GlspServerLauncher({
7          jarPath: '/path/to/server.jar',
8          serverPort: 5007
9      });
10
11     await serverProcess.start();
12
13     const serverProcessWrapper = new SocketGlspVscodeServer({
14         clientId: 'some.client.id',
15         clientName: 'SomeClientName',
16         serverPort: 5007
17     });
18
19     const glspVscodeConnector = new GlspVscodeConnector({
20         server: serverProcessWrapper
21     });
22
23     vscode.window.registerCustomEditorProvider(
24         'some.custom.editor',
25         new MyCustomEditorProvider(glspVscodeConnector),
26         {
27             webviewOptions: { retainContextWhenHidden: true },
28             supportsMultipleEditorsPerDocument: false
```

```
29          }
30      );
31
32      // Clean up disposables when extension deactivates
33      context.subscriptions.push(
34          serverProcess,
35          serverProcessWrapper,
36          glspVscodeConnector
37      );
38
39      workflowServer.start();
40  }
```

Listing 4.7: Activation function of the example extension.

We will start a GLSP server using our ‘GlspServerLauncher‘ quickstart component before wrapping it with the ‘SocketGlspVscodeServer‘ component to implement the correct interface for use with the ‘GlspVscodeConnector‘. Next, we need to register our Custom Editor Provider.

```
1  import * as vscode from 'vscode';
2
3  import { GlspVscodeConnector } from '@eclipse-glsp/vscode-integration'
       ;
4  import { GlspEditorProvider } from '@eclipse-glsp/vscode-integration/
       lib/quickstart-components';
5
6  export default class MyCustomEditorProvider extends GlspEditorProvider
        implements vscode.CustomEditorProvider {
7      diagramType = 'example-diagram';
8
9      constructor(
10         protected readonly glspVscodeConnector: GlspVscodeConnector
11     ) {
12         super(glspVscodeConnector);
13     }
14
15     setUpWebview(
16         _document: vscode.CustomDocument,
17         webviewPanel: vscode.WebviewPanel,
18         _token: vscode.CancellationToken,
19         clientId: string
20     ) {
21         const localResourceRootsUri =
22             vscode.Uri.file('/extension/path');
23
24         const webviewScriptSourceUri =
25             vscode.Uri.file('/path/to/webview/code.js');
26
27         webviewPanel.webview.options = {
28             localResourceRoots: [localResourceRootsUri],
29             enableScripts: true
30         };
```

```
31
32          webviewPanel.webview.html = `
33              <!DOCTYPE html>
34              <html lang="en">
35                  <body>
36                      <div id="${clientId}_container" style="height:
                            100%;"></div>
37                      <script src="${webviewPanel.webview.asWebviewUri(
                            webviewScriptSourceUri).toString()}"></script>
38                  </body>
39              </html>`;
40      }
41  }
```

Listing 4.8: The Example Extension's Custom Editor Provider extending the GLSP Editor Provider quickstart component.

After the provider is registered, we can initialize the client's JSON-RPC session, and the extension is ready to edit diagrams.

Now we can keep track of the selected elements using the connector's 'onSelectionUpdate' event, and we can register commands for users to interact with opened diagrams.

```
1  // Keep track of selected elements
2  let selectedElements: string[] = [];
3  context.subscriptions.push(
4      glspVscodeConnector.onSelectionUpdate(n => {
5          selectedElements = n;
6          vscode.commands.executeCommand(
7              'setContext',
8              'diagram.editorSelectedElementsAmount',
9              n.length
10         );
11     })
12 );
13
14 // Register various commands
15 context.subscriptions.push(
16     vscode.commands.registerCommand('diagram.fit', () => {
17         glspVscodeConnector.sendActionToActiveClient(
18             new FitToScreenAction(selectedElements));
19     }),
20     vscode.commands.registerCommand('diagram.center', () => {
21         glspVscodeConnector.sendActionToActiveClient(
22             new CenterAction(selectedElements));
23     }),
24     vscode.commands.registerCommand('diagram.layout', () => {
25         glspVscodeConnector.sendActionToActiveClient(
26             new LayoutOperation());
27     }),
28     vscode.commands.registerCommand('diagram.goToNextNode', () => {
29         glspVscodeConnector.sendActionToActiveClient(
30             new NavigateAction('next'));
```

```
31      }),
32      vscode.commands.registerCommand('diagram.goToPreviousNode', () =>
            {
33          glspVscodeConnector.sendActionToActiveClient(
34              new NavigateAction('previous'));
35      }),
36      vscode.commands.registerCommand('diagram.showDocumentation', () =>
             {
37          glspVscodeConnector.sendActionToActiveClient(
38              new NavigateAction('documentation'));
39      }),
40      vscode.commands.registerCommand('diagram.exportAsSVG', () => {
41          glspVscodeConnector.sendActionToActiveClient(
42              new RequestExportSvgAction());
43      })
44  );
```

Listing 4.9: Registration of custom commands and trackkeeping of selected elements.

