TU WIEN Informatics

Testing of GLSP-based Web Modeling Tools

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

eingereicht von

Haydar Metin, BSc.
Matrikelnummer 11776852

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, 21. August 2023

Haydar Metin   Dominik Bork

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Bibliothek
Your knowledge hub
TU WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Informatics

Testing of GLSP-based Web Modeling Tools

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Haydar Metin, BSc.
Registration Number 11776852

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, 21st August, 2023

Haydar Metin   Dominik Bork

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Haydar Metin, 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, 21. August 2023

Haydar Metin

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Testen spielt eine wichtige Rolle in der Qualitätssicherung von Softwareanwendungen. Es dient dazu, potenzielle Probleme aufzudecken, die durch Codeänderungen eingeführt werden könnten. Da moderne Softwaresysteme zunehmend komplexer werden, garantieren manuelle Testpraktiken oft keine umfassende Abdeckung mehr. Daher hat sich automatisiertes Testen als Lösung entwickelt, um eine gründliche Evaluierung zu gewährleisten. Diese Arbeit widmet sich dem Bereich des Testens von Diagrammeditoren und konzentriert sich insbesondere auf die Graphical Language Server Platform (GLSP), die das Language Server Protocol verwendet und Entwicklern ermöglicht, Webmodellierungswerkzeuge zu erstellen.

In diesem Zusammenhang untersucht diese Arbeit die Herausforderungen des Testens von Diagrammeditoren. Die Untersuchung geht auf die Mechanik der Browserautomatisierung ein und bewertet die Stärken und Einschränkungen moderner Web-Testframeworks wie Selenium, Cypress und Playwright. Diese Bewertung legt den Grundstein für die Entwicklung eines erweiterbaren und wartbaren Testframeworks, das auf GLSP zugeschnitten ist. Im Rahmen dessen behandelt die Arbeit wesentliche Fragen, wie Diagramme getestet und Interaktionen automatisiert werden können. Aus diesem Grund wird die Entwicklung einer Lösung erforscht, die die speziellen Anforderungen solcher Werkzeuge effektiv bewältigt. Dies umfasst verschiedene Faktoren, einschließlich Architekturelle und Design Prinzipien, um das Fundament für ein effizientes und anpassungsfähiges Testframework zu bilden.

Abschließend wird eine umfassende Testsuite erstellt. Diese Suite umfasst eine Reihe von Szenarien, von denen jedes verschiedene Aspekte von GLSP-basierten Diagrammeditoren abdeckt, um die Fähigkeiten des Testframeworks bei der Bewältigung dieser Herausforderungen zu bewerten.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Testing plays an important role in the quality assurance of software applications. It serves as a means to uncover potential issues introduced by code changes. As modern software systems become increasingly complex, manual testing practices often do not guarantee comprehensive coverage. Hence, automated testing has emerged as a solution to ensure thorough evaluation. This thesis delves into the domain of testing diagram editors, particularly focusing on the Graphical Language Server Platform (GLSP), which utilizes the Language Server Protocol, enabling developers to construct web modeling tools.

In accordance, this thesis investigates the challenges of testing diagram editors. The investigation delves into the mechanics of browser automation and assesses the strengths and limitations of modern web testing frameworks such as Selenium, Cypress, and Playwright. This evaluation lays the groundwork for developing an extensible and maintainable testing framework tailored to GLSP. Within this context, the thesis addresses essential questions concerning how diagrams can be tested and interactions automated. For this reason, developing a solution that effectively addresses the unique demands of such tools is explored and encompasses various factors, including architectural and design principles, to form a foundation of an efficient and adaptable testing framework.

To conclude, a comprehensive test suite is constructed. This suite spans a range of scenarios, each with different aspects of GLSP-based diagram editors to assess the testing framework's capabilities in addressing those challenges.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**
1.1 Problem & Motivation Statement 1
1.2 Aim of the Thesis & Expected Results 4
1.3 Methodology 6
1.4 Structure 6

2 Background **9**
2.1 Software Testing 9
2.2 GLSP 16
2.3 Discussion 22

3 Browser Automation **27**
3.1 Automation Protocols 27
3.2 Web Testing Frameworks 32
3.3 Evaluation 35
3.4 Discussion 46

4 GLSP-Playwright: Automation & Testing Framework **49**
4.1 Test Cases 50
4.2 Challenges 57
4.3 Design Principles 59
4.4 GLSP-Playwright 65
4.5 Architecture 67
4.6 Examples 91
4.7 Technical Outlook 97

5 Evaluation **101**
5.1 Preparations 101

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.2 Test Cases . . . 103
5.3 Report . . . 105
5.4 Discussion . . . 110

6 Future Work **113**

7 Conclusion **115**

List of Figures **119**

List of Tables **121**

List of Listings **123**

Bibliography **125**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

This chapter describes the problem this thesis will deal with and its motivation. Moreover, it will also elaborate on the benefits of the research to developers. It will also define the concrete research questions and the methodological approach. Finally, an overview with short descriptions concerning the remaining parts of the thesis will be given.

1.1 Problem & Motivation Statement

Humans are prone to making mistakes at different points. For the software development lifecycle, those errors can arise in various stages with varying degrees of impact, from minor inconveniences to severe consequences. For instance, a bug in the price calculation of a software system led to invalid prices on Amazon's website in 2014, with products being mistakenly priced at 1 pound each[1]. Accordingly, such miscalculations can have severe implications for businesses and be, in the worst case, fatal. On the other hand, in critical domains like medicine, software errors can be life-threatening, such as the well-known incident that occurred with the Therac-25 radiation therapy machine between 1985 and 1987. Here, due to race conditions in the software, patients were exposed to excessive radiation doses, leading to fatalities and serious injuries [LT93]. Fortunately, over the years, significant progress has been made in **testing and ensuring software quality** by establishing standards, regulations, and certifications. These efforts have contributed significantly to improving the reliability and safety of software systems.

Today, diagrams and models play a vital role in diverse organizations, workflows, and systems, providing visual representations of plans, structures, designs, concepts, and more. These visualizations enhance communication among individuals involved in various tasks and can be created by web-based diagram editors. However, if faulty software is used to create standardized diagrams, the common understanding could be lost, which

[1] Known as the Amazon - 1p glitch

leads to ambiguous results. Moreover, modeling tools serve purposes beyond visualization. For instance, tools like bigER [GB21] can automatically generate complex SQL code based on the model, applicable to various databases and applications.

Now this raises the question of what testing the software really means and how that can be accomplished. The IEEE definition defines software testing as follows:

> Software testing is a **formal process** carried out by a specialized testing team in which a software unit, several integrated software units or an entire software package are examined by **running the programs** on a computer. All the associated tests are performed according to **approved test procedures** on **approved test cases**
> - IEEE 610.12-90[2].

Accordingly, software testing is the process of assessing the quality of a software product, and it aims to enhance the overall quality by detecting issues and comparing the actual behavior of the software with its expected behavior, by leveraging a predefined set of test cases. In essence, software testing helps identify discrepancies between the intended functionality and the actual outcomes.

Based on the previous definition, this thesis focuses on testing web-based modeling tools built using the Graphical Language Server Platform (GLSP) [BLO23], which follows the Language Server Protocol (LSP) [Micb, Mica] and is specifically designed for graphical languages. The idea of extending LSP for graphical languages was previously discussed in 2018 in the paper [REIWC18], and today, it is possible. With GLSP, developers can construct custom diagram editors, like UML editors [MB23a, MB23b], using web technologies, and deploy them in various environments such as Theia[3], VS Code[4], or web pages. Notably, there are already existing examples of graphical (i.e., diagram) editors using the LSP approach, such as the bigER tool [GB21] based on the Langium[5] server. Besides, ongoing research and development surround GLSP, as evident in works like the paper on semantic zooming [CLB22] and disability-aware conceptual modeling [SMB23]. As more modeling tools are moving to the web, reliable testing mechanisms are becoming increasingly crucial. This thesis addresses this need by providing a comprehensive open-source testing framework for GLSP-based web modeling tools. The framework is already accessible on GitHub[6].

It needs to be highlighted that software testing, in general, and explicitly testing web applications, is a well-known area of research and development in academia and industry. Several testing tools and frameworks exist to write test cases and test web applications

[2] IEEE Std 610.12-1990 (R2002), IEEE Standard Glossary of Software Engineering Terminology IEEE, 1990. A more recent definition will be provided later.

[3] `https://theia-ide.org/`, Accessed: 19.08.2023

[4] `https://code.visualstudio.com/`, Accessed: 19.08.2023

[5] `https://langium.org/`, Accessed: 06.08.2023

[6] `https://github.com/eclipse-glsp/glsp-playwright`, Accessed: 19.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

running in browsers or browser-like environments, such as Selenium[7], Cypress[8], and Playwright[9]. At the core, these tools mainly offer browser automation capabilities. That means testers can write code to control the browser programmatically instead of doing it manually to compare the state of the web application for correctness. This approach is called end-to-end (E2E) testing of web applications, as the whole system is running similarly to the production environment and is tested from the user perspective through scenarios and user interactions. Available tools already allow fundamental interactions like button clicks and handling user input and more in the browser to simulate user interactions. They also offer the capability to check if the web application is correctly rendered by using image snapshot comparisons. Although those image comparisons tend to be unreliable. Even minor changes in diagram rendering, such as label positions, can cause the comparison to fail.

As visible, testers already have the necessary tools to automate the browser and to check the correctness. However, they are not powerful enough for graphical editors working with diagrams and models. They are general purpose tools and fall short because they are mainly designed to work with typical web applications, which is not the case for graphical editors. For this reason, testers often need to implement the testing logic themselves on top of the used testing framework to work with interactable diagrams and editors, which can be time-consuming and error-prone. Furthermore, diagrams and models and their respective graphical elements (i.e., diagram elements) can vary significantly in their structures and shapes as they are rendered as Scalable Vector Graphics (SVG)[10] on the web page. For instance, UML diagrams alone have 14 different models, each consisting of multiple diagram elements with varying interaction possibilities. As a result, each diagram becomes unique and requires specific testing approaches, making it challenging to achieve a standardized testing approach using existing tools. Also, these tools do not provide native support for using model semantics. That means accessing the diagram's specific edges or children of nodes is bothersome. Overall, the usual testing frameworks are primarily designed for general browser interactions and do not cater to the specific requirements of testing diagrams and graphical editors. This thesis aims to address these limitations by providing a comprehensive testing framework that caters specifically to GLSP-based web modeling tools that ease the effort required to test user interactions with the editor and the diagram within.

