Informatics

Kollaborative Modellierung in Echtzeit mit Eclipse GLSP

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Markus Hegedüs, BSc
Matrikelnummer 01526730

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 7. Dezember 2023

Markus Hegedüs Dominik Bork

Informatics

Real-time Collaborative Modeling with Eclipse GLSP

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Markus Hegedüs, BSc
Registration Number 01526730

to the Faculty of Informatics

at the TU Wien

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

Vienna, 7th December, 2023

Markus Hegedüs	Dominik Bork

Erklärung zur Verfassung der Arbeit

Markus Hegedüs, BSc

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, 7. Dezember 2023

Markus Hegedüs

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während dem Schaffen dieser Diplomarbeit unterstützt und motiviert haben.

Zuerst möchte ich meinen aufrichtigen Dank meinen Betreuern aussprechen: Ass. Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork und Univ.Lektor Dipl.-Ing. Dr.techn. Philip Langer. Ihre engagierte Betreuung und fachkundige Begleitung während meiner Diplomarbeit haben einen entscheidenden Beitrag zum erfolgreichen Abschluss dieser Arbeit geleistet. Für ihre hilfreichen Anregungen und die konstruktive Kritik, die maßgeblich zur Verbesserung meiner Arbeit beigetragen haben, bin ich aufrichtig dankbar.

Ein besonderer Dank gebührt allen Teilnehmern und Teilnehmerinnen meiner Evaluierungstests, ohne die, diese Arbeit nicht hätte entstehen können. Mein Dank gilt ihrer Informationsbereitschaft und ihren interessanten Beiträgen und Antworten auf meine Fragen.

Ebenfalls möchte ich mich bei meinen Kommilitonen Matthias Hofstätter, Marco Weber, Franz Kienegger und Philipp Fritz bedanken, mit denen ich gemeinsam fast alle Lehrveranstaltungen während des Studiums erledigt habe. Ein großer Dank gilt meinen Freunden, die mir das Studium erheblich erleichtert haben. Darüber hinaus möchte ich meiner Freundin Emma für das Korrekturlesen meiner Diplomarbeit danken.

Zum Abschluss möchte ich meinen herzlichen Dank an meine Eltern, Großeltern und Geschwister richten, die mir durch ihre Unterstützung mein Studium ermöglichten und stets an meiner Seite waren.

Ich widme diese Diplomarbeit meinem Opa „Didi“ und meiner Oma „Baba“. Danke für alles!

Markus Hegedüs

Kurzfassung

Der PC ist aus unserem Arbeitsumfeld nicht mehr wegzudenken. Er unterstützt uns während der Schul- und Studienzeit, beim Arbeiten, sowie auch in der Freizeit. Durch die Corona Pandemie ist diese Abhängigkeit nochmal enorm gestiegen. Viele Unterrichtsstunden wurden digital abgehalten, Freunde haben sich über Videotelefonie ausgetauscht, und viele von uns haben gänzlich remote gearbeitet. Jedoch bringt die dezentralisierte Verteilung einige Schwierigkeiten und Probleme mit sich, wie unter anderem die Gestaltung einer effizienten Zusammenarbeit mehrerer Personen. Hier kommt kollaboratives Arbeiten ins Spiel.

Im Kontext der Informatik, bedeutet kollaboratives Arbeiten, dass mehrere Personen, gleichzeitig an einem Dokument tätig sind. Dies gilt für textuelle, sowie für alle weiteren Arten von Dokumenten. Die folgende Diplomarbeit konzentriert sich auf Diagramme, sprich grafische Dokumente, welche mit GLSP (Graphical Language Server Platform) interagieren. GLSP bietet eine Plattform, welche ein Protokoll bereitstellt, mit dem Tools für die Modellierung von Diagrammen entwickelt werden können. Diese Arbeit soll das GLSP Protokoll erweitern, so dass es möglich ist, kollaborativ in Echtzeit an Diagrammen zu modellieren.

Das Untersuchen von existierenden kollaborativen Editoren, soll eine gute Grundlage schaffen, mit der klare Anforderungen für die Lösung definiert werden können. Die aufgestellten Fragen beschäftigen sich damit, wie GLSP erweitert werden muss, so dass es für kollaborative Zwecke genutzt werden kann. Des weiteren soll die Untersuchung einen Grundstein legen, um eine gute Benutzerfreundlichkeit innerhalb einer kollaborativen Session zu formen. Ein Prototyp, auf Basis des Workflow-Tools, soll das ausgearbeitete Konzept umsetzen und einen ersten Einblick in kollaboratives Modellieren mit GLSP bieten. Ebenfalls soll eine Evaluierung zeigen, dass es möglich ist, die Lösung ganz einfach auf andere Modellierungssprachen anzuwenden. Zum Schluss soll ein umfangreicher Echtzeittest die Lösung auf Performance, Benutzerfreundlichkeit und Zuverlässigkeit überprüfen.

Abstract

The PC has become an integral part of our working environment. It supports us at school and university, at work and in our free time. The coronavirus pandemic has increased this dependency enormously. Many lessons have been held remotely, friends have communicated via video calls and many of us have worked completely from home. However, working from home also brings a number of difficulties and problems with it, for example how several people can work together efficiently on the same thing. This is where collaborative working starts.

In the context of IT, collaborative work means that several people are working on a document at the same time. This applies to textual documents, as well as all other kinds of documents. This diploma thesis is concentrating on diagrams, i.e. graphical documents that interact with GLSP (Graphical Language Server Platform). GLSP offers a platform, which provides a protocol, to develop modeling tools for diagrams. This work is intended to extend the GLSP protocol, so it will provide collaborative real-time modeling on diagrams.

Analyzing existing collaborative editors should provide a good base upon to define clear requirements for the solution. The raised questions deal with how GLSP can be extended in order to use it for collaborative purposes. Furthermore, the investigation should lay the foundation for creating a good user experience within a collaborative session. A prototype based on the Workflow tool aims to implement the developed concept and provide a first insight into collaborative modeling with GLSP. An evaluation should also demonstrate that it is easily possible to apply the prototype to other modeling languages. Finally, a comprehensive real-time test will check the solution for performance, user-friendliness and reliability.

Contents

CHAPTER 1

Introduction

This chapter gives an insight into the motivation and the problem statement of the thesis. Furthermore, it defines the aim of the thesis. The research questions and the selected methodological approach are also defined and explained at this point. At the end there is an overview of all chapters of this diploma thesis.

1.1 Motivation & Problem Statement

The way we use our PC has changed a lot over the years. Thanks to modern technologies, it is almost entirely possible (e.g. for people who work in information technology) to perform their work remotely. The Corona pandemic has intensified this development even more.

An important technology that is often used in the course of remote working is collaborative working. A collaborative working environment deals with supporting eProfessionals to work together. The term collaborative software or also called groupware was already defined in 1991 by Ellis, Gibbs and Rhein: *"computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment"* [EGR91]. Instant messenger, software for video conferencing, application sharing, collaborative editing, etc. help to accomplish this. Collaborative editing means that several people can work on the same thing at the same time and changes are shown to all other people immediately [TRA$^+$12]. So it is possible to work together on a source file, on a wiki entry, on a video edit, or whatever. This diploma thesis will deal specifically with collaborative editing on diagrams. Therefore this research should give an insight on how it is possible to extend existing modeling tools with collaborative functionality.

A study [MBWM23] has shown that working with tools improves the use of models and modeling languages. It is important that these tools are easy to use, have no accidental complexity and good usability. However, existing modeling tools often have the problem that users struggle remembering the context information and therefore they cannot identify and fix errors and inconsistencies [PA18]. Tools should be as easy to use as possible, because it is assumed that in the future end users will design and manage models more and more themselves [SFB+14]. The support of modeling tools also increases the productivity of modeling. Integration with information systems should enable users to adapt the system structure and the behavior of the systems safely and conveniently [FS09]. Furthermore, tools that support collaborative modeling also enhance productivity [Rit10].

The Language Server Protocol (LSP) [lsp] was developed by Microsoft to make source code editors and IDEs (integrated development environments) independent of programming languages. The JSON-RPC based protocol is used for communication between editors/IDEs and language servers. It supports various features like code completion, syntax highlighting, refactoring, code navigation, etc.

However, LSP deals exclusively with textual languages. To make graphical modeling tools also independent of the modeling language, the Graphical Language Server Protocol (GLSP) was introduced by Eclipse Foundation. GLSP defines a language server protocol for diagrams of all kinds. Eclipse GLSP is a framework used to create custom diagram editors on the web or in the cloud. It is possible to integrate these editors into a web-based IDE or to host them standalone. However, this protocol is not yet designed to be integrated into a collaborative working environment. The general goal of this work is to solve this problem and make GLSP collaborative friendly [gls].

1.2 Aim of the Work

The aim of this work is to extend the Eclipse GSLP protocol so that it can also be used for collaborative purposes. Eclipse GLSP not only provides the protocol, it also provides a complete core library. This consists of a GLSP server, GLSP client, and integration projects in Eclipse Theia and Visual Studio Code. The server and client communicate with each other using actions and operations, which either modify the model or control the UI. GLSP server is a Java backend that holds the models, edits them, and then writes the associated files to the file system. GLSP client is a Node.js backend that controls the diagram editor. The two integration libraries for Eclipse Theia and Visual Studio Code are extensions to the GLSP client to simplify integration with the respective IDEs.

All these libraries are to be extended with the respective functionality to enable collaborative working. First and foremost, the goal will be to analyze the complete framework to see which method is the best to accomplish the synchronization of the actions and the model. First of all there is the possibility to implement the complete synchronization by yourself. Furthermore there is the option in Visual Studio Code to use their Live Share Server as support. This would solve some issues, such as sharing the workspace, out of the box.

Based of the information gained from the analysis, a prototype should be created, which performs initial synchronizations between several users and also visually displays these on the screen of the other users. An important goal of this diploma thesis will be to display one user's actions on the screen of other users in a clear and self-explanatory way. Afterwards a complete implementation of the prototype should enable the most important features of GLSP for collaborative work.

For evaluation purposes, the prototype will be applied to another modeling language. This should show that it can simply be used in other modeling platforms. By having a usability test with a number of test participants, the test should find out which strengths and weaknesses the first collaborative features have. The usability, performance and reliability of the prototype will be tested to gather information for further iterations of the implementation.

The following three research questions will be developed in the course of this diploma thesis and contribute to the validation of this work:

- **RQ1: What is the best possibility of the already existing implementation of the Eclipse GLSP framework to equip it with collaborative functionalities?**
  - This question is supposed to show different possibilities how it is feasible to extend the existing GLSP framework with collaborative functionalities. It should also determine which of these possibilities is the best for the prototype.
- **RQ2: What is an appropriate means to achieve a reliable and conflict-free synchronization of performed actions?**
  - To ensure collaborative working is feasible, it is essential that all performed actions are processed in a reliable and conflict-free way.
- **RQ3: Which representation of the performed actions shows them to the other users in the best possible and self-explanatory way?**
  - An important part of this work is to find out how the prototype can present performed actions to other users in such a way that they can understand and comprehend them. Collaborative work is only possible efficiently if all users of a collaborative session understand what is happening in it and where exactly everyone is currently working.

1.3 Methodology

To implement a suitable solution and answer the research questions the Design Science Research (DSR) [ARHR04] methodological approach with following methods will be used:

1. **Library & Framework Analysis**

- In order to understand how existing libraries and frameworks work, they must be analysed in advance. To achieve this, it is primarily important that the GLSP framework and the VS Live Share API are characterised and investigated in detail. This is needed to answer **RQ1**.

2. **Collaboration Literature & Requirements Engineering**
   - Collaboration functionality is already built into many existing tools. It is used in both textual and graphical editors. This method will draw conclusions from this and potentially integrate some already existing and well-established functionalities. This helps to answer the research questions **RQ2** and **RQ3**. Based on the accumulated knowledge, the requirements for the artifact will be determined.
3. **Conceptualization**
   - Before the prototype, which represents the artifact, can actually be developed, a concept needs to be created. This concept is designed based on the two previous methods and should describe the prototype in an abstract way.
4. **Prototyping**
   - In the next phase, a prototype is developed from the designed concept. This prototype extends the GLSP framework to include collaborative functionality. Developed functionality has to be evaluated and adapted in periodic cycles with the authors of GLSP. This prototype represents the artifact of the process.
5. **Evaluation**
   - The completed feature must finally be evaluated in two phases. First, it must be tested whether it can also be applied to modeling languages other than the one of the prototype, and it should also be tested with a group of test persons to see if it meets the usability requirements.

1.4 Structure

The diploma thesis consists of the following chapters:

1. **Introduction**
   - The current chapter gives an introduction to the thesis and describes research questions and applied methods.
2. **Foundations**

   – This chapter will explain the fundamentals necessary to understand the problem. A description of Visual Studio Code, Visual Studio Live Share, Collaborative Editing, Language Server Protocol (LSP) and the Graphical Language Server Platform (GLSP) is important for the whole thesis and the development of the prototype.

3. **Related Work**
   – An overview of existing collaborative editors provides a basis for determining the requirements of the prototype. Textual as well as graphical editors will be compared and analysed for different characteristics.

4. **Concept**
   – In this chapter, a concept for the prototype will be designed based on the requirements. This concept should define the protoype's extension of the GLSP protocol on an abstract level.

5. **Prototype**
   – Here the current implementation of the prototype based on the previously designed concept is presented. Code listings and screenshots of the current solution are shown here.

6. **Evaluation**
   – The implemented prototype will finally be evaluated. An important point is to check if the prototype can also be implemented to other modeling languages. This chapter also lists the results of two real-time usability tests.

7. **Conclusion**
   – Finally, this chapter will summarize the whole work by presenting answers of the research questions and provide an outlook for the future, where possible improvements to the solution will be described.

CHAPTER 2

Foundations

The following chapter presents the foundations of this thesis. That includes an explanation of Visual Studio Code (VS Code). This research explores especially on the GLSP implementation in VS Code. Furthermore, this chapter covers collaborative editing, which is essential for the rest of this thesis. This technique will be implemented in the next section: Visual Studio Live Share (VS Live Share). Since the existing GLSP tool is integrated into VS Code, VS Live Share is a perfect option as a collaborative framework. Language Server Protocol (LSP) and Graphical Language Server Platform (GLSP), which is based on LSP, are introduced at the end.

2.1 Visual Studio Code

Visual Studio Code is a source code editor that is available for various operating systems such as Windows, macOS and Linux. Visual Studio Code, also referred to as VS Code, was created by Microsoft. VS Code provides built-in support for programming languages such as JavaScript, TypeScript, Node.js. VS Code provides features like debugging, refactoring, syntax highlighting, code jumping out of the box. VS Code also offers the possibility to add other programming languages like C++, C#, Java, Python, PHP, Go, .NET via plugins, also called extensions. VS Code is largely an open source project and provides regular updates to its users. VS Code also has one of the largest communities on Github worldwide [vscb].

An important point of VS Code is the extensibility. The extension API offers the user the possibility to customize VS Code almost completely. Here, from the user interface to the editing can be customized pretty much everything. Furthermore, many main functionalities of VS Code are actually extensions that have been developed via the extension API [vscc].

Using the Extension API, a developer can customize the following areas:

- **Theming:** By means of a color or file icon theme the look of VS Code can be customized.
- **Extending the Workbench:** Custom components and views can be added in the UI.
- **Webview:** A webview can be created to display custom webpages developed with HTML, CSS and JavaScript.
- **Language Extension:** New programming languages can be added.
- **Debugging:** Add debugging functionality for a specific runtime.

2.1.1 Webview API

The Webview API [vscd] allows the developer to build a fully customizable webview into VS Code. Also the already provided Markdown extension uses the Webview API to display Markdown previews. The extension determines what is displayed in the webview as an iframe. HTML elements are rendered in the webview and the webview communicates with the extension via a message pipeline. There are three ways to build webviews into VS Code:

- The easiest way to create a webview is to use the `createWebviewPanel` method. Webview panels are displayed here as their own editors. So it is possible to display and visualize own UI components.
- Another option is to use a custom editor over the Customer Editor API. By means of a custom editor the developer can write own editors to edit every possible file in the workspace.
- And it is also possible to include webview views in sidebars or panel areas.

2.1.2 Custom Editor API

A custom editor [vsca] allows the developer to replace the standard VS Code editor with a custom editor for certain files. There are two parts of a custom editor. The first part is the view with which the user interacts and the second part is a document model, which is used by the extension to interact with the associated resource. The view part uses a webview to render the UI. This allows it to design the user interface using HTML, CSS and JavaScript. An important point is that the webview cannot access the VS Code API directly, but has to communicate via messages to VS Code and back as well. It is also possible that one custom editor opens multiple editor view instances for one document. The document model describes how the extension understands the resource or the file and how it has to work with it.

A distinction is made between customer text editors and custom editors. The biggest difference is how the document model looks like. A `CustomTextEditorProvider` uses `TextDocument` as its data model. A `CustomTextEditor` can be used for all text based file types and are easier to implement, because VS Code already provides a lot of features to work with text based files. With a `CustomEditorProvider` the extension defines the form of the document model itself. Binary file types can take a `CustomEditor` to edit them. But of course this makes it more complex for the extension, because it has to take care of much more and implement the features itself. A `CustomEditor` can also be used in `PreviewMode`, which is then only used to display binary file types. This reduces the complexity of course.

To activate a custom editor, there is a so-called contribution point, which is there to tell VS Code about all custom editors. A contribution point can easily be inserted in the `package.json`. There are several attributes like `displayName` of the custom editor, or the `filenamePattern`. So VS Code can determine for which files which custom editor should be called instead of the default editor.

An example of a contribution point to activate a custom editor could look like this:

```
 1  {
 2      ...
 3      "contributes": {
 4          "customEditors": [
 5              {
 6                  "viewType": "catEdit.catScratch",
 7                  "displayName": "Cat Scratch",
 8                  "selector": [{
 9                      "filenamePattern": "*.cscratch"}
10                  ],
11                  "priority": "default"
12          }]
13      }
14      ...
15  }
```

Listing 2.1: Example of a conribution point for a custom editor in package.json file.

This example at Listing 2.1 shows how a custom editor in `package.json` file is registered. For this purpose, one or more selectors are defined in line 8. The `filenamePattern` is one way to define a selector. In this example in line 9 all files with the file type `.cscratch` are opened with this `CustomEditor`. The `viewType` is a unique identifier for this custom editor. The implementation of the custom editor must also provide this unique type to create the mapping between contribution point and implementation. VS Code uses the `displayName` defined in line 9 to display the custom editor on the screen in various cases. VS Code uses the `priority` field to indicate which custom editor should be opened if there are multiple registered custom editors for a given file.

2.2 Collaborative Editing

The introduction of the mobile Internet has led to more and more people working together on all sorts of things using collaborative tools [EG89]. With the help of collaborative systems, groups of people spread all over the globe can communicate, work and exchange information with each other. Collaborative editing is the process of several people working on a document at the same time. There are different types of collaborative editing systems like collaborative text editing, collaborative graphical editing system, and so on [GG96].

Real-time collaboration uses a common protocol to exchange data across multiple users concurrently. To ensure a consistent view of the shared document, operational transform is a technique which is applied to the document. With this technique, each user owns their own copy of the document. When a user broadcasts changes to other users, these users must transform these changes before executing them. This is to ensure that no wrong changes are made to the document and that the own state is consistent with the state of other users [SE98].

A collaborative real-time editor is a software that allows working collaboratively on a document in an editor. The first collaborative real-time editor was demonstrated by Douglas Engelbart in 1968 in `"The mother of all demos"`. The first real concrete tool that supported collaborative editing was Instant Update in 1991, developed by `ONTechnology` for Mac OS. Within the tool, users could work together on a document over LAN on a server [TMGS97].

2.3 Visual Studio Live Share

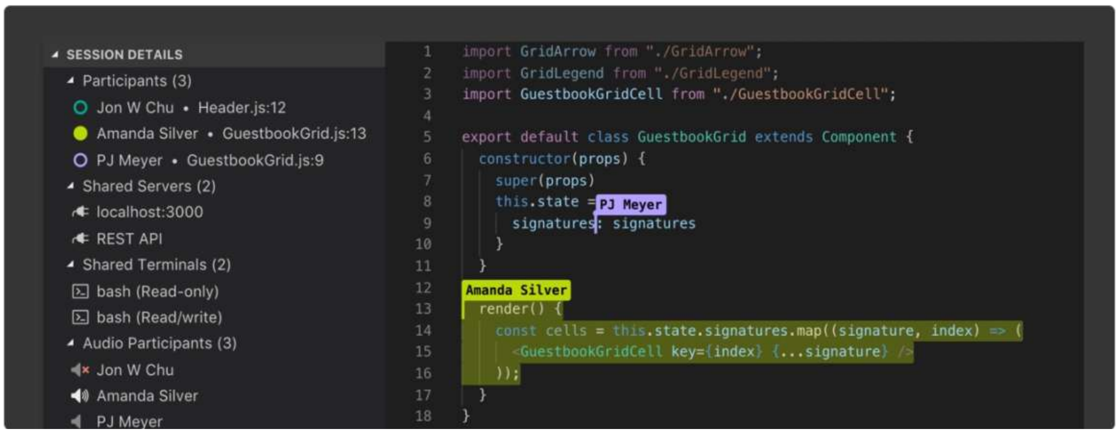
With Visual Studio Live Share [visa], it is possible for multiple developers to develop software together and do collaborative editing. Developers can simultaneously edit and debug a file in real-time, regardless of the programming language. It is easy to share the current project, host a debugging session, share the terminal, share web applications, communicate with each other via voice calls and much more.

The big advantage over classic pair programming is that developers can work with each other using Visual Studio Live Share while keeping their personal editor preferences. This includes the theme, keybindings, font size and so on. Furthermore, each participant has an own cursor. Visual Studio Live Share also allows following other users while they are working, as it shows all the cursors of all participants on the screen. Of course, in a real-time collaboration session it is also possible to work alone on own parts of the project. Visual Studio Live Share can be used in the Visual Studio IDE, as well as in Visual Studio Code or directly in the browser.

2.3.1 Concepts & Features

In a Visual Studio Live Share collaboration session, there is a single collaboration session host and one or more guests. The host starts the collaboration session and shares the share link with other people. They use this link to join a collaboration session. By joining the session, these participants automatically become guests. Guests join this session via an end-to-end encrypted peer-to-peer-connection and obtain access to the virtual filesystem. A collaboration session host can use all of its tools and services during the session. A guest, however, has only a selected set of things to do during the session. These allowed things are enabled by the host. Pretty much all shared content stays on the host machine and is not shared to the cloud or anywhere else. This increases security and results in shared content being shared when a guest enters the session. Once the host ends the session, all shared content is no longer accessible outside the host machine. Additionally, Visual Studio Live Share also deletes all temporary files created by the IDE/Editor during a session as soon as the host ends the session. Sharing means that the host shares the contents of a project or folder with guests. They can access the shared content via an invitation link. Also not only files are shared, also the debugging can be shared. In turn, joining means that a guest uses the invitation link to join a collaboration session. With joining the session, the guest gets access to all shared content unlocked by the host.

This subsection will also describe the features that come out of the box with Visual Studio Live Share.



```
import GridArrow from "./GridArrow";
import GridLegend from "./GridLegend";
import GuestbookGridCell from "./GuestbookGridCell";

export default class GuestbookGrid extends Component {
  constructor(props) {
    super(props)
    this.state =
      signatures: signatures
    }
  }

  render() {
    const cells = this.state.signatures.map((signature, index) => (
      <GuestbookGridCell key={index} {...signature} />
    ));
  }
}
```

Figure 2.1: Co-editing with multiple participants in a Visual Studio Live Share collaboration session.
Source: `https://learn.microsoft.com/en-us/visualstudio/liveshare/use/coedit-follow-focus-visual-studio-code` (Accessed: 2023-05-20)

Co-editing

Co-editing means that a participant of a collaboration session sees all other participants who are currently in this document when opening a document. The participant also sees the cursors of all other participants. In this way, the participant also sees selections or where other participants are currently located in the document. Furthermore, Visual Studio Live Share visually displays changes made by other participants. Figure 2.1 shows three participants in a collaboration session co-editing a document with each other. In line 9, the cursor of PJ Meyer gets highlighted in purple. From line 13 to line 16, the selection of participant Amanda Silver gets highlighted in yellow.

Following and focusing

If a participant wants to see what another participant is doing, it is possible to follow other participants. This means that jumps in the code, between files and so on are immediately followed. The participant who follows, also sees all the changes of the participant in focus. It is also possible that all participants follow a single participant and that this participant transmits all the steps live.

Co-debugging

Sometimes it is very useful if another participant helps debugging an issue. For this purpose it is quite easy to do collaborative debugging in Visual Studio Live Share. Here the debugging session is shared with all other guests.

Share server / Share port

The host of the debugging session can also share parts of its application to all guests. So it is possible that the host shares a port of its web application across the whole collaboration session. Not only a web application can be shared, but also local databases or REST endpoints. With this feature a local port of the host machine is shared with all guests. They can easily access this port on their local machine and communicate with the application/database/etc. of the host machine via Visual Studio Live Share.

Share terminals

As a host it is also possible to share the terminal with other guests. The terminal can be shared with other participants in read mode or read-write mode. This means that guests can also execute commands and read the output of the commands.

Other features

The host can grant or share access to special files or folders to selected guests. There are also two connection modes: direct and relay. In direct mode, guests connect to the host without going over the Internet. In relay mode, guests connected to a completely different network can communicate with the host over the Internet. In auto mode, a connection attempt is first made via direct mode, and if this fails, Visual Studio Live Share attempts to make the connection via relay mode. This auto mode is selected by default by Visual Studio.

2.3.2 VS Live Share Extension API

With the VS Live Share Extension API [visb] it is possible to include the features of Live Share in a Visual Studio extension. For this purpose, it is possible that the own extension establishes a collaboration session via Visual Studio Live Share and this extension can use the connection to send messages between participants. Also here, a host shares the collaboration session and afterwards one or more guests join this collaboration session.

Messaging

Messages are sent across RPC services. This service is shared by the host. The host can send notifications to all guests using this service. Guests can send notifications to the host via a proxy or they can also send requests to the host. When a request is sent, the host responds synchronously with a reply. Notifications do not have a response.

- **`shareService()`:** Provides an RPC service to all guests. The named service is shared by the host and creates a `SharedService` with the following methods:
  - **`onRequest()`:** When a request is sent to the service, the callback is executed and a response is sent back to the requester.
  - **`onNotify()`:** This callback is called when a notification is sent to this service.
  - **`notify()`:** This method sends a notification to all listeners.
- **`getSharedService()`:** Provides an RPC service to a guest which was provided by a host. This method returns a `SharedServiceProxy` with the following picked methods:
  - **`request()`:** Sends a request to the service and waits for a response.
  - **`onNotify()`:** This callback is called when a notification is sent from the service.
  - **`notify()`:** This method sends a notification to the service.

URI conversion

Guests can access the host's workspace via the URI schema `vsls:`. The path to a file is then relative to the root workspace. The `convertLocalUriToShared` or `convertSharedUriToLocal` method can be used to convert the URI from one format to another.

2.4 Language Server Protocol (LSP)

This section describes the Language Server Protocol [Bün19]. Year after year, new programming languages come onto the market. Furthermore, the number of integrated development environments is increasing. Many of these IDEs support a large number of programming languages. Language providers are also interested in supporting as many IDEs as possible. Concluding this means that if each IDE wants to support each programming language, the complexity increases to m-times-n (m*n).

To counteract this, the Language Server Protocol (LSP) was designed. The Language Server Protocol solves those problems by separating language-specific parts from the integrated development environment.

The Language Server Protocol was driven forward with the development of VS Code and is one of the reasons for the success of VS Code. LSP benefited a lot from its large community through its usage at this open-source editor [BL23].

This leads to the fact that programming language specific features like code completion, Goto definition, refactoring, syntax highlighting are no longer computed in the IDE itself. Instead, a language-specific server process, which can be developed in any programming language, takes over these calculations. The server process communicates with a client via the standardized Language Server Protocol. This client is then integrated into the IDE, e.g. via an extension. Decoupling the language implementation from the IDE the complexity leads to the fact that the complexity falls on m-plus-n (m+n). Since the launch in 2016, already more than 50 programming languages have been developed with the Language Server Protocol, including Java, TypeScript and COBOL. Also there are already client integrations for more than 10 IDEs. There are integrations for e.g. IntelliJ, Eclipse, VS Code.