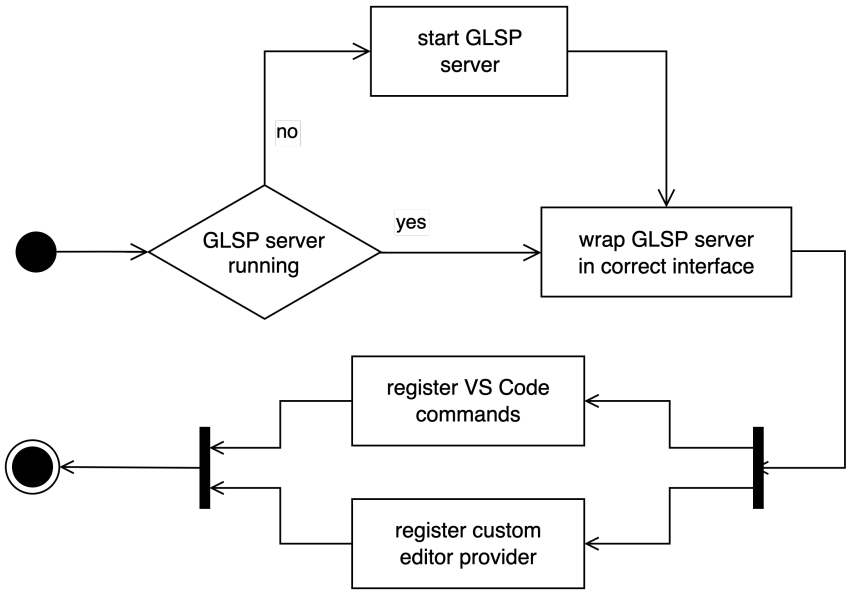


Figure 4.2: Activity flow for creating a GLSP VS Code extension.

The next chapter will showcase the functionalities we provided with our proposed integration based on our example extension.

CHAPTER 5

Showcase

This chapter will focus on the features and enhancements that have been made possible by our contribution. We will showcase the features using an example extension, which uses the default GLSP client, the default GLSP server and the default "workflow" diagram example provided by the GLSP project.

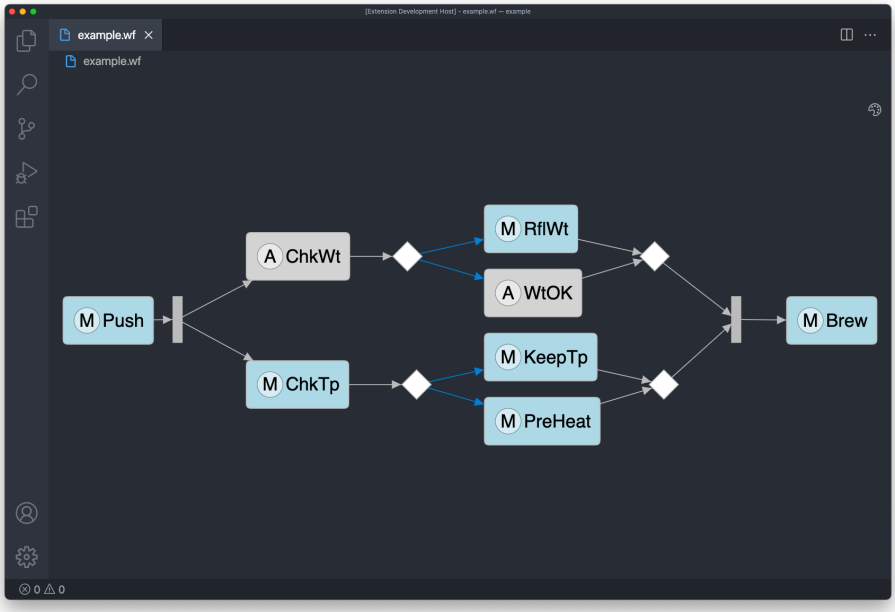


Figure 5.1: Default workflow diagram in our proposed example extension.

The example workflow in Figure 5.1 models the inner workings of a coffee machine: When a user pushes the button on the device, two parallel task flows get triggered - one to

check the water levels and one to check the machine's temperature. The machine will refill water and pre-heat itself if necessary before brewing the coffee. An "M" denotes manual tasks in the workflow, an "A" automatic ones. Parallelization is indicated by tall rectangles (i.e., "fork nodes" and "merge nodes"). Rotated squares are "decision nodes" and "merge nodes" and handle control flow. Edges may connect any two nodes and blue edges represent special "weighted edges". This example is intended to give an overview of the basic functionalities provided by GLSP.

5.1 Validations

The GLSP VS Code integration supports the native display of GLSP Markers. Using the "Validate Model" button in the pallette of the GLSP client, we can request the GLSP markers for the current model from the GLSP server. The integration will update the VS Code's diagnostics menu with the received markers. Diagnostics are displayed in the diagnostics view, the status bar, and their existence is also inidcated by the text color of the filename in the file explorer and editor tab (Fig. 5.2).