As previously mentioned, errors can have a wide range of different impacts depending on the context. They can be trivial or fatal. For GLSP, it depends on the use case of the custom editor. Errors could be without profound impact on drawing custom diagrams (depending on the diagram). On the other hand, errors in the editor could have catastrophic effects on mission-critical systems. Consequently, proper testing can prevent errors that might otherwise lead to serious consequences in real-world applications.

[7] `https://www.selenium.dev/`, Accessed: 19.08.2023
[8] `https://www.cypress.io/`, Accessed: 19.08.2023
[9] `https://www.playwright.dev/`, Accessed: 19.08.2023
[10] `https://developer.mozilla.org/en-US/docs/Web/SVG`, Accessed: 19.08.2023

TU Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

1.2 Aim of the Thesis & Expected Results

The primary objective of this thesis is to investigate and address the challenges and constraints associated with testing diagram editors (i.e., in particular, GLSP-based web modeling tools). A portion of the research will be devoted to exploring the limitations of existing tools for automating browsers and understanding their practical restrictions. Based on the insights gained from this analysis, a new testing framework will be developed specifically for GLSP-based diagram editors, known as GLSP-Playwright.

To achieve the desired goals, the following key objectives will be pursued in the development of GLSP-Playwright:

1. **Editor-Abstraction**: The framework needs to abstract, streamline, and automate common interactions with the editor. By providing this abstraction layer, testers can efficiently interact with the editor without dealing with complex implementation details. The framework will offer a set of methods and utilities to handle common actions, making it easier for testers to create test scenarios and verify the expected behavior of the diagram editor.

2. **Diagram-Abstraction**: One challenge of testing lies in the unique nature of each diagram, as they offer specific interaction options, shapes, and semantics. While existing testing frameworks can handle interactions with the editor's general elements like buttons and forms (i.e., previous key aspect), they often lack support for diagram-specific actions like resizing, connecting elements, and accessing elements, as those depend on the underlying implementation of the tested application. Moreover, typical testing frameworks rely on element identifiers (ID) to locate the element on the web page and then to simulate interactions. Usually, those IDs are static and stay the same. However, interacting with diagrams can create or remove diagram elements, and the respective IDs are not easily understandable because they are UUIDs[11], e.g., `"ca4142fb-4916-4c93-90d8-7909dc147690"`. As a result, the testing framework needs to intelligently utilize semantic knowledge from the diagram, such as element names, types, or other information, to accurately identify and interact with the desired elements. Summarized, interacting with the diagram elements more semantically (e.g., by using labels) can improve the readability of the test scenarios. To address these complexities, the testing framework should provide dedicated interaction capabilities with diagrams and allow customization to suit different scenarios. At the same time, solving this challenge is also the main goal of this thesis.

3. **Integrations**: GLSP offers various integrations for tool platforms, such as Theia and VS Code. Therefore, running and interacting with these diverse environments is essential. Traditional frameworks typically focus on differentiating between browsers, like Google Chrome and Mozilla Firefox. However, GLSP operates in a

[11] `https://developer.mozilla.org/en-US/docs/Glossary/UUID`, Accessed: 12.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

more diverse set of environments, including web pages and Electron[12] applications. Consequently, each environment has its specific requirements for execution, which makes it necessary to be tailored and handled within the testing framework.

4. **Integration-Parallelization**: Based on the previous objective, it needs to be possible to execute the same test case on different environments without much effort and to run all test cases in parallel to reduce the time required to run the whole test suite.

Concretely, the following research questions will drive the research and development of the new testing framework and need to be answered:

- **RQ1: What do most GLSP-based diagrams and editors have in common regarding possible user interaction possibilities, respective user interfaces that support those interactions, and tool platforms?**
  - The answer determines what abstractions and functionalities the framework must provide the developers to enable working with GLSP-based editors.
- **RQ2: How can the new testing framework be implemented by respecting extendability and maintainability so that different GLSP-based diagrams and editors can be tested?**
  - The testing framework should be a general-purpose framework that should provide the necessary methods such that the developers can efficiently implement their test cases for their custom diagram editors.
- **RQ3: What is the necessary metadata the testing framework needs from the GLSP-based diagram editor to process the diagram?**
  - The answer is necessary to allow the testing framework to retrieve semantic information to determine how the diagram is connected or set up. The rendered diagrams have for the users a meaning. For computers, they are only SVGs without any semantics. In other words, the computer can not easily understand how to process the diagram; for this reason, it requires metadata.

This thesis aims to develop a new testing framework that allows the developer to test their custom diagram editors on different environments (i.e., tool platforms). Testing-wise, the focus of this thesis is to be able to programmatically interact with diagrams and to test those interaction possibilities. Interaction possibilities include semantic interactions that the editor allows the users, such as creating, renaming, deleting, and similar. Testing for visual differences and the correct rendering is not part of this research.

[12] `https://www.electronjs.org/`, Accessed: 19.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1.3 Methodology

The methodological approach underlying the work in this thesis is Design Science Research (DSR) [HMPR04]. The following steps will be followed:

1. **Testing Literature & Requirements Analysis**:
   Testing is a widely explored field. Analyzing the state-of-the-art of existing industrial tools, design principles, and approaches are necessary.

   Aside, the testing framework will be used by developers and testers. Gathering information regarding their expectations, requirements, and workflows is essential to support them. This step is also necessary to answer **RQ1**.

2. **Library & Framework Analysis**:
   Different tools, frameworks, and libraries already exist for testing purposes with different strengths and drawbacks. Therefore investigating and analyzing them is necessary to determine if they are appropriate and can be used to build upon for fulfilling the requirements.

3. **Conceptualization**:
   Based on the previous two steps, a concept needs to be created as the basis for the new testing framework. This concept includes details like how the developers can use the framework to write tests and the architecture of the new testing framework (e.g., Page Objects, code structure). Additionally, programming language and implementation strategies are also determined. The goal is to answer **RQ2**.

4. **Build & Evaluate Artifacts**:
   GLSP will be extended, and one new artifact will be created:

   a) **GLSP**: Extend GLSP to provide the necessary metadata to third party applications such as the GLSP testing framework (**RQ3**).

   b) **GLSP Testing Framework**: Develop and incrementally extend the GLSP testing framework with new features and evaluate them with the authors of GLSP.

5. **Final Evaluation**:
   Evaluate the GLSP testing framework by implementing a test suite for a GLSP-based diagram modeling tool using the functionalities of the framework.

1.4 Structure

This thesis consists of the following chapters:

Chapter 2 gives background information by introducing software testing and GLSP. The necessity of testing and how software applications can be tested will be discussed in more

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

detail. Additionally, GLSP will be introduced, and the difficulties concerning software testing it raises will be elaborated upon.

Chapter 3 illustrates how browser automation works. There are different approaches to how browsers can be controlled through third parties. Each method has its own unique characteristics and, consequently, distinct advantages and disadvantages with a significant impact on the testing framework. Also, in this chapter, different browser automation and testing frameworks will be showcased, compared, and then the most appropriate selected.

Chapter 4 delves into a comprehensive explanation of the testing framework's architecture. It will encompass a detailed examination of the individual components, design patterns, and technologies used, along with an explanation and exploration of the testing methodology employed in the framework. It will also compare testing scenarios with and without GLSP-Playwright to demonstrate the benefits.

Chapter 5 evaluates the GLSP-Playwright testing framework against a real-world example. Thereby, it will determine which aspects of the GLSP framework the GLSP-Playwright will be able to cover.

Chapter 6 will look at what work can be conducted to expand the functionality and improve the user experience.

Chapter 7 finally concludes this thesis with a summary of what has been achieved and answers the research questions introduced in the beginning.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Background

This chapter will first clarify the necessity of software testing and then introduce GLSP. For software testing, it will also go into detail about existing approaches and terminologies. After defining GLSP, it will also discuss what will be tested in the context of GLSP and provide information on why GLSP is more complicated to test compared to other applications.

2.1 Software Testing

Software testing has become an essential part of any software application or system in this era of rapidly evolving technologies and changing requirements [Jor13, BP06, BZZ20]. It is the central factor ensuring the software's quality, reliability, and functionality. Moreover, it helps to identify issues early and saves cost and time in the long run [JAAA16]. This is done by a systematic process where the system is evaluated to detect deviations from the expected behavior. In most cases, the system's expected behavior is already defined beforehand or given through requirements. Due to this reason, validating the specified requirements and checking if the system performs as intended can uncover bugs or other flaws that can negatively influence the quality and stability of the system. The process of testing software is too general in practice as it encompasses a wide range of techniques and methodologies on how it can be done. For this reason, there are different foci, each with individual approaches and goals focusing on various aspects of a software system including but not limited to [IST23, DLF06]:

- **Functional Testing:** Type of testing to determine if the features work according to the software requirements.
- **Performance Testing:** Evaluation to determine how the system performs in terms of responsiveness and stability under a particular workload.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Usability Testing:** Observation of the behavior of users who try to accomplish tasks to determine their effectiveness.

The ensuing sections will further provide theoretical aspects for software testing and define on which levels software applications can be tested.

2.1.1 Definitions

Previously, it was mentioned that software testing ensures software quality, but what is the quality of the software? To better understand, two definitions from ISTQB and IEEE will be provided. The International Software Testing Qualification Board (ISTQB) provides the defacto standard and terminology in the industry concerning software testing. The definitions of ISTQB [IST18] and IEEE [bac17, nr. 3.3259, 3.3835] can be taken from Table 2.1.

Table 2.1: (Software) Quality definition

	ISTQB	**IEEE**
Quality	*The **degree** to which a component, system or process **meets specified requirements** and/or **user/customer needs** and **expectations**.*	***degree** to which the system **satisfies** the **stated and implied needs** of its **various stakeholders**, and thus **provides value***
Software Quality	*The **totality of functionality** and **features** of a software product that bear on its **ability** to **satisfy stated or implied needs**.*	*capability of software product to **satisfy stated** and **implied needs** when used **under specified conditions***

While there are subtle differences in the wording and specific details of the definitions, the general principle and perspectives align. Both definitions emphasize the satisfaction of user needs and requirements as crucial for software quality. Moreover, both acknowledge stated and implied needs. Stated needs are explicit needs that refer to requirements or specifications that are clearly defined, documented, or communicated by the stakeholders. Typically, those needs are expressed in user stories, functional requirements, or other specifications. They are formal, meaning they can be measured with almost no ambiguity or multiple interpretations. The opposite is implied needs. They are not explicitly stated but are inferred based on the context such as user behavior, common practices, or industry standards. Usually, they are derived from the experience of the development team or identified through analysis or interactions with the users and stakeholders. Common implied needs are for example performance, usability, and security. They are not directly mentioned but are still considered essential for software to fulfill its purpose. Moreover, both definitions highlight the ability of the software to perform as expected and meet user needs in the intended environment.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

A small difference between the definitions is that ISTQB highlights the *"totality of functionality and features"*, whereas the definition of IEEE emphasizes satisfaction *"under specified conditions"*. The former definition uses *"totality"* which means that all aspects of the software contribute to the quality. In contrast, the IEEE definition implies the overall effectiveness in meeting use requirements by writing *"under specified conditions"*. This indicates that the evaluation takes into account the particular environment or circumstances in which the software is assessed.