2.4.1 Architecture of the Language Server Protocol

The protocol is sent in both directions in a server-client architecture. Communication is performed using the Language Server protocol, which is based on the stateless and lightweight JSON-RPC. Most implementations run on two different processes on the same machine. JSON-RPC is designed in such a way that server and client do not necessarily have to be on different machines.

JSON-RPC is a protocol that defines how a request, response or notification sent between server and client should look like [jso]:

- **Request**: A request can be sent by both the client and the server. A request always expects a response from the other instance. For this purpose, each request must have a method (string to define which function should be called) and can have parameters, which can be defined differently depending on the method. An ID must be included. It is then used so that the caller can map a response message back to the initial request message.
- **Notification**: A notification is a special request that does not contain an ID and therefore does not expect a response. This can also be sent by both client and server. Of course, the calling instance does not expect a response, but is simply processed by the other instance.

- **Response:** A response is sent in response to a request and contains the ID that is sent in the request. It also contains a result if the request was successful or an error object if the request failed.

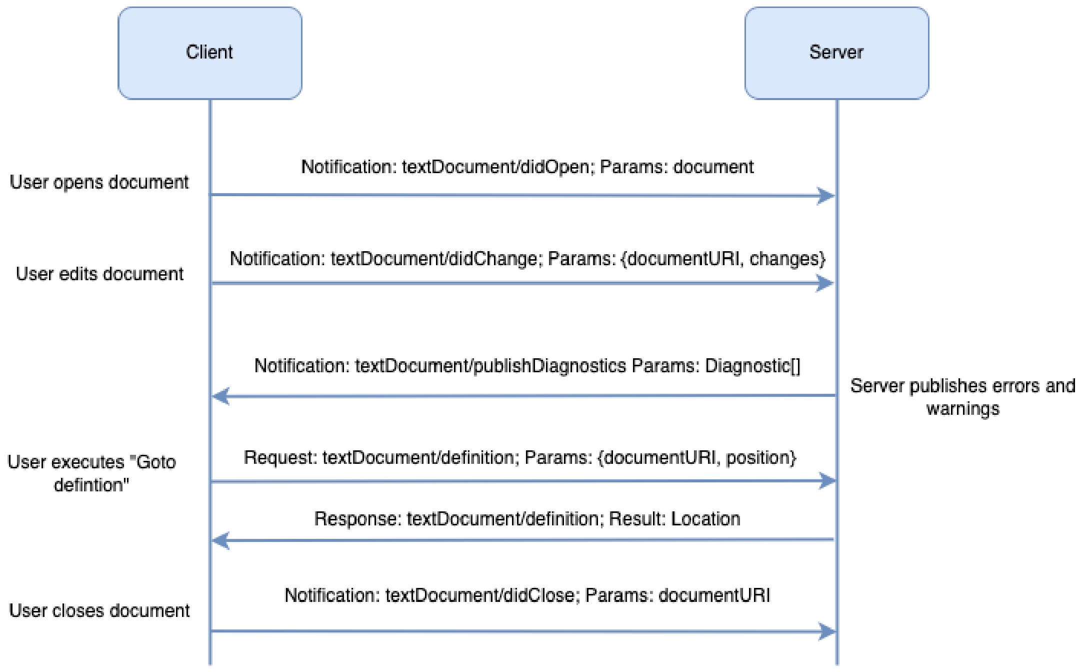

Client
Server
User opens document
Notification: textDocument/didOpen; Params: document
User edits document
Notification: textDocument/didChange; Params: {documentURI, changes}
Notification: textDocument/publishDiagnostics Params: Diagnostic[]
Server publishes errors and warnings
User executes "Goto defintion"
Request: textDocument/definition; Params: {documentURI, position}
Response: textDocument/definition; Result: Location
User closes document
Notification: textDocument/didClose; Params: documentURI

Figure 2.2: Communication between language server and client with LSP.
Source: `https://www.eclipse.org/community/eclipse_newsletter/2017/may/article1.php)` (Accessed: 2023-11-28)

Figure 2.2 shows two instances communicating with each other. On the left side it displays the client, which is integrated into the IDE and on the right side it displays the server, which acts as a language server. Between these two actors either requests, responses or notifications are sent back and forth. In this example, which is shown here, the user opens a document and thus a `textDocument/didOpen` notification with parameter `document` is sent to the server. Furthermore, the user also edits the document and a `textDocument/didChange` notification is sent with parameters: `documentURI and changes`. These notifications are then processed by the server. Next, the server sends a notification with all errors and warnings (`textDocument/publishDiagnostics`). The user now wants to execute a Goto definition, sends a request (`textDocument/definition`) and receives the `location` of the request as a response. The client uses this location to perform a Goto definition in the IDE. Finally the user closes the document and sends a `textDocument/didClose` notification with parameter `documentURI`. So the server knows that the client has not opened this document anymore.

2.4.2 Features of the Language Server Protocol

The Language Server Protocol supports more than 40 different message types in five operational categories. These categories are general, window, client, workspace and document. However, not every language and not every editor supports all features of the Language Server Protocol. Language and editor, use the `Capabilities` to check which features are supported. Therefore the initialize request tells the language server the `Capabilities`. This subsection will show features that are covered by the Language Server Protocol.

Code Completion

A code completion request sends the current position of the cursor and the respective document to the server. The server calculates all possible completion items and sends them back to the client. If the user then selects a completion from the list, another request is sent to the server to complete the completion.

Hover

A hover request sent to the server calculates information which should be displayed in a hover menu. The position is sent to the server and the server returns information like object information, text formatting, line-breaks, lists or indentations. Figure 2.3, at line 13, shows how a hovering at `Person` class over `Address` symbol would look like for TypeScript language. The hover menu shows all information like definition and attributes with their types about referencing `Address` class.

Goto Definitions

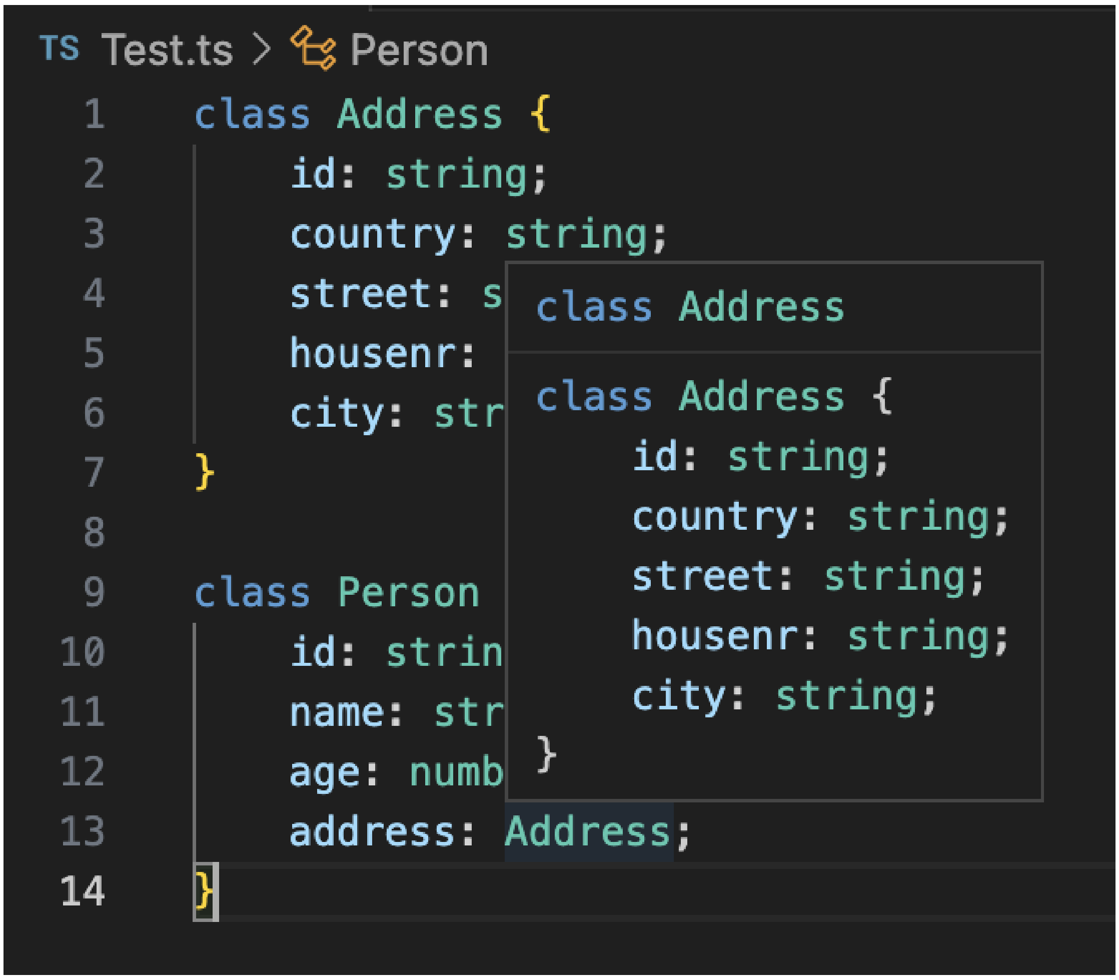
With the "Goto Definition Request" the server sends the position of the definition of a symbol within a document back to the client. Furthermore there is the type "Goto Type Request" which returns the position where a certain type has been defined. The "Goto Implementation Request" returns the position of the implementation of a certain symbol.

Find-References

The Find-References request returns a list of all occurrences of a given symbol within the selected project. This is also calculated using the current cursor position.

Diagnostics

Another important functionality covered by the Language Server Protocol are diagnostics. Errors and warning messages are calculated on the server and sent to the client. The task of the client is quite simple, because it is only responsible for displaying them.



```
TS Test.ts > Person
1    class Address {
2        id: string;
3        country: string;
4        street: s
5        housenr:
6        city: str
7    }
8
9    class Person
10       id: strin
11       name: str
12       age: numb
13       address: Address;
14   }
```

```
class Address

class Address {
    id: string;
    country: string;
    street: string;
    housenr: string;
    city: string;
}
```

Figure 2.3: Hovering over line 13 in Person class to show additional information about Address class.

2.5 Graphical Language Server Platform (GLSP)

The Graphical Language Server Platform (GLSP) [gls] is an open-source framework that enables developers to create web-based diagram editors. GLSP is developed by the Eclipse Foundations. GLSP is built on a client-server architecture and uses a protocol based on the Language Server Protocol to send data in both directions. The Language Server Protocol has been adapted to fit graphical modeling and diagram editors. All important functionalities like loading the model, interpreting, and editing the diagram are outsourced to a server. This means that editors do not have to worry about these issues. Furthermore, it is possible that GLSP can be easily integrated into various web-based editors. GLSP supports platforms like VS Code, Eclipse Theia, Eclipse RCP, standalone or also web apps.

A GLSP server works as a separate process and can host multiple sessions. It contains all language-specific content and editing capabilities. In addition, there is always a GLSP client that communicates with the GLSP server via a JSON-RPC protocol. Furthermore, a GLSP client can also have multiple sessions, which then interact with the integrated development environment.

The flexibility of GLSP also allows the deployment of the server to be flexible. In the standard case, the server and client are deployed on the identical machine. Alternatively, it is also possible for the server to be hosted on a different machine or, in the case of multiple servers, for these to be distributed across several machines. Then there is also the possibility that no server is required and the client has all required knowledge [BLO].

The GLSP server fetches the source model via a JSON file, EMF model or database and converts it into a graphical model. The graphical model can be serialized and is then sent to the GLSP client. The GLSP client renders this model in a webview. The client queries the server to find out which operations can be applied to the diagram. These can then be applied by the user in the webview to edit the diagram. Examples are adding new nodes and edges, moving elements, editing labels, etc. When the user performs an operation a message is sent to the server, which then adjusts the graphical model and sends an update back to the client. This client then renders the obtained graphical model in its webview.

2.5.1 Server

A GLSP server can be developed in any programming language, as long as it is possible to communicate with the client using the JSON-RPC protocol. Eclipse GLSP provides two frameworks that can be used to develop GLSP servers. These two have been developed in Java and in TypeScript. Both frameworks are very similar in structure and work with dependency injection (DI) to initialize all providers, services and handlers. For this purpose, the default implementations can be used or custom implementations that override the default variant.

The Java GLSP framework uses Google Guice [1] as dependency injection framework. In turn, the Typescript GLSP framework, which runs in a Node container, uses inversify.js [2] as its dependency injection framework. In both cases, an abstract `DiagramModule` class is provided. This must be overridden by the developer to be able to implement a concrete GLSP server. There are some abstract methods which have to be implemented. But there are also some already implemented methods which can be overridden to customize the functionality of the GLSP server. Mandatory are methods which determine where the source model is stored and a factory which should build the graphical model. Voluntarily editing operations and model validation can be overridden as well.

There are two dependency injection containers in the GLSP server framework:

[1] `https://github.com/google/guice`
[2] `https://inversify.io/`

- **Server DI Container**: All components and services that are not assigned to a session are configured in the Server DI container. These are also components and services that are used in parallel across all sessions.

- **Diagram Session DI Containers**: There is a plurality of session containers, because each client session gets assigned its own container. In this container handlers, states, etc. are initialized. Each session also gets its own diagram language and diagram module, which is responsible for this diagram language.

2.5.2 Architecture

Since the GLSP client is based on Sprotty, many concepts and also the model are largely adopted from Sprotty. Also the architecture was already conceptualized in Sprotty and will only be extended in GLSP. The architecture of Sprotty and GLSP is based on the Flux pattern and is also used by many other reactive web frameworks. The main point is the unidirectional event flow which is already used in several Flux frameworks. In contrast to the classic Model-View-Controller pattern, the data flow in the Flux pattern is clearly regulated.

The components in the Sprotty architecture [spr], displayed in Figure 2.4, show how the event flow is unidirectional. The main data flow is marked by the red arrows.

Starting from the view, an action is dispatched. These actions indicate a certain operation which is executed on the graphical model. Since they are also JSON objects they can be easily serialized and used as a protocol between client and server for exchange. Model elements are referenced by their ID in actions. The `ActionDispatcher` gets an action directly from the `ModelSource` or from the `Viewer`. It uses registered `ActionHandlers` which listen for actions to create commands. These commands are sent to the `CommandStack`. All operations run through the `ActionDispatcher`. This leads to the fact that `CommandStack` and also Viewer are never directly addressed. The `ActionHandler` listens for actions and converts them to commands.

`Commands` are the actual behavior of an action on the graphical model. A command has the methods `execute()`, `undo()`, `redo()`. Where all three methods get the current graphical model, modify it and return it. The `CommandStack` executes all commands it receives from the `ActionHandler` via the `ActionDispatcher`. It also holds the state of all changes executed via the `execute()` method. Additionally it stores the undo and redo stack. As soon as the new graphical model is calculated it is sent to the `Viewer`.

The `Viewer` works with the graphical model and creates a virtual DOM (Document Object Model). The `Viewer` also takes care of event listeners and listens for mouse events, for example. To create the virtual DOM the `Viewer` uses various `Views`. These describe how a virtual DOM is created from a graphical model. For this the `Viewer` looks up in the `ViewRegistry` which `View` is intended for which element type.

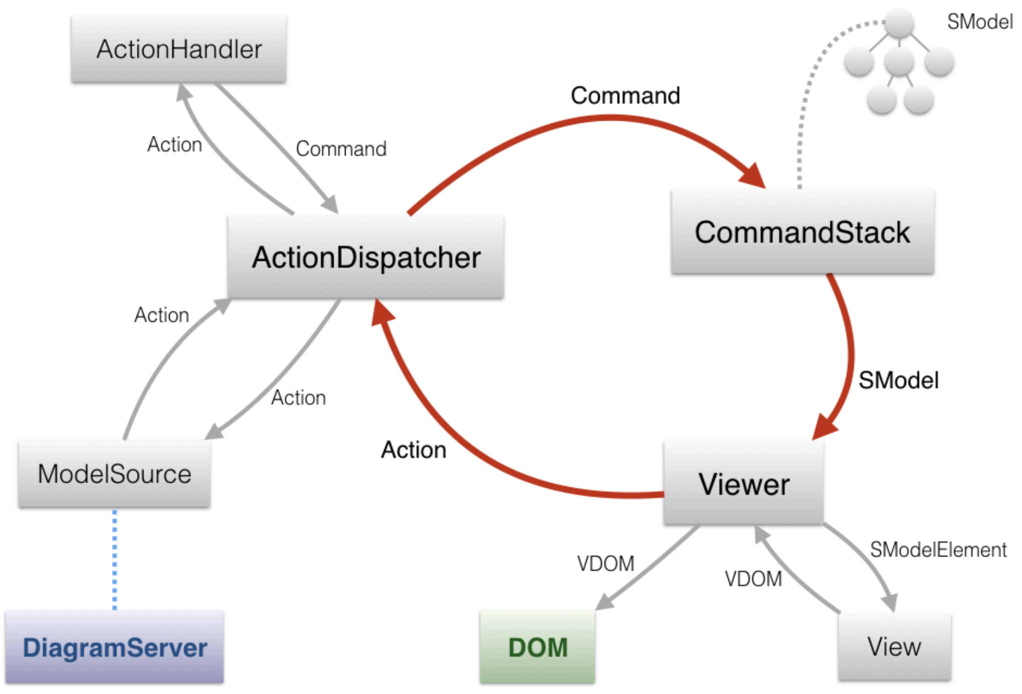


Figure 2.4: Event flow of the Flux pattern implemented in Sprotty Framework.
Source: `https://github.com/eclipse-sprotty/sprotty/wiki/Architectural-Overview` (Accessed: 2023-05-24)

2.5.3 Source Model

GLSP can handle any source model format or source model framework, since the loading of the source model and the transformation into a format understandable for the diagram has to be done by the developer herself. However, there are some default implementations for certain source model frameworks like EMF models [3], EMF.cloud model server [4] or could be simply transferred from JSON files.

To implement a concrete GLSP server a developer must provide the following three things for the source model:

- **Source model storage**: The developer must define how the source model is loaded and stored.

[3] `https://github.com/eclipse-glsp/glsp-server/tree/master/plugins/org.eclipse.glsp.server.emf`
[4] `https://github.com/eclipse-emfcloud/modelserver-glsp-integration`

- **Graphical model factory**: Those are responsible for defining how to create a graphical model from a source model.
- **Edit operation handlers**: These handlers edit the source model if the user performs actions on the diagram.

To load a source model and display it as a diagram, the client sends a RequestModelAction with a URI or other arguments to define the source model. Next, the server loads the source model via the source model storage. For this it uses the URI or other arguments given by the sent action. In the next step the server executes the graphical model factory to create a graphical model from this source model. Then as a response to the sent action a SetModelAction or UpdateModelAction is sent, which sends the graphical model from the server to the client. This client then renders the graphical model in the webview.

2.5.4 Graphical Model and Rendering

The graphical model is a serializable model that is transferred between the server and the client to render it on the client. This graphical model is created on the GLSP server from a source model via a factory. On the GLSP server all model elements typically have the prefix G. Each graphical model has a GModelRoot element which can have children. These can be either GShapeElement elemenst or GEdge elements. GShapeElements have a position and a size. This makes it possible for the webview to visualize the element later. There are different GShapeElements like GNode, GPort, GLabel, GCompoarment. GEdge in turn has no position and size. They have a source element and a target element which should be connected to each other. Normally GNode or GPort elements are then connected via GEdge elements.

At the GLSP client Sprotty [5] is used to render the diagram. Sprotty is a diagramming framework that uses SVG to render its model in a webview. Sprotty uses the SModel where all elements are marked with the prefix S. The GLSP model is based on the Sprotty model and therefore both models are compatible with each other. The Sprotty model has a SNode element in contrast to the GNode element in the GLSP model. For the GEdge element there is the SEdge element in Sprotty.

Of course, it is also possible to extend the graphical model with own elements. This is possible on the GLSP server as well as on the GLSP client. This work will also introduce new graphical elements to display the interactions of all users in the webview. The new element type has to be configured on the GLSP client.

```
export class WeightedEdge extends SEdge {
    override type = 'edge:weighted';
    probability: string;
```

[5] https://github.com/eclipse-sprotty/sprotty

```
 4 }
 5
 6 const workflowDiagramModule = new ContainerModule((bind, unbind, isBound,
       rebind) => {
 7     ...
 8     configureModelElement(context, 'edge:weighted', WeightedEdge,
           WeightedEdgeView);
10     ...
11 }
12
13 @injectable()
14 export class WeightedEdgeView extends EdgeView {
15   render(
16     edge: WeightedEdge,
17     context: RenderingContext
18   ): VNode | undefined {
19     const propabilitySVG = <text class-weighted-edge={true}>{edge.propability
       }</text>;
20     const edgeSVG = super.render(edge, context);
21
22     return (
23         <g>
24             {propabilitySVG}
25             {edgeSVG}
26         </g>
27     );
28   }
29 }
```

Listing 2.2: Example of how to create a new SModel element and then configure it in the container.

The Listing 2.2 shows how to define a new Sprotty element `WeightedEdge`, which inherits from `SEdge`. It overrides `type` so it has a new element type. Additionally it has another attribute `probability` of type `string`. This `WeightedEdge` element is then defined at line 8 using the `configureModelElement` function in the new container. The first parameter is the context, the second is an element type to identify the element (edge:weighted), the third parameter is the class of the model `WeightedEdge`, and the fourth and last parameter is the class of the view `WeightedEdgeView`, which then returns an SVG and renders the element. Starting from line 13, an example of a Sprotty view implementation is demonstrated. Overriding the `render(edge, context)` method provides an SVG out of the graphical model. In this example, a new SVG element from type `text` is created, which visualizes the probability. With `class-weighted-edge` an CSS-class to the referred element is set. Additionally it renders the edge from the inherited class `EdgeView`. Last but not least the method returns a `g` element, which is a container and can combine multiple SVG elements. This example combines the probability as a text and the edge.

2.5.5 GLSP protocol & Action types

This part summarizes some of the action types referred to the GLSP protocol.

Server-Client Lifecycle

The sequence in which the client and server communicate is actually always the same. To initialize the server, the client sends an `InitializeServer` request at startup. The server responds with an `InitializeResult` and until then all communication is blocked. A client identifies itself over an `applicationId` and sends with which `protocolVersion` is used. The `InitializeResult` also provides information about all action kinds which are supported.

When the client is initialized at the server and the user then opens or creates a diagram, the client sends an `InitializeClientSession` request to the server. For each opened diagram a new session is created again at the server. The server also creates a DI session container for this diagram and this client. As parameters a `clientSessionId` and a `diagramType` is sent.

When closing a diagram a `DisposeClientSession` request is sent to the server. This also disposes the DI session container at the server. As parameter the previously created `clientSessionId` is used as identification. When the client switches off, it can send a notification to the server so that it can clean up all remaining resources.

Between the initialization and the dispose of the client session, the server and the client communicate exclusively via `ActionMessages`.

If the GLSP client wants to stop communicating with the GLSP server, another `ShutdownServer` message is sent to the server.

Model Data

The `RequestModelAction` is normally the very first action sent from the client to the server. With this request the client gets back a `SetModelAction` or `UpdateModelAction`. These both return the graphical model and are rendered at the client. With the `UpdateModelAction` a transition of the graphical model from the old to the new model can also be animated.

Model Saving

A `SaveModelAction` stores the graphical model back to the source model. A `fileUri` can be passed as an attribute, which would save the model to another location. The server sends a `SetDirtyStateAction` if the current model state differs from the persisted model state of the source model. This is used at the client to indicate to the user that the document is not saved. An `ExportSvgAction` takes the sent SVG and saves it to the defined file system.

Model Layout

GLSP takes over the calculation of all bounds of all elements. Calculating the correct bounds is not so simple and depends partly on client properties. To handle this the server sends a `RequestBoundsAction` to the client. The client gets a model, renders it invisibly and sends back a `ComputedBoundsAction` with the elements and the corresponding bounds.

Model Edit Mode

In GLSP it is possible to edit the model in various ways. For this there are different modes which are set by the client to the server by a `SetEditModeAction`. The server can then react differently to operations.

Client Notifications

In GLSP there are also actions which are executed specifically on the client. These client-side actions can be initiated by the client as well as by the server. Viewport actions change the viewport to improve the usability of the diagram. For notifications, the client distinguishes between a status and a message. Both can be displayed differently. For example, in Theia integration, status updates are displayed directly in the diagram and messages are displayed as popups. Furthermore, there is a `SelectAction` or a `SelectAllAction`, which changes the state of the selected elements. With this state the client can mark elements as selected.

Element Hover

As soon as the user hovers over an element a `RequestPopupModelAction` is sent to the server. The server sends a `SetPopupModelAction` to the client with new elements which are rendered in a popup by the client.

Element Validation

As in many modeling languages, it is also possible to validate elements with GLSP. For this purpose, the client requests `Markers` for `elementsIDs` sent along using `Request-MarkersAction`. A `Marker` contains the validation result for an element. The server then sends `Marker` to the client using a `SetMarkersAction`. A `DeleteMarkersAc-tion` removes all markers on an element.

Element Navigation

In GLSP there are different types of navigations. There are default variants which are used by the client and server to implement concrete navigation types. The client can request navigation targets from the server by means of an action for a certain navigation type. The `NavigationTarget` describes an object to which the client should navigate.

Element Type Hints

Element Type Hints are used to determine which modifications are allowed on different element types. `ShapeTypeHints` are for elements of type `GShapeElement` and `EdgeTypeHint` for elements of type `GEdge`. The client sends a `RequestTypeHints-Action` and gets as response a `SetTypeHintsAction` with all allowed modifications for requested element types.

Element Creation and Deletion

To process the model, the client sends so-called operations to the server. `Operations` edit the graphical model and as a response the server sends an `UpdateModelAction` to the client. New elements are added either by the `CreateNodeOperation` for nodes or `CreateEdgeOperation` for edges. This is done by sending either the coordinates for nodes or source and target element for edges as attributes. To delete elements the client sends the `DeleteElementOperation`.

Node Modification

A `ChangeBoundsOperation` changes the position or size of a node. The client sends this to the server when a node element is modified. A `ChangeContainerOperation` places a node into a new container.

Edge Modification

To modify an edge, the client can send a `ReconnectEdgeOperation` to the server. This operation connects an edge to a new node. Likewise, a `ChangeRoutingPointsOp-eration` can edit the routing points of an edge.

Element Text Editing

In GLSP, textual inputs from a user are checked in advance by a validation. For example, if a user edits the label of a node, the client sends a `RequestEditValidationAction`. In response the server sends a `SetEditValidationResultAction` which contains a `ValidationStatus`. To execute the changes to a label performed by an user the client sends an `ApplyLabelEditOperation` to the server.

Clipboard

In GLSP clipboards are handled directly on the client. However, the conversion of a selection into a clipboard readable format, as default `application/json`, is handled by the server. For this there is the `RequestClipboardDataAction` which gets a `SetClipboardDataAction` as response. Additionally there is a `CutOperation` which deletes the selected element and copies it to the elements in the clipboard. The `PasteOperation` loads the clipboard data into the graphical model back.

Undo / Redo

The server stores the command stack of all executed commands on a graphical model in its state. The `UndoAction` performs an undo of the last executed command. A `RedoAction` executes a redo of the last undone command.

Contexts

Within a context the client can execute special actions of a `contextId`. There are the following contexts: Context Menu with id `context-menu`, Command Palette with id `command-palette`, Tool Palette with id `tool-palette`.

2.5.6 Platform Integrations

It is also possible to integrate GLSP based editors into any web application. Most IDEs offer the possibility to build extensions or plugins that offer a webview in which the GLSP editor can be integrated. There are already modules for the integration in Eclipse Theia [6], VS Code [7] or Eclipse IDE [8]. Furthermore, it is also possible to implement a standalone web editor, which is built on plain JavaScript.

[6] `https://github.com/eclipse-glsp/glsp-theia-integration`
[7] `https://github.com/eclipse-glsp/glsp-vscode-integration`
[8] `https://github.com/eclipse-glsp/glsp-eclipse-integration`

CHAPTER 3

Related Work

This chapter describes already existing collaborative editing tools and also talks about the advantages and disadvantages of these tools. Textual and graphical collaborative tools will be demonstrated in this part. Finally, a comparison of all the tools is presented in a table to define requirements for the prototype.

3.1 Textual collaborative document editing tools

This section focuses on textual collaborative editing tools. These tools are checked for defined criteria. Google Docs Editors and Etherpad, an open source alternative to Google Docs, will be introduced. And it will also give a look into Jetbrains' Code With Me, an alternative to VS Code's Live Share.