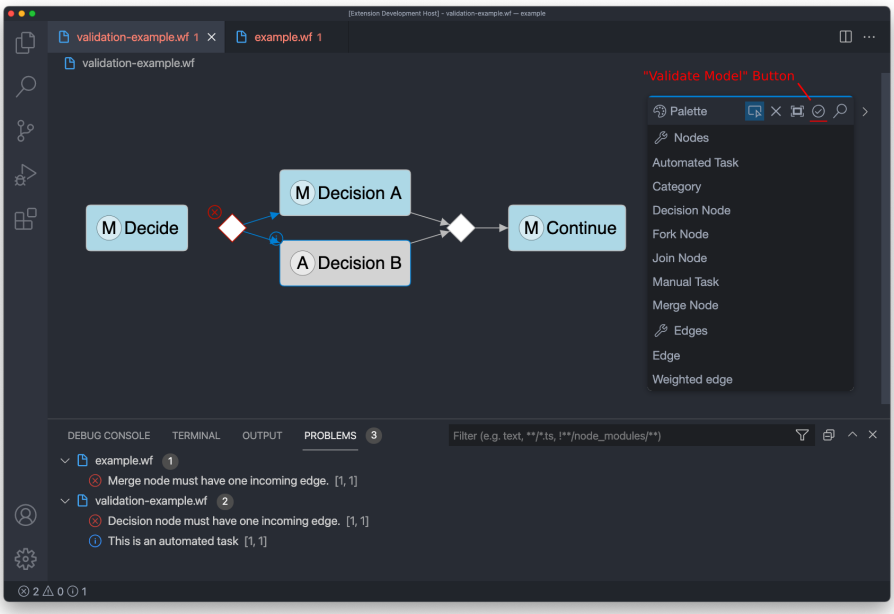
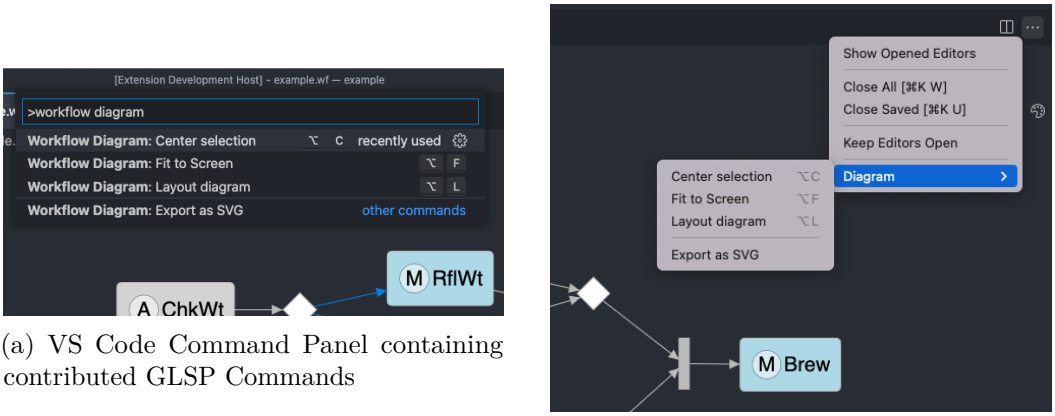


Figure 5.2: GLSP Markers displayed natively in VS Code.

Validations work across multiple files. Clicking on a validation opens the corresponding document. When a file is closed, its validations are also removed from the VS code UI.

5.2 Menu and Command Actions

Our example extension contributes a set of commands and menus to VS Code (Fig. 5.3). The GLSP VS code integration enables triggering GLSP actions for the active diagram through these native menus.



(a) VS Code Command Panel containing contributed GLSP Commands

(b) VS Code menu contributions

Figure 5.3: Interaction with the GLSP diagram through native VS Code UI elements.

The "Center selection" menu item and command send a 'CenterAction' to the client. This action will center the viewport around all currently selected elements (Fig. 5.4). If the command is triggered while no elements are selected, the viewport is centered around the entire diagram.

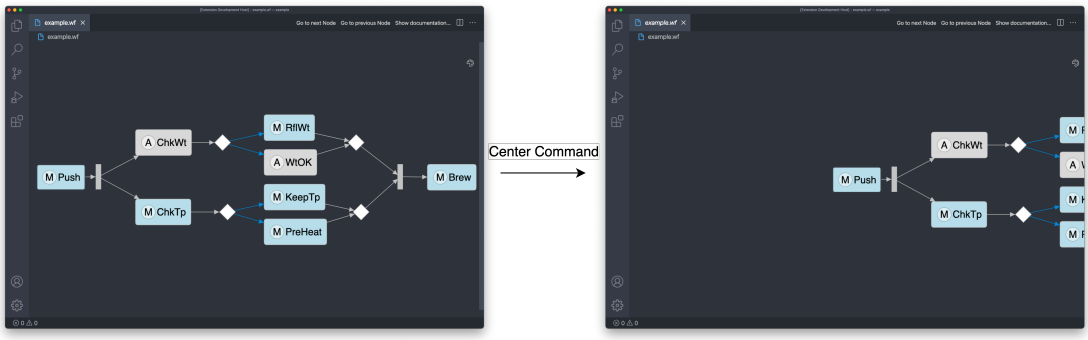


Figure 5.4: Diagram before and after centering viewport on the "Push" task.

The "Fit to Screen" command behaves similar to the "Center selection" command. It will fit the viewport to the selected elements (or the whole diagram) by sending a 'FitToScreenAction' to the GLSP client (Fig. 5.5).