The process that assures the previously defined (software-) quality is called quality assurance (QA) [IST18, bac17, nr. 3.3260] or software quality assurance (SQA) [bac17, nr. 3.3836] and is defined in Table 2.2.

Table 2.2: (Software) Quality assurance definition

<table>
<tr><th></th><th>ISTQB</th><th>IEEE</th></tr>
<tr><td>Quality Assurance</td><td colspan="2">Part of quality management focused on providing confidence that quality requirements will be fulfilled.</td></tr>
<tr><td>Software Quality Assurance</td><td>-</td><td>set of activities that define and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate for and produce software products of suitable quality for their intended purposes</td></tr>
</table>

(Software) Quality assurance is an ongoing process within the software development lifecycle to ensure that the software product complies with the established and standardized quality specifications. The goal is to catch shortcomings and deficiencies before releasing the software. There are different ways to accomplish this such as *organizational*, *static*, and *dynamic* methods [BZZ20, IST23].

- **Organizational** methods provide checklists and guidelines to reduce the emergence of quality deficiencies.
- **Static** methods analyze the software without executing the system, for example, scanning for code smells.
- **Dynamic** methods evaluate the system at runtime and monitor the behavior.

2.1.2 Testing

Testing is an activity performed to identify defects and problems as defined in Table 2.3. According to the ISTQB definition [IST18], it includes all static and dynamic methods, yet, in the IEEE definition [bac17, nr. 3.4272], it concerns only dynamic verification. Still, the focus is on evaluating the system's actual behavior against the expected behavior.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 2.3: Testing definition

ISTQB	IEEE
The process consisting of all ***lifecycle activities****, both* ***static*** *and* ***dynamic****, concerned with planning, preparation and* ***evaluation*** *of* ***software products*** *and* ***related work products*** *to determine that they* ***satisfy specified requirements****, to demonstrate that they are* ***fit*** *for* ***purpose*** *and to* ***detect defects****.*	***activity*** *in which a* ***system or component*** *is* ***executed*** *under* ***specified conditions****, the results are* ***observed or recorded****, and an* ***evaluation*** *is made of some* ***aspect*** *of the* ***system or component***

In accordance, the resulting problems and defects found in software can be further categorized into [BZZ20, IST23]:

- **Software Errors:** A software error refers to human mistakes made during the designing, coding, or implementation of software. It results in incorrect or unintended behavior as it deviates from the expected or intended functionality. They are typically introduced by incorrect logic, coding mistakes, misunderstandings, or flawed implementation.
  - *Example: A developer forgets to implement a NULL check in Java.*
- **Software Faults:** Software errors lead to software faults. A fault is an incorrect step or process that leads to incorrect functioning of the software.
  - *Example: Missing a NULL check leads to an exception at runtime.*
- **Software Failures:** Failure is the observable discrepancy between the intended and delivered result. It happens when a software fault is encountered or triggered. It manifests as crashes, freezes, incorrect outputs, and similar undesired behaviors that prevent the correct functioning.
  - *Example: Data is not saved after the user clicks on save.*

Using those three different terminologies enables testers and developers to communicate what they know about the problem. *Failures* mean that something went wrong, but the cause is unknown. *Fault* means the cause is known, but not why the fault occurred. An *error* means the origin and reason for the fault are known. An error of the developer leads to a fault in the code that results in failure for the user. Consequently, software testing has the intent to find and prevent failures while gaining confidence about the level of quality.

Before continuing, it needs to be clarified what the difference between *verification* and *validation* is [BZZ20, BP06]. *Verification* is the assurance that a system conforms to the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

specification. The question here is if the product is built right. *Validation* is the assurance that the system satisfies the user's needs or expectations and fulfills its intended use. The focus here is if the right product is built. In the context of software testing, the goal is *verification*. The tests check if the software component conforms to the provided specification.

2.1.3 Levels

In the literature [BP06, BZZ20, Jor13, IST23], testing of software can be done on four different levels. Those levels group activities that belong and need to be executed together. The testing pyramid illustrated in Figure 2.1 shows the different levels of testing based on their scope and importance. At the bottom of the pyramid is the foundation, called *Unit Testing*. It forms the largest portion of the test suite and is the most cost-effective to implement and execute. The levels above are separated regarding whether the focus is on the tested component's internal working or external behavior [JAAA16]. *White Box* means that the tester has full disclosure about the source code and can test it considering its internal functioning. *Black Box* means, that the tester writes tests based on the exposed API instead of the source code. To achieve a robust testing strategy, it is crucial to maintain a balanced distribution of tests across all levels. A strong foundation of unit tests forms the basis, followed by a smaller set of integration tests, and a smaller yet significant number of end-to-end tests. This approach allows for quicker feedback, cost-effectiveness, and efficient identification of issues at various levels of the application. On the other hand, the inclusion of acceptance testing in the testing pyramid is controversial. The reason why will be elaborated later on.

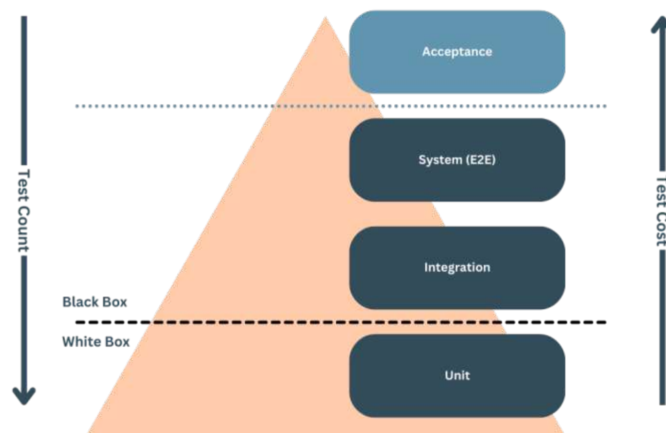

Figure 2.1: Test pyramid

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Unit Testing

Unit tests focus on verifying the correctness of individual units or components of a system as defined in Table 2.4 [IST18, bac17, nr. 3.4429].

Table 2.4: Unit testing definition

ISTQB	IEEE
The testing of ***individual software components****.*	*test of* ***individual programs*** *or* ***modules*** *in order to* ***ensure*** *that there are* ***no analysis or programming errors***

In this context, a unit refers to the smallest testable part. This is usually either a function, method, or class. Typically, they are written by developers during development to ensure the code meets the specified requirements. They can be written in isolation from the rest of the system. That means that the unit can be tested while minimizing dependencies on other parts. Additionally, they also allow easily to test targeted or specific functionality. This advantage allows for writing unit tests early in the development cycle which makes it easier to detect and address those issues.

Unit tests are often automated and execute rapidly. However, it requires access to the code base and development environment. Nevertheless, they can be executed frequently to detect regressions. Further, they act as documentation for the expected behavior of the unit. This ensures that modifications or enhancements in the system do not introduce new defects in those units.

Integration Testing

Integration tests focus on the interfaces and interactions between different components and units (see Table 2.5 [IST18, bac17, nr. 3.2034]).

Table 2.5: Integration testing definition

ISTQB	IEEE
Testing performed to ***expose defects*** *in the* ***interfaces*** *and in the* ***interactions*** *between* ***integrated components*** *or* ***systems****.*	*testing in which* ***software components, hardware components, or both*** *are* ***combined*** *and* ***tested*** *to* ***evaluate*** *the* ***interaction among*** *them*

The goal is to verify the system's behavior in terms of how these units work together. In contrast, unit tests target individual units in isolation. The aim is to identify issues that may arise due to interactions between components. For example, compatibility problems and communication/protocol errors are the focus. Furthermore, compared to unit tests, integration tests have a larger scope to test. They cover larger sections or combinations of code bases involving multiple interconnected units or modules. Besides, while dependencies should be mocked in unit tests, for integration tests they should not.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

The dependencies are taken into account while testing. Practically, integration tests check two aspects. First, if interfaces and protocols are followed by the components. Second, the overall workflow/order of execution and the simulated environment in which the component is executed is considered.

System Testing

System testing concerns evaluating the complete and integrated software system as a whole as defined in Table 2.6 [IST18, bac17, nr. 3.4122].

Table 2.6: System testing definition

ISTQB	IEEE
*Testing an **integrated system** to verify that it meets **specified requirements**.*	*testing conducted on a **complete, integrated system** to **evaluate** the **system's compliance** with its **specified requirements***

The test environment should mirror the production environment. This minimizes environment specific-errors. The objective is to ensure that the system meets the specified requirements in a production-like environment. The primary focus is compliance with the stated and implied needs. That are functional and non-functional (e.g., security, usability, performance) aspects of the system. It involves testing the system behaviors from an end-to-end (E2E) perspective. Typically, the tests involve user scenarios that mimic real-world user interactions and workflows to check if the system delivers the expected outcomes in those scenarios.

System testing is often conducted by dedicated testers or from an independent team who are not directly involved in the development of the software. The reason is to have an independent evaluation of the behavior to ensure the requirements and functions are as expected without bias.

Acceptance Testing

Acceptance testing is conducted mainly by the users, customers, or authorized entities and not by the developers/dedicated testers (see Table 2.7 [IST18, bac17, nr. 3.33]).

It is done to determine if the system fulfills the acceptance criteria. It involves validating whether the system meets the requirements and expectations of the stakeholders. Uncovering defects is not the main purpose. It ensures that the software aligns with the user's needs and achieves the business objectives. Stakeholders go through the system and evaluate if the system is ready for acceptance and use in a production environment.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 2.7: Acceptance testing definition

ISTQB	**IEEE**
Formal testing with respect to ***user needs, requirements, and business processes*** *conducted to* ***determine*** *whether or not a* ***system satisfies*** *the* ***acceptance criteria*** *and to enable the* ***user, customers or other authorized entity*** *to* ***determine*** *whether or not to* ***accept*** *the* ***system****.*	***formal testing*** *conducted to enable a* ***user, customer, or other authorized entity*** *to determine whether to* ***accept*** *a* ***system or component***

2.2 GLSP

The Graphical Language Server Platform (GLSP)[1] is an open-source framework developed by EclipseSource. Its primary purpose is to create diagram editors using web technologies and serves as a foundation for building graphical modeling tools and applications with sophisticated graphical features. One of the key advantages of GLSP is its adaptability and modularity, which allows developers to tailor it to their specific requirements.