3.1.1 Google Docs Editors

Google Docs Editors [goo] is a set of document collaboration tools which are used not only in companies but also by individuals and students. Google Docs Editors is an office suite released by Google in 2006. All tools are provided as part of the Google Drive service. This office suite consists of following applications:

- **Google Docs** is an online word processor.
- **Google Sheets** is a spreadsheet application.
- **Google Slides** is a presentation software.
- **Google Drawings** is a vector drawing program.
- **Google Forms** is used for online forms and surveys.

- **Google Sites** is a graphical website editor.
- **Google Keep** is an application for taking notes.
- **Google Fusion Table** was a data table management tool (until 2019).

Features

All these tools build on the collaborative functionality that allows multiple users to work on them simultaneously. At the forefront, Google Docs Editors can be used as a web-based browser application, as a mobile app on Android and iOS, or as a desktop application for Google's ChromeOS. With Google Docs, it is possible for multiple people to work on a document in real-time. All changes are immediately visible for all other participants. Also cursor positions and selections of other users are shown in a special color to other users like in Figure 3.1.

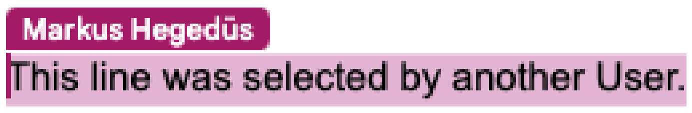
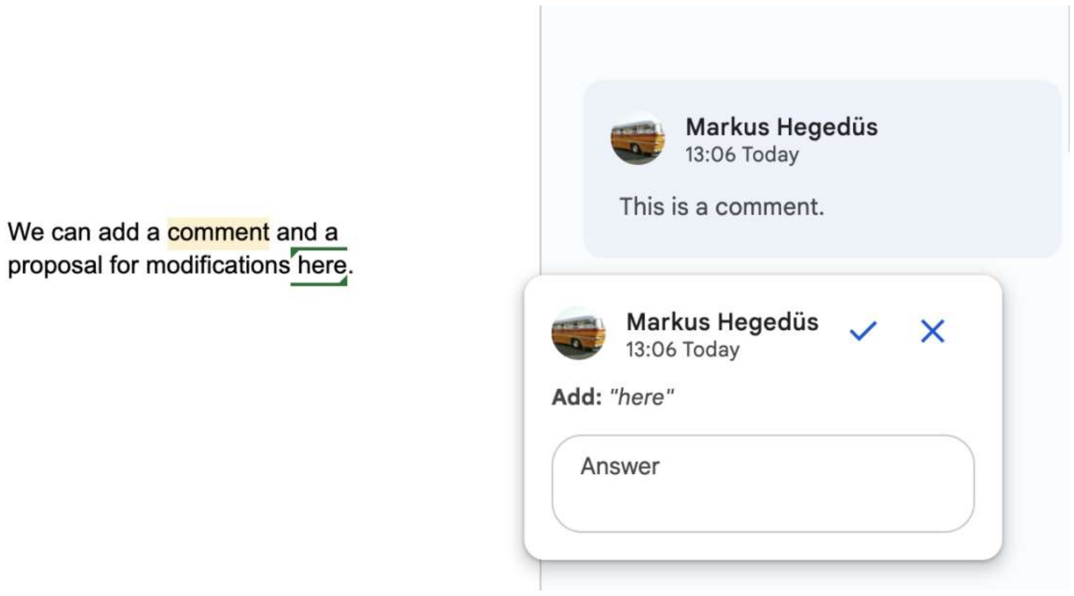


Figure 3.1: Example on how Google Docs displays selections in assigned colors.

It is also possible to add comments and proposals for modifications at various places in the document and distribute them to other users. Figure 3.2 shows how this works. This example demonstrates how a comment to word *comment* is added and it shows a proposal for a modification to add word *here* at the end of the line.

A permission system regulates the permissions a user has. For example, users can share or lock parts of the document with other users. By means of the revision history it is possible to track all changes. Thus, participants can see what other participants have changed, when and where. Also with the revision history the color, which is assigned to a participant, is used to show changes of other participants visually. If a user does not like changes made by other users, edits can still be reverted afterwards.

If a user does not like changes made by other users, edits can still be reverted afterwards. All changes are stored on Google's server. Simultaneous changes from different users are processed via the Jupiter algorithm and an operational transformation. All changes of the corresponding document are saved and applied, as well the document gets autosaved per change. The document is stored in the creator's Google Drive space by default. However, it is also possible for the creator to create or move the document to collaborative Google Drive spaces.


We can add a comment and a proposal for modifications here.

Markus Hegedüs
13:06 Today
This is a comment.

Markus Hegedüs
13:06 Today
Add: "here"

Figure 3.2: Example on how Google Docs displays comments and proposals for modifications.

There is also a Google Chrome extension [1] which allows the user to use Google Docs offline. With the mobile app for Android and IOS it is possible to use the application offline out-of-the-box. Various file types are supported for importing and also for exporting. This includes Open Document, Microsoft Office, HTML, Rich text format.

Pricing & Source Code availability

All these main features are also available for all other Google Docs Editors like Google Sheets, Google Slides, etc. Google Docs Editors is free for private use, but for use in professional environments a Google Workspace plan has to be bought. This also includes other collaboration tools such as Google Meet. Although the use of Google Docs Editors is free, the code is not freely available and thus the whole project is closed source.

3.1.2 Etherpad

Etherpad [etha] is an open source alternative to Google Docs. Etherpad is a web-based text editor which can be used for collaborative editing. Several participants can work together on a document in real-time. Etherpad was released in 2008 and was bought by Google in 2009 and then released to the public as open source code. Etherpad has been implemented in Java, JavaScript and Scala. Etherpad Lite has been implemented entirely in JavaScript. The constant update logic is implmenet via Ajax [ethb].

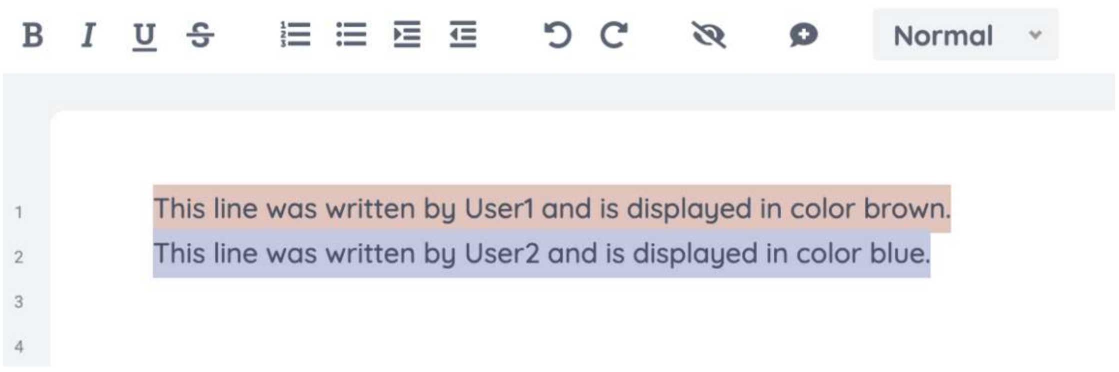
[1] https://chrome.google.com/webstore/detail/docs/aohghmighlieiainnegkcijnfilokake?hl=de

Features

Etherpad can be downloaded as open source code and installed on an own server. Thus, all data remains stored on the own server and is not necessarily sent through the Internet. However, there are public servers on which Etherpad is installed and which can also be used freely by all people. However, the development still lies with the Etherpad foundation. As a slimmer alternative to Etherpad, Etherpad Lite was developed, which is faster and more modern. In addition to Etherpad, which is purely a word processor, there is EtherCalc, which is a collaborative spreadsheet program.

To create a new document, a URL is generated which is unique and which the creator can share with other users. A document is named as "pad". It is also possible that the creator assigns a password for the document. Also here the document gets autosaved and simultaneous changes by multiple users are merged by means of operational transform.

Just like Google Docs, Etherpad assigns different colors to different users. The assigned colors make it possible to display written text sections in the colors of the user who wrote this section. In the example at Figure 3.3, User1 wrote the first line and User2 wrote the second line. The first line is highlighted in the color of User1, namely brown, and the second line in the color of User2, namely blue.



This line was written by User1 and is displayed in color brown.
This line was written by User2 and is displayed in color blue.

Figure 3.3: Example on how Etherpad displays two lines written by two different participants.

All changes are saved and can be reviewed in a history view by all participants. For this purpose, Etherpad provides a timeline that can be used by any participant. It is possible to track the complete change history of the document. It is also possible to see which authors have contributed to the document. The user can either manually operate the timeline and jump forward or backward or press the autoplay button to automatically display all changes from the start to the end. Figure 3.4 demonstrates how Etherpad displays the timeline functionality to the user. Two authors, namely Alex and Markus, have worked on this document, and Version 82 from the document is currently shown.

It is also possible to write own plugins [2] for EtherpadLite and install them on an

[2] https://github.com/ether/etherpad-lite/wiki/Available-Plugins

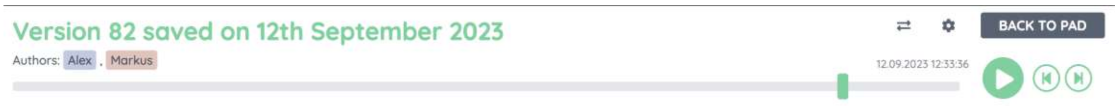


Figure 3.4: Example on how Etherpad displays timeline functionality to go back and forward in time through all changes.

own server. Etherpad also provides its own registry with public plugins, which can be downloaded and used on a server [3].

Pricing & Source Code availability

Because Etherpad is open source, it is possible to host an Etherpad server on an own server. However, there are also free online servers which provide Etherpad services such as *yopad.eu* [4].

3.1.3 Jetbrains Code With Me

Visual Studio Live Share has already been introduced in more detail in Chapter 2, since the prototype will base on Visual Studio Live Share. In this section another collaboration IDE will be introduced. It will focus on Jetbrains' Code With Me [cod] feature, which is built into most Jetbrain IDEs by default. So Code With Me is already built in by default for *IntelliJ IDEA, PyCharm, WebStorm, PhpStorm, CLion, GoLand, RubyMine*. It is available for Ultimate and also Community Edition. For *Android Studio* there is a plugin in the JetBrains Marketplace [5] which can be downloaded and installed. For *Rider* it will be added in the near future.

Features

The main idea of Code With Me is that co-workers in a team can work together on a project in real-time from a wide variety of locations around the world. Participants can examine and edit the code of a project completely independent of location. Developers can do this with their own screen and keyboard. With this option, developers can copy their session link and send it to their colleagues. They can then simply click on the link to join the session.

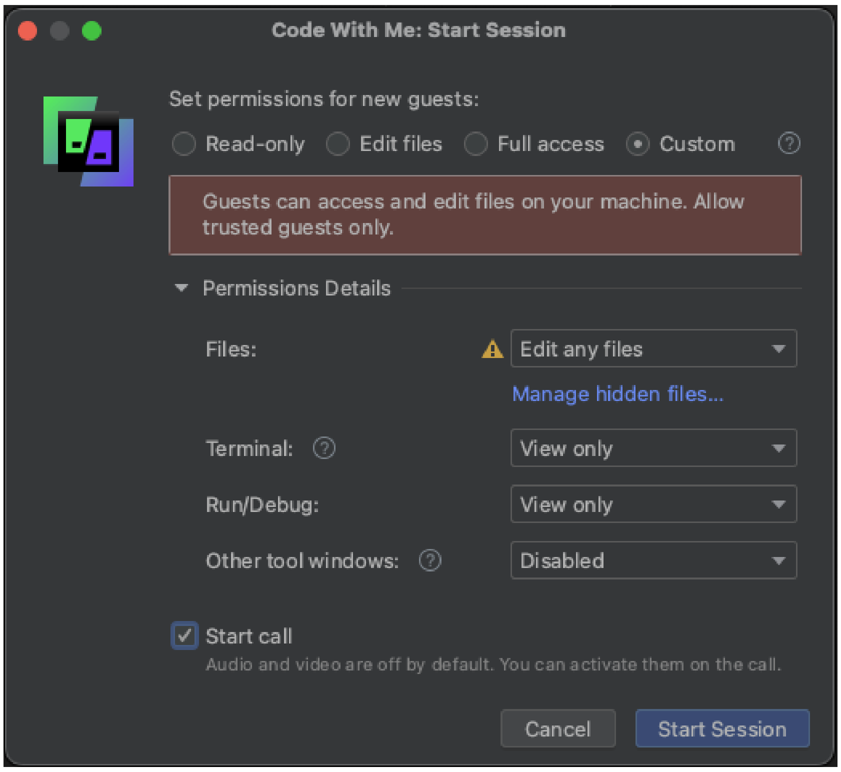
One of the main features of Code With Me is that users can work on and edit a file simultaneously. All changes are applied in real-time to all participants. Other users' cursors and selections are displayed in their own colors, as in other collaborative real-time editors. This means that changes are clearly traceable and multiple developers do not get in each other's way so quickly.

[3] `https://static.etherpad.org/index.html`
[4] `https://yopad.eu`
[5] `https://plugins.jetbrains.com/plugin/14896-code-with-me`

A big and important part of Code With Me is the following feature. With this feature a user can follow the cursor of another user or force all participants to follow the own cursor. With Run and Debug it is possible for a single user to start the application to be developed locally and for several developers to access this application. The whole team can then also debug together and thus find existing bugs together.

Code With Me allows each participant to keep their own settings and interface. Also keymaps are kept and developers can use them as if they were not in a collaborative session. Since Code With Me is also asynchronous there is no delay time as is common from other remote desktop solutions. Features like code autocompletion, navigation, code insights are not lost and can be used natively.

Files are not stored on the hard disk of the guest machines. The host can set which files it wants to share with guests. The host can also set permissions for certain guests in a session and thus manage sharing differently depending on the guest. Using Code With Me Enterprise, it is possible for sessions to run on their own private network. Thus, no traffic is sent outside the private network.


Code With Me: Start Session

Set permissions for new guests:

Read-only Edit files Full access [✓ Custom]

Guests can access and edit files on your machine. Allow trusted guests only.

Permissions Details

Files: Edit any files

Manage hidden files...

Terminal: View only

Run/Debug: View only

Other tool windows: Disabled

[✓] Start call

Audio and video are off by default. You can activate them on the call.

Figure 3.5: Example on how Jetbrains' Code With Me allows Host to set permissions for guests when starting a session.

In Figure 3.5, a depiction illustrates how a host can start a collaboration session. The host can share the project on three different modes. These are Read-Only, Edit files and Full access. In Full access mode, guests have full access to the project and can, for example, create new files. In Custom mode, the host can decide for herself how she wants to choose the permissions for files, terminal, run/debug and other tool windows. It is also possible to open an audio and video call immediately at the start of the session.

Another special feature, which is also available in VS Code Live Share, is port-forwarding, which allows users to release their ports to other participants. This allows local applications and resources to be shared with all participants. Not only ports, also application windows can be released to other participants. Classic screensharing is known from many remote desktop applications and is also supported here. Audio, video calls and chats are possible within the IDE and do not have to be externalized to other applications.

Pricing & Source Code availability

Code With Me offers free versions for Community Editions of IntelliJ IDEA and PyCharm. In this free version, up to three guests can work together simultaneously for 30 minutes in a session. The community version of IntelliJ IDEA is also open source. To enjoy unlimited session length and support multiple guests per session, an upgrade to the Premium or Enterprise version [6] is needed.

3.1.4 Comparison

Finally, this subsection compares all three presented textual collaborative document editing tools in Table 3.1. In addition to the three editors from this chapter, the table will also compare Visual Studio Live Share for text editors, which was already introduced in the last chapter and is essential for this work.

For this purpose, various criteria are used for comparison: On which platforms does the client platform run, is this application still maintained, is the code open or closed source, does cut/copy/paste operation work, does undo/redo operation work, what functionality does the application offer to avoid conflicts already during the collaboration process, what architecture is behind the collaboration functionality, and is the application easy to use from a collaboration perspective.

[6] https://www.jetbrains.com/code-with-me/buy/

	Google Docs	**Etherpad**	**Jetbrains Code With Me**	**Visual Studio Live Share**
Client platforms	Available as a browser application, chrome extension, desktop application for Google's ChromeOS and mobile app for Android and iOS.	Browser application	IntelliJ IEDA, PyCharm, WebStorm, PhpStorm, CLion, GoLand, RubyMine, Android Studio	Visual Studio and Visual Studio Code
Maintained	Yes	Yes	Yes	Yes
Open / Closed source	Closed source	Open source	Community version open source, premium and enterprise version are closed source.	Server is closed source, client is open source.
Cut / Copy / Paste	Yes	Yes	Yes	Yes
Undo / Redo	Yes	Yes	Yes	Yes
Conflict reduction during collaboration process	Every user gets an own color assigned. Color is used for selection and cursor position of other participants. There is a chat and comment functionality.	Every user gets an own color assigned. Etherpad shows written words highlighted in their author's color. Timeline functionality can display every change of every user.	Every users gets an own color assigned. Color is used for cursor position and selection highlighting. Chat functionality. Extra: Video and Audio calls possible.	Every user gets an own color assigned. Color is used for cursor position and selection highlighting. Chat functionality.

Collaboration architecture	Google's collaboration server used.	Own Etherpad server needed.	Jetbrains provides collaboration server.	VS Live Share provides collaboration server.
Easy to use from collaboration perspective	Yes, since every user is aware of where other users are currently working. Chat and comment functionality is also good for collaborative working. Version history also very helpful.	Good open-source alternative for Google Docs. Author's word highlighting and timeline very useful for collaborative working.	Very powerful collaboration developing tool with basic functionality (cursor position, selection highlighting) and extra stuff. Port forwarding and collaborative debugging also possible. Audio and video calls good for collaboration sessions.	Easy to use. Cursor position and selection highlighting basic functionality for textual collaborative editors. Port forwarding and collaborative debugging also possible.

Table 3.1: Comparison of all three textual collaborative document editing tools and Visual Studio Live Share.

3.2 Graphical collaborative document editing tools

This section will describe graphical collaborative document editing tools. In contrast to textual editors, graphical editing tools usually work with graphical files consisting of diagrams. The final application will also work with diagram files. For this, tools like MetaEdit+, Graphity, Google Drawings and VS Code TURN are compared.

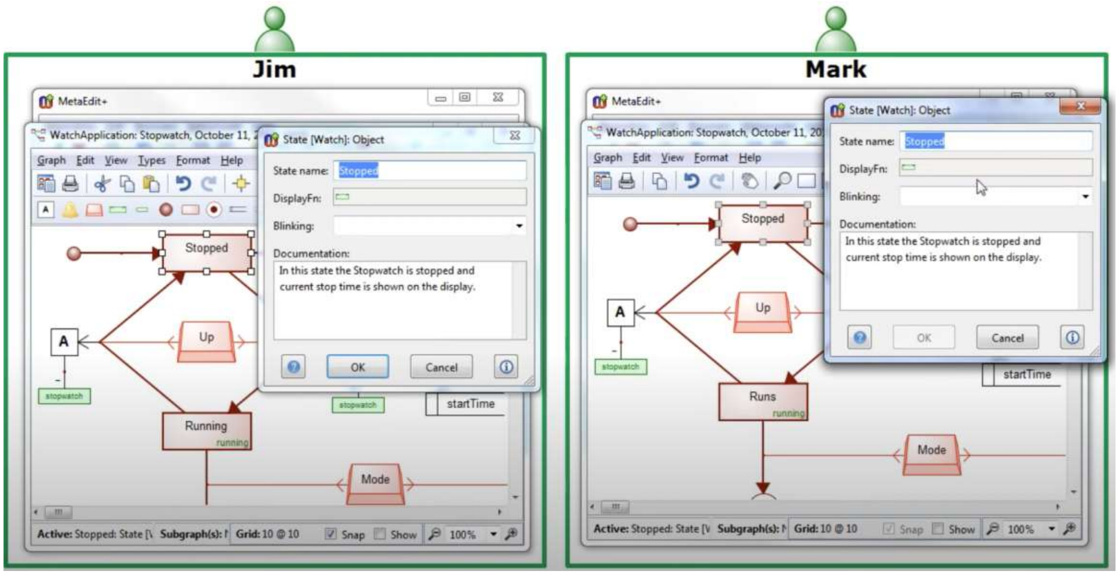
3.2.1 MetaEdit+

MetaEdit+ [met] is a collaborative modeling tool that is successfully used in the industry. MetaEdit+ is built on a multi-user client-server architecture where user, clients, server and repository can be located in geographically different places. Participants have the ability to access data via local applications, a remote desktop application or via the browser. MetaEdit+ supports a variety of modeling languages and different ways of presenting the model such as metrics, tables, text and of course diagrams. MetaEdit+ relies on automatic fine-granularity locking for high concurrency to avoid conflicts [KLR96] and works with a version control integration [Kel17].

Features

MetaEdit+ also has a collaborative functionality [KT21] that allows users to work on a model simultaneously. This also works via the graphical diagram modeling interface. MetaEdit+ works with transactions, which means that any change to the model must first be committed before it is written to the model. Only when the user has edited and committed the focused element is it saved in the model and thus transferred to the server. Other users get the change by also committing their transaction and thus fetching the latest status on the server. This means that a user is not interrupted in her work during a transaction and latest updates are displayed only when a new transaction starts and this happens after a commit.

To avoid conflicts, MetaEdit+ has a real-time locking system. If a user edits an element, it is locked for all other users in real-time. This also happens during a transaction, i.e. MetaEdit+ tries to keep a collaboration unit as small and granular as possible. In this way, the server tries to lock as few elements as possible for other participants. Furthermore, locking works automatically, which means that the user does not have to explicitly lock and unlock an element, but MetaEdit+ takes over these activities when an element is edited or the edited element is committed. This leads to the fact that no conflicts can occur in the diagram and thus also no conflict resolution or merging must be operated.



State name: Stopped
DisplayFn:
Blinking:
Documentation:
In this state the Stopwatch is stopped and current stop time is shown on the display.

Figure 3.6: Example on how MetaEdit+ locks Elements when two Users are trying to work on the same Element.
Source: `https://www.youtube.com/watch?v=JQzt4cd8ppc&t=145s&ab_channel=MetaEdit` (Accessed: 2023-09-11)

In Figure 3.6, a diagram is shown that is supposed to represent the running of a clock. In this example, Jim and Mark are trying to edit the *Stopped* element at the same time.

Jim has opened the element first, Mark second. In Jim's dialog box the OK button is active, whereas in Mark's dialog box the OK button is disabled. The element is visible to Mark, but not editable. Once Jim edits the element and commits the transaction, Mark can edit the element again.

Pricing & Source Code availability

MetaEdit+ as a whole with server, repository and clients has a 31-day trial version, but is not free to use beyond this time. The application is therefore also closed source. However, there are plugins, for example for Eclipse [7] and Visual Studio [8], which are open source and free and thus allow integration into IDEs.

3.2.2 Graphity

Graphity [grab] is a collaborative diagram editor designed for Atlassian Confluence [9] and developed by yWorks. Unlike MetaEdit+, Graphity offers real-time collaborative editing. Which means that changes are immediately presented to all other participants.

Features

Graphity [graa] is an extension for Atlassian's Confluence, which allows to embed diagrams directly into Confluence pages. Graphity supports multiple diagram types and modeling languages out of the box, such as Flowacharts or BPMNs. Graphity also allows storing diagrams not only on an own server but also in the cloud. To use Graphity on produciton, a collaboration server must be hosted on its own infrastructure. Graphity itself offers a test server, but that should only be used for testing.

As already mentioned, Graphity offers real-time collaboration. This means that all changes are displayed to all other users immediately after execution. Graphity also does not perform conflict resolution or merging as well. This means that changes are always made to the current model, even if this does not correspond to the local model. However, Graphity offers a locking system, which locks currently used elements. The locking is only active if an element for example is moved, transformed or maybe also if the title of the element is changed. Not only the element, but also all directly dependent nodes and labels are also locked. After editing, the element is immediately unlocked for all other participants. With this algorithm the graph model is always consistent. In the example at Figure 3.7, User 2 on the right side of the image performs an operation on Element 1. This scales the element and makes it bigger. At the same time, User 1 locks this element, namely Element 1, and cannot select or edit it.

Undo/redo operations are possible with Graphity. However, there is only one global command stack for the whole diagram. This means that when an undo operation is

[7] https://github.com/MetaCase/metaedit-plugin-for-eclipse
[8] https://github.com/MetaCase/metaedit-extension-for-visual-studio
[9] https://www.atlassian.com/software/confluence

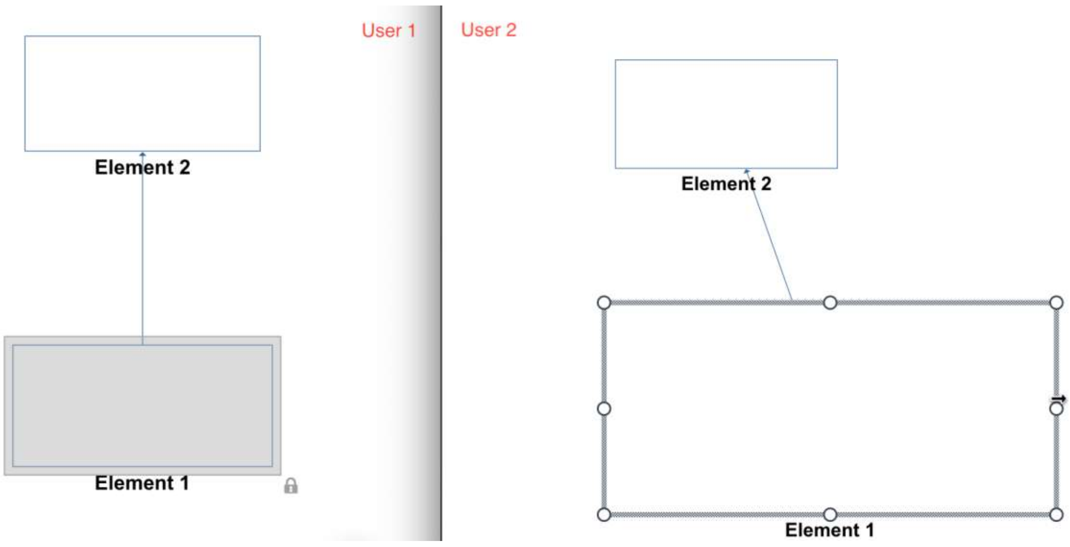
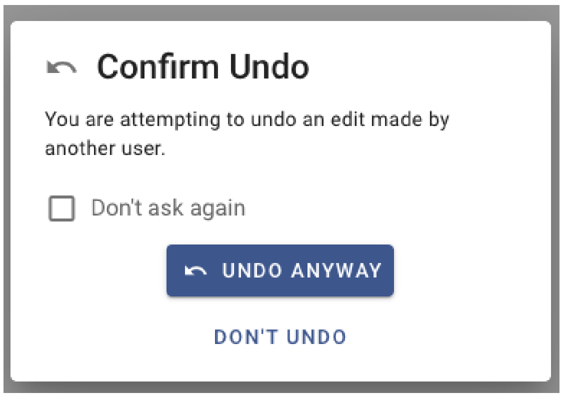


Figure 3.7: Example on how Graphity locks Elements when a user performs an operation. **Source:** `https://www.graphity.com/collaborative-editing` (Accessed: 2023-09-13)

performed, the operation is globally undone regardless of the user who performed the operation. However, if a non-own operation is to be undone, a confirmation dialog is displayed to make sure that this operation really should be undone. In Figure 3.8 Graphity asks the user if she still wants to do the undo operation even this operation was initially performed by another user.


Confirm Undo

You are attempting to undo an edit made by another user.

Don't ask again

Figure 3.8: Example on how Graphity shows a confirmation dialog if a user wants do undo another user's operation.

Another special characteristic is that the number of users and documents is not limited. However, it depends on the performance of the server how many users can work on a document at the same time. The current position of all users as cursors are displayed on the screens at all other users. In Figure 3.9, the positions of Test User 1, Test User 2 and Test User 3 are displayed instantaneously.