The same pattern applies to the "Layout diagram" command, except it will reorganize diagram elements with a "LayoutOperation" (Fig. 5.6). This command changes the

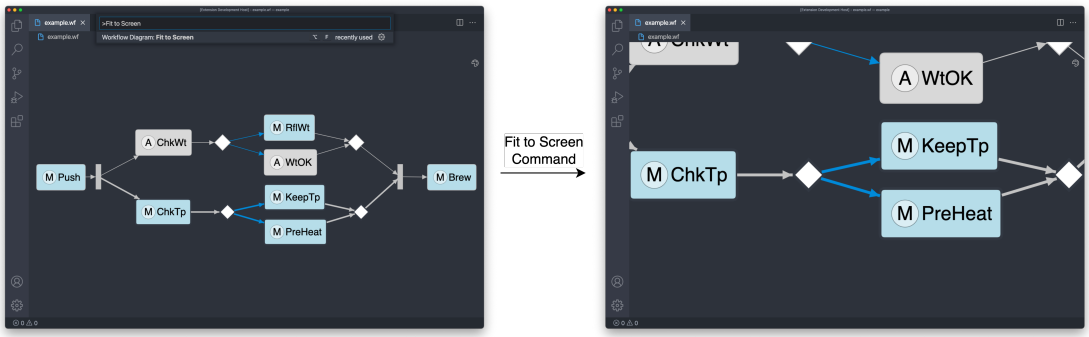


Figure 5.5: Diagram before and after fitting the selected elements to the viewport.

model, marking the document as dirty in the process.

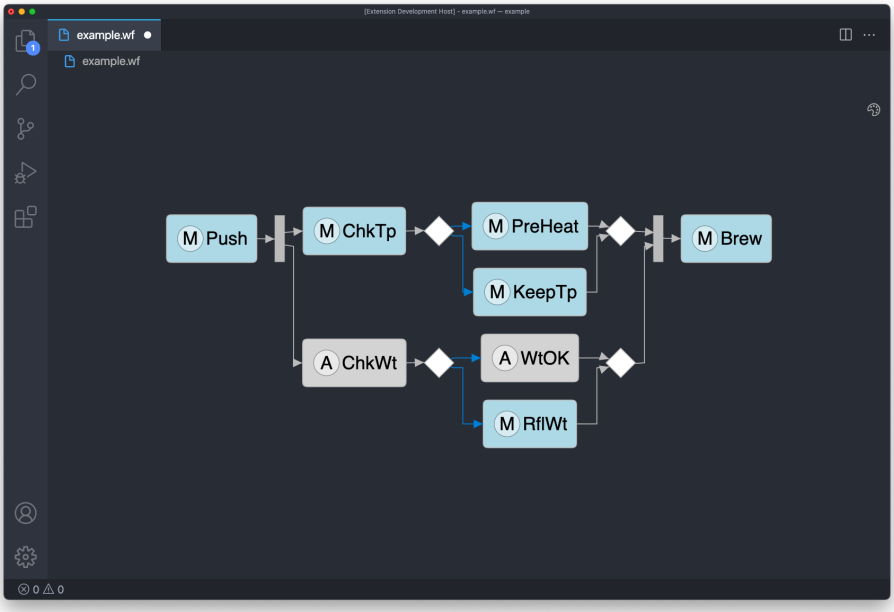


Figure 5.6: Diagram after running the “Layout diagram” command.

5.3 Selection Context

The GLSP VS Code integration makes it possible to control the visibility of menu items depending on the currently selected items. Our example extension contributes navigation menus that are only visible when a single diagram node is selected. This functionality is realized using VS Code’s “custom when clause contexts” [29] in combination with the ‘onSelectionUpdate‘ event exposed by the GLSP VS Code Connector.

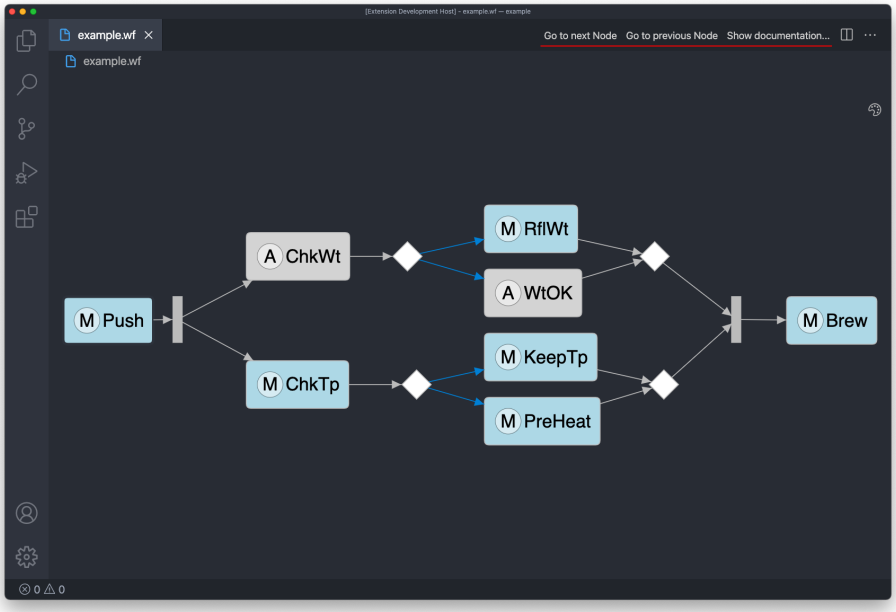


Figure 5.7: Navigation menu items are displayed because only a single node is selected.