GLSP operates on a client-server architecture [BLO23, Phi]. The client aspect is responsible for displaying the diagrams and handling user interactions, while the server is tasked with managing the business logic and the underlying source model. Figure 2.2 illustrates the various components, including the client, protocol, server, and source model that comprise the GLSP framework.

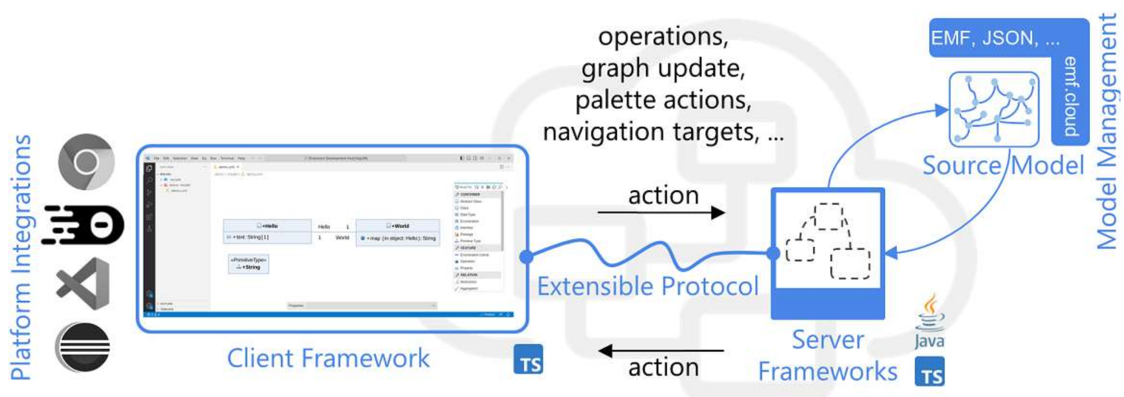

Figure 2.2: Overview of GLSP components and their interplay [BLO23]

Server-side The server part plays a critical role in enabling the diagram editor. It processes client requests, manages diagram states, enforces constraints, and provides services for modifying and reading the diagram data. It also maintains the *source model*,

[1] `https://www.eclipse.org/glsp/`, Accessed: 19.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

which represents the underlying data structure of the diagram. A source model defines the elements, their attributes, their relationship, and other relevant information. It is the only source of truth regarding the diagram. The communication between the client and server is handled through the GLSP *protocol*. The protocol defines a set of messages and commands that the client and server understand. It is used to exchange information regarding diagram updates, actions, and other relevant events.

Client-side The client leverages web technologies such as HTML, CSS, and JavaScript (in particular TypeScript) to create the graphical/diagram editor. One of the key responsibilities of the client is to render the diagram elements based on the data received from the server. The client also takes care of creating an interactive and visually appealing representation of the diagram by utilizing SVG. More about this topic will be provided in Subsection 2.3.1. Additionally, the client provides user interface components necessary to work with the diagram, such as tool palettes, context menus, and property palettes. Aside, the same GLSP-Client can be reused on different *tool platforms*. Tool platforms are browser-like instances such as integrated development environments (IDEs) like Eclipse IDE, Eclipse Theia, and Visual Studio Code (VS Code) or browsers in general. This enables integrating GLSP-based diagram editors seamlessly into their preferred environment.

A comprehensive understanding of the underlying technology is paramount when creating a customized testing framework [JAAA16] because the features available in the technology significantly impact the design and implementation of the tailored testing framework. It sets the groundwork for all subsequent steps, making error prevention critical. For this reason, the Workflow Example and the respective integrations for Theia and VS Code will be investigated and characterized. The Workflow Example is a consistent and widely used example by the GLSP authors. It serves as a simple flow chart diagram editor, allowing developers to explore different GLSP features, and it effectively demonstrates the inherent functionalities offered by GLSP. For this reason, in the subsequent sections, aspects and characteristics of GLSP that can have an impact on the testing framework will be described.

2.2.1 Characteristics

In this section, the characteristics of the architecture of GLSP that are important for the testing framework will be explained[2]. First, GLSP is a web-based application that leverages various modern web technologies to facilitate the creation of editors [BLO23, Phi].

Upon closer examination of the client side[3], it is implemented in TypeScript. It is also possible to reuse the same client implementation across different tool platforms, such

[2] `https://eclipse.dev/glsp/documentation/`, Accessed: 06.08.2023
[3] `https://github.com/eclipse-glsp/glsp-client`, Accessed: 11.05.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

as browsers, Theia[4], and VS Code[5]. This approach allows the diagram editor and its internal implementations to remain consistent across tool platforms without being affected by platform-specific requirements. In essence, the core functionality for rendering the diagram and other important parts is encapsulated within the GLSP-Client and exposed by an API to generate the editor from scratch at the target location requested. This approach allows different tool platforms to call the GLSP-Client API, place the editor at the correct location, and customize the GLSP-Client by providing custom functionality or implementing new features directly to the tool platform. For example, developers could decide to implement an outline view, which provides a textual overview of the current diagram, directly to the GLSP-Client or the tool platform. The former allows to implement once and reuse it on all other tool platforms; however, directly implementing it into the tool platform usually feels more natural to the user experience and not detached.

Upon further investigation of the server side in the GLSP architecture, it becomes apparent that the deployment approach can vary. Typically, the server and client are deployed together on the same machine. However, this is not the only option available. The server can also be deployed on a different machine, allowing the client to interact with it remotely. Alternatively, the GLSP-Server can be embedded directly into the client, creating what is known as a *fat client*, though this is a less common scenario. Nevertheless, in most cases, the GLSP-Server operates independently in the background as a separate process, either in a Java Runtime Environment or with NodeJS. Aside from the GLSP-Server, the deployment could also require other servers.

Summarized, the following characteristics are standing out in the context of testing GLSP:

- **Web Technologies:** GLSP leverages modern web technologies and the client-server pattern to create a highly interactive and user-friendly diagram editor.
- **Runtime Dependencies:** The GLSP-Client may have runtime dependencies on other language servers and various other services during its execution aside of the GLSP-Server.
- **Cross Tool Platform:** By utilizing web technologies GLSP ensures cross-platform compatibility and allows the editor to run seamlessly in various web browsers and browser-like environments.
- **Tool Platform Differences:** Tool platform-specific integrations have the ability to influence the behavior of GLSP, resulting in different functionalities and interactions within the editor.

[4] `https://github.com/eclipse-glsp/glsp-theia-integration`, Accessed: 11.05.2023
[5] `https://github.com/eclipse-glsp/glsp-vscode-integration`, Accessed: 11.05.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.2.2 User Interface

This section will examine the User Interface (UI) of the Workflow editor[6] with version *1.1.0-RC10*. Figure 2.3 provides an overview of the UI and displays several components that facilitate diverse interaction capabilities within the editor. These components employ various methods to enable users to interact with the editor effectively. The subsequent parts will delve deeper into each of these components to gain a comprehensive understanding of their functionalities and how they contribute to enhancing the user experience.

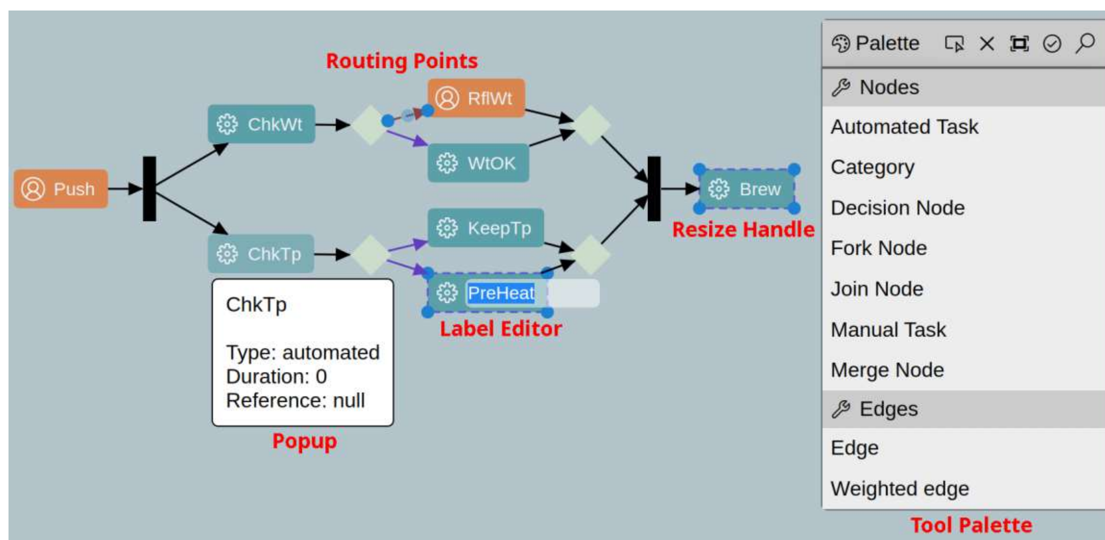

Figure 2.3: Overview of the UI (Standalone Version)

Diagram

The central component of the editor is the diagram. However, in the GLSP documentation[7] and source code it is called a *graph*, which represents the rendered model as a diagram and serves as the primary area for user interactions. To enhance clarity, this thesis will maintain the term *diagram* rather than *graph* for better readability. However, when addressing source code, such as in the architectural details presented in Chapter 4, the accurate term *graph* will be employed. Within the diagram, diagram elements are visualized as nodes and edges, which can also possess child elements and display additional visual elements based on the specific context and functionality implemented. For example, the diagram elements could have a marker showing the state of the diagram element visually. Next, the whole diagram and the diagram elements within are technically SVG elements rendered by the browser, allowing diverse shapes and styles to be created. By utilizing SVG, the users can interact with both the diagram elements and the diagram

[6] `https://github.com/eclipse-glsp/glsp`, Accessed: 06.08.2023
[7] `https://eclipse.dev/glsp/documentation/gmodel/`, Accessed: 18.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

itself. All those elements are rendered on top of the viewport, which also allows user actions like adjusting the visible area.

Aside, it is possible to do actions, such as but not limited to changing position, size, renaming, hovering, and deleting, which can be performed directly on the diagram elements. Since the modifications can affect the underlying model, the rendered part of the diagram may need to be updated. These changes are asynchronous and might take varying amounts of time (e.g., milliseconds to seconds) to trigger the re-rendering process.

UI Extensions

UI extensions enhance the editor's functionality by introducing custom user-facing interactions or providing feedback to users. For instance, the tool palette displaying available nodes and edges to create is a UI extension. Another example is the popup that appears when hovering over an diagram element. These extensions enrich the editor by adding new features and user interfaces, thus improving the overall user experience. The outstanding components visible to the user are as follows.