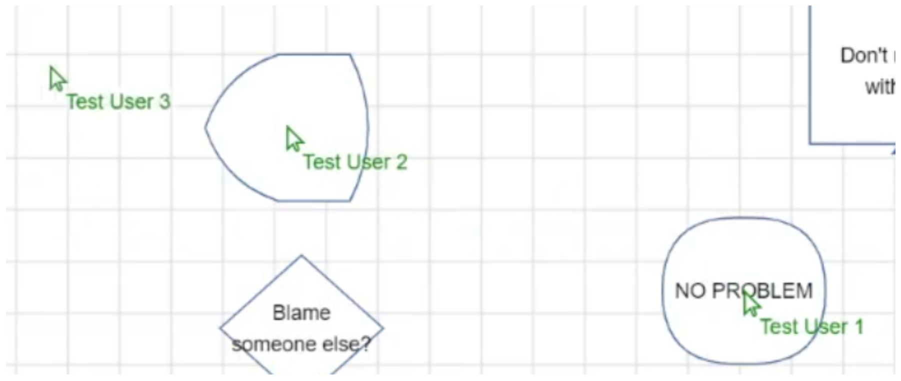


Figure 3.9: Example on how Graphity shows current mouse pointers position of all other users.

Pricing & Source Code availability

Graphity is closed source and can be purchased in the Atlassian Marketplace and is available for a fee. However, there is also a trial version, as well as academic and community versions for selected organizations. The collaboration server code can be downloaded and hosted on a local instance.

3.2.3 Google Drawings

Google Drawings [dra] is an application of Google Docs editors and was first launched in 2010. Google Drawings was developed in JavaScript and can be used via the browser or as a desktop application for Google's ChromeOS. Google Drawings, like all other Google Docs editors, has a collaboration functionality.

Features

With Google Drawings users can create diagrams of all kinds. Thus, flowcharts, org charts, mind maps and many other types of diagrams can be created and edited. Using a share link, guests can open and edit the same document. Not only shapes, arrows and text but also images can be imported and edited.

Other users can enter via the share link and see the document on their screen. As with other Google Docs editors, there is the option of a chat function and comments can also be created by users. Unlike many collaborative diagramming tools, Google Drawings does not lock elements, even if they are being edited, moved or scaled. Different from Graphity, Google Drawings has its own operations stack per user. This means that undo/redo operations can be performed per user.

Google Drawings also uses colors and names to make the location of other users visible. So Google Drawings shows selections of elements in the color of the respective user. Each

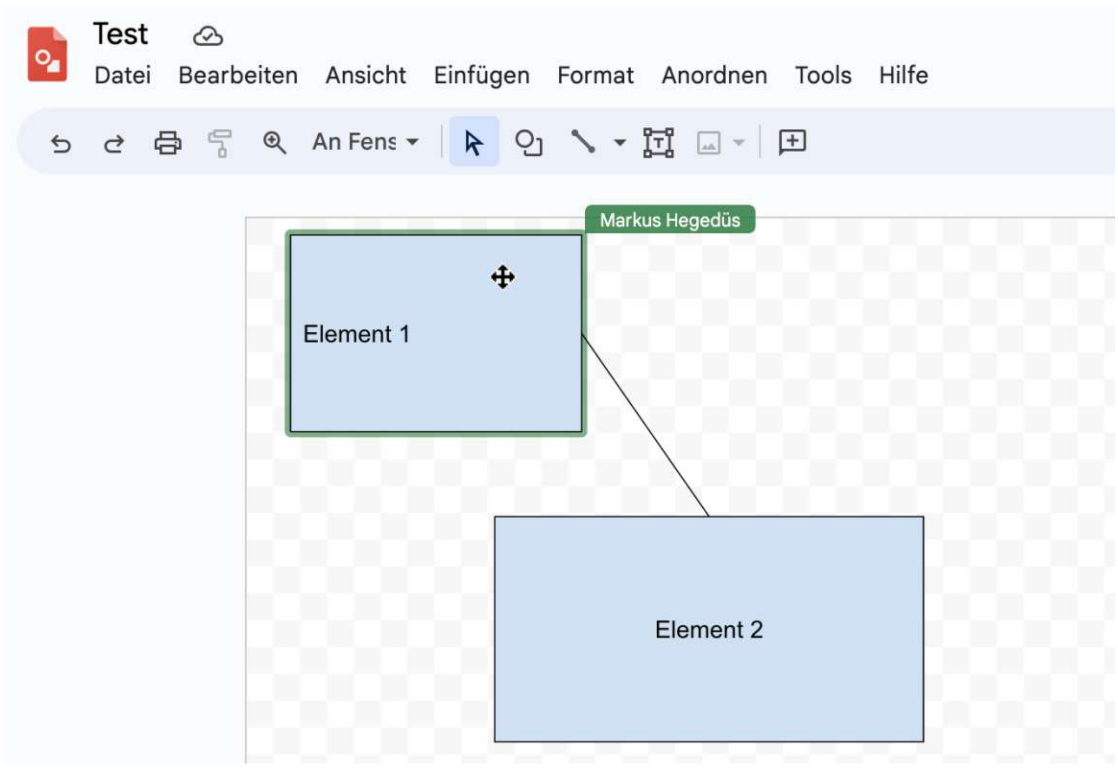


Figure 3.10: Example on how Google Drawings shows selections of other users with a specific color an label.

user is assigned a unique color. In addition, a label with the name of the user in the same color is displayed next to it. In Figure 3.10 another user, which is marked with the color green, is currently selecting Element 1.

Pricing & Source Code availability

The same principle applies here as for Google Docs. Private use is free, only for commercial use a Google Workspace plan has to be purchased. Google Drawings is also a closed source code.

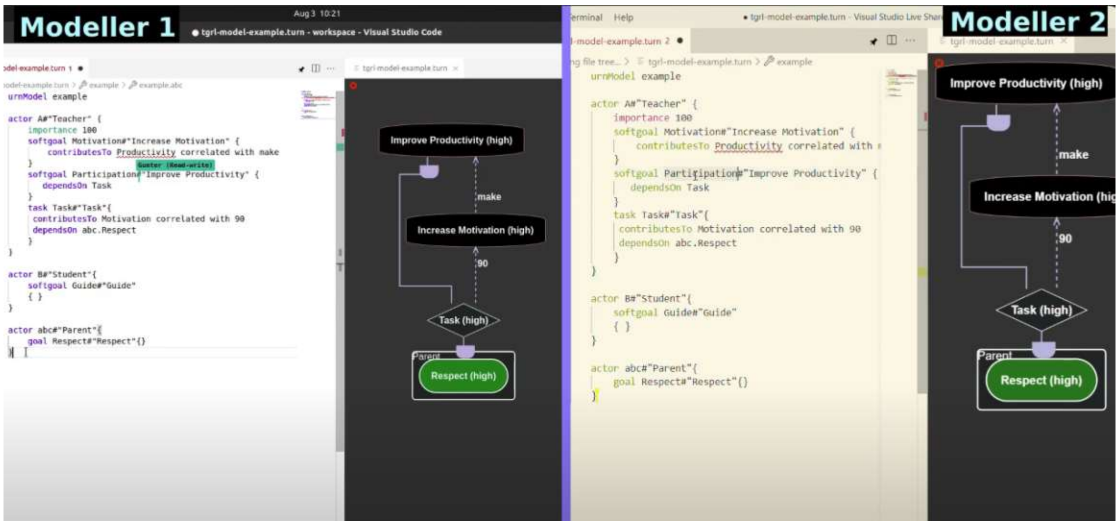
3.2.4 VS Code TURN

VS Code TURN is a Visual Studio Code plugin that uses a language server in the background. This plugin works with the TGRL language server and provides full support for their language (Textual Goal-oriented Requirement Language). Unlike GLSP, the model is not created graphically, but by code, and then later generated and displayed visually. VSCode TURN was built as a collaborative editor. Also VSCode TURN uses Visual Studio Live Share to synchronize their actions via their Visual Studio Live Share server. The biggest problem is resolving the conflicts created by undo operations and the synchronization in a multi-tenancy view [SM21].

Processing via VS Code Teletype CRDT was considered in a theoretical concept. VS Code Teletype CRDT also supports other IDEs like Atom [10] and Eclipse Theia [11]. However, this is not yet as mature as Visual Studio Live Share [vsce].

Features

VS Code TURN extension uses the functionality of Visual Studio Live Share to enable collaborative work. Since VS Code TURN uses only a textual language to model a diagram, it is easy to use Live Share functionality to enable collaborative modeling. To do this, Live Share must be enabled during modeling and each user must control their own language server. Since modeling is purely textual, Visual Studio Live Share can inherently push all changes to all other users. The user then accesses their own language server with the updated model and creates a new graphical model.



Modeller 1
Modeller 2
tgrl-model-example.turn - workspace - Visual Studio Code
tgrl-model-example.turn - Visual Studio Live Share
Gunter (Read-write)
Improve Productivity (high)
make
Increase Motivation (high)
90
Task (high)
Parent
Respect (high)

```
urnModel example

actor A#"Teacher" {
    importance 100
    softgoal Motivation#"Increase Motivation" {
        contributesTo Productivity correlated with make
    }
    softgoal Participation#"Improve Productivity" {
        dependsOn Task
    }
    task Task#"Task"{
     contributesTo Motivation correlated with 90
     dependsOn abc.Respect
    }
}

actor B#"Student"{
    softgoal Guide#"Guide"
    { }
}

actor abc#"Parent"{
    goal Respect#"Respect"{}
}
```

Figure 3.11: Example on how VS Code TURN uses Visual Studio Live Share to enable collaborative modeling.
Source: `https://www.youtube.com/watch?v=3fqXI3tiQjw&ab_channel=RijulSaini` (Accessed: 2023-09-13)

Visual Studio Live Share ensures that a conflict-free execution is achieved. Undo/redo operations are performed at user level. By the implementation of Visual Studio Live Share the undo/redo operations also work conflict free. The participants' cursors are also displayed in their own color, as is usual with Live Share. This also reduces the potential for conflict, since each user knows where other users are currently working. In

[10] `https://github.com/atom/atom`
[11] `https://theia-ide.org/`

the example in Figure 3.11, two modelers work on one document. Each modeler has the textual editor on the left and the visualized diagram on the right. Visual Studio Live Share is showing the cursor of Modeler 2 on the left side of Modeler 1's text editor in green and with a label.

Pricing & Source Code availability

VS Code TURN is open source and available on Github [12]. It is also free to install and use.

3.2.5 Comparison

Here all four presented graphical collaborative document editing tools are compared in Table 3.2. In addition to the criteria already used in the last chapter, two more criteria are introduced: which modeling languages or diagram types are supported and which methods are used to resolve modeling conflicts.

	MetaEdit+	**Graphity**	**Google Drawings**	**VS Code TURN**
Modeling languages / Diagram types	Uses own modeling languages, which can be designed with *MetaEdit+ Workbench*.	BPMN, Flowchart, Network diagrams, Organization charts, etc.	No real modeling languages supported. User can work with generic elements like forms and arrows. There are templates for specific diagram types.	TGRL (Textual Goal-oriented Requirement Language)
Client platforms	Desktop application for Windows, Linux and Mac OS X. Integration for Eclipse and Visual Studio.	Available as plugin for Atlassian Confluence and thus via the browser.	Available as a browser application and desktop application for Google's ChromeOS.	Plugin for Visual Studio Code
Maintained	Yes	Yes	Yes	Last change 2021.

[12] https://github.com/sainirijul/vscode-turn

Open / Closed source	Closed source, only plugins for IDEs are open source.	Closed source	Closed source	Open source
Cut / Copy / Paste	Yes	Yes	Yes	Yes
Undo / Redo	Yes	Yes, but only on global level.	Yes	Yes
Conflict reduction during collaborating process	No visible widgets. User can see if element is actually locked when editing this element.	User can see if other users are working on elements since they get locked then. And user can see other users' mouse pointers.	User can see selections of other users in their assigned colors with labels. Users have the possibility to chat and comment stuff.	Uses VS Live Share collaboration functionality to e.g. show current cursor in other user's color. Chat available.
Resolving modeling conflicts upon operations	Locking elements and unlocking them when committing transaction. No conflict resolution.	Elements and dependencies get locked while editing. No conflict resolution.	No locking and no conflict resolution. Latest operation wins.	Done by VS Live Share collaboration functionality for text editors.
Collaboration architecture	Clients are working with one server, which is responsible for handling transactions and collaboration.	Own collaboration server needed.	Google's collaboration server used.	Uses VS Live Share server.

Easy to use from collaboration perspective	Not easy to use, since it's not a real-time collaboration functionality. Participants don't event know what actually is edited by other users. Elements get locked until whole transaction is finished.	Yes, since it can lock elements in real-time without clicking on them. Undo / redo operations on global level are not perfect. Mouse pointers for participant visibility is good.	Yes, since every user can see other user's selections and there is a possibility to chat with other participants and create comments to different elements.	In the end only a textual modeling language which shows diagram in a read-only window. Working with elements on graphical level is not possible.

Table 3.2: Comparison of all graphical collaborative document editing tools.

CHAPTER 4

Concept

With this implementation a prototype to adapt the GLSP protocol to a multi-user architecture is created. It is clear that the prototype will use Visual Studio Live Share and its collaborative server to share data between multiple users. Also clear is that the solution should only extend the GLSP protocol and not completely rebuild it. Eclipse' graphical language server protocol is constantly maintained and state of the art. This modern technology stack provides everything which is needed to implement the prototype. The definition of the rest of the architecture is elaborated in this chapter.

In order to meet the requirements, these must be worked out and defined. Another question to be answered is whether a single or multi language server architecture is better suited to the requirements. The first and most important step of the prototype is to modify the GLSP protocol so that it is suitable for collaborative purposes. Furthermore it has to be clarified how to create a conflict-free handling of all operations. The Visual Studio Live Share API is used to exchange data. Here it is possible to share data synchronously or asynchronously. Since GLSP is implemented not only on the client, but also on the language server, changes will also have to be made to the server. As soon as the GLSP client and GLSP server have been implemented collaboratively, the concept will address the second important issue of the prototype. It must be ensured that conflict potential is kept as low as possible already during collaboration. Users need to know where in the diagram and on which element other users are working currently.

4.1 Requirements

This section discusses how to build requirements from the analyzing work about tools, introduced at Chapter 3. The prototype is given as a precondition that it must use the GLSP protocol and that it must run over Visual Studio Live Share. Although, the application should be extended to allow easy migration to a different collaborative server and architecture (instead of Visual Studio Live Share).

Following requirements will be covered within the prototype:

- **Modeling languages / Diagram types:** It should be possible to handle all modeling languages and diagram types supported by GLSP collaboratively. This prototype will focus on the Workflow sample and implement it. In the further consequence also the BIGUML [1] extension should be realised.
- **Client platforms:** For the first prototype, this should be executable at client level only as a plugin for Visual Studio Code.
- **Maintained:** GLSP is maintained, so this is not a real criteria for the prototype.
- **Open / Closed source:** Since GLSP is open source, the prototype will also be open source.
- **Cut / Copy / Paste:** Since GLSP supports cut, copy and paste operations, the collaborative version of GLSP should continue to support these operations.
- **Undo / Redo:** GLSP also supports undo and redo operations. The collaborative version should continue to support these two operations. In contrast to Graphity these are to function however on user level. This means that each user must have its own command stack.
- **Conflict reduction during collaborating process:** Since the prototype uses the VS Live Share extension, it can use collaborative functionalities like chat out-of-the-box. Also VS Live Share assigns each user an own color, as it is common from other collaborative editors. Graphity uses mouse pointers of other users and displays them live at their current position. The prototype should also have this functionality. Like Graphity, the name of the user should be displayed in a label next to the mouse pointer. Both the mouse pointer and the label should be displayed in the assigned color of VS Live Share. Another functionality is that sections of other users should be highlighted. Google Drawings will outline the selected element in the assigned color and also add a label with the name of the user in this color. However, since this is not optimal for multiple selections from the same element, the prototype will display icons in a special form next to the element. This icon should also have the assigned color of the user. When multiple elements are selected, icons will be positioned at multiple elements. This also makes it easy for multiple users to select the same element and still have it displayed clearly. There was no displaying of the viewport of other users in any of the presented tools, which should be implemented in the prototype. For this purpose, a rectangle is to be displayed in the color of the user. This rectangle should show the exact viewport in real-time of other users. However, users do not always want to see all other activities such as the current mouse pointer, selections and viewport. Therefore it should be possible that a user can hide and show these features.

[1] `https://marketplace.visualstudio.com/items?itemName=BIGModelingTools.umldiagram`

- **Resolving modeling conflicts upon operations:** The prototype should not have any locking. As already seen in Google Drawings, the most current operation on an element should win. The prototype should take much value on visual representation of activities of other users and therefore a locking system should not be needed. And thus no conflict resolution is necessary as well.
- **Collaboration architecture:** Since the extension is based on the VS Live Share extension, the VS Live Share Server is also used as collaboration server.
- **Easy to use from collaboration perspective:** As already mentioned, much emphasis should be placed on this feature. Mouse pointers, selections and viewports of other users should help. Since also no elements are locked, a simple use of the tool should be possible. The chat function also serves for better collaborative work.

4.2 Single vs. multi GLSP language server architecture

The first step is to answer a fundamental question. Besides the collaboration server of Live Share, the GLSP language server is used to make modeling possible at all. In a non-collaborative architecture, a GLSP client (e.g. Visual Studio Code extension) has an associated GLSP server which usually runs locally on its own computer. The Workflow extension starts e.g. the GLSP server when loading the extension. Just like the `BIGUML` extension, but this one also starts a `ModelServer`. However, it is not in the scope of this thesis to implement this `ModelServer` collaboratively.

4.2.1 Single GLSP language server architecture

In the single GLSP language server architecture, there is one GLSP server per collaboration session. This GLSP server is placed on the local instance of the host. This server takes care of all the important things that are language specific, like code completion, syntax highlighting, code navigation, etc. This GLSP server also holds the model state, which contains all information about the diagram. This model state is the only one during the whole collaboration session. All guests work with this one GLSP server and this one model state. Because there is only one model state for all users, no merge algorithm is necessary.

The biggest challenge of a single server architecture is that guest actions and operations are not executed on their own server, but are first sent from their client over VS Live Share to the host client, which forwards them to the host server. The subsequent response from the host server must then be distributed to the correct guest on the host client again over VS Live Share.

Another problem of a single server architecture is that in the current GLSP server implementation there is only one command stack. But because the application shall support undo/redo commands on user level, i.e. per user, this GLSP server has to be modified in a way that each participant of a collaboration session on the host server has an own command stack.

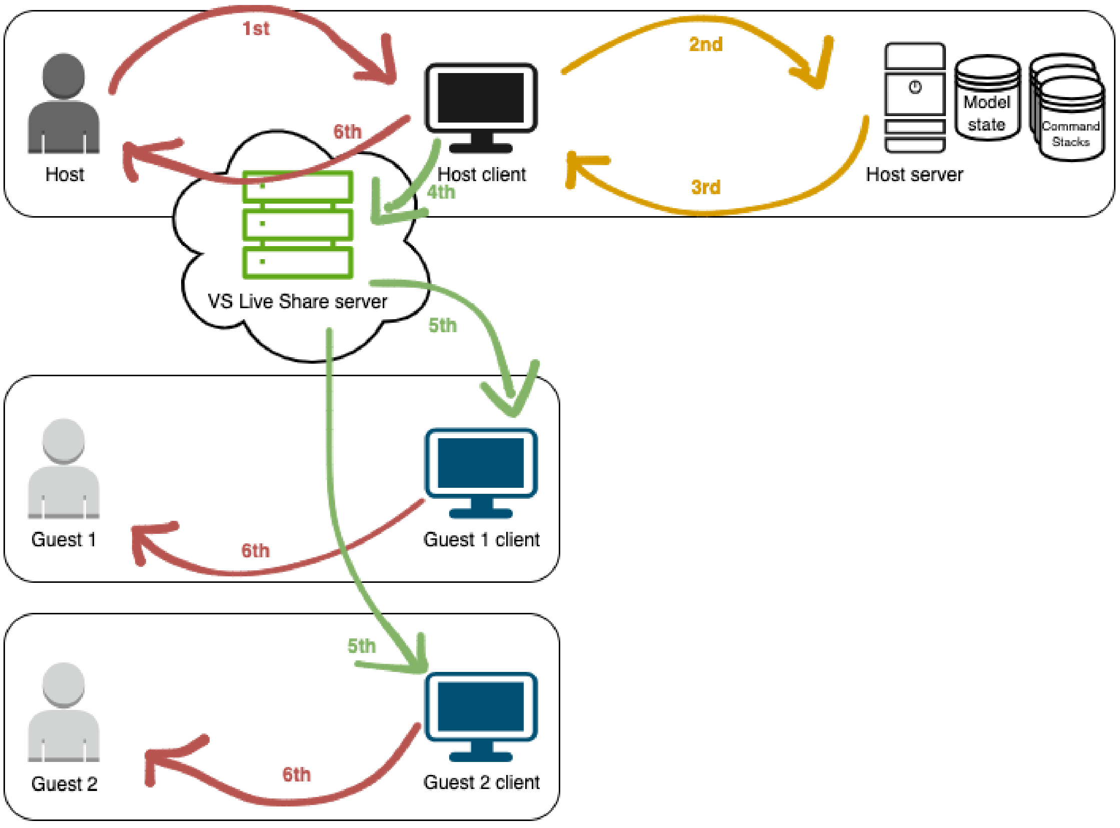


Figure 4.1: Data flow for a model operation within a collaboration session in a single server architecture.

The Figure 4.1 shows the data flow of a model operation within a collaboration session in a single server architecture. This example illustrates a host and two guests, where each user has a GLSP client, but only the host has its own GLSP server. The host owns the only model state of the collaboration session, and as explained before, one command stack per collaboration user. Additionally there is the VS Live Share which is responsible for the communication between the users. In the first step **(1st)**, the user makes a change to the diagram, such as creating a new node. The host client creates a `CreateNodeOperation` from this action and sends it to the host server **(2nd)**. The host server executes this operation on the existing model state. The new model state is stored on the server and the operation is stored on the command stack of the executing user. The host server sends the corresponding response as an `UpdateModelAction` back to the host client **(3rd)**. The host client recognizes that it is an `UpdateModelAction` and broadcasts **(4th)** it to the VS Live Share server. This server distributes this message to all other clients **(5th)**. In the end, each client transfers the new model into a graphical representation of the diagram and displays it to all users on the screen **(6th)**.

4.2.2 Multi GLSP language server architecture

In the multi GLSP language server architecture, each participant in a collaborative session owns its own GLSP server. This GLSP server holds the model state, which contains all information about the diagram. In a multi server variant, this would mean that each participant in the collaboration session has its own model state and own command stack. The main difficulty would be to maintain this model state so that there would be no deviations among all these model states. This would require a merge algorithm such as operational transformation to keep all model states at the same value.

Another challenge of a multi server architecture is to load the model state into all the guest servers. Since the diagram files are usually stored locally on the host, these files would first have to be sent via VS Live Share to all guest clients and from there to all guest servers. With this file the guest server can initiate its own model state. However, this only works if all information is contained in the diagram file. In some modeling languages it is common that the GLSP server fetches further information about the diagram from a `ModelServer`. This `ModelServer` may be installed locally on the host and therefore it is difficult for the guest to reach this server. But not only loading the model state, but also writing it back to the diagram file or to the `ModelServer` is a challenge. Another possibility would be that not the diagram file, but the model state already initiated on the host server is sent to all guests via VS Live Share. This would have the advantage that the guest server does not have to access the diagram file or the `ModelServer` directly. Also each GLSP server has its own command stack. The command stack per user is necessary for example to execute undo/redo operations on user level.

The Figure 4.2 shows how a data flow would look like for a model operation within a collaboration session in a multi server architecture. This example shows also one host and two guests. Each user has a GLSP client and its own GLSP server. So each GLSP server also has its own model state and command stack. There is also the VS Live Share for communicating between users. The same example like for the single server architecture is used here. In the first step **(1st)**, the user creates a new node. The host client converts this action into a `CreateNodeOperation` and sends a message to the VS Live Share server **(2nd)** that an operation has been performed. The VS Live Share server broadcasts **(3rd)** this message to all guest clients. At the same time, but independently, all clients send a message to the server **(4th)** that a `CreateNodeOperation` has taken place on the model. Each GLSP server now performs this operation on the model and transforms the model. The new model gets stored into every model state. Besides that this command gets placed on every command stack. After the transformation, the host **(5th)** sends an `UpdateModelAction` with the updated model back to the respective clients. Finally, each client converts the new updated model into a graphical representation of the diagram and displays it to all users **(6th)**.

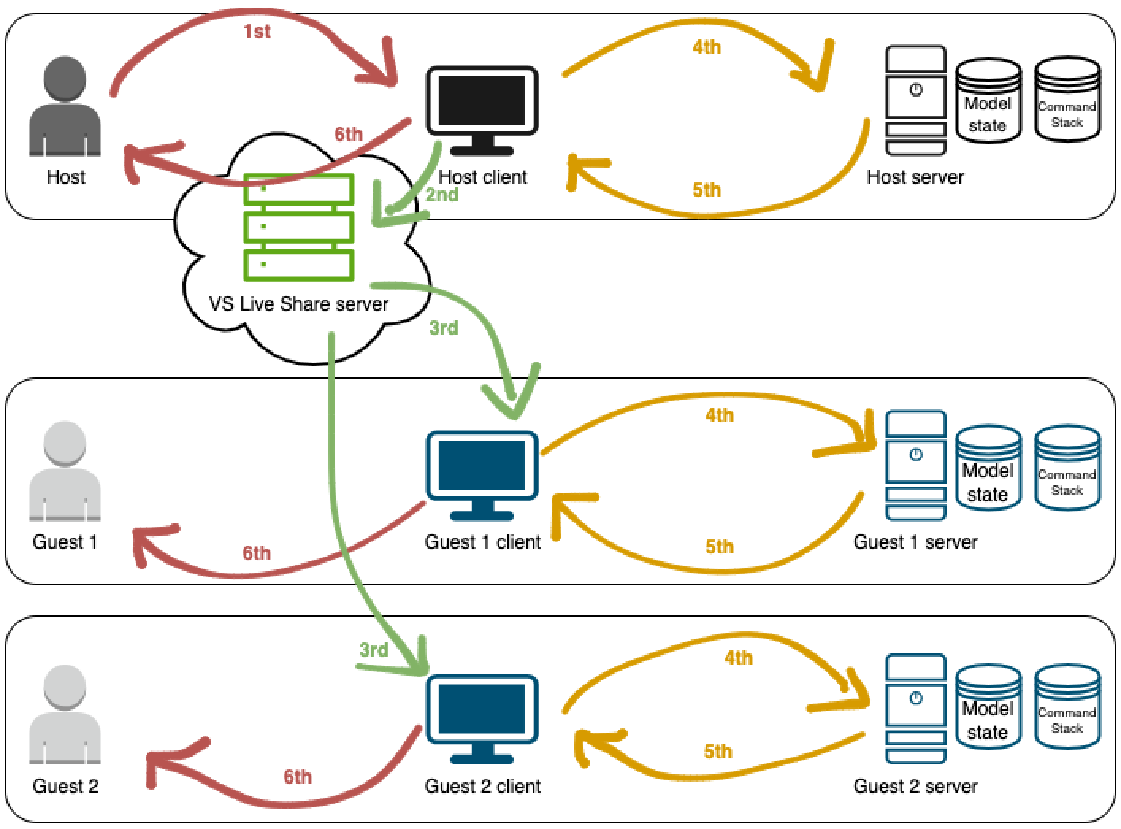


Figure 4.2: Data flow for a model operation within a collaboration session in a multi server architecture.

4.2.3 Decision

For the prototype, the variant with the **single server architecture** was chosen. This has the reason that:

- no merge algorithm is necessary to keep the model state consistent,
- and no logic is needed to load the model state at guest servers and as well no logic is needed to write the state back to the source.

However, in this solution it will be necessary that:

- the host client will take care of forwarding user-addressed responses from the server back to the initiator client,
- and that the host server stores command stacks per user to allow undo/redo operations at user level.

4.3 Challenges

This section discusses all possible challenges and problems to be solved in the course of the prototype implementation. This includes challenges that arise from working with GLSP messages over a GLSP server in a collaborative session. How to enable command stacks to be stored at user level, will be defined next and finally it has to be defined how the prototype can achieve conflict reduction already during the collaborating process.

4.3.1 Working with GLSP messages through one single GLSP server

This subsection deals with the challenges and problems of sending messages through a single GLSP server. This discussion tries to find solutions how to solve these problems. This GLSP server is placed on the host instance and has to receive all messages, like initialization/dispose messages, actions and operations, of all users of the collaboration session, process them and send them back to the host client. All guest clients are connected to the host client via the VS Live Share server. The goal is to build the collaboration module in a way that it is independent from VS Live Share and works with other collaborative servers as well. As a prerequisite, the solution will have a collaboration server communicating with one host client and multiple guest clients, since VS Live Share is also built on this architecture. The host client has a special role in this architecture, as it takes care of accepting and forwarding messages to the guest clients through the collaboration server. In order to understand the problem it needs to take a closer look at the different messages between GLSP client and GLSP server.