The “Go to next Node” and “Go to previous Node” buttons (Fig. 5.7) will select the next/previous node along an edge, depending on the currently selected node. In addition to selecting the next/previous element, this command will center the viewport around the new selection (Fig. 5.8). The “Show documentation...” button will navigate to an external documentation file “example.md” (Fig. 5.9). External documentation only works for the manual “Push” task - the GLSP server defines this behavior.

5.4 Document State

Our integration will keep track of changes to a document in VS Code. This means we mark a document as dirty each time changes happen through the user or the server. VS Code will highlight dirty documents with a dot in the editor tab . When the user reverts a document to the persisted state, either through “undo” commands or through VS Code’s native “File: Revert File” command, a document will become clean again.

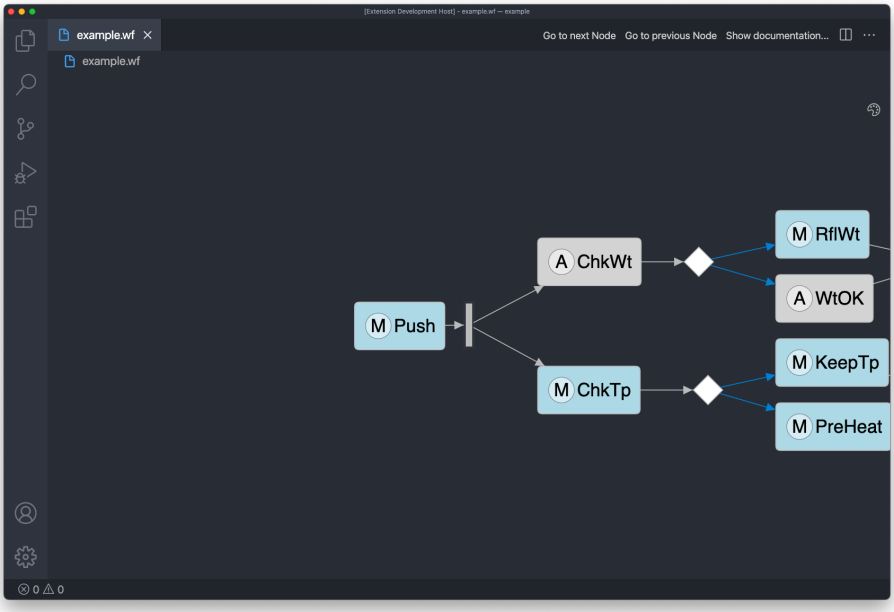


Figure 5.8: Fork node is selected and centered after pressing "Go to next Node". (see Fig. 5.7)

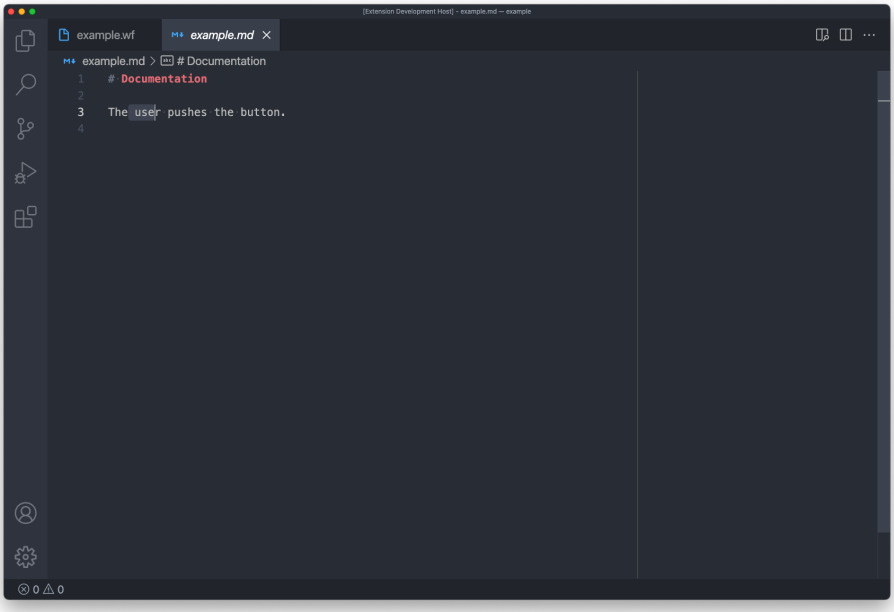


Figure 5.9: External navigation target (i.e. documentation) is opened after pressing "Show documentation...". (see Fig. 5.7)

CHAPTER 6

Discussion

This chapter aims to present observations during the research and development phase of this thesis, and critically reflect upon our contribution and the surrounding software components.

6.1 Limitations of the VS Code Extension API

During the development phase of this thesis, we identified some desired features that were either not fully supported or not supported at all by VS Code. We collect the found limitations below. It should be mentioned that the VS Code API is subject to change and thus our findings might be resolved in the future.