Tool Palette: The tool palette is a core component that allows users to add new elements to the diagram. It comprises two main parts. The top section is the toolbar, which enables users to perform various actions, such as deleting elements or using the marquee tool for selection. The body of the tool palette contains a collection of addable diagram elements such as nodes or edges. These elements can be organized and grouped under specific names for easier user access and usage.

Command Palette: The command palette lists available commands that users can search for and execute. Users can access it by using the keyboard shortcut $CTRL + SPACE$. There are two types of command palettes - one for the diagram and one for the diagram element. The commands displayed in each palette depend on the specific context in which it is accessed.

Label Editor: SVG elements do not support direct user input. As a solution, GLSP introduced the label editor as an input field displayed above the rendered SVG to address this limitation. Its purpose is to take user input for the diagram element and, accordingly, to update the label. Additionally, it may incorporate a validator to ensure that only valid input is accepted from users.

Popup: Certain diagram elements provide supplementary information when users hover over them. These additional details are displayed as a popup located next to the diagram element, offering contextual information or additional options related to that element.

Routing Points & Resize Handle: When users select an edge in the diagram, they can create new routing points for that edge. Similarly, when clicking on a node, users

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

can modify its size by dragging one of the four resize handles at the node's corners. This allows users to customize the appearance and layout of the elements in the diagram according to their specific needs.

Mouse & Shortcuts

GLSP heavily relies on mouse interactions, where most of the user actions are performed through mouse clicks and movements. However, the keyboard also plays a significant role in supplementing mouse interactions, enhancing the overall user experience and accessibility [SMB23]. GLSP employs shortcuts in three main ways:

- **Shortcut-Only:** Some interactions can be only initiated by a shortcut, e.g., command palette.
- **Shortcut-Counterpart:** Actions mainly done by the mouse are also available as keyboard shortcuts, e.g., copy-paste and delete.
- **Mouse-Supplementation:** Pressing a specific key switches the mode; e.g., pressing the *CTRL* key allows the mouse to create the same diagram element multiple times.

2.2.3 Integrations

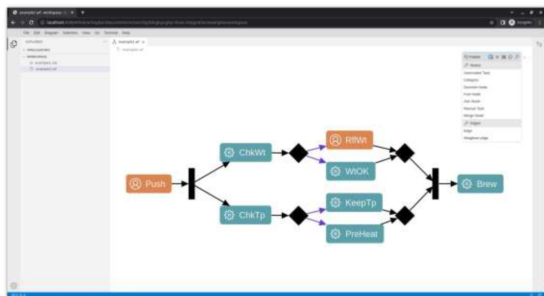

Figure 2.4: Workflow example in Theia

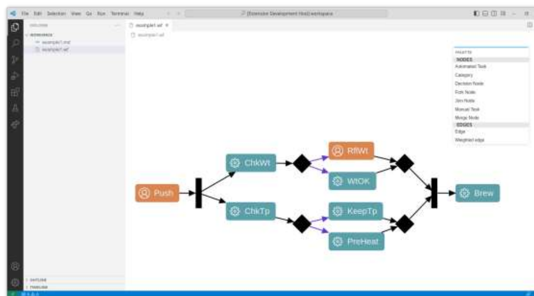

Figure 2.5: Workflow example in VS Code

GLSP offers support for various tool platforms through integrations. These integrations act as connectors between the GLSP-Client and specific tool platforms, facilitating the seamless integration of GLSP within these platforms. By using those integrations, developers gain the flexibility to use GLSP in the particular tool platform and to provide custom user interfaces within the tool platform. That allows delegating certain aspects of the diagram editor's functionality to the platform. By doing so, the implementation becomes integrated into the tool platform instead of residing solely within the GLSP-Client. This approach enhances the user experience as it creates a cohesive interface that seems to originate from the tool platform (e.g., IDE) rather than being a separate entity.

As previously stated, various tool platforms are built differently. Table 2.8 illustrates these differences to provide a clearer understanding.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 2.8: Tool platform differences

	Standalone	**Theia**	**VS Code**
Environment	Browser	Browser	Electron
Integration Level	Fully	Fully	Extension
Location	Direct	Widget	Widget
DOM Placement	Direct	Direct	iFrame

- **Environment:** The environment refers to the execution context of the tool platform. The Standalone and Theia platforms run within the browser. On the other hand, VS Code is a native application that requires installation and operates on top of Electron, a browser-like environment.
- **Integration Level:** The integration level determines the extent to which the integration can customize the tool platform by adding new functionalities or user interfaces within it. The Standalone integration operates alongside the web application that loaded it with complete control over the environment. Theia also offers similar flexibility, enabling full customization of the editor. In contrast, VS Code only allows customization through extensions[8]. These extensions run in a sandboxed environment, ensuring security but limiting their access to the platform.
- **Location:** The location specifies how the GLSP-Client is accessed in different integrations. In the Standalone integration, developers have the flexibility to place it wherever they need it within their application. In Theia, it becomes accessible when a file is opened in a tab within Theia, and when it is registered that the file should be handled by the GLSP diagram editor. The same principle applies to VS Code, where the GLSP-Client becomes available after opening a file associated with it.
- **DOM Placement:** DOM placement determines where the GLSP-Client constructs the editor within the web page's Document Object Model (DOM)[9]. The editor is directly added to the existing DOM elements in the Standalone and Theia integrations. However, in VS Code, the GLSP-Client creates the editor within an iFrame[10], a separate document embedded within the main document.

2.3 Discussion

This chapter explored testing software and introduced the Graphical Language Server Platform (GLSP). Now, the focus is on applying the principles and methodologies discussed in Section 2.1 to the specific context of GLSP.

[8] `https://code.visualstudio.com/api`, Accessed: 06.08.2023

[9] `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction`, Accessed: 06.08.2023

[10] `https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe`, Accessed: 06.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The testing framework developed in this thesis aims to simplify the functional testing process for GLSP-based editors. By using it, the testers can easily create test cases to verify that their editor adheres to specifications and, consequently, improve software quality by detecting and addressing potential software failures through extensive testing. Likewise, in the context of the four levels of testing, the focus lies on end-to-end testing, which falls under the system testing layer. This layer presents a greater challenge compared to the layers below [DLF06], as it requires interacting with a running system, and interactions need to be typically done by simulating users. As mentioned in Section 2.2, GLSP provides a wide range of user interactions through diverse user interfaces of varying complexity. Consequently, automating these interactions for testing purposes can be challenging, as testers need to find ways to simulate users and perform tasks programmatically.

Ultimately, the primary objective is to empower (software) quality assurance teams with an efficient tool to write efficient tests for identifying shortcomings in the software product before its release to improve the overall quality of the software. To answer how the testing can be done, the upcoming chapter will delve into practical methods for testing the user interface required for end-to-end testing and to automate those tasks programmatically. However, before proceeding, it is essential to understand the complexities involved in testing GLSP. One main factor contributing to this complexity, namely the dynamic nature of the diagram (including diagram elements) will be introduced. More about the same topic and the other challenges the testing framework faces will be discussed later in Chapter 4.

2.3.1 Testing Diagrams

As aforementioned, Scalable Vector Graphics (SVG)[11] are employed for diagrams and their corresponding diagram elements. SVG is a popular format for representing two-dimensional vector graphics, and it allows for easy scaling without losing image quality. An example of SVG can be found in Listing 2.1.

However, this format poses a challenge when it comes to testing. During manual testing, testers can visually observe the rendered elements and interpret their meaning and depending on the test case, click on the correct location of the diagram element. However, programmatically deducing the semantic meaning of the elements and doing the same becomes difficult. For instance, the link between the example provided in Listing 2.1, that the program/tool retrieves and the corresponding visual representation in Figure 2.6 that the browser renders and the user sees may not be easily interpretable without manual inspection.

In the context of GLSP, all the diagram elements are SVG elements, yet it must be possible to interact with specific diagram elements in the diagram like an user would do. For this reason, working with the diagram and the SVGs within poses three challenges concerning testing.

[11] https://developer.mozilla.org/en-US/docs/Web/SVG, Accessed: 06.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 2.1: SVG Example

```
<?xml version="1.0" encoding="utf-8"?>
<svg version="1.1" id="Layer_1"
        xmlns="http://www.w3.org/2000/svg"
        xmlns:xlink="http://www.w3.org/1999/xlink" x="0px"
        ↪  y="0px" width="122.88px" height="116.864px"
        ↪  viewBox="0 0 122.88 116.864" enable-background="new
        ↪  0 0 122.88 116.864" xml:space="preserve">
        <g>
                <polygon fill-rule="evenodd"
                ↪  clip-rule="evenodd" points="61.44,0
                ↪  78.351,41.326 122.88,44.638 88.803,73.491
                ↪  99.412,116.864 61.44,93.371 23.468,116.864
                ↪  34.078,73.491 0,44.638 44.529,41.326
                ↪  61.44,0"/>
        </g>
</svg>
```

Figure 2.6: Visual representation of the SVG example

- Diagrams often have complex structures with multiple interconnected elements, shapes, and relationships. Those intricate configurations make it difficult to use automated tests to verify the correctness of these elements.
- Ensuring the consistent visual rendering of diagram elements across different browsers and browser-like environments, devices, and screen resolutions is a challenge. Minor differences in rendering engines can lead to visual discrepancies.
- Diagrams often support dynamic interactions. They allow the users, for example, to select, click, drag and drop, connect, or resize elements. Replicating these behaviors can be complex as user actions need to be simulated and the outcome needs to be verified accordingly.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Due to the complexity of working with visual representations and rendered elements, there is a lack of standardized testing approaches for diagrams in web applications. While there are established testing practices for standard web components, finding a way to test the complex behaviors and interactions of diagrams is mostly up to testers. The widely adopted testing frameworks are primarily designed on functional aspects of web applications. They test the data flow rather than the diagram's intricate elements and interactions. For this reason the verification of diagram elements is usually done manually.

Overall, the complexity of diagram structures, visual rendering variations, dynamic interactions, lack of standardized testing approaches, and limitations of testing tools contribute to the difficulties of testing diagrams in web applications. For this reason, in this thesis, the verification of visual rendering for diagram elements will be omitted due to the complexity of interpreting SVG representations. Instead, the main focus will be on simulating user interactions with the diagram elements to test their behavior and functionality. By automating these interactions, it can be ensured that the diagram elements respond as expected to user actions and that the overall functionality of the GLSP-based editor is thoroughly tested.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 3

Browser Automation

In the previous chapter, software testing was introduced, GLSP examined, and the objective of conducting end-to-end tests for the user interface of such applications was established. In this chapter, various approaches to accomplish this task will be explored.