InitializeServer & ShutdownServer

`InitializeServer` is the first message a GLSP client sends to a GLSP server in the lifecycle of an instance. This message tells the server that a new GLSP client is present and wants to communicate with the server. In the case of the VS Code extension, this message is sent immediately after the extension is initialized. In response to the `IntializeServer` request the GLSP server sends back an `IntializeResult` which tells the client which protocol version and which possible actions the server supports. In contrast, `ShutdownServer` is the very last message that the GLSP client sends to the GLSP server at the end of an instance's life cycle. With this message the GLSP client informs the GLSP server that the client will terminate its communication with the server. With the `ShutdownServer` message there is no response from the server.

Since the GLSP client sends the `InitializeServer` message when the extension is started, and before the collaboration session is started, this message type is not relevant for the collaboration implementation and therefore does not need to be modified. Likewise, the client sends the `ShutdownServer` message after the collaboration session has ended, and is therefore also not relevant and does not need to be modified as well.

InitializeClientSession & DisposeClientSession

The GLSP client sends an `InitializeClientSession` to the GLSP server the first time a diagram file is loaded. The GLSP server then creates a new session in the dependency injection container. This happens once per diagram file per GLSP client. In a collaborative session, this message should also be sent only once per file for the entire session. The server does not send a response for the `InitializeClientSession`, it just creates a new session. Since only the host client communicates with the server, and it may also be possible for a guest client to open this file first, this initialization can also be sent from the guest client to the server via the host client. The communication takes place via the VS Live Share. Thus, the host initializes this file on behalf of the guest. So that each user can be identified in the collaborative session, each user is assigned a `subclientId`. The host gets a special ID, namely an 'H'. All guests are assigned a sequential number as ID. This important `subclientId` is sent with every message.

Furthermore, each client assigns a unique ID for this session, which is used as mapping for further messages to this diagram. However, this `clientSessionId` is only unique per client. This means that different clients can be assigned different IDs for the same document. But in order to allow a unique mapping between message and file, each client must send the file name or the relative path to the collaborative workspace. But in order to enable a unique allocation between message and file, each client must send the file name (or the relative path) to the collaborative workspace within every message. A `relativeDocumentUri` is inserted for this purpose.

Whether the file (or the diagram) has already been initialized must be stored by the host client. The host client remembers which file has already been initialized by which client. If a file is initialized for the very first time, then a message is also sent to the server.

As already mentioned the server needs a `clientSessionId` to be able to allocate the diagram. Since this is only unique within a client and the server logic should not be changed that it uses `relativeDocumentUri` on the server (instead of `clientSes-sionId`), a temporary global `clientSessionId` is added at the host client, which should be used for any communication with the server for this diagram, no matter from which client. To be clear which temporary global `clientSessionId` to use when calling the server, the host client also stores which temporary global `clientSessionId` belongs to which `relativeDocumentUri`.

Figure 4.3 shows an example of what it looks like when Guest 1 sends an `Initial-izeClientSession`, and this diagram has not yet been initialized by the host or another guest. To do this, the user first opens a file **(1st)**. The guest client creates an `InitializeClientSession` request from it. This request contains the local `clientSessionId`, which is used locally on the client to display the diagram, the `relativeDocumentUri`, which should identify the file in the collaborative workspace and the `subclientId`, which should identify the client/user. The request is sent to the VS Live Share server instead to its own server **(2nd)**. This server forwards the request to the host client **(3rd)**. The host client now checks if this file has already been initialized

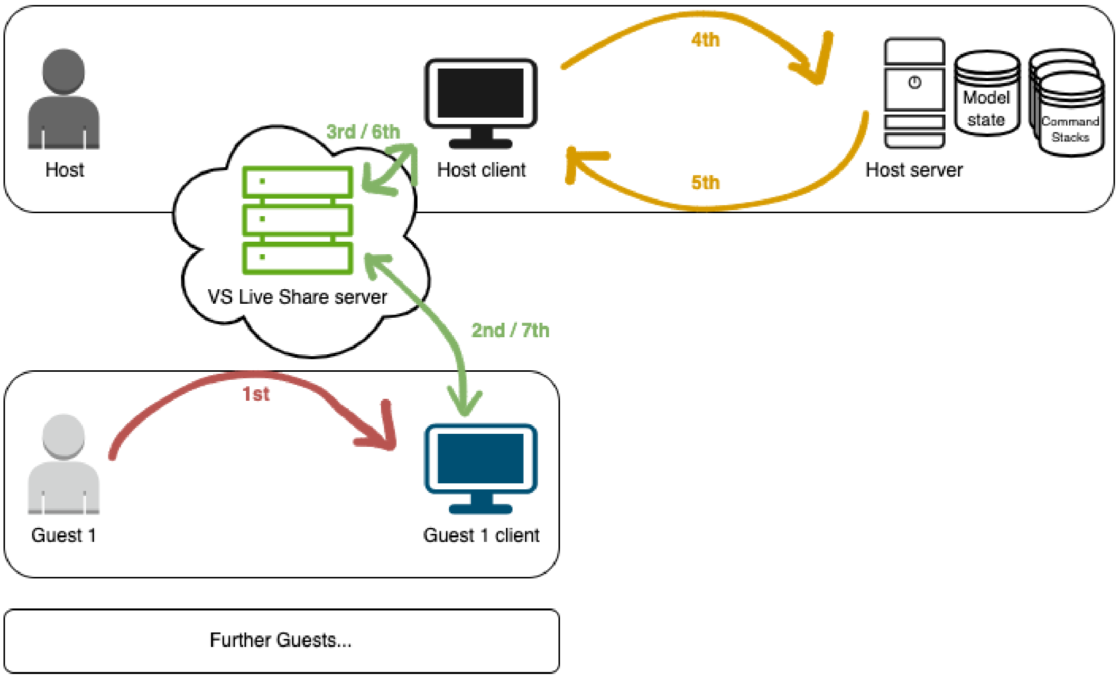


Figure 4.3: Data flow for an InitializeClientSession message initiated by Guest 1.

or not. If yes, then it stores only the `subclientId` to the `relativeDocumentUri`. If not, then the host client creates a new temporary global `clientSessionId` and stores this also to the `relativeDocumentUri`. Then the host client sends the request to the host server **(4th)**. The host server processes the request and sends an empty response back to the host client **(5th)**. Finally, the host client sends the empty response back to the guest client via VS Live Share **(6th/7th)**.

In contrast to the `InitializeClientSession`, the `DisposeClientSession` request is sent from the host client to the GLSP server when closing a diagram file. Here the GLSP server clears all data related to the client session. Also here the request contains the information like local `clientSessionId`, `relativeDocumentUri` and `subclientId`. At the host client it works exactly like the initialization, except that the request is only sent when each client closes this file. Thus, the request is sent when the last client closes the diagram. In addition, the host client removes the information that the sending client has opened the document. When all clients have closed the file, the host client also removes the temporary global `clientSessionId` because it is no longer needed. Finally, in case of a request from a guest, the empty response is sent to the guest client via VS Live Share.

ActionMessage sent from the client

As with all `ActionMessages` sent from the client, the host client replaces the local `clientSessionId` with the temporary global `clientSessionid`. This allows the server to assign the message to a session. The host client gets the temporary global `clientSessionId` from the `relativeDocumentUri` sent with the message.

For an `ActionMessage` which isn't a `RequestAction` no `requestId` is sent, since no response is expected from the client. For an `ActionMessage` with response the `requestId` is added at the client and is an indicator in the GLSP protocol that a response message is expected from the server. Here the GLSP server takes the `requestId` and sets it as `responseId` in the response message. So the client can assign the response to the request. The initiator's `subclientId` is added, so the server maps the possible response to the correct subclient. If this is a `RequestAction` again as for the `requestId` the `subclientId` is also mapped to the `ResponseAction` initiated later on from the server.

However, as for all `ActionMessages`, two cases must be considered: (a) the host client sends this message or (b) a guest client sends this message. If, as in (a), the host client sends the message to the server, it goes directly to the host server and the task is done. If as in case (b) the guest sends this message, then it is first sent to the host client via the VS Live Share server, which then sends it to the host server. Thus the task is also done.

There is a special case, namely at a `RequestModelAction` in the context of a collaborative session, a flag `disableReload` must be given at the host client, which tells the server that it should not reload this diagram if it has already been loaded. Namely this would then overwrite the model on the server with each new subclient. Additionally, in case a guest client sends this message, the `sourceUri` on the host client must be replaced. This is because a guest client does not set the correct `sourceUri`. The guest client has only one URI modified by VS Live Share for the collaborative session.

ActionMessage sent from the server

An `ActionMessage` sent from the host server is processed at the host client. If it is a `ReponseAction`, a `subclientId` and a `requestId` are set. This allows the host client to forward it to the correct client via VS Code Live Share. The appropriate host client then processes the message further. If the special `subclientId` 'H' is set in the message, then the message is processed at the host client itself. If it is a `SetModelAction` or `UpdateModelAction`, which the server sends in the course of an operation on the model, then this message is broadcast to all guests and also executed on the host itself. If it is not a `ResponseAction` at all, because no `subclientId` and `requestId` is set, it will also be broadcast to all guests and also executed on its own host client.

Data flow

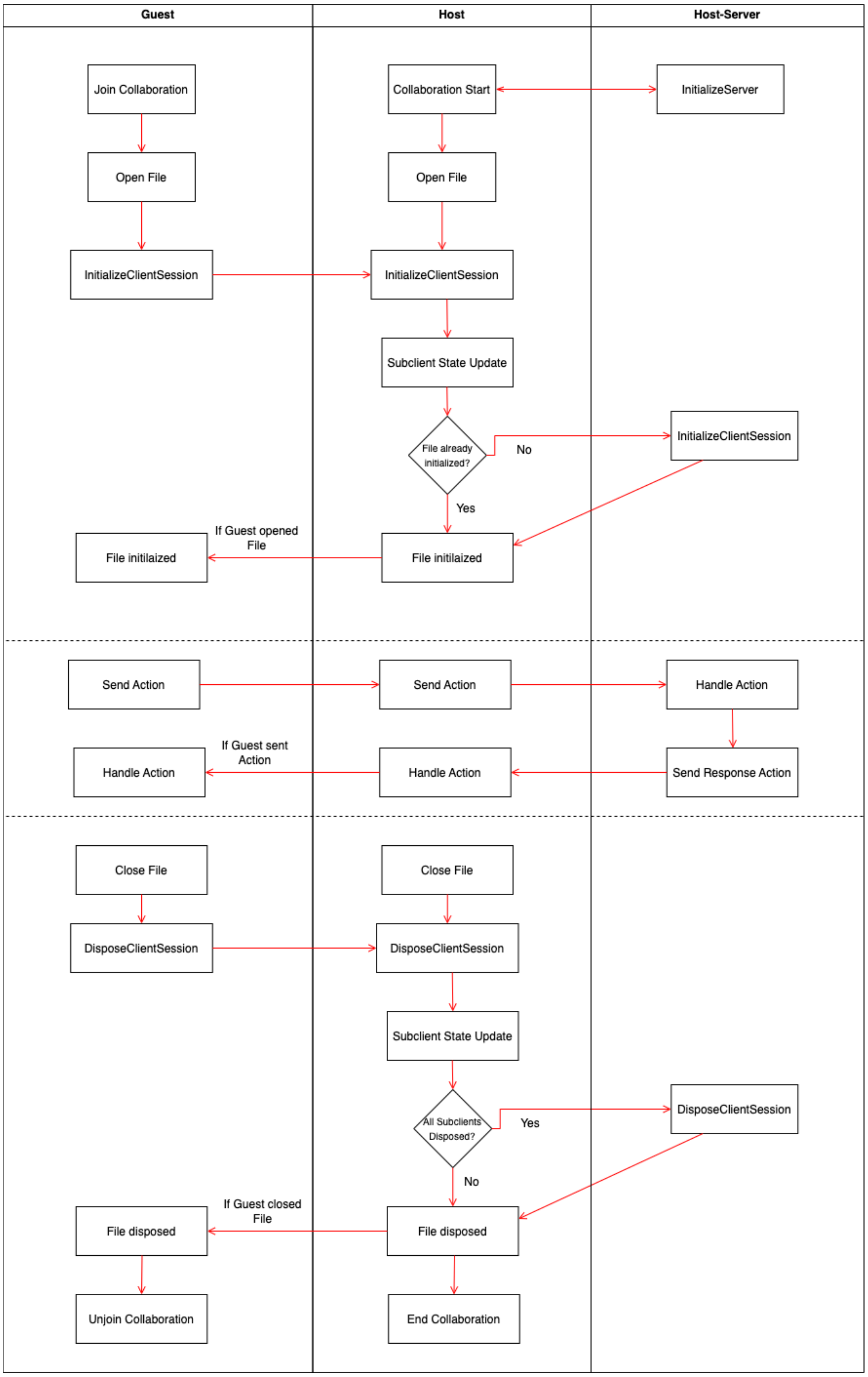


Figure 4.4: Data flow for an InitializeClientSession, Action and DisposeClientSession message in a collaborative session.

The data flow diagram in Figure 4.4 represents the three main message types of the GLSP protocol as they would behave in a collaborative session between a guest, host and host server.

In the first part, the host starts the collaboration session and the host joins it. Then the file is opened and an `InitializeClientSession` is initiated. The guest sends this to the host, and the host updates the subclient state. If this file has not been initiated yet, the message is sent to the server. In case the guest has initiated the message, the guest is told that the work is done now.

In the second part an `Action` is sent to the server. If the guest initiates it, it is sent to the host server via the host. If the message was initiated by the guest, the response is forwarded to the guest. The host or the guest then process the response via their `ActionHandlers`.

In the last part a file is closed and thus a `DisposeClientSession` message is initiated. Also here the subclient state is updated on the host. If all subclients have closed this file, the host sends the message to the server. If the message is initiated by the guest, the guest is again told that the work is done. When the file disposed, the guest can leave the collaboration session and the host can close it.

4.3.2 Enable command stacks on user level through one single GLSP server

Command stacks are unique per session in the current implementation of the GLSP server. However, since multiple users are working on a file within a session, a solution must be found that keeps multiple command stacks within a session. To implement command stacks on user level a `CommandStackManager` as an intermediate layer is introduced.

At all points where the GLSP server accesses the session-wide command stack via dependency injection, this command stack is accessed via a `CommandStackManager`. The `CommandStackManager` works with a `subclientId`, which is passed to the server with every access. The `CommandStackManager` stores the respective command stack for each `subclientId` in a `Map`. It is important that the client sends a `subclientId` with all messages. If no `subclientId` is sent, a fallback `subclientId` is used, which holds all operations that cannot be assigned to a subclient.

4.3.3 Conflict reduction during collaboration process

An important part of a collaborative editor is to be able to avoid possible conflicts already during the session. This can be achieved in many different ways. One way is to provide a chat functionality that allows participants of a session to communicate with each other in real-time. VS Live Share includes this functionality out of the box. VS Live Share also assigns each user an exact color, which makes it possible to recognize and associate activities of other users. The prototype will incorporate in additional ways that allow participants to collaborate efficiently and conflict-free.

CollaborationAction

To make this work a new action type called `CollaborationAction` is added. These actions are only sent between the subclients via VS Live Share and are not sent to the server. `CollaborationActions` have no response and are broadcast to all other subclients. Again, a distinction must be made whether the `CollaborationAction` is initiated by the host or by a guest. In the case that the host initiates this message, it is broadcast to all guests and processed there by their `ActionHandler`. If the guest initiates a `CollaborationAction`, it is first sent to the host via VS Live Share and then to all other guests. The host and all other guests then process this `CollaborationAction` via their `ActionHandler`. To include all necessary information about the initiator of the `CollaborationAction`, this information is enhanced to the message. This includes the `subclientId` of the initiator, the `name` of the user and the assigned `color` of the user. This information is stored in an attribute called `initialSubclientInfo`.

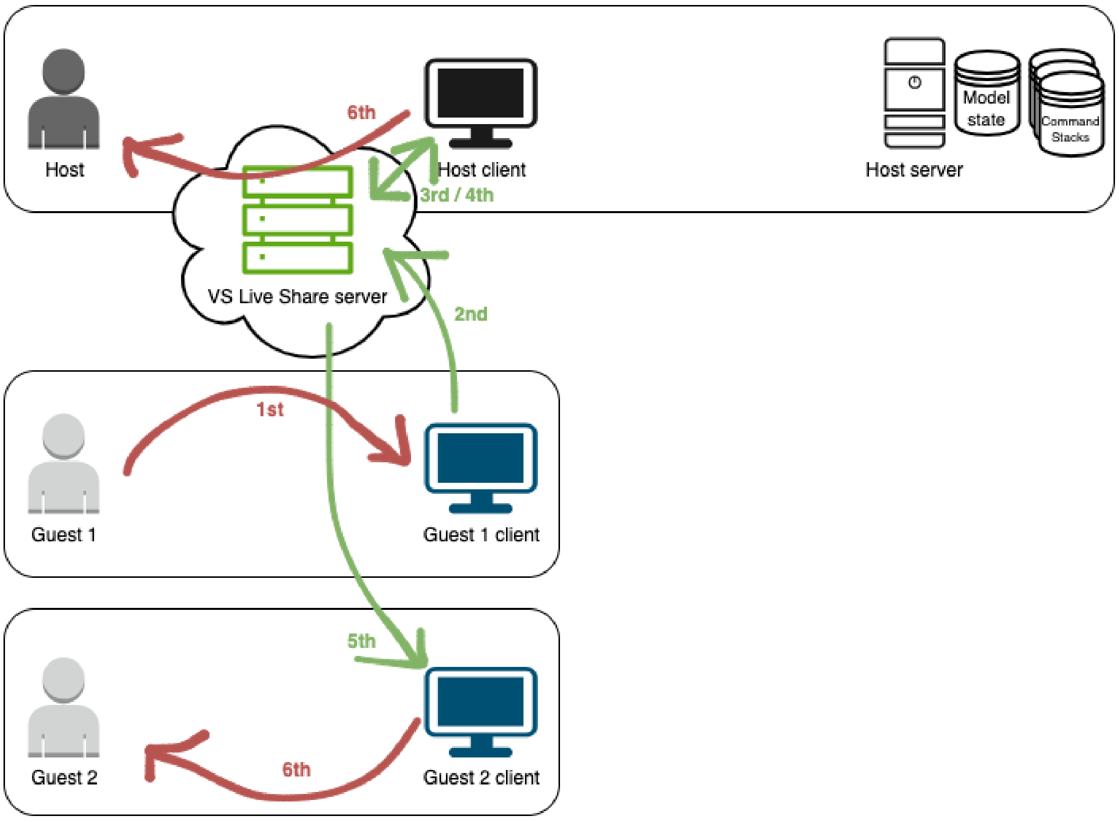


Figure 4.5: Data flow for a CollaborationAction initiated by Guest 1.

The Figure 4.5 shows how the data flow of a `CollaborationAction` initiated by Guest

1 looks like. In the first step **(1st)** a `CollaborationAction` is initiated by Guest 1. This can be, for example, moving the mouse cursor. The guest client sends this action with the `initialSubclientInfo` via VS Live Share to the host client **(2nd/3rd)**. Once there, the information is broadcast to all other guests via VS Live Share **(4th/5th)**. Finally, the `ActionHandlers` of the respective clients execute this action **(6th)**. This message is not executed at the own client, because it is only relevant for other subclients. This can be for example the display of the initiator's mouse pointer on the screen.

Mouse pointer

As mentioned before displaying the mouse pointers of other participants in a collaborative session is a way for conflict reduction. For this the initator host sends a `MouseMove-Action` which is a `CollaborationAction`. This contains the absolute `position` of the initiating user represented as a `Point` (x and y position). Furthermore this action has the `initialSubclientInfo` like all `CollaborationActions`. With the `position`, the `name` and the `color` of the initiator it is possible to display a mouse pointer in the desired color with the name of the user as a label on the actual position for all other hosts.

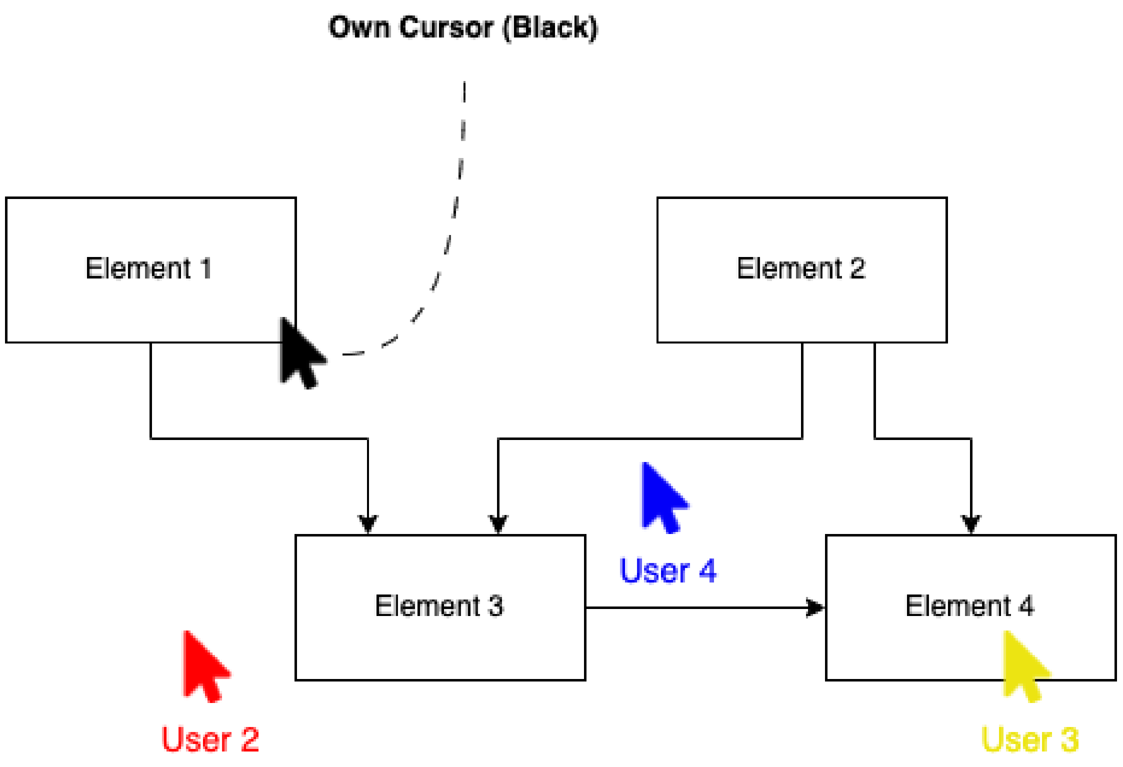


Figure 4.6: Mock-up of participant's cursor representation in a collaboration session.

The Figure 4.6 represents a mock-up of what a collaboration session with four participants would look like. It illustrates a simple example with four nodes and five edges. The example shows the own cursor, which is displayed normally by the operating system. All other cursors are displayed in the assigned color and a label with the name of the user. The position of the cursor is always shown in real-time.

Viewport

The prototype will also display viewports of other participants in a collaborative session. To do this, the initiating subclient sends a `ViewportBoundsChangeAction`, which is also a `CollaborationAction`. With the info of the initiator this action holds the `bounds` of the viewport. `Bounds` consist of a `Point` (x and y position) and a `Dimension` (width and height of the viewport). With the information of the `color` and the `bounds` the prototype will display other viewports, as a dashed transparent rectangle, in real-time.

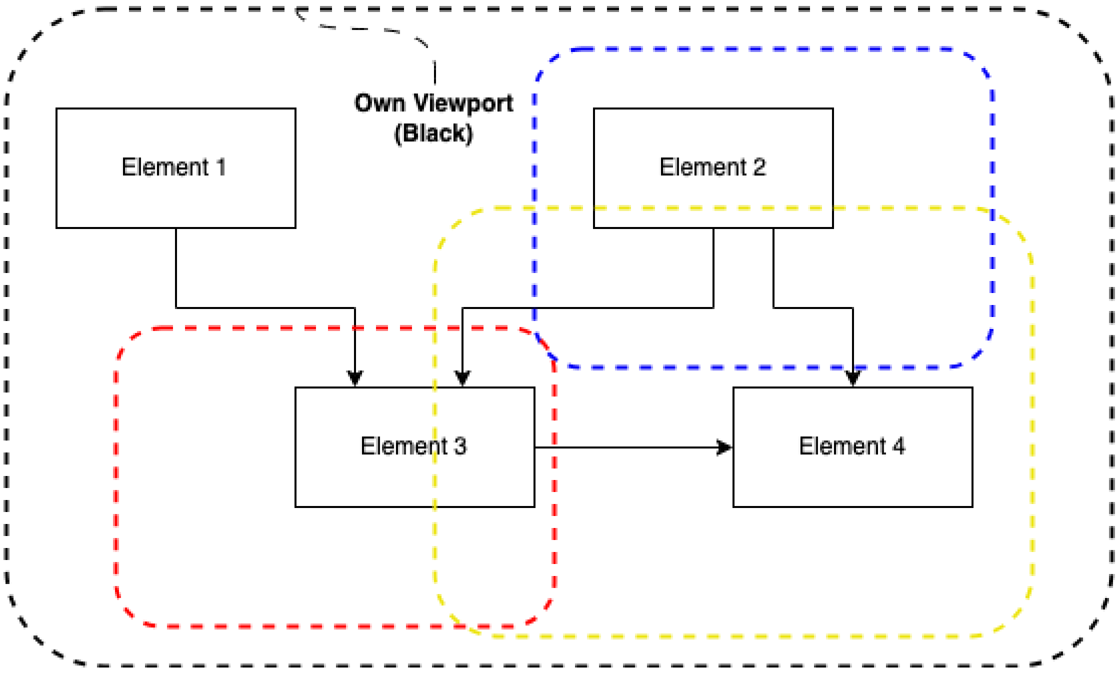


Figure 4.7: Mock-up of participant's viewport representation in a collaboration session.

The example in Figure 4.7 is the same one which was used for mouse pointer. However, this mock-up shows the viewports of all other participants in the assigned color. The own viewport is only used for the example, and will not be included in the prototype.

Selections

A third functionality which is included in the prototype, to do conflict reduction during collaboration time, is displaying selections of other users' elements. For this purpose, the initiating subclient sends a `SelectionChangeAction`, which also inherits from the `CollaborationAction`. This new action has an array with the ID of all selected elements. Since a multi-selection is possible, several elements can be selected. To efficiently display selections of other participants filled selection icons, as rectangles and circles, in the color of the initiating user are used. The prototype displays them at the upper left edge for nodes and directly at the start for edges.

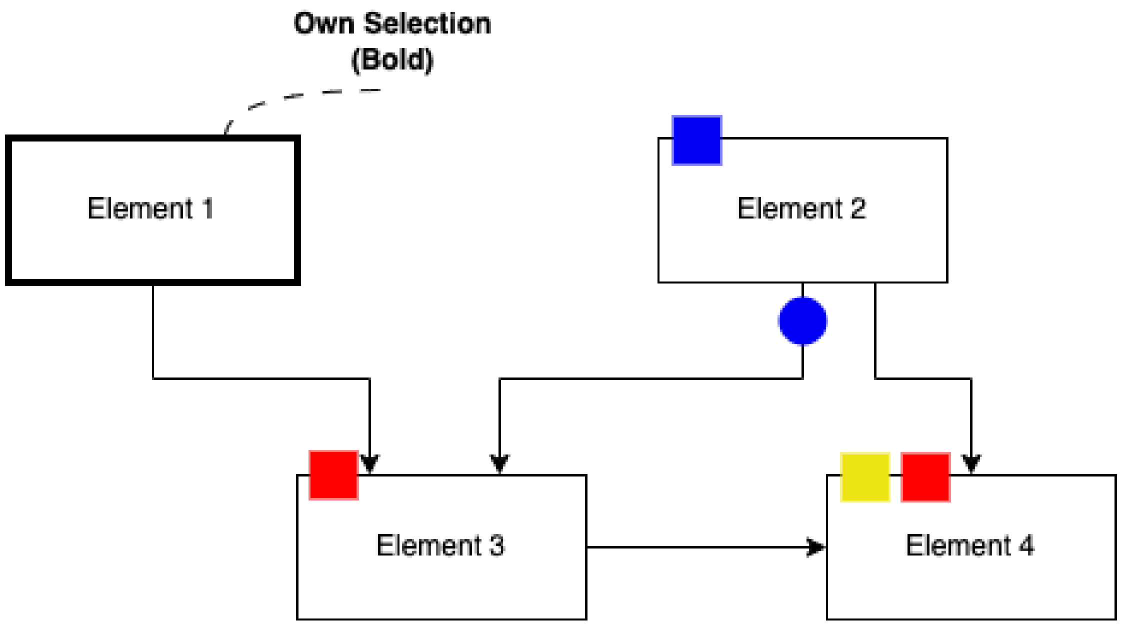


Figure 4.8: Mock-up of participant's selections representation in a collaboration session.