6.1.1 Title Bar Menu Items

In diagram editing software, it is typically possible to interact with a diagram through the title bar (i.e., the menu generally provided by the Operating System). VS Code does not provide a way to contribute menu items to the title bar through extensions. Extension developers can contribute menus in other places, like in the editor title bar or directly in the webview. While those menu items in the title bar are not strictly necessary, the deviation from expected UI practices may steepen the learning curve or hinder the adoption of GLSP through VS Code. For our proposed example extension, we used the editor title bar as a workaround.

6.1.2 Context menu in custom editors

While it is possible to contribute to the context menu (i.e., the menu that usually appears on a right-click) in text editors and other regions in the interface, context menus in webviews cannot be extended natively via Extensions. This issue does not prevent webviews from implementing custom context menus: It is possible for extensions

to prevent the native context menu from being shown, and webviews have enough capabilities to render a fully custom context menu when the user presses the right mouse button. Our integration library currently does not provide a context menu implementation. Nonetheless, future contributions may profit from the interceptor pattern we introduced in the GLSP VS Code Connector. In the future, GLSP context menus may be rendered using the information obtained through intercepting 'SetContextActions' defined by the GLSP protocol.

6.1.3 Obligatory Range Indicator on Diagnostics

When registering diagnostics for a document in VS Code, it is also necessary to provide a textual range the diagnostic applies to within the document. Omitting the range during registration results in an error, and the diagnostic will not be applied. This assertion is a problem for our integration, as we want to provide diagnostic functionality for non-textual documents. We considered the approach of defining diagnostic ranges to match the syntactic elements of the serialized data model. However, this would introduce significant development overhead for individual languages while providing no meaningful benefit to users. As a compromise, the GLSP VS Code Connector will set ranges to range from line 1 and column 1 to line 1 and column 1 when registering diagnostics. These ranges are guaranteed to exist, even for empty documents.

6.1.4 Diagnostic Navigation in non-textual Documents

When a user clicks on a diagnostic in the diagnostic view, VS Code will navigate to the affected file and highlight the affected code range with the cursor. The goal for our integration would be to navigate to the affected diagram element when a user clicks on a diagnostic. VS Code does not yet provide a clean way to control the behavior of diagnostic navigation, making the desired functionality only possible via a workaround that has considerable side effects.

The workaround we found during research involved appending the element id as a parameter to the URI parameter of the affected diagnostic resource. This change caused VS Code to always create a new document-webview-pair for the resource when the user clicks on a diagnostic. First, during the "open"-phase of the document, the VS Code integration would check for an appended URI parameter. If it exists, the integration would instantly reveal or open a webview for the original resource (without the parameter) and send a 'NavigateAction' to its client, triggering a visual navigation. Right after, in the "resolve"-phase, the integration will dispose of the newly created document (with the additional parameter).

While this workaround makes element navigation through diagnostics possible, it had the side effect of an editor tab appearing for a split-second each time a user clicked on a diagnostic belonging to a GLSP document. We decided against including this workaround in our contribution because the side-effect felt unnatural and interfered with the goal of having a native editing experience. In an attempt to make diagnostic navigation in

non-textual documents possible in the future, we opened a feature request in the public VS Code repository explaining the problem [30].

6.2 SWOT Analysis of the Graphical Language Server Platform

To dissect the Graphical Language Server Platform critically, we will conduct a SWOT analysis. A SWOT analysis looks at a subject's strengths, weaknesses, opportunities, and threats. We aim to gain insight into what we can use GLSP for in the future and how to improve it further. The points we list below should not be seen as a holistic overview but rather as observations while working with the technology.

6.2.1 Strengths

For its age, GLSP already has a large community of contributors and involved developers. Additionally, the project is hosted under the umbrella of the *Eclipse Foundation* [31], and it is actively developed and backed financially by *EclipseSource* [32]. A strong community and support is especially important for frameworks to ensure their adoption and development in future stages. We identify a promising outlook for GLSP, as the platform is very able to harness the community and backing behind its already established network.

Apart from the many advantages of client-server architectures we already mentioned, GLSP clients are strongly decoupled from the server - in many cases, even more than clients using the textual LSP. This paves the way for a lot of different use-cases and scenarios where a strong separation of concerns is critical. The high extensibility of the platform amplifies this strength, and the number of working integrations (i.e., standalone, Eclipse, Eclipse Theia, and VS Code) shows that its pattern of independent software components is working as intended.

6.2.2 Weaknesses

From a technical standpoint, GLSP chose a very opinionated architectural tradeoff with the server fully maintaining the state of documents and controlling possible editing operations. Compared to textual clients using the LSP, GLSP clients are very reliant on their language servers. In the textual LSP architecture, the language server never interferes with the actual editing of a document - a client will never be held up while editing text - this is not the case in GLSP. Even though GLSP servers have tight control over the clients, which can be beneficial, they also have a lot of responsibilities. For instance, when a GLSP server runs slow, the editing experience on the Client will also suffer. These kinds of dependencies might show even stronger in distributed environments, where language servers and clients are running on different machines and latency is a concern. In the future, it may be beneficial to explore use-cases where graphical clients

are entirely independent of graphical language servers, managing their own state and using servers only for auxiliary functionality.