As web applications become more intricate, similar to native applications, more sophisticated testing frameworks are needed. Today testing frameworks for the web provide generic functionality to interact with web pages and to verify their state. In essence, they control the browser to simulate user actions. The topic of how the browser can be automated to simulate users will be the first area of investigation in this chapter. Based on the findings, the typical characteristics of a web testing framework will be declared at a higher level. Subsequently, existing web testing frameworks will be compared, and one will be selected as the basis for further customization. It is important to note that web testing frameworks do not provide domain-specific functionalities. Testers must implement these functionalities themselves. Consequently, making it necessary to tailor the used testing frameworks to the specific application's requirements. Therefore, choosing the most suitable existing testing framework to meet GLSP's testing needs is crucial. No single framework can cover all aspects of testing for an application. Each testing framework has its strengths, weaknesses, and focus areas. Careful consideration is essential to ensure an effective foundation for testing GLSP-based editors.

3.1 Automation Protocols

So how can a testing framework automate the browser? To automate the browser, there exist established browser automation protocols [YS23, Aha23, Saw22]: WebDriver, DevTools, and native protocol. Each protocol has a different architecture and advantages and disadvantages. Web testing frameworks are typically categorized based on their protocols, and they often share similar strengths and limitations within their category.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.1.1 WebDriver-Protocol

> ***WebDriver*** *is a* ***remote control interface*** *that enables* ***introspection*** *and* ***control*** *of* ***user agents****. It provides a* ***platform- and language-neutral wire protocol*** *as a way for* ***out-of-process programs*** *to* ***remotely instruct*** *the behavior of* ***web browsers****. Provided is a set of* ***interfaces*** *to* ***discover and manipulate*** *DOM elements in* ***web documents*** *and to control the behavior of a user agent. It is primarily intended to allow* ***web authors*** *to write* ***tests*** *that* ***automate a user agent*** *from a* ***separate controlling process****, but may also be used in such a way as to* ***allow in-browser scripts*** *to control a —* ***possibly separate*** *— browser.* - [W3Cb]

The WebDriver protocol [Aha23, Saw22, W3Cb] is a platform and language-neutral protocol that allows other processes to interact and control the behavior of browsers. This is done by providing a well-formed HTTP-based API which is currently managed by W3C. Those APIs allow the processes to discover and manipulate the DOM elements on web pages and perform relevant operations (i.e., click). Since all recent browsers are considered W3C compliant, the browser vendors include WebDriver capabilities in their browsers.

In essence, every browser provides a browser driver (e.g., ChromeDriver) binary which is responsible for managing the respective browser. The browser-specific implementation is then hidden by those browser drivers. Those browser drivers also implement the WebDriver protocol and enable interaction with the outside. This is done by starting a web server to enable communication over the web (i.e., REST-API). Now libraries can use those well-defined WebDriver APIs to communicate over the browser driver with the corresponding browser. Each request to the web server will be translated to the respective instructions needed to trigger the change in the corresponding browser. The execution status and result are sent back to the requester. Figure 3.1 illustrates the whole approach.

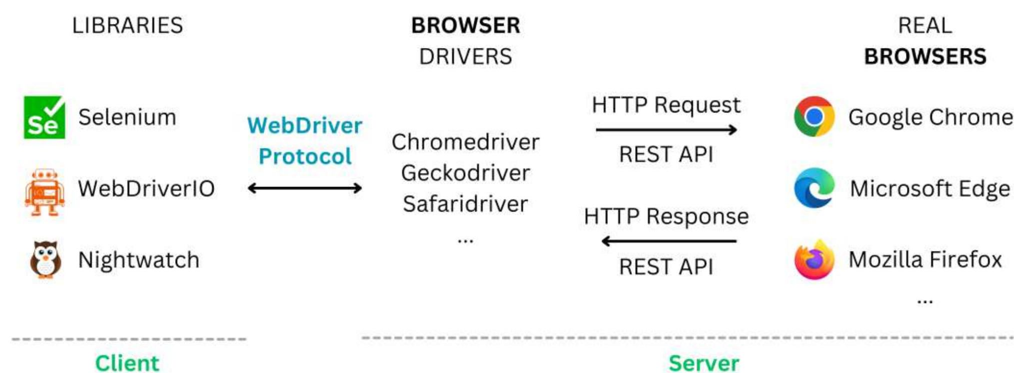

Figure 3.1: WebDriver protocol

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

This approach makes automation libraries cross-browser compatible as the browser-specific implementation is implemented by the browser drivers and exposed by a single API.

3.1.2 DevTools-Protocol

Development Tools [Aha23, Saw22, YS23], commonly known as DevTools, are essential components available in modern browsers that enable developers to analyze and debug web applications. The DevTools protocol offers a set of APIs that allow developers to instrument, inspect, debug, and control browsers natively, similar to the WebDriver protocol. In simple terms, the DevTools protocol is the interface (i.e., API) provided by the browser itself. Depending on the browser vendor, different libraries and tools can use these exposed APIs in various ways. For example, in Figure 3.2, the browser driver of Google Chrome and Playwright use the same protocol to control the browser.

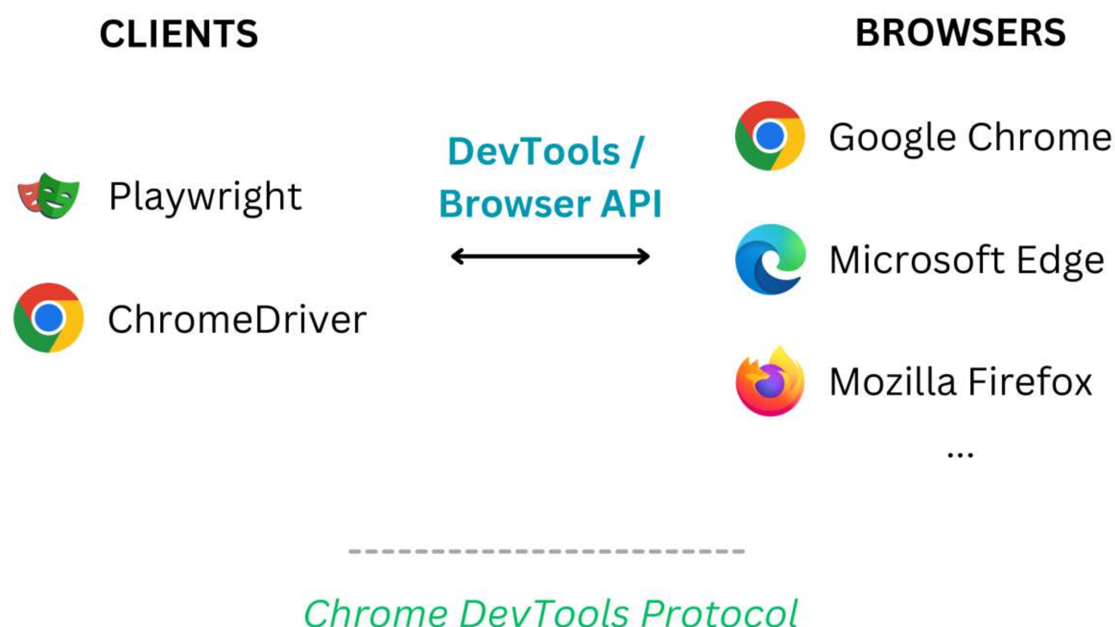

Figure 3.2: DevTools protocol

Due to the direct nature of the interactions with the browser, the DevTools protocol allows for low-level control and offers better performance compared to other approaches. This is because communication with the browser happens directly, without the need to go through multiple layers, resulting in more efficient and faster interactions.

Currently, the most comprehensive DevTools protocol is offered by Google Chrome, referred to as the Chrome DevTools protocol (CDP) [Goo]. Other browser vendors implement a subset of this protocol in their respective browsers to enable similar functionalities

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

for developers. This means that while the Chrome DevTools protocol may have more extensive features and capabilities, other browsers still aim to provide a compatible set of APIs to allow developers to inspect and debug their web applications effectively. However, that does not imply that the CDP enables cross-browser automation because it is not a standard and how well the CDP is implemented and if it is even implemented depends on each browser vendor.

3.1.3 Native

The native protocol [Aha23, Saw22] takes a distinct approach compared to the WebDriver and DevTools protocols, where an API (e.g., browser) is utilized for browser interaction.

In the native protocol (see Figure 3.3), both the application and the tests are executed side-by-side within the same process in the browser. This enables the test case to directly access all of the browser's features, the Document Object Model (DOM), and the application loaded on the web page. That gives the test cases the same ability as the web application has on the web page. In this approach, the application and the tests alternate their execution. When the application is running, the tests wait, and when the tests are running, the application pauses. Typically, this approach is supported by a NodeJS server running in the background, facilitating the communication and coordination between the application and the tests.

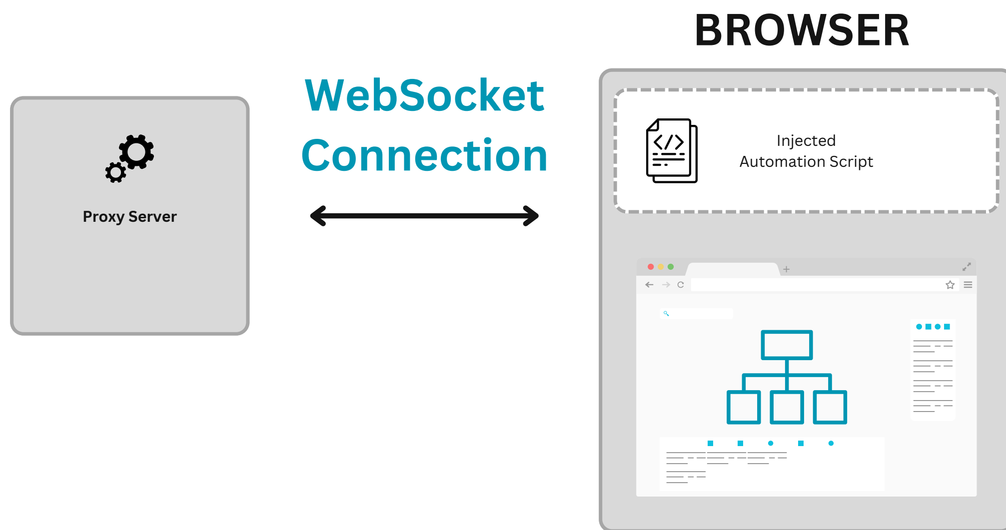

Figure 3.3: Native protocol

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.1.4 Future: WebDriver BiDi

WebDriver Bi-Directional (BiDi) [YS21, W3Ca] is a new specification built upon the WebDriver protocol and is currently under development. It aims to combine the WebDriver protocol's and CDP's strengths with cross-browser support, fast performance, and low-level access. Currently, various vendors from different industries collaborate to create such a specification. The goal is to combine the two worlds and to have cross-browser support by utilizing the benefits of CDP while ensuring low latency.