The Figure 4.8 shows a mock-up of how the prototype will represent selections as rectangles and circles. The custom selection is shown here in bold. However, it will be displayed by the diagram tool as before. Selections of other users are displayed by means of a rectangle for nodes and circle for edges filled in the color of the user. In this example, User 1 (red) selects Element 3 and Element 4, User 2 (yellow) selects Element 4 and User 3 (blue) selects Element 2 and the edge between Element 2 and Element 3. If one element is selected by more than one user multiple selection icons are displayed, e.g. for Element 4.

DisposeSubclientAction

To remove displayed elements like mouse pointers, viewports and selections of other participants in case of closing the document, an additional action is introduced to inform about this. A `DisposeSubclientAction` contains only the `initialSubclientId` of the participant which is closing some document. Just like `CollaborationActions`, the `DisposeSubclientAction` is broadcast to all other participants via VS Live Share server. However, this action does not inherit from the `CollaborationAction` because it is a special case that does not require a name and color. Also, this action does not have a response and is only considered as a notification, which is not sent to the server. All other subclients then use this action to remove inserted elements of the initiating user.

However, there may also be cases where a user who leaves a collaborative session does not send a `DisposeSubclientAction` to all other participants. This can happen, for example, if the internet connection fails or the program terminates abruptly. However, the solution can add a listener for VS Live Share that emits when participants of a collaborative session change. If a participant is no longer in the session, but this participant has not yet been properly disposed, the host sends a `DisposeSubclientAction` to all other guests. The host and all guests then execute this action to remove any elements still displaying from the missing user.

CHAPTER 5

Prototype

This chapter discusses the implementation of the prototype. For this purpose, necessary changes to the GLSP protocol will be presented. The implementation of the prototype will be performed on top of the Workflow example. For this purpose, necessary changes and extensions to the existing GLSP server and GLSP client implementation will be presented by using code listings and descriptions. Furthermore, the following part will show screenshots of the Workflow VS Code example within a collaborative session.

5.1 Extend the GLSP protocol

This section demonstrates what is necessary to extend the GLSP protocol in general to make it work for collaborative purposes. Further specific changes for certain specifications will be outlined in the next chapters.

5.1.1 Initialize- & DisposeClientSessionParameters

The `Initialize-` and `DisposeClientSessionParamters` are used to send data from the host client to GLSP server during the initialization and disposition of a client session. Always a `clientSessionId` and additionally a `diagramType` is sent during initialization process. Additionally there is another field called `args` which can be used to send additional custom arguments. This field is of type `Args` and consists of a simple object containing key-value pairs. Since the prototype's extension will send the same information for both methods between two GLSP clients, namely when a guest subclient sends its initialization or disposition to the host subclient, the GLSP client will use the `args` attribute to communicate the **`subclientId`** and the **`relativeDocumentUri`**. The `subclientId` tells which guest subclient sent the information to the host subclient and the `relativeDocumenUri` tells which document or file is addressed.

5.1.2 ActionMessage

An `ActionMessage` is used to send an action between a GLSP server and host client. This can be used in both directions. This type of message is also used to communicate between the host subclient and the guest subclient. An `ActionMessage` has the `clientId (clientSessionId)` to associate the action with a client session and the `action` itself. To identify the message later, certain information must be sent along. For this purpose an `args` attribute of type `Args` is introduced to send the **`relativeDocumentUri`** with the message. The `subclientId` is not sent in the `ActionMessage`, but directly in the action. This provides an advantage that the server can map it back to the `ResponseAction` more easily.

5.1.3 Action

The implementation extends the `Action` class with a property **`subclientId`**. This is used so that the host client can forward responses from the server to actions sent to the intended subclient. For this it is necessary that the GLSP server maps the sent `subclientId` to all outgoing server actions, as the GLSP server does with the `requestId`.

5.1.4 CollaborationAction

As already described in the previous chapter, a new type of action is introduced: `CollaborationAction`. These are not sent to the GLSP server, but only between subclients to display visual elements such as mouse pointers, viewports and selections of other collaboration participants. A `CollaborationAction` has a collaboration flag, which identifies it as such and therefore it is not sent to the GLSP server. It also has an `initialSubclientInfo`, which contains a `subclientId`, a `name` and a `color`. These are used to visually display the initiator of the `CollaborationAction`. A `visible` flag left determines whether this element should be displayed or hidden, in the case that the function is switched off by the user.

The following three actions inherit from the `CollaborationAction`: `MouseMoveAction` with `position` as an attribute, `ViewportBoundsChangeAction` with `bounds` as an attribute and `SelectionChangeAction` with `selectedElements` as an attribute.

The prototype adds the `DisposeSubclientAction`, which has the `initialSubclientId` as an attribute. This is used if a subclient closes a document. All visual elements of the initiator are then removed from all other subclients.

5.2 Extend the GLSP server

This section shows all the necessary customizations to the GLSP server to make it suitable for collaborative requirements. This includes changes to the `ModelState`, the mapping of the subclientId and the implementation of a CommandStack on user (subclient) level.

5.2.1 ModelState

Since the solution with a single GLSP server only needs one ModelState for all subclients per client session, no major adjustments to the GLSP server are necessary. One client session means one document/file for the entire collaborative session.

There is now a problem with multiple subclients sending a RequestModelAction to the same client session and the action reloads the SourceModel into the ModelState each time. This causes the GLSP server to overwrite saved changes. To prevent this an additional configuration named DISABLE_RELOAD is added to the key-value pair attribute called options of the RequestModelAction. This field is of type Boolean and tells the GLSP server whether the ModelState should be reloaded for an already loaded state or not. By default this flag is false and that means that in this case the state would be reloaded.

The customized RequestModelActionHandler checks if this flag is set to false or the ModelState has not been loaded yet. If that is the case, the ModelState is loaded from the SourceModel. If this is not the case, this part is simply skipped and the already loaded model is submitted.

5.2.2 SubclientId mapping

As defined in the previous Section 5.1, all subclients send a subclientId to the host client and thus further to the host server with each action. If this action results in another action that is sent from the server back to the client, then the host client must forward this to the correct subclient.

To make this happen a new static function addSubclientId(initialAction, extendedAction) (Listing 5.1) is added to the Action class, which takes the subclientId of an initialAction and maps it to an extendedAction. Server-created actions, in the course of a client action, pass through the DefaultActionDispatcher to map, for example, the incoming requestId to the ResponseAction. With this extension, the subclientId is also mapped to the incoming action at this point using the Action.addSubclientId(initialAction, extendedAction) function.

```
public abstract class Action {
    ...
    public static Action addSubclientId(
        final Action initialAction,
        final Action extendedAction
    ) {
      if (initialAction.getSubclientId() != null) {
        extendedAction.subclientId = initialAction.subclientId;
      }
```

```
        return extendedAction;
    }
    ...
}
```

Listing 5.1: Function addSubClientId which addes subclientId from initial to extended action.

Also in other places this function is called to map the `subclientId` to outgoing actions. For example, the `GModelCreateNodeOperationHandler` creates a `SelectAction` in the flow of a `CreateNodeOperation`. This handler is running when a new node is created in the diagram and creates an action which should automatically select the new node.

```
actionDispatcher.dispatchAfterNextUpdate(
    Action.addSubclientId(
        operation,
        new SelectAction()
    ),
    Action.addSubclientId(
        operation,
        new SelectAction(List.of(element.getId()))
    )
);
```

Listing 5.2: Function-call of Action.addSubclientId to map subclientId from CreateNodeOperation to SelectAction

Listing 5.2 illustrates how the `GModelCreateNodeOperationHandler` first creates a new empty `SelectAction` to deselect all elements and also a new `SelectAction` with the ID of the new node. In both cases the `subclientId` sent by a `CreateNode-Operation` (variable `operation`) is mapped to the newly created action.

5.2.3 CommandStack on user level

To implement `CommandStacks` at user (subclient) level, the access to the `Command-Stack` at the GLSP server has to be adapted. In the non-collaborative implementation of GLSP server, a single `CommandStack` is held directly in the `ModelState` implementation. The implementation extends the GLSP server with a `CommandStackManager` that stores all `CommandStacks` per `subclientId` in a `Map`-object.

To do this, an interface `CommandStackManager` with following functions is created:

- The function **`getOrCreateCommandStack(subclientId)`**, which returns a `CommandStack` for a `subclientId`. If no `CommandStack` is found, a new one is created first and then returned.

- The function **`getAllCommandStacks()`** returns all `CommandStacks`.
- The function **`setCommandStack(commandStack, subclientId)`** allows to store a `CommandStack` for a `subclientId` in the `Map`-object.

In addition to the interface, a default implementation that fulfills this logic is created. Listing 5.3 shows how the `DefaultCommandStackManager` class implements this interface. In case null was passed as `subclientId` because the command cannot be assigned to a subclient, a `FALLBACK_SUBCLIENT_ID` is used.

Also noticeable is that this class injects a `CommandStackFactory`, which is an interface that contains a function `createCommandStack()` that should create a `CommandStack`. The default implementation of this factory pattern is implemented in the `DefaultCommandStackFactory` class.

```java
public class DefaultCommandStackManager implements CommandStackManager {

    @Inject
    CommandStackFactory factory;

    protected Map<String, CommandStack> commandStackMap = new HashMap<>();

    @Override
    public CommandStack getOrCreateCommandStack(final String subclientId) {
        String subclientIdOrFallback = getSubclientIdOrFallback(subclientId);
        if (commandStackMap.containsKey(subclientIdOrFallback)) {
            return commandStackMap.get(getSubclientIdOrFallback(
    subclientIdOrFallback));
        }

        CommandStack commandStack = factory.createCommandStack();
        commandStackMap.put(subclientIdOrFallback, commandStack);
        return commandStack;
    }

    @Override
    public List<CommandStack> getAllCommandStacks() {
        return new ArrayList<>(commandStackMap.values());
    }

    @Override
    public void setCommandStack(final CommandStack commandStack, final String
     subclientId) {
        String subclientIdOrFallback = getSubclientIdOrFallback(subclientId);
        if (commandStackMap.containsKey(subclientIdOrFallback)) {
            commandStackMap.get(subclientIdOrFallback).flush();
        }
        commandStackMap.put(subclientIdOrFallback, commandStack);
    }

    private String getSubclientIdOrFallback(final String subclientId) {
        if (subclientId != null) {
            return subclientId;
        }
        return CollaborationUtil.FALLBACK_SUBCLIENT_ID;
    }
}
```

Listing 5.3: Default implementation of CommandStackManager interface.

Previously, `ModelState` implementations accessed the `CommandStack` directly. In the new implementation, the `CommandStackManager` is injected and the chosen `CommandStack` is accessed via the `subclientId`. Through this extension it is now possible that each user has its own `CommandStack` and undo/redo operations thus work at user level.

```
public abstract class DiagramModule extends GLSPModule {
   ...
   @Override
   protected void configureBase() {
      ...
      bind(CommandStackFactory.class)
        .to(bindCommandStackFactory()).in(Singleton.class);
      bind(CommandStackManager.class)
        .to(bindCommandStackManager()).in(Singleton.class);
   }

    protected Class<? extends CommandStackFactory> bindCommandStackFactory()
    {
      return GModelCommandStackFactory.class;
   }

   protected Class<? extends CommandStackManager> bindCommandStackManager() {
      return DefaultCommandStackManager.class;
   }
   ...
}
```

Listing 5.4: Configure CommandStackFactory and CommandStackManager for dependency injection.

Listing 5.4 displays how the dependency injection container gets extended to bind the default implementations of both, namely `CommandStackFactory` and `CommandStackManager` interfaces, to their corresponding interfaces at the `DiagramModule`. So it is possible to inject the `CommandStackManager` at all places within a client session. Furthermore, it is possible to simply overwrite the default implementation.

5.3 Extend the GLSP VS Code integration

The biggest customization to enable collaborative work using GLSP, takes place at GLSP VS Code integration. This is because VS Live Share shares data between subclients and VS Live Share is clearly best located in the GLSP VS Code integration layer.

5.3.1 CollaborationGlspClient

In the default implementation, the GLSP client uses a JSON-RPC implementation of `GLSPClient` to communicate with the GLSP server. **`GLSPClient`** should not be confused with the term **GLSP client**. `GLSPClient` is an interface used for communication between the GLSP client and GLSP server. GLSP client is the name for the client part of the server-client architecture. To realize the prototype, the point where the GLSP client would normally send messages to the GLSP server, namely the JSON-RPC `GLSPClient` implementation, will be adapted. Here for, a new `GLSPClient`, named `CollaborationGlspClient`, is introduced. This new class communicates via VS Live Share with other subclients and in the case of the host subclient also with the GLSP server. Since this class is the main component of this customization, this subsection presents and describes code excerpts in detail.

```
export class CollaborationGlspClient implements GLSPClient {
    protected readonly BROADCAST_ACTION_TYPES = [SetModelAction.KIND,
    UpdateModelAction.KIND];

    readonly id: string;

    protected commonProvider: CommonCollaborationGlspClientProvider;
    protected hostProvider: HostCollaborationGlspClientProvider;
    protected guestProvider: GuestCollaborationGlspClientProvider;

    protected registeredSubclientMap = new Map<string, Map<string, string>>()
    ;

    protected serverClientIdMap = new Map<string, string>();

    protected actionMessageHandlers: ActionMessageHandler[] = [];

    constructor(
        protected glspClient: GLSPClient,
        config: CollaborativeGlspClientConfig
    ) {
    ...
```

Listing 5.5: Member variables and constructor of CollaborationGlspClient.

The **`CollaborationGlspClient`** class implements the `GLSPClient` interface (Listing 5.5). Another `GLSPClient` is passed in the constructor, which is then responsible for communication with the GLSP server. Furthermore, config `CollaborativeGlsp-`

ClientConfig is passed, which takes over the three providers for communication with other subclients. The architecture requires that there is exactly one host subclient and any number of guest subclients. If the subclient acts as a host, the hostProvider is used; if the subclient acts as a guest, the guestProvider is used. Methods that are used by both host and guest are implemented in the commonProvider.

The registeredSubclientMap stores a further Map with all subclientIds for all documents (identified by the realtiveDocumentUri) for their local clientSessionIds. In addition, another state that holds the global clientSessionIds for each relativeDocumentUri is needed. The prototype uses the serverClientIdMap to store this information. It also needs a list of all ActionMessageHandlers that are triggered at certain points.

```
async initializeClientSession(params: InitializeClientSessionParameters):
    Promise<void> {
    if (!params.args?.subclientId) {
        params.args = {
            ...params.args,
            subclientId: this.commonProvider.getSubclientIdFromSession()
        };
    }

    if (!this.commonProvider.isInCollaborationMode() || this.commonProvider.
    isHost()) {
        const relativeDocumentUri = this.getRelativeDocumentUriByArgs(params.
    args);

        const subclientId = params.args?.subclientId as string;
        const subclientMap = this.registeredSubclientMap.get(
    relativeDocumentUri) || new Map<string, string>();

        const initialized = subclientMap.size > 0;
        subclientMap.set(subclientId, params.clientSessionId);
        this.registeredSubclientMap.set(relativeDocumentUri, subclientMap);
        if (initialized) {
            return;
        }
        params.clientSessionId += `_${subclientId}`;
        this.serverClientIdMap.set(relativeDocumentUri, params.
    clientSessionId);

        return this.glspClient.initializeClientSession(params);
    } else if (this.commonProvider.isGuest()) {
        return this.guestProvider.initializeClientSessionForGuest(params);
    }
}
```

Listing 5.6: Method initializeClientSession of CollaborationGlspClient.

The **intializeClientSession(params)** method is called when a new document is initialized. Both `subclientId` and `relativeDocumentUri` are defined in the `params` parameter. If no `subclientId` is stored, the `commonProvider` is called to set the `subclientId` in the `params` pararmeter.

In line 9 in Listing 5.6 the method checks whether the GLSP client is currently not in a collaborative session OR whether it is currently the host of the collaborative session. In this case, it stores this subclient with the local `clientSessionId` in the `registeredSubclientMap` of the respective document. If this document has already been initialized on the server, the function terminates. If not, a global `clientSessionId` is set in line 21, which is a combination of the local `clientSessionId` and `subclientId`. With the adjusted `params`, a request is sent to the GLSP server and the result is returned. The reason why this information is also stored in non-collaborative mode is that the information will be needed if one user later on subsequently switches to a collaborative session and this GLSP client would become the host of this session.

In case this GLSP client is the guest of a collaborative session, an `intializeClientSession` request is sent via the `guestProvider` to the host subclient. How the providers use VS Live Share to implement the data transfer between subclients will be shown later. Technically any library could be used, instead of VS Live Share, to implement providers for data exchange. The host then executes this request, sent by the provider, as a proxy on its environment as described above in the previous paragraph.

```
async disposeClientSession(params: DisposeClientSessionParameters): Promise<void> {
    if (!params.args?.subclientId) {
        params.args = {
            ...params.args,
            subclientId: this.commonProvider.getSubclientIdFromSession()
        };
    }

    if (this.commonProvider.isInCollaborationMode() && this.commonProvider.isHost()) {
        this.handleDisposeSubclientMessage(params);
    }

    if (!this.commonProvider.isInCollaborationMode() || this.commonProvider.isHost()) {
        const relativeDocumentUri = this.getRelativeDocumentUriByArgs(params.args);

        const subclientId = params.args?.subclientId as string;
        const subclientMap = this.registeredSubclientMap.get(relativeDocumentUri) || new Map<string, string>();

        subclientMap.delete(subclientId);
        this.registeredSubclientMap.set(relativeDocumentUri, subclientMap);
        if (subclientMap.size > 0) {
            return;
23          }
24          this.serverClientIdMap.delete(relativeDocumentUri);
25          return this.glspClient.disposeClientSession(params);
26      } else if (this.commonProvider.isGuest()) {
27          return this.guestProvider.disposeClientSessionForGuest(params);
28      }
29  }
```

Listing 5.7: Method disposeClientSession of CollaborationGlspClient.

The **`disposeClientSession(params)`** method (Listing 5.7) is called when a GLSP client closes a document.

As with the `initializeClientSession` method, `relativeDocumentUri` and `subclientId` are set in the `params`. If the `subclientId` is not set, it is set at the start of the method via the `commonProvider`.

First, line 9 checks whether this GLSP client is in a collaborative session AND is the host of the session. If that is the case, the implementation broadcasts a `DisposeSubclientAction` to all subclients via the private method `handleDisposeSubclientMessage` and executes it on their `ActionHandlers`. The `ActionHandlers` of the host also execute this action. This is done so that all subclients know as soon as a subclient closes a document or leaves a session.

Then in line 13 it is checked again whether this GLSP client is not in a collaborative session OR the host of the session. If that is the case, the entry of the calling subclient for this document is deleted from the `registeredSubclientMap`. If all entries for a document have been deleted, this means that the document can also be disposed from the GLSP server. To do this, the entry in the `serverClientIdMap` is first deleted and then a request is sent to the GLSP server in line 24.

If the method is called by a guest subclient, a request is again sent in line 27 via the `guestProvider` to the host subclient, which then processes the request as a proxy.

```
sendActionMessage(message: ActionMessage): void {
    if (!message.action.subclientId) {
        message.action.subclientId = this.commonProvider.
    getSubclientIdFromSession();
    }

    if (CollaborationAction.is(message.action)) {
        this.handleCollaborationAction(message as ActionMessage<
    CollaborationAction>);
    } else if (!this.commonProvider.isInCollaborationMode() || this.
    commonProvider.isHost()) {
        const relativeDocumentUri = this.getRelativeDocumentUriByArgs(message
    .args);

        message.clientId = this.serverClientIdMap.get(relativeDocumentUri) ||
     '';
12
13         if (message.action.kind === RequestModelAction.KIND) {
14             const requestModelAction = message.action as RequestModelAction;
15             requestModelAction.options = {
16                 ...requestModelAction.options,
17                 disableReload: true
18             };
19             if (message.action.subclientId !== SUBCLIENT_HOST_ID) {
20                 requestModelAction.options = {
21                     ...requestModelAction.options,
22                     sourceUri: getFullDocumentUri(relativeDocumentUri)
23                 };
24             }
25         }
26         this.glspClient.sendActionMessage(message);
27     } else if (this.commonProvider.isGuest()) {
28         this.guestProvider.sendActionMessageForGuest(message);
29     }
30 }
```

Listing 5.8: Method sendActionMessage of CollaborationGlspClient.

The **`sendActionMessage(message)`** method (Listing 5.8) gets called by the GLSP client when it sends an `ActionMessage` to the GLSP server. Only the host subclient sends a message to its GLSP server. In contrast, guest subclients send their message to the host subclient, which then forwards this message to the GLSP server.

As in the other two methods, the system first checks whether the `subclientId` is set. If this is not the case, it is set.

Line 6 checks whether a `CollaborationAction` is transmitted. Messages of this type are not sent to the GLSP server, but are only executed by the host subclient itself and broadcast to all guest subclients.

Line 8 checks whether the GLSP client is not in a collaborative session OR the host subclient is. If that is the case, the global `clientSessionId` is attached to the message through the `relativeDocumentUri` and the `serverClientIdMap`. If it is a `RequestModelAction`, the function sets the `disableReload` flag to `true`, so that the `sourceModel` is not reloaded each time. And if the message originates from a guest subclient, the `sourceUri` is replaced with the full URI, as the guest subclient only knows the `relativeDocumentUri` inside the collaborative workspace. Finally, the GLSP client sends the `message` to the GLSP server

If the method is called by a guest subclient, it is sent via the `guestProvider` to the host subclient, which then processes it further and sends it to the GLSP server.

```
1  this.glspClient.onActionMessage((message: ActionMessage) => {
2      const relativeDocumentUri = this.getRelativeDocumentUriByServerClientId(
       message.clientId);
3      const subclientMap = this.registeredSubclientMap.get(relativeDocumentUri)
       ;
4      if (!subclientMap) {
5          return;
6      }
7      if (!this.commonProvider.isInCollaborationMode()) {
8          const localClientId = subclientMap.get(SUBCLIENT_HOST_ID) || '';
9          this.handleMessage(SUBCLIENT_HOST_ID, message, localClientId);
10     } else if (this.commonProvider.isHost()) {
11         const subclientId = message.action.subclientId;
12         if (subclientId == null || this.BROADCAST_ACTION_TYPES.includes(
       message.action.kind)) {
13             this.handleMultipleMessages(subclientMap, message);
14         } else {
15             const localClientId = subclientMap.get(subclientId) || '';
16             this.handleMessage(subclientId, message, localClientId);
17         }
18     }
19 });
```

Listing 5.9: Method onActionMessage of CollaborationGlspClient.

If the GLSP server sends a message to the GLSP client, this message is handled in the callback implementation of the **`onActionMessage(callback)`** method (Listing 5.9).

First, the registered subclients for this `relativeDocumentUri` are loaded in line 3. If the GLSP client is not in a collaborative session, the `ActionMessage` is simply executed by the local `ActionMessageHandler` as in the default `GLSPClient` implementation.

In a collaborative session, only a host subclient can receive messages from a GLSP server (line 10). If no `subclientId` is set in the incoming message OR the type of action is within the list of `BROADCAST_ACTION_TYPES`, then this message is executed by the local `ActionMessageHandler` and broadcast to all other guest subclients. This is done in the `handleMultipleMessage(subclientMap, message)` method. `BROADCAST_ACTION_TYPES` contains the `SetModelAction` and `UpdateModelAciton`, which are sent by the GLSP server during an operation on the model.

If a `subclientId` is set AND this message is not broadcast, it is either executed itself or sent to the initial sender of the initiator message, all done at the `handleMessage(subclientid, message, localClientId)` method . Before this, the local `clientSessionId` is loaded from the `subclientMap`, which is needed on the local GLSP client to execute the message.

5.3.2 CollaborationGlspClientProviders

To ensure that the `CollaborationGlspClient` runs, a `CollaborativeGlspClientConfig` must be passed. This contains an implementation of the following three interfaces: `CommonCollaborationGlspClientProvider`, `HostCollaborationGlspClientProvider` and `GuestCollaborationGlspClientProvider`. It is also possible for all three interfaces to be implemented in just one so-called `CollaborationGlspClientProvider`. This subsection shows all methods that are implemented to enable collaborative working with GLSP. In the default implementation, which is presented later, VS Live Share is used to send data between subclients. However, it is possible to use any other form of data exchange to implement these providers. The only important thing is that the architecture remains the same and that there is only a maximum of one host subclient that handles the communication with the GLSP server. The number of guest subclients is not limited.

```
export interface CommonCollaborationGlspClientProvider {
    initialize(config: CollaborationGlspClientProviderInitializeConfig):
    Promise<void>;
    isInCollaborationMode(): boolean;
    isHost(): boolean;
    isGuest(): boolean;
    getSubclientIdFromSession(): string;
    getSubclientInfoFromSession(): SubclientInfo;
}
```

Listing 5.10: Interface with methodes of CommonCollaobrationGlspClientProvider.

The **`CommonCollaborationGlspClientProvider`** implements methods that must be supported by both a host and a guest subclient (Listing 5.10). The `initialize(config)` method is executed when the `CollaborativeGlspClient` is started and it waits until the method is terminated. As it returns a `Promise<void>`, the method can also execute asynchronous calls. The `isInCollaborationMode()` method returns a Boolean value as to whether this subclient is currently in a collaborative session. The `isHost()` and `isGuest()` methods return a Boolean value with the information as to whether the subclient is currently acting as host or guest. The `getSubclientIdFromSession()` function returns the `subclientId` of the subclient and the `getSubclientInfoFromSession()` function returns a `SubclientInfo` of the subclient. A `SubclientInfo` contains the `subclientId`, a human-readable name that is displayed on the screen for this subclient and a color that is assigned to the subclient. This can also be displayed on the screen to characterize users.

```
export interface HostCollaborationGlspClientProvider {
    handleActionMessageForHost(message: ActionMessage): void;
    handleMultipleActionMessagesForHost(messages: ActionMessage[]): void;
    onGuestsChangeForHost(handler: GuestsChangeHandler): void;
}
```

Listing 5.11: Interface with methodes of HostCollaobrationGlspClientProvider.

The **`HostCollaborationGlspClientProvider`** only needs to be implemented by a host subclient (Listing 5.11). The postfix *"ForHost"* and later also *"ForGuest"* are used in all these methods, as it is possible for both interfaces to be implemented by just one provider. The `handleActionMessageForHost(message)` method is called by the host subclient when an `ActionMessage` arrives from the GLSP server and this is sent to all guest subclients. The `handleMultipleActionMessagesForHost(messages)` does more or less the same, but only for an array of messages. Last but not least, the method `onGuestsChangeForHost(handler)` registers a `GuestsChangeHandler`, which is called when the number of guest subclients changes. A `GuestsChangeHandler` returns all current `subclientIds`.

```
export interface GuestCollaborationGlspClientProvider {
    initializeClientSessionForGuest(params: InitializeClientSessionParameters
    ): Promise<void>;

    disposeClientSessionForGuest(params: DisposeClientSessionParameters):
    Promise<void>;

    sendActionMessageForGuest(message: ActionMessage): void;
}
```

Listing 5.12: Interface with methodes of GuestCollaobrationGlspClientProvider.

In contrast, the **`GuestCollaborationGlspClientProvider`** only needs to be implemented by guest subclients (Listing 5.12). Here, the *"ForGuest"* postfix is shown again. The `initializeClientSessionForGuest(params)` is called when the `CollaborationGlspClient` handles an `IntializeClientSession` request from a guest subclient. The implementation of this function is intended to send this request to the host subclient, which afterwards can process it further and send it to the GLSP server. The `disposeClientSessionForGuest(params)` method does the same for the `DisposeClientSession` request. The `sendActionMessageForGuest(message)` method is called when a guest subclient wants to send an `ActionMessage` to the server. This is also forwarded to the host subclient, which then sends the information to the GLSP server.