The GLSP protocol includes references to the Eclipse Sprotty protocol in the form of interface descriptions and entire protocol messages. We assume the protocol was designed this way to avoid duplication across sister projects. From our observations, the dependency on the Sprotty protocol has several disadvantages: First, the reference to a different protocol decreases transparency. In order to fully understand the GLSP protocol, one has to understand the Sprotty protocol first. Second, dependencies are always in danger of drifting apart - a change in the Sprotty protocol may not always be compatible with features of the GLSP protocol. Lastly, the strong coupling with the Sprotty protocol may disincentivize the use of non-Sprotty clients. Because the GLSP protocol works very intensively with Sprotty's data types, it may be hard for existing modeling clients to adapt the GLSP protocol without considerable development efforts. A change in this regard may open up the market for GLSP to invite new client vendors to integrate the GLSP protocol and consequently the entire platform.

6.2.3 Opportunities

The market for decoupled editing tools is still relatively young. Until recently, reigning modeling software exclusively had a monolithic architecture, and the upsides of a decoupled architecture are evident. Especially in the VS Code ecosystem, there is a lot of room to grow for GLSP. There are very few extensions to generate and visually edit diagrams in VS Code, even less to edit diagrams for domain-specific languages.

The GLSP platform strongly profits from expansions of the VS Code feature set. It would be in the interest of the GLSP platform to support the development of webviews and diagnostics in VS Code to address the issues we identified in section 6.1.

6.2.4 Threats

The main threats to GLSP are competing technologies. While there have been advances in web-based modeling with client-server architectures [33, 34], we have not been able to identify similar technologies that expose and utilize a strictly defined protocol like the GLSP protocol. This makes it all the more likely that competing products will appear at some point since the potential market has not yet been fully tapped.

Interestingly enough, VS Code might present itself as a threat to the GLSP VS Code integration. It may very well happen that the editor's limitations we identified above will never be resolved. If this is the case, it would be detrimental to the integration as the usability hindrance imposed is non-trivial and might decrease the adoption of GLSP in the future.

6.3 Opportunities for improving the GLSP VS Code Integration

For our VS Code integration, we prioritized a set of features. This means that some features are still up for implementation or improvement. We will outline those features below:

- **Context menu.** VS Code does not support native context menus in webviews. Future development efforts could include a custom context menu implementation based on web technologies to handle GLSP's 'SetContextAction' message.
- **Disposal of the GLSP server.** The current GLSP VS Code integration simply kills the process hosting the GLSP server when the extension becomes deactivated. While this is a valid approach, a future implementation may shut down the server gracefully by sending a shutdown notification.
- **Restoring diagram viewport.** Currently, when an editor containing a GLSP document is closed and re-opened, the viewport will be reset to the original location and zoom level instead of the location the viewport had when the editor was closed. Future contributions may implement functionality to store a document's viewport state and restore it when a document is re-opened.
- **Document backups.** VS Code has a concept for backups of opened documents. Backups are used for unexpected application exits and to prevent data loss. The GLSP VS Code integration does currently not support this functionality.
- **Server message notification.** The GLSP VS Code integration does not yet intercept 'GLSPServerStatusActions' and 'ServerMessageActions' from the GLSP server. In the future, the integration could handle those actions and display the actions' content in the native UI (i.e., as "information messages", "error messages", or as "status bar messages").
- **Tool palette in VS Code interface.** Future contributions could experiment with having the tool palette right in the native VS Code interface instead of the webview, for example, in the "activity bar". Exploring this feature might give new insights from a usability perspective and also from a technical viewpoint.

CHAPTER 7

Conclusion

Summing up, we analyzed various technologies and architectures, including VS Code, the Language Server Protocol, and the Graphical Language Server Platform. We found that distributed client-server architectures have significant advantages over monolithic architectures when implementing modeling software with support for multiple DSLs. When comparing Microsoft's LSP with the Eclipse GLSP protocol, two protocols that aim to improve editing experience for two different domains, we found notable differences in design decisions. The most substantial differences lie in the handling of file synchronization and document ownership. While textual language servers using the LSP only provide auxiliary functionality and leave document ownership up to the clients, graphical language servers using the GLSP protocol have more authoritative responsibilities, like retrieving and persisting models and preventing the user from making faulty edits.
Concluding this thesis, we present our improved VS Code integration for GLSP as contribution to the open source Eclipse GLSP project[1]. Through this integration, VS Code extension authors may include GLSP components in VS Code extensions to enable a near-native model editing experience. We found that it was not possible to support all features we set out to implement due to the limitations of the VS Code extension API. The primary constraints are a lack of support for native context menus and diagnostics in non-textual documents. Future work may build on our contribution to expand the supported feature set of GLSP or to facilitate the integration of GLSP into VS Code even further.

[1] Our contibution to the Eclipse GLSP project as Pull Request on GitHub: https://github.com/eclipse-glsp/glsp-vscode-integration/pull/15

List of Figures

Listings

Bibliography

[1] GitHub. GitHub's engineering team has moved to Codespaces. https://github.blog/2021-08-11-githubs-engineering-team-moved-codespaces/. Accessed: 20.08.2021.

[2] Microsoft. Language Server Protocol. https://microsoft.github.io/language-server-protocol/. Accessed: 31.01.2022.