3.1.5 Comparison

All three protocols follow a fundamentally different approach [Aha23].

The WebDriver [W3Cb] uses a sophisticated protocol standardized by the W3C, enabling browser vendors to provide their browser drivers implemented against the protocol to enable cross-browser support. This approach reduces the effort required by the developers to automate multiple browsers, as they only need to develop against the WebDriver API and let the browser drivers instruct the browser. This architecture constrains the experience by making it only one way. There is no bi-directional communication between the test framework and the browser. It always goes from the test framework to the browser. Additionally, installing and managing those browser drivers can confuse users and introduce more errors if not done correctly as the browser driver has to be compatible with the used browser.

DevTools provide better control and stability, as the browser is controlled directly instead of adding multiple layers of abstraction. However, the DevTools protocol is not the same in all browsers. The DevTools protocols are not standardized like for the WebDriver. The Chrome DevTools Protocol (CDP) [Goo] is integrated into all Chromium-based and Blink-based browsers (e.g., Google Chrome, Edge, Opera) and allows to interact with the browser extensively. However, not all DevTools of browsers have the same wide-ranging integration as CDP. Firefox supports only a subset of CDP. Ultimately, most browsers have their own DevTools protocol (e.g., Safari) implementation and do not follow a specification or a de facto standard making it harder to find documentation or examples. Fortunately, these days most browser vendors try to realize CDP in their browsers.

The last approach is letting the test code run together with the application in the same context (i.e., in-process). This approach already limits the usable programming languages, as the browser only understands JavaScript. However, it enables full support as there is no difference between the application and the tests for the browser. This enables intercepting the network traffic, consequently allowing to use of server mocks and working with the DOM directly. However, the tests can not utilize multiple tabs/windows or iFrames depending on the framework.

The closer the tests are to the browser, the more stable the tests become. The reason is simple, in the WebDriver approach, the tests and the browser are not aware of each other. The WebDriver approach sends the commands to the browser driver and waits for

TU Wien Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

a response. This approach limits available features in the browser and introduces timing issues, as the tests have no information regarding the current state of the browser. The other approaches are closer to the browser or even in-process. This approach enables them access to more information and enables them more fine control. In turn, it gets harder to support cross-browsers.

Overall, the WebDriver approach provided the first steps forward in web automation [YS23]. It provides true cross-browser support and allows large-scale testing in the cloud. However, the limited feature set, like the inability to listen and interact with network or DOM events and the timing issues producing false negatives make testing more flaky [Mic16, Pla20]. Working with the browser through the DevTools protocol or directly through the browsers exposed native API dramatically improves the performance and provides more control. Nevertheless, it can not easily provide cross-browser support.

The frameworks build upon these approaches. They have different advantages and disadvantages, and there is (currently) no one-size-fits-all approach. Aside from the underlying approach, the testing framework also has a significant influence. The next sections focus on defining and deciding on an appropriate web testing framework for GLSP.

3.2 Web Testing Frameworks

This section presents a general overview of common capabilities found in various web testing frameworks. This overview is based on a manual research approach, where multiple modern frameworks such as Selenium[1], Playwright[2], Cypress[3], WebDriverIO[4], Puppeteer[5] and more were analyzed. Accordingly, their advertised features were also studied and their documentation was inspected. The goal was to identify these frameworks' common components and capabilities. These frameworks primarily automate the user's interactions with the browser, which is achieved by abstracting and utilizing one of the available browser automation protocols. By using those frameworks, testers can now define a series of step-by-step instructions on how the framework should control the browser by specifying actions and expected outcomes. The technical implementation may differ among testing frameworks.

3.2.1 Components

In addition to browser control and data retrieval from web pages, prevalent testing frameworks often include other essential components required for testing, such as:

[1] `https://www.selenium.dev/`, Accessed: 19.08.2023
[2] `https://www.playwright.dev/`, Accessed: 19.08.2023
[3] `https://www.cypress.io/`, Accessed: 19.08.2023
[4] `https://webdriver.io/`, Accessed: 06.08.2023
[5] `https://pptr.dev/`, Accessed: 06.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Test Runner:** The test runner is a crucial component of a testing framework. Its main responsibilities include scanning for test files, executing them, and generating test reports. It also plays a role in managing the testing environment, such as isolating the context between the tests, where the actual test cases are executed. Essentially, the test runner orchestrates and oversees the entire testing process.
- **Assertion Libraries:** Web testing frameworks usually include a built-in assertion library. They enable testers to verify the correctness of their test cases by simplifying the process of comparing the expected values with the actual results obtained during test execution. By using built-in functions and methods provided by the assertion library, testers can write validations more efficiently and with less effort rather than creating complex validation logic from scratch.
- **Testing Plugins:** Testing plugins are additional tools that extend the capabilities of testing frameworks by offering features that simplify the process of writing test cases similar to assertion libraries. These plugins provide functionalities that help reduce the complexity involved in creating tests. For instance, they allow testers to create mock objects or spy on certain elements without the need to manually write complex code for these tasks. Using testing plugins allows testers to achieve more efficient and effective testing, as they can leverage pre-built functionalities to handle common testing scenarios.

The testing framework typically includes the test runner, assertion library, and testing plugins by default to make it easier for testers to use. Depending on the specific testing framework, these components can be replaced with alternative implementations. Nonetheless, most testing frameworks come with these essential components integrated and readily available to simplify the testing process and reduce the effort testers require. This approach removes the worry about manually setting up these components separately.

3.2.2 Characteristics

An ideal web testing framework (in particular for GLSP) should have several qualities. First and foremost, it should be well-suited to the specific project and the challenges it aims to address. This means the framework should support the programming languages and functionalities required for the testing tasks. For example, doing functional testing differs from doing performance testing. Consequently, a suitable framework needs to be selected. Furthermore, the framework should be easy to set up and seamlessly integrate into the Continuous Integration/Continuous Deployment (CI/CD) pipeline. This ensures that the testing process becomes integral to the development workflow by facilitating automated testing. Another crucial aspect of a good web testing framework is the ability to generate detailed test reports. Usually, those cases are run without the testers observing them. For this reason, the testing framework needs to create detailed reports to understand what happened after each run. Finally, the framework should be equipped with modern testing features, such as auto-waiting for actions and the capability to trace

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

errors efficiently. These features help streamline the testing process and enhance the overall testing experience for the testers, ensuring more accurate and reliable results.

Aside from those general characteristics, an ideal framework needs to have the following qualities:

- **Test Reliability:** Test reliability is a crucial aspect of automated testing. When testing applications with diverse runtime requirements, the automation process can become fragile and prone to errors. That means to ensure test reliability, it is essential to maintain consistency in the testing environment. Accordingly, if the testing environment remains unchanged, the test results should also be consistent and reproducible. This consistency ensures that the tests produce the same outcomes when executed under similar conditions, enhancing the reliability and validity of the testing process [LGRW23].

- **Smart Waiting:** Smart waiting eliminates the need for manual waits and delays, which can introduce unpredictability and cause test failures. Instead, the framework should automatically wait until the application is ready for the next action. This ensures a stable state before proceeding with the next step. This intelligent waiting mechanism ensures that tests run consistently and reliably without unnecessary pauses or errors caused by timing issues.

- **Performance:** End-to-end (E2E) tests typically take longer to execute compared to unit tests. To address this issue, a testing framework should prioritize performance to minimize the overall time needed to run the entire test suite. One way to achieve this is by running tests in parallel. That allows multiple tests to be executed simultaneously. This parallel execution can significantly reduce the overall test duration. Nevertheless, running tests in parallel for E2E may be complicated due to side effects introduced by the running application [LGRW23].

- **Comprehensible Tests:** When writing tests, it is essential to ensure that the test cases are easy to understand and follow, even for non-technical individuals. Test scripts should be written in a way that feels intuitive and natural to enable anyone, regardless of their technical background, to comprehend the purpose and flow of the tests.

- **Debug Experience:** Creating end-to-end tests can be challenging because they simulate user interactions within the application automatically. Accordingly, to follow and understand the actions the framework does, it needs to offer debugging capabilities. These debugging features enable testers to inspect and analyze the test execution in detail. This helps them to identify any errors or unexpected behavior during the test run, which can be harder by employing only logs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.3 Evaluation

There exists a multitude of different testing frameworks for web applications. Nevertheless, they share some similarities depending on the underlying automation protocol, consequently having their intrinsic advantages and disadvantages. Accordingly, most testing frameworks only provide an abstraction layer above the browser automation protocol to remove the necessity to implement the client part by the testers. Moreover, as mentioned, they also include other libraries (e.g., assertion libraries) to reduce the required effort or provide new features on top of the protocol.

For this reason, the following sections will outline the desired features and expectations for the testing framework concerning GLSP-based applications. It will establish specific criteria that the evaluation process will follow to assess the testing framework's suitability for testing GLSP applications. By defining these expectations and criteria, the goal is to select the most appropriate testing framework that aligns with GLSP's specific needs and ensures a successful testing experience for testers.

3.3.1 Criteria

Choosing the suitable testing framework is pivotal, given the high complexity of writing E2E tests, the necessity to be able to interact with the diagrams, and the complexity of GLSP in general. Unfortunately, no testing framework can fulfill all possible testing scenarios. Different frameworks have different advantages and disadvantages and are more aligned to specific cases. The decision on the testing framework depends on the intended goals and the problems that have to be solved. The most appropriate testing framework will be selected based on the following criteria.

- **Language/Browser Support:** The choice of supported programming languages and browsers is a significant consideration for the testing framework. Since GLSP itself is developed using mainly TypeScript and Java, the testing framework must also support either TypeScript (i.e., JavaScript) or Java to align with the existing codebase and to ease the testing process. Moreover, testing on different browsers is especially important to gain confidence.
- **Automation Protocol:** The testing framework's capability is closely linked to the used browser automation protocol. Each automation protocol has distinct advantages, which can influence what the framework can do and what it cannot.
- **Parallelization (Performance):** The ability to run test cases in parallel is a crucial factor to consider because UI tests can be time-consuming compared to other tests. Parallelization allows running multiple test cases simultaneously, which significantly reduces the overall time required to complete the entire test suite.
- **Electron Support:** Another consideration is whether the testing framework supports testing Electron applications. GLSP can be executed in browser and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

browser-like environments, and while most web testing frameworks already support regular browsers, they often lack specific support for testing Electron applications.

- **Features:** Another evaluation aspect is the features provided on top of the used browser automation protocol. An ideal testing framework should offer valuable and user-friendly features that can simplify the process of writing test cases. This reduces the effort required by testers and enhances the overall testing efficiency.
- **Popularity:** The last consideration is regarding popularity. A widely used framework often has a larger community of developers actively working with it. This leads to more shared resources, such as tutorials, examples, and documentation, making finding information and solutions to common issues easier.