5.3.3 LiveshareGlspClientProvider

The `LiveshareGlspClientProvider` is a default implementation of the `CollaborationGlspClientProvider`, as part of the prototype. This is where VS Live Share comes into play for the first time, as VS Live Share supports in exchanging data between subclients. In general, VS Live Share, or any other library for exchanging data, only needs to be used in this class. This makes it very easy to switch the collaboration library.

VS Live Share offers the possibility to:

- send an asynchronous notification without a response, used from the host to all guests - `notify(name, args)`
- send a synchronous request with response, used from the guest to the host - `request(name, args)`

The `name` parameter identifies a unique name that is used to assign the request or notification on the receiving side. On this side there are the methods `onNotify(name, handler)` for guest subclients and `onRequest(name, handler)` or for the host subclient to receive the arguments (parameter `args`) in the `handler` callback and process them further. This part briefly presents all four methods and how they are implemented for the prototype.

```
handleActionMessageForHost(message: ActionMessage): void {
    this.hostService.notify(ON_ACTION_MESSAGE, message);
}

async initializeSession(session: Session): Promise<void> {
    ...
    else if (session.role === Role.Guest) {
        this.service.onNotify(ON_ACTION_MESSAGE, (message: any) => {
            this.checkActionMessageAndSendToClient(message);
        });
    }
    ...
}

checkActionMessageAndSendToClient(message: ActionMessage): void {
    const subclientId = message.action.subclientId;
    if (this.getSubclientIdFromSession() === subclientId) {
        this.collaborationGlspClient.handleActionOnAllLocalHandlers(message);
    }
}
```

Listing 5.13: Asynchronous notification over notify(name, args) and onNotify(name, handler) at LiveshareGlspClient.

Listing 5.13 shows how an asynchronous notification is transferred between a host subclient and guest subclient. The `handleActionMessageForHost(message)` method, implemented from the `HostCollaborationGlspClientProvider`, is called by the `CollaborationGlspClient` when a message comes from the GLSP server and is to be forwarded to a guest subclient. The **`notify(ON_ACTION_MESSAGE, message)`** method is called here. `ON_ACTION_MESSAGE` is a unique name which is used in the **`onNotify(ON_ACTION_MESSAGE, message=>void)`** method to map the notification correctly. The callback function is defined once per session, if it is a guest subclient, in the `initializeSession(session)`. As the message is sent to all guest subclients, a check in the `checkActionMessageAndSendToClient(message)` method is needed to ensure the `subclientId` of the session matches the one of the message. If that is the case, the method can execute the received `message` on all local `ActionHandlers`.

```
sendActionMessageForGuest(message: ActionMessage): void {
    this.guestService.request(SEND_ACTION_MESSAGE, [message]);
}

async initializeSession(session: Session): Promise<void> {
    ...
    if (session.role === Role.Host) {
        this.service.onRequest(SEND_ACTION_MESSAGE, async params => {
            const message = params[1] as ActionMessage;
            this.collaborationGlspClient.sendActionMessage(message);
        });
    }
    ...
}
```

Listing 5.14: Synchronous notification over request(name, args) and onRequest(name, handler) at LiveshareGlspClient.

The other direction using a synchronous message is shown in Listing 5.14. For that purpose the `sendActionMessageForGuest(message)` method, implemented from the `GuestCollaborationGlspClientProvider`, is called by the `CollaborationGlspClient` when the guest subclient sends an `ActionMessage` to the GLSP server. To do this, it is first sent to the host subclient using the **`request(SEND_ACTION_MESSAGE, [message])`** method. The callback function at **`onRequest(SEND_ACTION_MESSAGE, params=>any)`** is defined in the `initializeSession(session)` method once per session for the host subclient and accepts all messages of the type `SEND_ACTION_MESSAGE`. The `CollaborationGlspClient` is called in the callback function to send a `message` to the GLSP server.

5.3.4 ToggleFeatureTreeDataProvider & CollaborationFeatureStore

Another feature is the ability to hide and show collaborative elements such as mouse pointers, viewports and selections of other participants. VS Live Share offers the possibility to extend the tree view with own elements. The `registerTreeDataProvider(viewId, treeDataProvider)` method can be used to pass a provider that provides new elements.

For this purpose, the `ToggleFeatureTreeDataProvider` class was invented, which implements the `TreeDataProvider` interface. This implementation creates a `TreeItem` for each of the three collaborative features mentioned above, which contains a `name`, `tooltip` and `command`. Also an `EventEmitter` is implemented, which is fired when one of the elements is clicked.

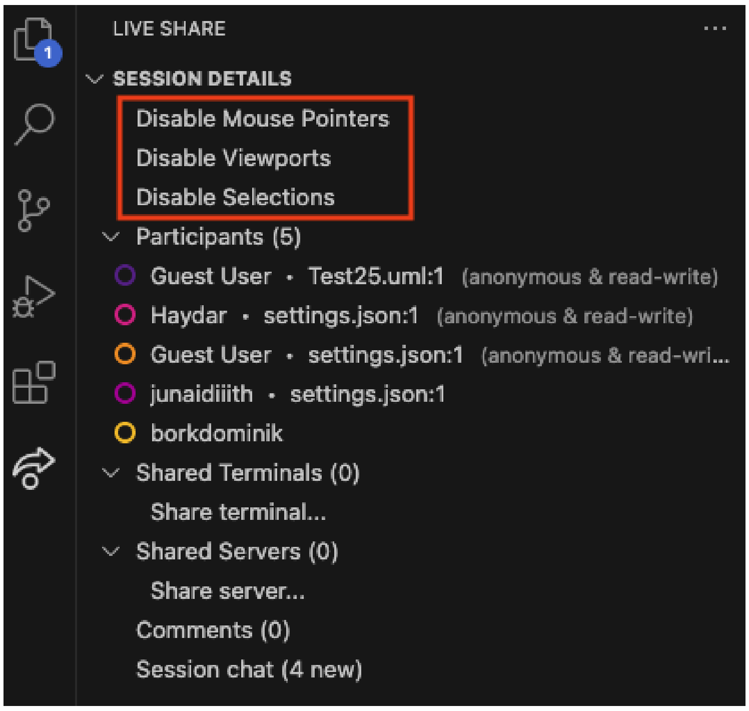

LIVE SHARE
SESSION DETAILS
Disable Mouse Pointers
Disable Viewports
Disable Selections
Participants (5)
Guest User • Test25.uml:1 (anonymous & read-write)
Haydar • settings.json:1 (anonymous & read-write)
Guest User • settings.json:1 (anonymous & read-wri...
junaidiiith • settings.json:1
borkdominik
Shared Terminals (0)
Share terminal...
Shared Servers (0)
Share server...
Comments (0)
Session chat (4 new)

Figure 5.1: VS Live Share tree view with toggle elements for collaborative features.

In addition to this provider, an extra class `CollaborationFeatureStore` is introduced, which implements the interface `ICollaborationFeatureStore`. This stores for each of the three collaborative actions whether it is currently enabled or disabled for its own GLSP client.

Finally, the created commands of the `TreeItems` are registered to VS Code within the introduced `configureCollaborationCommands(context)` function. Each command has a unique name with which that command is identified. If a command is performed, the client sends a `ToggleCollaborationFeatureAction` to all `ActionHandlers`, which show and hide the visual element in the diagram view. Toggle elements are displayed in Figure 5.1, which make it possible to enable or disable mouse pointers, viewports and selections. Clicking on one of these toggle buttons executes the command which sends the action for showing or hiding features to the GLSP client.

5.4 Extend the GLSP client

Finally, the GLSP client has to be extended. The GLSP client is displayed in a webview using Sprotty [1] and receives data such as Actions via the GLSP server. Furthermore, Actions can be sent to the GLSP server via the GLSP VS Code integration. Not all Actions are sent outside the webview, some are only used to transfer data within the GLSP client. In context of this thesis they get called WebviewActions, because there is no certain naming convention in the code for them. The GLSP client gets extended with three main features: visually displaying mouse moves, viewport changes and selection changes of other subclients. This is to counteract the important point of *conflict reduction during the collaboration process*.

The implementation adds a dependency injection ContainerModule called collaborationModule, which can be easily integrated. This contains a Tool, a Provider, Commands, WebviewActions, ModelElements and a View for each of the three main features. A Tool is a service class that can do several things at the same time, e.g. contain a state, register listeners (like a MouseListener) or act as an ActionHandler itself. Providers are also ActionHandlers that listen for certain Actions (mostly from the server or CollaborationActions) and dispatch WebviewActions. A Command defines the behavior of a WebviewAction. In that case it adds or deletes ModelElements of a certain type to or from the graphical model. A View specifies exactly how a ModelElement is rendered at the DOM (Document Object Model).

As each feature is basically structured in the same way, this thesis will introduce one of them in more detail in this section and point out the differences between those features. This section will focus on the mouse move feature.

5.4.1 Tools

First, this subsection takes a look at the created tools, which capture various user actions such as mouse moves, viewport changes or selection changes and ensure that CollaborationActions are dispatched, which are then sent via the CollaborationGlspClient to other subclients and transferred to their webview.

```
@injectable()
export class MouseMoveTool extends BaseGLSPTool implements IActionHandler {
    ...
    protected mouseListener: MouseMoveListener;

    handle(action: Action): void {
        if (SetViewportAction.is(action)) {
            this.lastViewport = action.newViewport;
        }
    }

```

[1] https://github.com/eclipse-sprotty/sprotty

```
12      enable(): void {
13          this.mouseListener = new MouseMoveListener(this);
14          this.mouseTool.register(this.mouseListener);
15      }
16
17      disable(): void {
18          this.mouseTool.deregister(this.mouseListener);
19      }
20  }
21
22  export class MouseMoveListener extends MouseListener {
23      constructor(protected tool: MouseMoveTool) {
24          super();
25      }
26
27      override mouseMove(target: SModelElement, event: MouseEvent): Action[] {
28          const lastViewport = this.tool.lastViewport;
29          const x = lastViewport.scroll.x + (event.pageX / lastViewport.zoom);
30          const y = lastViewport.scroll.y + (event.pageY / lastViewport.zoom);
31
32          return [MouseMoveAction.create({ position: { x, y }})];
33      }
34  }
```

Listing 5.15: Implementation of MouseMoveTool and MouseMoveListener dispatching MouseMoveActions.

The `MouseMoveTool` (Listing 5.15) inherits from `BaseGLSPTool` and implements `IActionHandler`. The implemented `handle(action)` method listens to a `SetViewportAction` and the passed `lastViewport` is stored in an own state variable. In addition, a `MouseMoveListener` is registered in the `enable()` method, which listens for mouse moves and dispatches a `MouseMoveAction` (is a `CollaborationAction`). The absolute position is calculated using the `lastViewport` and the relative position of the `MouseEvent`. By returning the `MouseMoveAction` in line 32, this action is dispatched automatically.

The `ViewportBoundsChangeTool` listens also for `SetViewportAction` to store the `lastViewport`. It also listens for an `InitializeCanvasBoundsAction`, which is dispatched when the window is resized. It contains the `height` and `width` of the webview. These two entries are used to dispatch a `ViewportBoundsChangeAction` (`CollaborationAction`).

The `SelectionChangeTool` implements a `SelectionListener`, which calls the `selectionChanged(root, selectedElements)` when the selections of elements are changed. By calling this the tool dispatches a `SelectionChangeAction` (`CollaborationAction`).

5.4.2 Providers

Providers catch `CollaborationActions` (already sent through VS Live Share to other subclients) and use them to dispatch `WebviewActions`. All providers implement the `IActionHandler` interface and inject a `feedbackActionDispatcher`. A `FeedbackActionDispatcher` ensures that these actions are executed again after an `UpdateModelAction`, which leads to a re-rendering of the graphical model. If no `FeedbackActionDispatcher` would be used, all displayed visual elements would be removed after an `UpdateModelAction`.

```
@injectable()
export class DrawMousePointerProvider implements IActionHandler {
    @inject(TYPES.IFeedbackActionDispatcher)
    protected feedbackActionDispatcher: IFeedbackActionDispatcher;

    protected lastActions: Map<string, DrawMousePointerAction> = new Map();

    protected lastViewport: Viewport = DEFAULT_VIEWPORT;

    handle(action: Action): Action | void {
        if (SetViewportAction.is(action)) {
            this.lastViewport = action.newViewport;
            Array.from(this.lastActions.values()).forEach(a => a.zoom = this.
    lastViewport.zoom);
            this.dispatchFeedback();
        }

        if (MouseMoveAction.is(action) && action.initialSubclientInfo != null
    ) {
            const feedbackAction = DrawMousePointerAction.create({
                position: action.position,
                initialSubclientInfo: action.initialSubclientInfo,
                zoom: this.lastViewport.zoom,
                visible: action.visible
            });
            this.lastActions.set(feedbackAction.initialSubclientInfo.
    subclientId, feedbackAction);
            this.dispatchFeedback();
        }

        if (ToggleCollaborationFeatureAction.is(action) && action.actionKind
    === MouseMoveAction.KIND) {
            Array.from(this.lastActions.values()).forEach(a => a.visible = !a
    .visible);
            this.dispatchFeedback();
        }

        if (DisposeSubclientAction.is(action) && action.initialSubclientId !=
     null) {
            this.lastActions.delete(action.initialSubclientId);
            this.dispatchFeedback();
```

```
                return RemoveMousePointerAction.create({
                    initialSubclientId: action.initialSubclientId
                });
            }
        }
        ...
}
```

Listing 5.16: Implementation of DrawMousePointerProvider dispatching DrawMousePointerActions and RemoveMousePointerActions.

Listing 5.16 shows the `DrawMousePointerProvider`. This injects the `Feedback-ActionDispatcher` in line 2. In addition, there are member variables for a list of `lastActions` per `subclientId` and the `lastViewport`. This class saves the last `DrawMousePointerAction` for each subclient, so it is possible to re-draw each mouse pointer after re-rendering.

The `handle(action)` method listens for `SetViewportActions` so that it can save the current zoom value and, in the case of a zoom change, re-render all mouse pointers with new zoom value. This is necessary so that in the case of a zoom-in, the size of the mouse pointer does not zoom-in as well.

Line 17 listens for `MouseMoveAction` with `initialSubclientInfo` is not `null`. If this `initialSubclientInfo` is not `null`, this means that the `MouseMoveAction` is not dispatched by this GLSP client and has already been sent to other `subclients` via VS Live Share. A `DrawMousePointerAction` is created from this `MouseMove-Action`, where the last zoom value is assigned. In line 24, this new action is added to the `lastActions` and then all `DrawMousePointerActions` are dispatched using the `FeedbackActionDispatcher`.

In line 28, this method listens for a `ToggleCollaborationFeatureAction` and checks for type `MouseMoveAction`, which causes a inversion of the visible flag. Depending on this flag, mouse pointers are displayed or hidden.

`DisposeSubclientActions` are handled in line 33. A not-null-check for `initial-SubclientId` is processed here as well. This actions deletes the `DrawMousePoint-erAction` of the sending subclient from the `lastActions Map` and dispatches a `RemoveMousePointerAction`, which removes the mouse pointer of the subclient.

The `DrawViewportRectProvider` works the same way, except that it dispatches a `DrawViewportRectAction` and a `RemoveViewportRectAction`.

The `SelectionIconProvider` does not have a list of `lastActions` per `subcli-entId`. Instead, it has an additional inner `Map` with `lastActions` per `elementId` per `subclientId`. The reason for this is that a user can select not just one, but several elements. The `SelectionIconProvider` dispatches `DrawSelectionIconAction` and `RemoveSelectionIconAction`.

5.4.3 WebviewActions & Commands

WebviewActions are actions that are only used within the webview or the GLSP client and are not sent to the GLSP server or other subclients. Each WebviewAction has a command that handles its execution. In that case, a command adds or removes `ModelElements` from the graphical model.

```
export interface DrawMousePointerAction extends DrawCollaborationAction {
    kind: typeof DrawMousePointerAction.KIND;
    position: Point;
    zoom: number;
}

@injectable()
export class DrawMousePointerCommand extends FeedbackCommand {
    static readonly KIND = DrawMousePointerAction.KIND;

    constructor(@inject(TYPES.Action) protected action:
    DrawMousePointerAction) {
        super();
    }

    execute(context: CommandExecutionContext): CommandReturn {
        const id = mousePointerId(context.root, this.action.
    initialSubclientInfo.subclientId);

        removeElementFromParent(context.root, id);

        const mousePointerSchema = {
            id,
            type: DefaultTypes.MOUSE_POINTER,
            position: {
                x: this.action.position.x,
                y: this.action.position.y
            },
            color: this.action.initialSubclientInfo.color,
            name: this.action.initialSubclientInfo.name,
            zoom: this.action.zoom,
            visible: this.action.visible
        };

        context.root.add(context.modelFactory.createElement(
    mousePointerSchema));

        return context.root;
    }
}
```

Listing 5.17: Implementation of DrawMousePointerCommand and relating DrawMousePointerAction creating ModelElements.

The DrawMousePointerAction showed in Listing 5.17, which is called WebviewAction within this thesis, has a position and a zoom value. The position specifies the absolute position of the mouse pointer in the webview. The local zoom value is used to ensure that the rendered mouse pointer is not scaled, but is always displayed in the same size. The DrawMousePointerAction inherits from the DrawCollaborationAction. This contains the initialSubclientInfo and a visible flag.

The DrawMousePointerAction is used in the DrawMousePointerCommand to create a MousePointer ModelElement in the execute(context) method and add that to the root element. In line 16, the ID of the element is created using the root element and the subclientId. This is used to remove the old element from the root element. The schema with all attributes such as position, color, name, etc. is then generated. A ModelElement is created from this schema via the modelFactory, which is added to the root element at the end.

A RemoveMousePointerCommand handles a RemoveMousePointerAction, which is dispatched when a subclient is disposed from the client session. This Command removes the element completely from the graphical model.

Also there are the same two commands for the ModelElements ViewportRect and SelectionIcon.

5.4.4 ModelElements & View

ModelElements are elements that are created using the modelFactory and a defined schema. Each of these elements can be assigned to a view using an ID. This view is used to render the ModelElement. The view is declared using JSX [2] and inherits from a dedicated view class, which comes with a number of functionalities.

```
export class MousePointer extends CollaborationElement {
    override type = DefaultTypes.MOUSE_POINTER;
    name: string;
    zoom: number;
}

@injectable()
export class MousePointerView extends ShapeView {
    override render(mousePointer: MousePointer, _context: RenderingContext):
    VNode | undefined {
        if (!mousePointer.visible) {
            return undefined;
        }

        const invertedZoom = 1 / mousePointer.zoom;
        const pointerX = -8.3 / mousePointer.zoom;
        const pointerY = -7.3 / mousePointer.zoom;
```

[2]https://facebook.github.io/jsx/

```
17         const textY = 30 / mousePointer.zoom;
18 
19         const pointerTransform = 'translate(' + pointerX + ', ' + pointerY + ') scale(' + invertedZoom + ')';
20         const textTransform = 'translate(0, ' + textY + ') scale(' + invertedZoom + ')';
21 
22         const graph = (
23             <g>
24                 <g transform={pointerTransform}>
25                     <polygon fill="#FFFFFF" points="8.2,20.9 8.2,4.9 19.8,16.5 13,16.5 12.6,16.6 "/>
26                     <polygon fill="#FFFFFF" points="17.3,21.6 13.7,23.1 9,12 12.7,10.5 "/>
27                     <rect fill={mousePointer.color} x="12.5" y="13.6" transform="matrix(0.9221 -0.3871 0.3871 0.9221 -5.7605 6.5909)" width="2" height="8"/>
28                     <polygon fill={mousePointer.color} points="9.2,7.3 9.2,18.5 12.2,15.6 12.6,15.5 17.4,15.5 "/>
29                 </g>
30                 <text class-mouse-pointer-text={true} style={{ fill: mousePointer.color }} transform={textTransform}>{ mousePointer.name }</text>
31             </g>
32         );
33 
34         return graph;
35     }
36 }
```

Listing 5.18: Implementation of MousePointerView using MousePointer to render for webview.

In Listing 5.18 the `ModelElement` `MousePointer` gets defined. This defines the `type`, which it inherits from the `CollaborationElement` class. In addition, the class inherits the attributes for `color` and the `visible` flag. As its own attributes, the class has a `name`, which is displayed below the mouse pointer, and the local `zoom` value, which ensures that the element and the text are always displayed at the same size, regardless of the current zoom level. The `CollaborationElement` class inherits from the `SShapeElement` class, which contains a `position` and a `size`.

The `MousePointerView` class inherits from the abstract `ShapeView` class. The abstract class `ShapeView` has an `isVisible(model, context)` method, which is used to calculate whether the `ModelElement` is in the current webview. To do this, it is important that the passed `ModelElement` inherits from the `SShapeElement` class so that it has a `position` and a `size`, which is essential for the calculation.

Line 10 checks whether the `ModelElement` is visible; if not, the function is aborted. In the next lines, the position of the mouse pointer and the text is calculated using the zoom value. Starting with line 22 the graph is defined using JSX. A `<rect>` element and a `<polygon>` element are used to render the mouse pointer in combination with the `color`. A `<text>` element is used to display the `name` of the user.

The `SelectionIcon` element is rendered in the `SelectionIconView` class. The `ViewportRectView` class takes care of the element of type `ViewportRect`.

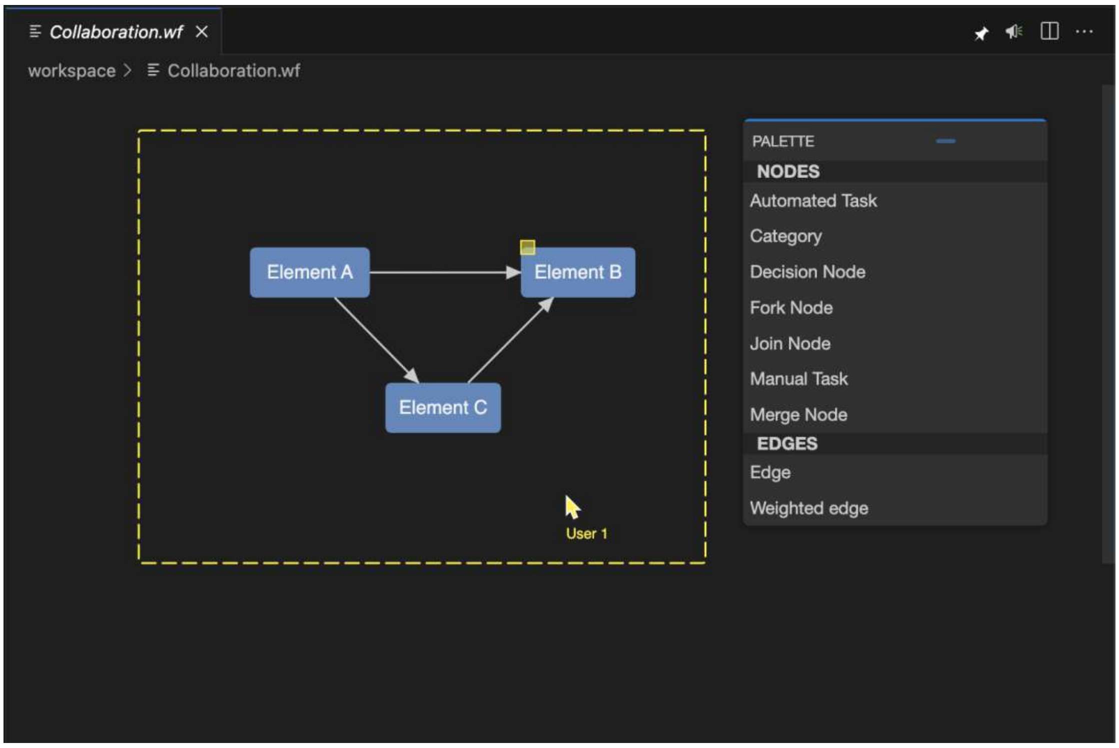


Figure 5.2: Prototype implementation showing mouse pointer, viewport and selection feature for a another user within a collaboration session.

Figure 5.2 displays the Workflow example with the collaboration implementation. This simple example diagram has three elements and three edges. This example shows a collaborative session with a host subclient and a guest subclient. The other participant (User 1) gets the color yellow assigned. In this example, the mouse pointer of User 1 is shown in the middle of the image with its name as a text label. Additionally, the current viewport of User 1 is displayed using a dotted rectangle. User 1 selects element B. This is represented by a small yellow square directly on the element.

CHAPTER 6

Evaluation

Chapter 6 shows the feasibility of using the generic implementation, which was developed using the Workflow example, for arbitrary modeling languages realized with GLSP. The VS Code extension BIGUML [MB] [big] is used for this purpose. BIGUML is an open-source UML modeling tool using the Graphical Language Server Protocol and was leaned on the concept of Eclipse Papyrus [1] [MB23]. It is implemented as a Visual Studio Code extension. That extension works with *Unified Modeling Language* (UML) as a modeling language. Two usability tests on the collaborative feature will be performed. This will be done with a number of participants using BIGUML and the Workflow example.

6.1 Applying collaborative modeling to the BIGUML VS Code extension

First, collaborative modeling will be applied to the BIGUML VS Code extension. Just like the Workflow example, BIGUML was implemented on the basis of GLSP. The tool uses a GLSP server in the background. In contrast to the Workflow example, BIGUML also has a `ModelServer`, which is responsible for loading, saving and validating the model. The `ModelServer` also runs locally on the own computer and accesses the local file system. Since it was not part of this diploma thesis to make it collaborative, and it accesses the local file system, it is not possible to create or delete files on a guest subclient. Therefore the `ModelServer` for guest subclients are completely deactivated. The BIGUML tool also uses the GLSP client and the GLSP VS Code integration to build a fully-fledged VS Code extension. When starting the BIGUML VS Code extension, it automatically starts a GLSP server and a `ModelServer` in the background.

The tool BIGUML supports various UML diagrams such as Activity diagram, Class diagram, Communication diagram, etc. In this section the feasibility of adapting the

[1] `https://projects.eclipse.org/projects/modeling.mdt.papyrus`

collaborative feature to another modeling language will be evaluated. Since BIGUML supports the GLSP protocol, it should be straightforward to extend BIGUML to support real-time collaborative modeling.

6.1.1 Add VS Live Share to the VS Code extension

The first step is to add VS Live Share to the `package.json` of the VS Code extension. In the case of BIGUML, this file has the path `client/packages/uml-vscode-integration/extension/package.json`. It is very important that this is done in the `package.json` of the extension and not in another `package.json` file.

```
 1  {
 2      ...
 3      "dependencies": {
 4          ...
 5          "vsls": "^1.0.4753"
 6      },
 7      "extensionDependencies": [
 8          "ms-vsliveshare.vsliveshare"
 9      ]
10      ...
11  }
```

Listing 6.1: Add VS Live Share to the package.json of the extension.

At Listing 6.1, the npm package `vsls` [2] is added to the `dependencies` in line 5. In addition, `ms-liveshare.vsliveshare` is configured as one of the `extensionDependencies`. This is all to add VS Live Share as a dependency for the VS Code extension.

6.1.2 Grant permission to VS Live Share extension

VS Live Share by default does not allow any other extension to use it. One possibility is to get an entry in the public registry of VS Live Share. Since this is not so easy, the permissions have to be granted manually. A local file with the name `.vs-liveshare-settings.json` is created in the user's home directory, in which the public registry for the local VS Live Share extension can be adapted.

[2] `https://www.npmjs.com/package/vsls`

```json
{
    "extensionPermissions": {
        "BIGModelingTools.*": "*"
    }
}
```

Listing 6.2: Grant local permissions to publisher BIGModelingTools for VS Live Share in .vs-liveshare-settings.json file.

Listing 6.2 shows how this file could look like. For this purpose, an entry with the publisher `BIGModelingTools` of the extension is created under `extensionPermissions`. The value `"*"` specifies that all permissions are granted. The `publisher` of the extension is defined in the `package.json` file of the extension, same file like mentioned in Subsection 6.1.1.

Of course it is always possible to create this file manually. However, a helper function `writeExtensionPermissionsForLiveshare(publisher)` has been developed. If there is no `.vs-liveshare-settings.json` file, this function creates it and adds an entry for the specified `publisher`. An entry is added under the `extensionPermissions` attribute with granting all permissions. When the file is created and the entry is added, a prompt is displayed asking the user to restart VS Code. This is necessary for the changes to take effect.

This function gets called at the start of the VS Code extension, namely when invoking the `activate(context)` method. VS Code calls this method automatically when a VS Code extension is activated.