[3] Eclipse Foundation. The Eclipse Graphical Language Server Platform. https://www.eclipse.org/glsp/. Accessed: 15.01.2022.

[4] Eclipse Foundation. GLSP protocol. https://github.com/eclipse-glsp/glsp/blob/master/PROTOCOL.md. Accessed: 15.01.2022.

[5] Eclipse Foundation. Eclipse ide. https://www.eclipse.org/eclipseide/. Accessed: 31.01.2022.

[6] Eclipse Foundation. Eclipse Theia. https://theia-ide.org/. Accessed: 31.01.2022.

[7] Microsoft. Visual Studio Code. https://code.visualstudio.com/. Accessed: 16.02.2022.

[8] Eclipse GLSP. Eclipse GLSP VSCode Integration repository. https://github.com/eclipse-glsp/glsp-vscode-integration. Accessed: 16.02.2022.

[9] Eclipse Foundation. Eclipse GLSP GitHub repository. https://github.com/eclipse-glsp/glsp. Accessed: 21.01.2022.

[10] Microsoft. Visual Studio Code for the web. https://vscode.dev/. Accessed: 15.01.2022.

[11] Can I use... Browser support for File System Access API. https://caniuse.com/native-filesystem-api. Accessed: 15.01.2022.

[12] Microsoft. Visual Studio Marketplace - VS Code extensions. https://marketplace.visualstudio.com/VSCode. Accessed: 06.11.2021.

[13] Microsoft. Visual Studio Code Extension API. https://code.visualstudio.com/api. Accessed: 06.11.2021.

[14] Microsoft. List of tools supporting LSP. https://microsoft.github.io/language-server-protocol/implementors/tools/. Accessed: 06.11.2021.

[15] Microsoft. List of implementations of language servers. https://microsoft.github.io/language-server-protocol/implementors/servers/. Accessed: 06.11.2021.

[16] Microsoft. VS Code language server node package. https://github.com/microsoft/vscode-languageserver-node. Accessed: 15.01.2022.

[17] Philip Langer, Eclipse Foundation. Web-based Diagram Editors with GLSP | Cloud IDE Day 2021. https://www.youtube.com/watch?v=u4Wa_tx7R6Y, June 2021.

[18] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. Towards a language server protocol infrastructure for graphical modeling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, page 370–380, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Hendrik Bünder. Decoupling language and editor - the impact of the language server protocol on textual domain-specific languages. *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development*, 2019.

[20] Hendrik Bünder and Herbert Kuchen. Towards multi-editor support for domain-specific languages utilizing the language server protocol. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selić, editors, *Model-Driven Engineering and Software Development*, pages 225–245, Cham, 2020. Springer International Publishing.

[21] Eclipse Foundation. Eclipse Sprotty. https://github.com/eclipse/sprotty. Accessed: 15.01.2022.

[22] Eclipse Foundation. Eclipse Sprotty architecture. https://github.com/eclipse/sprotty/wiki/Architectural-Overview. Accessed: 15.01.2022.

[23] Eclipse Foundation. Sprotty client server protocol. https://github.com/eclipse/sprotty/wiki/Client-Server-Protocol. Accessed: 15.01.2022.

[24] Philipp-Lorenz Glaser and Dominik Bork. The biger tool - hybrid textual and graphical modeling of entity relationships in VS code. In *25th International Enterprise Distributed Object Computing Workshop, EDOC Workshop 2021, Gold Coast, Australia, October 25-29, 2021*, pages 337–340. IEEE, 2021.

[25] Microsoft. LSP proposal for pull-based diagnostic messages. https://github.com/microsoft/vscode-languageserver-node/blob/main/protocol/src/common/proposed.diagnostics.md#L1. Accessed: 19.01.2022.

[26] Microsoft. VS Code Disposable type. https://github.com/DefinitelyTyped/DefinitelyTyped/blob/05f47ff39cc69f63051b97933172cbb879ba2715/types/vscode/index.d.ts#L1495. Accessed: 06.02.2022.

[27] Mozilla and individual contributors. The Inline Frame element. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe. Accessed: 06.02.2022.

[28] Eclipse Foundation. Eclipse sprotty-vscode library. https://github.com/eclipse/sprotty-vscode. Accessed: 06.02.2022.

[29] Microsoft. VS Code when clause contexts. https://code.visualstudio.com/api/references/when-clause-contexts. Accessed: 08.02.2022.

[30] GitHub. Feature request for custom Diagnostic Navigation in VS Code repository. https://github.com/microsoft/vscode/issues/126623. Accessed: 09.02.2022.

[31] Eclipse Foundation. Eclipse GLSP project page. https://projects.eclipse.org/projects/ecd.glsp. Accessed: 10.02.2022.

[32] Eclipse Source. Glsp. https://eclipsesource.com/technology/eclipse-glsp/. Accessed: 10.02.2022.

[33] Christian Thum, Michael Schwind, and Martin Schader. Slim—a lightweight environment for synchronous collaborative modeling. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, pages 137–151, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[34] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Huseyin Ergin. Atompm: A web-based modeling environment. In *Joint proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013): September 29-October 4, 2013, Miami, USA*, pages 21–25, 2013.