6.1.3 Configure CollaborationCommands

The prototype has implemented the `CollaborationFeatureStore` to enable the user to switch collaborative features on and off, as shown in Subsection 5.3.4. For this to be enabled, the collaboration commands must be configured when the extension is activated. The helper function `configureCollaborationCommands({context, glspConnector})` was implemented for this purpose. Calling this function when activating the extension configures the commands for VS Code.

```typescript
async function activate(context: vscode.ExtensionContext): Promise<void> {

    writeExtensionPermissionsForLiveshare('BIGModelingTools');

    ...

    const connector = diContainer.get<UVGlspConnector>(TYPES.Connector);

    configureDefaultCommands({
        extensionContext: context,
        connector,
12            diagramPrefix: VSCodeSettings.commands.prefix
13        });
14
15        configureCollaborationCommands({
16            extensionContext: context,
17            connector
18        });
19    }
```

Listing 6.3: Configure collaboration commands in activate(context) method.

Listing 6.3 shows the `activate(context)` method, which is placed in the `extension.ts` file. In line 15, the helper method `configureCollaborationCommands({context, connector})` is called. The function also calls the method `writeExtensionPermissionsForLiveshare('BIGModelingTools')` at the very beginning in line 3.

6.1.4 Configure SocketGlspVscodeServer

There is an implementation of the `SocketGlspVscodeServer` in `BIG`UML. This is realized in the class `UVGlspServer`. To make it collaborative-capable, a `GlspClientProvider` must be passed to the `super constructor` for the `collaboration` attribute. It is possible to develop own providers, but it is also possible to use the already developed `LiveshareGlspClientProvider`, which uses VS Live Share for data exchange. The Listing 6.4 shows how the `SocketGlspVscodeServer` can be configured using the `LiveshareGlspClientProvider`.

```
@injectable()
export class UVGlspServer extends SocketGlspVscodeServer {
    constructor(...) {
        super({
            clientId: 'glsp.uml',
            clientName: 'uml',
            serverPort: glspServerConfig.port,
            collaboration: new LiveshareGlspClientProvider()
        });
    }
    ...
}
```

Listing 6.4: Configure SocketGlspVscodeServer using LiveshareGlspClientProvider.

6.1.5 Set relativeDocumentUri to GLSPClient parameters in GlspVscodeConnector

As the `CollaborationGLSPClient` requires a `relativeDocumentUri` for transmitting of messages to the GLSP server, this has to be set manually. The best way to do that is in the GlspVscodeConnector, as the associated file path of the calling message can be read there. The absolute file path must be converted into a relative file path. The helper function getRelativeDocumentUri(absolutePath) returns the relative path.

Since the GlspVscodeConnector calls the CollaborationGlspClient through the configuration, it requires a defined relativeDocumentUri when calling the functions initializeClientSession(params), disposeClientSession(params), sendActionMessage(message). The parameters of all three of these functions have an args attribute, which can be freely extended. This args attribute should be extended with the relativeDocumentUri property.

```
async registerClient(client: GlspVscodeClient){
    const relativeDocumentUri = getRelativeDocumentUri(client.document.uri.
   path);

    initalizeParams.args = {
        ...initalizeParams.args,
        relativeDocumentUri
    };

    await glspClient.initializeClientSession(initializeParams);
}
```

Listing 6.5: Setting relativeDocumentUri to InitializeClientSessionParameters.

Listing 6.5 shows an example of how the relativeDocumentUri is calculated, then set as a property on the args attribute, and finally these parameters are passed to the initializeClientSession(params) function. The absolute path can be retrieved from the client attribute, which is passed to the registerClient function. In the BIGUML extension, the UVGlspConnector class overwrites the default implementation of the GlspVscodeConnector. This class sets also the relativeDocumentUri.

6.2 Real-time usability

To test the usability of the collaborative feature as part of GLSP, two real-time usability tests with human test participants were carried out. Both tests were run under pretty much the same conditions, but for comparison the tests used firstly the BIGUML extension and secondly the Workflow extension.

6.2.1 Preparation

In order to obtain a meaningful test result, good preparation and planning for the tests is essential. A group of PhD students and BIG employees were invited to perform a stress test on the two tools. To make it easy for the test participants to set up the environment for themselves, a guide explains how to set up the tool so they can test it on their own computer. This required the installation of VS Code, Java 11+, VS Live Share and the specific VS Code extension. A new version of the VS Code extension with the collaborative feature was built. Afterwards the resulting `.vsix` file was uploaded to a public server from which it could be downloaded. The test participants were able to install this `.vsiv` file on their VS Code version and try it out in advance.

The tests should show if and how it is possible for a different number of simultaneously participating users to work collaboratively on the same diagram. The tests should also show how the behavior changes when the number of elements in the diagram changes. To make this as uncomplicated as possible, a small, medium and large diagram file was created in beforehand. The small one had 25, the medium one 100 and the large one 500 elements.

It was also necessary to think about interview questions in advance, which were sent to the test participants by email. Afterwards they were asked to send back textual feedback on the questions by email so that it is possible to incorporate this into the work and further iterations of this tools. The questions are:

- How did the BIGUML/Workflow extension perform for you in collaborative mode for the small, medium and large diagram?
- Would you do collaborative modeling with other people in the future?
- Do you have any suggestions for the future how to improve the collaboration (e.g. so that it is possible to have a conflict-free model synchronization)?

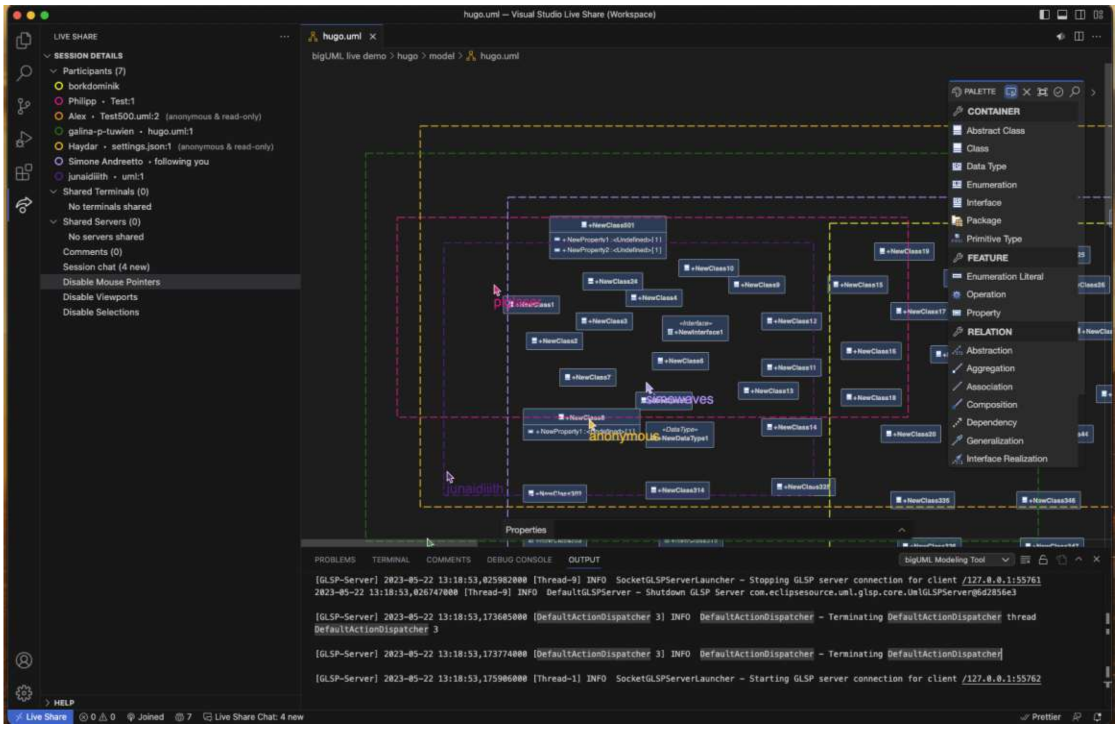
To help the test participants familiarize themselves with the tools, a short presentation about the work and this tools was held during the first test. The first test was planned to take two hours, the second one one hour. The next subsections presents how the tests went and what the outcome of the tests was.

6.2.2 First test with BIGUML VS Code extension

In the first usability test, the test participants used BIGUML as a tool. The test started with only two participants and the small diagram. The author hosted the session at the beginning and shared the invitation link with the other participants via email. Joining of a guest to a session worked seamlessly. The first test with only two participants and a small diagram also worked well. There was also no long delay time between action and feedback. Mouse pointer and viewport changes, as well as the selection of elements, were transferred smoothly to the screens of other participants and displayed there.

After some time, more participants joined the session. Also the medium and large diagrams were used. It became clear that the more participants joined the session, the longer the waiting time between action and feedback became. However, it turned out that this has less to do with the number of simultaneous participants in a session, but more with the number of actions. If there were many participants in a session, but only one participant performed actions, the performance was increased again.

Another point that has also been shown is that actions performed on the host subclient were displayed faster than on the guest subclient. The reason for this is that it has direct access to the GLSP server located on its own computer and therefore does not have to take the detour via VS Live Share to the host subclient, from there to the GLSP server, and then back to itself again through the host subclient. Since all guest subclients have a peer-to-peer connection with the host subclient, it is a great advantage if the host subclient has a good internet connection. Due to the countryside location, the author's internet connection was not ideal for this. For this reason, a new live share session was started with a host located in Vienna as part of the test. Since the new host had a better internet connection, it quickly became obvious that this also made a difference to the performance when working with many participants.



```
[GLSP-Server] 2023-05-22 13:18:53,025982000 [Thread-9] INFO  SocketGLSPServerLauncher - Stopping GLSP server connection for client /127.0.0.1:55761
2023-05-22 13:18:53,026747000 [Thread-9] INFO  DefaultGLSPServer - Shutdown GLSP Server com.eclipsesource.uml.glsp.core.UmlGLSPServer@6d2856e3

[GLSP-Server] 2023-05-22 13:18:53,173605000 [DefaultActionDispatcher 3] INFO  DefaultActionDispatcher - Terminating DefaultActionDispatcher thread
DefaultActionDispatcher 3

[GLSP-Server] 2023-05-22 13:18:53,173774000 [DefaultActionDispatcher 3] INFO  DefaultActionDispatcher - Terminating DefaultActionDispatcher

[GLSP-Server] 2023-05-22 13:18:53,175906000 [Thread-1] INFO  SocketGLSPServerLauncher - Starting GLSP server connection for client /127.0.0.1:55762
```

Figure 6.1: Collaboration session with eight participants working with BIGUML tool.

With a large number of actions by many participants, there were also "bouncing" elements. For example, this means that a participant moves an element, but this jumps back to the original point and then to the desired point again. This happens when an action B of another user is sent before the user's own action A, but the feedback from action B is not received until after the sending of action A. Figure 6.1 shows how such a collaboration session with eight participants would look like. Every user gets an own color assigned, which are then used to show visual elements like mouse pointers, viewports and selection markers to other users.

It also happened that some actions were lost entirely. The reason for this could be that the ModelServer cannot cope with concurrent requests. It has also been shown that the ModelServer throws error messages and timeouts, which of course leads to actions not being completed and therefore they get lost.

What has also been shown, is that the size of the diagram says much less about the performance than the number of actions performed and the number of simultaneous participants. The performance and usability of the tool did not really degrade while collaborating on the large diagram with a small number of participants.

6.2.3 Second test with Workflow VS Code extension

As the ModelServer did lead to some problems with performance and consistency, a second usability test with the Workflow tool should bring more insight into the problems, as this does not have a ModelServer in the background. The second test was planned to be shorter, but had the same structure as the first one. Also in this test, three diagram files were created in advance, ranging from small to medium to large (25 - 100 - 500 elements).

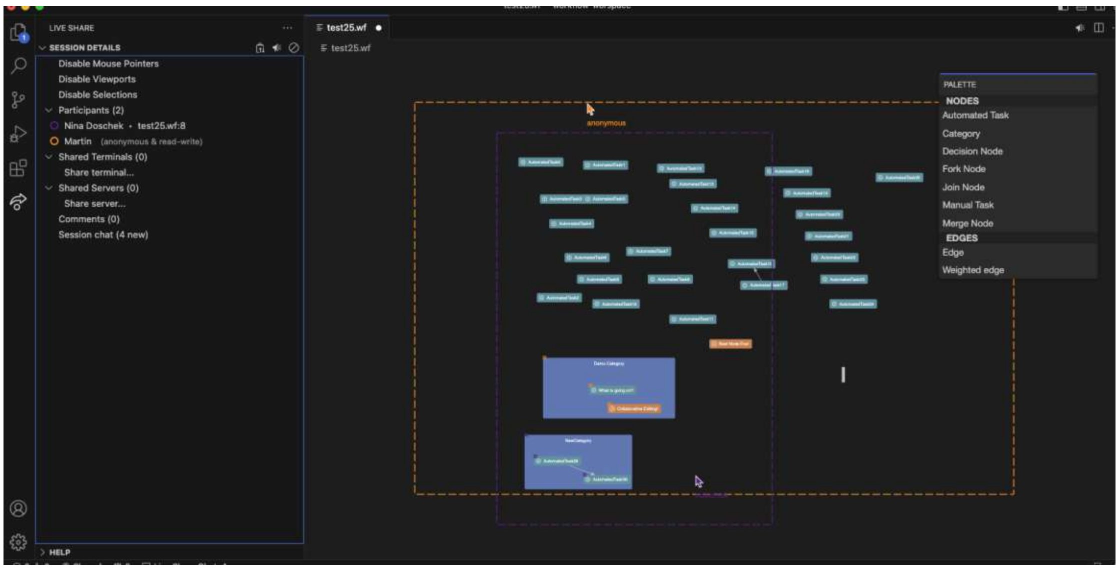


Figure 6.2: Collaboration session with three participants working with Workflow tool.

The test started again with only two participants and the small diagram with only 25 elements. As with the first test, this worked very well without any delay times. It also worked perfectly with three participants as shown in Figure 6.2. This time the test switched to the medium and larger diagram before more participants were added. Here, it was also noticed that the number of elements did not have a major impact on performance and usability. This shows that collaborative working with GLSP operates well with a manageable number of participants.

At the end of the session, several participants joined. This showed that also with the Workflow tool, a large number of participants or concurrent actions had some negative impact on performance and usability. However, since there was no `ModelServer` in the background, it also became clear that this did not have as much of an impact as with the BIGUML tool. It also didn't happen that some actions were not completed at all. The Workflow tool also made it possible to create and delete diagram files for guest subclients.

6.2.4 Conclusion & Feedback

The feedback from the test participants, sent by email, is very positive. The feedback reveals that collaborative working with a small number of participants worked very well and without much time latency. However, this is for the reason that more concurrent actions are sent with more participants. What has also been shown is that the number of elements does not affect usability and performance as much as the number of actions.

Participants who took part in both tests reported that they were more pleased with the usability of the Workflow tool when collaborating with several participants. The reason for this is that BIGUML has a `ModelServer` in the background which cannot handle concurrent requests, as the `ModelServer` has not yet been adapted for collaborative requests. The `ModelServer` also caused some actions to be lost completely, as it causes timeouts and error messages if there are too many concurrent actions.

Some participants reported differences with low internet connections. Swapping the host subclient over to a participant with a better connection resulted in better performance over the entire collaboration session.

When the test participants were asked whether they could imagine collaborative modeling in the future or not, the feedback was primarily positive. Most of them can imagine using GLSP modeling in a collaborative session. One participant came up with idea to rather use it in a reviewing session, where they only have to revise a few things and they want to show these changes to other participants or discuss them with them.

The last question, namely whether there were any suggestions for improvement, also received a number of responses. One suggestion was to show all participants who have just opened the same file. At the moment, only users who are within the collaborative session are displayed. The solution could display an info window with all active users within that file placed inside the viewport. Another suggestion was to implement incremental updates of the graph. This should speed up the execution time, as the queue of pending actions is not as large and the delay would be less noticeable. Not updating the model until an own action has been executed would also mean that the elements would not bounce so much, as described in the first usability test. A final suggestion was to integrate some progress reporting. For example, a spinner could be displayed to show the user that their action is currently being processed.

To summarize, the tests were a success and the collaborative feature with GLSP, both with BIGUML and the Workflow tool, was very well endorsed by the test participants. Collaborative sessions with a small number of participants can also be used in productive sessions. Suggestions from test participants can be incorporated into later iterations of the feature to further improve usability and performance during a collaborative session.

CHAPTER 7

Conclusion

This chapter will list the answers to the research questions formulated in Section 1.2. Finally, a look into the future will present and discuss possible improvements and extensions to this solution.

7.1 Summary

This diploma thesis studied how GLSP can be extended with a collaborative functionality. In particular, this work focused on accomplishing this in the context of Visual Studio Code and its VS Live Share extension. First of all, it was necessary to gather background information on collaborative editing, GLSP, Visual Studio Code and Visual Studio Code Live Share. Afterwards, existing collaborative editors were analyzed. In this step, textual and graphical document editors were examined for certain characteristics. The thesis used the obtained information to set up and define requirements for the prototype. In the next step, the concept for this prototype was declared in an abstract form. The prototype was developed in the main part of the work. The first part of the prototype consisted of turning GLSP into a collaborative platform. The second step consisted of presenting performed actions by a user to other participants of the collaborative session as self-explanatory as possible. In the last step of the work, the solution was evaluated in two ways. The first one was to apply the solution to a different modeling language. The thesis focused specifically on the Workflow example in the prototype. The evaluation phase applied the solution to the BIGUML tool. In the second step of the evaluation phase, the solution was validated in two usability tests with actual test participants. For this purpose, the BIGUML as well as the Workflow tool were subjected to a usability and stress test.

The following presents the answers to the research questions:

- **RQ1: What is the best possibility of the already existing implementation of the Eclipse GLSP framework to equip it with collaborative functionalities?**
  - The research investigated two possible ways how to solve this problem. The first option was a single GLSP server architecture. Here, there is only one GLSP server per collaborative session, which also only communicates with the host subclient. Guest subclients have to send any messages to the GLSP server via the host subclient, which acts as a proxy. To send messages between the host and the guest subclients, a provider such as Visual Studio Live Share is used, which exchanges messages between subclients via peer-to-peer connections. This provides an advantage that there is only a single `ModelState`. The second option is a multi GLSP server architecture, where each subclient has its own GLSP server and `ModelState`. Again the messages between subclients would be sent via a provider, such as VS Live Share.

    The prototype has gone with a single GLSP server architecture because in this case it needs only one `ModelState` and therefore it does not have to worry about synchronizing several of them. In addition, the advantage is that only the host has to take care of loading and saving the `ModelState` from and to the source of the model.

    VS Live Share handles the communication of all messages between subclients. The extension has been implemented in such a way that it is very easy to exchange this provider. This may be useful, if the solution wants to support other GLSP implementations than the one used for Visual Studio Code.
- **RQ2: What is an appropriate means to achieve a reliable and conflict-free synchronization of performed actions?**
  - This question is actually fulfilled by the answer to **RQ1**. Since the prototype uses a single GLSP server architecture and consequently only one `Model-State`, no synchronization of the performed actions is necessary. Since each user also has their own `CommandStack`, undo/redo operations are performed on the global `ModelState`, as long as they are valid for the current state.
- **RQ3: Which representation of the performed actions shows them to the other users in the best possible and self-explanatory way?**
  - The second part of the solution dealt with the question of how it is possible to present the performed actions of a user to other users in such a way that all participants in the collaboration session understand what is happening during that session and therefore no actions are overlooked. To realize this, the solution shows permanently where exactly somebody is working on the diagram.

    The prototype uses three new action types, which were introduced to be only sent between the subclients. The actions display a mouse pointer, a viewport and selections of an initiator in an assigned color to all other users. In this way, all users know who is currently working on which objects and at which position of the diagram. These measures also lead to conflict reduction during the collaboration process.

7.2 Future Work

To conclude the work, this chapter discusses possible extensions for the future and present them briefly. Some of these extensions have arisen in the process of developing the current solution, other extensions have resulted out of the feedback given during the usability tests.

One of the most important tasks for the future is to register the publisher of the VS Code extension in the VS Live Share registry. This is necessary for VS Live Share to accept the extension and grant permissions to it. In the current solution, when the VS Code extension is started, a special function is called which creates a local file which then grants the permissions for the local VS Code instance.

Since the solution was developed in such a way that it is not dependent on VS Live Share, it is very easy to replace the collaboration provider and use another framework or library that takes care of the data exchange between host and guest subclients. It would also be possible to implement a custom solution for this.

This thesis has its focus on the GLSP server and the GLSP client. Extending the `ModelServer` with collaborative functionality was not part of this thesis. As it was displayed in the tests, this leads to timeouts and errors if there are too many concurrent actions, which means that actions are lost and consequently not executed. With the current implementation, it is also not possible for guest subclients to create and delete files within a collaborative session.

Incremental updates to the graph would make the entire execution faster. This would mean that the queue of pending actions would not grow as quickly and the delay would therefore be less noticeable. A progress bar that indicates the waiting time of pending actions would support users to enjoy better usability.

Due to VS Live Share it is possible to identify active users in the collaborative session. VS Code provides an overview of all participants and the associated color. A further improvement would be to display all active users in the current file. For instance, it would be possible to display an info box with all users and their color in the current viewport.

Finally, it is of course possible to introduce further `CollaborationActions` which would lead to more usability. For example, it would be possible to add a "Follow my cursor" functionality, allowing users to automatically follow the cursor of another user.

List of Figures

List of Tables

List of Listings

Bibliography

[ARHR04] Jinsoo Park Alan R. Hevner, Salvatore T. March and Sudha Ram. Design science in information systems research. *Management Information Systems Quarterly*, 28:75–105, 03 2004.

[big] biguml github. `https://github.com/borkdominik/bigUML`. Accessed: 2023-11-17.

[BL23] Dominik Bork and Philip Langer. Language server protocol - an introduction to the protocol, its use, and adoption for web modeling tools. *Enterprise Modelling and Information Systems Architectures - International Journal of Conceptual Modeling*, 18(9):1–16, 2023.

[BLO] Dominik Bork, Philip Langer, and Tobias Ortmayr. A vision for flexible glsp-based web modeling tools. In *16th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modelling (PoEM'2023)*. Springer International Publishing.

[Bün19] Hendrik Bünder. Decoupling language and editor - the impact of the language server protocol on textual domain-specific languages. In *International Conference on Model-Driven Engineering and Software Development*, 2019.

[cod] Jetbrains code with me. `https://www.jetbrains.com/code-with-me/`. Accessed: 2023-09-11.

[dra] Google drawings. `https://support.google.com/docs/answer/179740?visit_id=638302244312377836-2388093958&hl=en&rd=2`. Accessed: 2023-09-13.

[EG89] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, page 399–407, New York, NY, USA, 1989. Association for Computing Machinery.

[EGR91] Clarence A. Ellis, Simon J. Gibbs, and Gail Rein. Groupware: Some issues and experiences. *Commun. ACM*, 34(1):39–58, jan 1991.

[etha] Etherpad. https://etherpad.org/#. Accessed: 2023-09-11.

[ethb] Etherpad lite github. https://github.com/ether/etherpad-lite. Accessed: 2023-09-11.

[FS09] Ulrich Frank and Stefan Strecker. Beyond erp systems: An outline of self-referential enterprise systems. requirements, conceptual foundation and design options. ICB-Research Report 31, Essen, 2009.

[GG96] Carl Gutwin and Saul Greenberg. Workspace awareness for groupware. pages 208–209, 01 1996.

[gls] Eclipse - graphical language server protocol. https://www.eclipse.org/glsp/. Accessed: 2023-05-12.

[goo] Google docs editors. https://www.google.com/intl/de_at/docs/about/. Accessed: 2023-09-11.

[graa] Collaborative editing with graphity - the diagram editor - youtube. https://www.youtube.com/watch?v=2NFSs0Gpu2Y&ab_channel=yWorks. Accessed: 2023-09-12.

[grab] Graphity. https://www.graphity.com/. Accessed: 2023-09-12.

[jso] Json-rpc specifications. https://web.archive.org/web/20080517011921/http://json-rpc.org/wiki/specification. Accessed: 2023-05-22.

[Kel17] Steven Kelly. Collaborative modelling with version control. In *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages 20–29. Springer, 2017.

[KLR96] Steven Kelly, Kalle Lyytinen, and Matti Rossi. Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In *Advanced Information Systems Engineering: 8th International Conference, CAiSE'96 Heraklion, Crete, Greece, May 20–24, 1996 Proceedings 8*, pages 1–21. Springer, 1996.

[KT21] Steven Kelly and Juha-Pekka Tolvanen. Collaborative modelling and metamodelling with metaedit+. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 27–34. IEEE, 2021.

[lsp] Microsoft - language server protocol. https://microsoft.github.io/language-server-protocol/. Accessed: 2023-05-12.

[MB] Haydar Metin and Dominik Bork. Introducing bigUML: A flexible open-source glsp-based web modeling tool for uml. In *Companion Proceedings of the 26th International Conference on Model Driven Engineering Languages and Systems, MODELS 2023*. IEEE.

[MB23] Haydar Metin and Dominik Bork. On developing and operating glsp-based web modeling tools: Lessons learned from bigUML. In *Proceedings of the 26th International Conference on Model Driven Engineering Languages and Systems, MODELS 2023*. IEEE, 2023.

[MBWM23] Judith Michael, Dominik Bork, Manuel Wimmer, and Heinrich C. Mayr. Quo vadis modeling? *Software and Systems Modeling*, Oct 2023.

[met] Metaedit+. `https://www.metacase.com/mep/`. Accessed: 2023-09-12.

[PA18] Parsa Pourali and Joanne M. Atlee. An empirical investigation to understand the difficulties and challenges of software modellers when using modelling tools. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, page 224–234, New York, NY, USA, 2018. Association for Computing Machinery.

[Rit10] Peter Rittgen. Collaborative modeling: Roles, activities and team organization. *Int. J. Inf. Syst. Model. Des.*, 1(3):1–19, jul 2010.

[SE98] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, page 59–68, New York, NY, USA, 1998. Association for Computing Machinery.

[SFB$^+$14] Stefan Strecker, Peter Fettke, Jan vom Brocke, Jörg Becker, and Elmar Sinz. The research field "modeling business information systems". *Business & Information Systems Engineering*, 6, 02 2014.

[SM21] Rijul Saini and Gunter Mussbacher. Towards conflict-free collaborative modelling using vs code extensions. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 35–44. IEEE, 2021.

[spr] Sprotty architercture. `https://github.com/eclipse-sprotty/sprotty/wiki/Architectural-Overview`. Accessed: 2023-05-24.

[TMGS97] S.G. Tammaro, J.N. Mosier, N.C. Goodwinn, and G. Spitz. Collaborative writing is hard to support: A field study of collaborative writing. pages 19–51, 03 1997.

[TRA+12] Bill Tomlinson, Joel Ross, Paul Andre, Eric Baumer, Donald Patterson, Joseph Corneli, Martin Mahaux, Syavash Nobarany, Marco Lazzari, Birgit Penzenstadler, Andrew Torrance, David Callele, Gary Olson, Six Silberman, Marcus Stünder, Fabio Romancini Palamedi, Albert Ali Salah, Eric Morrill, Xavier Franch, Florian Floyd Mueller, Joseph 'Jofish' Kaye, Rebecca W. Black, Marisa L. Cohn, Patrick C. Shih, Johanna Brewer, Nitesh Goyal, Pirjo Näkki, Jeff Huang, Nilufar Baghaei, and Craig Saper. Massively distributed authorship of academic papers. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '12, page 11–20, New York, NY, USA, 2012. Association for Computing Machinery.

[visa] Visual studio live share. `https://learn.microsoft.com/en-us/visualstudio/liveshare/`. Accessed: 2023-05-27.

[visb] Vs live share extension api. `https://www.npmjs.com/package/vsls`. Accessed: 2023-05-28.

[vsca] Visual studio code - custom editors api. `https://code.visualstudio.com/api/extension-guides/custom-editors`. Accessed: 2023-05-19.

[vscb] Visual studio code - docs. `https://code.visualstudio.com/docs`. Accessed: 2023-05-19.

[vscc] Visual studio code - extension api. `https://code.visualstudio.com/api`. Accessed: 2023-05-19.

[vscd] Visual studio code - webview api. `https://code.visualstudio.com/api/extension-guides/webview`. Accessed: 2023-05-19.

[vsce] Vs code teletype. `https://github.com/sainirijul/vscode-teletype`. Accessed: 2023-09-13.