The following parts will describe common testing frameworks for each browser automation protocol, which have been selected in agreement with the authors of GLSP. The evaluation presented in the following sections is based on practical experience gained from using the tools and insights obtained through research from various sources on the web (e.g., articles, developer forums, documentations). It is essential to acknowledge that these assessments may carry inherent bias and subjectivity due to the individual perspective and experience.

While the presented information aims to be as objective and balanced as possible, readers should be aware that personal biases and preferences may influence the assessments. Therefore, it is recommended to consider multiple sources and conduct further research to form a well-rounded understanding when evaluating the advantages and disadvantages of the selected frameworks.

3.3.2 Selenium

Selenium[6], initially released in 2004, is one of the oldest and most known testing frameworks available. Historically, before Selenium, testing web applications was done primarily manually [YS23]. However, manual testing is error-prone and could lead to degrading software quality. Initially, Selenium which allowed controlling browsers directly revolutionized the technology landscape for that time. Today, Selenium is a group of automation testing tools used for automation. It also greatly impacted the WebDriver protocol standardized by the W3C. Selenium is open-source and provides different components to support writing tests, namely Selenium IDE, Selenium Grid, Selenium RC, and Selenium WebDriver [GGGMO20]. In brief, the Selenium IDE is used to record and playback tests by using an extension on the browser. That removes the necessity to write test cases explicitly. Selenium Grid is an intelligent proxy server that allows running tests in parallel on multiple machines for different browsers, consequently allowing load balancing. The load balancing is achieved by creating a hub server that routes the commands to remote web browser instances registered as Grid nodes. This

[6] https://www.selenium.dev/history/, Accessed 19.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

approach enables running the same test case on multiple machines and with different browsers. Selenium RC (Remote Control) was the main Selenium project for a very long time. It allowed writing tests in any scripting language by implementing, however, it is not used anymore because of the release of the Selenium WebDriver as Selenium WebDriver replaced Selenium RC.

As the name implies, Selenium WebDriver uses the WebDriver protocol and also is the precursor of the WebDriver specification released by the W3C. In 2012, the working draft was released to make the WebDriver protocol an internet standard and recommended in 2018 by the W3C. Nevertheless, until the recent version of Selenium 4 (released in 2021), it did not use the WebDriver specification directly. It used an custom implementation to communicate with the browser drivers making it more fragile. Selenium 4 now uses the standard specified by the W3C directly. That means that Selenium can directly communicate with the browser drivers in a standardized way, thus making it more consistent than older versions.

Advantages and Disadvantages

Selenium offers several advantages and disadvantages, which are outlined below [LGRW23, GGGMO20]:

+ **Language Support:** Selenium supports many programming languages, including Java, JavaScript, Ruby, Python, C#, and others. This versatility allows developers and testers to choose their preferred programming language for writing test cases.

+ **True Cross-Browser Compatibility:** Selenium adheres to the W3C standard, which ensures comprehensive cross-browser support through the WebDriver protocol. This enables the creation of tests that can seamlessly run on multiple browsers without requiring extensive modifications. As the WebDriver protocol and browser drivers continue to evolve, Selenium will gain access to additional features, in turn, which will enhance its capabilities and compatibility with modern browsers.

+ **Rich Tools:** Apart from its browser interaction capabilities, Selenium offers a suite of useful tools. Selenium Grid enables the execution of tests across various browser types, versions, and operating systems on remote machines. Selenium IDE facilitates the recording and playback of tests. Additionally, Selenium exhibits seamless integration and extensibility with other tools and frameworks.

+ **Large Community and Support:** Selenium has a vast and active community and comes with extensive documentation and tutorials. The active community provides their knowledge and experience and collaborates with each other making it easier to overcome challenges and learn from others.

- **Beginner Unfriendly:** Unfortunately, the API is complicated and noticeable that it is an old framework. Further, setting everything up and managing the browser drivers can be bothersome and requires some initial effort and time investment.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Time-Consuming Test Execution:** Due to the different abstraction layers and not communicating with the browser directly makes executing the test cases slower compared to other automation frameworks [Rag20, Gag23]. This is intrinsic to the architectures which use the WebDriver protocol.

- **Fragile Automation:** Selenium has no direct access to the browsers state and thus making it more fragile than other testing frameworks. The fragility is especially visible compared to frameworks that have auto-wait functionality. That leads to more unexpected errors.

- **Lack of Reporting:** Selenium does not have built-in reporting capabilities[7]. It can capture screenshots and log details. However, the reporting itself depends on third party libraries.

While Selenium has its limitations compared to modern automation frameworks, it is still one of the most known test automation tool available [CLR20]. It is great for applications and testers that utilize a mix of old and new technologies.

3.3.3 Cypress

Cypress is a frontend automation tool for browser tests designed with modern JavaScript frameworks (e.g., Angular, React, Vue) in mind, released in the year 2017. First and foremost, Cypress is not a general-purpose browser automation tool like Selenium. It follows the native protocol and runs in the same run-loop as the tested application. Aside from E2E testing, it supports Unit, Integration, and API testing. The simplicity of setting everything up and running tests made it popular in recent years, especially for frontend developers[8].

Cypress follows a fundamentally different approach[9] compared to other testing frameworks. Cypress runs in the browser, whereas most automation tools run outside the browser and execute remote commands to control the browser through either the browser drivers or the browser's API. This approach enables Cypress to use features that are usually not accessible by other testing frameworks, like interacting with the application directly or manipulating the DOM and the network requests. For this reason, everything regarding the browser itself is a much better experience in Cypress. Yet, concerns outside of the browser (particularly the browser tab) may take extra work or not be possible. For example, starting servers or communicating with outside processes are more complicated.

[7] https://www.selenium.dev/documentation/test_practices/encouraged/improved_reporting/, Accessed: 18.08.2023

[8] https://www.cypress.io/blog/2020/03/06/guest-post-modern-web-testing-with-cypress/, Accessed: 19.08.2023

[9] https://docs.cypress.io/guides/overview/key-differences, Accessed: 06.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Advantages and Disadvantages

The advantages and disadvantages of Cypress can be summarized as follows [Pat23, Gag23]:

+ **Easy Setup and Use:** Cypress focuses on a user-friendly and simple testing experience. It provides an intuitive API enabling comfortable writing tests together with the built-in GUI for easy test execution and debugging.

+ **Robust and Reliable:** Running directly in the browser allows Cypress to run the test cases intelligently. It can auto-wait before triggering an action, making tests more reliable. It can also retry assertions in cases when there are network issues or the UI is in a transient state.

+ **Performant Test Execution:** Cypress runs directly in the browser together on the same level as the application. This approach provides faster feedback and reduces the overall time required to run the test suite.

+ **Debugging Experience:** Cypress has a unique time travel functionality for debugging. Time travel allows the users to step through each step of the execution to inspect the application state at that time to identify issues.

+ **Full Control:** The tests run together with the application, allowing full access to the browser and the application. This approach enables full control over the application. It can modify the application state or network requests and interact with frontend frameworks like Angular and React.

- **Architecture Limitation:** Cypress does not allow to interact with multiple tabs simultaneously[10] and will never allow it. There is also no support for multiple browser instances. Although there are some workarounds, it still poses a limitation for more complex scenarios. Finally, solving operating system level tasks is also complicated.

- **Limited Language Support:** As the tests run directly next to the application in the browser, only JavaScript/TypeScript is supported. This factor can limit developers as they can have Java or C# backgrounds.

- **Complicated Local Parallel Execution:** Parallel testing is complicated. Cypress allows running the tests in parallel on different machines. Yet, it is not easily possible to run the tests locally in parallel. Workarounds are utilizing third party libraries and workarounds with docker containers, but this approach complicates the test setup.

[10] `https://docs.cypress.io/guides/references/trade-offs`, Accessed: 06.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Nevertheless, Cypress is still a very popular testing framework due to its unique approach to testing and the developer-friendly and easy-to-setup approach. It provides fast execution and powerful features as the tests run at the same level as the application, making it a compelling choice.

3.3.4 Playwright

Playwright[11] is an open-source browser automation and testing framework developed by Microsoft and released in 2020. It is similar to Puppeteer[12], another web testing framework developed by Google, which only focuses on Chromium-based browsers. It was developed to address the limitations other automation frameworks like Puppeteer and Selenium face. For example, Selenium is mature and used in various projects. Yet, it is not robust, and the setup process can be complex, and Puppeteer only supports Chromium browsers. To overcome those drawbacks, Playwright automates different browsers through a single API and interacts with the browsers directly with the DevTools protocol, similar to Selenium, but with the benefits of the underlying protocol. This approach enables Playwright low-level access to the browser and allows more reliable and stable tests by handling automatic race conditions and using smart waiting mechanisms. Although CDP is not implemented everywhere, Playwright still enables cross-browser support. Playwright is shipped together with patched versions of WebKit and Firefox browsers in which the DevTools protocol is extended to work with the Playwright API.

As the architecture introduced in Subsection 3.1.2, Playwright runs outside the browser like Selenium but communicates with the browser through the DevTools protocol. Some characteristics of the WebDriver protocol and the native approach accompany Playwright, which allows it to use features of both approaches. In Selenium, the clients communicate with the browser drivers through HTTP. In Playwright, the client communicates with the Playwright server, and the Playwright server controls the browser instances[13]. The communication between the client and the server happens through a WebSocket connection. WebSockets increase the performance as the client establishes a persistent connection with the server, which stays open until one side closes it. All the requests and responses are tunneled through this connection. WebSockets allow bi-directional communication and improve performance as the connection stays open. In the end, based on the request sent, the server communicates through the DevTools protocol (CDP) with the browsers. Playwright is free from the typical in-process limitations that automation frameworks like Cypress have, and using the browser's API still allows a high degree of freedom, even if it is not as powerful as running the test cases in the browser directly.

Advantages and Disadvantages

Playwright has the following advantages and disadvantages [Pat23, Gag23]:

[11] `https://playwright.dev/`, Accessed 19.08.2023

[12] `https://pptr.dev/`, Accessed: 06.08.2023

[13] `https://www.programsbuzz.com/article/playwright-architecture`, Accessed: 06.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

+ **Language Support:** Like Selenium, Playwright supports various programming languages like JavaScript (TypeScript), Python, Java, .NET.

+ **Cross-Browser Support:** Although Playwright does not use the WebDriver protocol, it still supports multiple web browsers like Chrome, Firefox, and Safari.

+ **Reliable and Stable:** As Playwright has direct access to the browser through the DevTools protocol it can use functionality that the WebDriver protocol currently can not. One such feature is the auto-wait capability which waits for network and DOM events by using the browser's native APIs resulting in more stable and resilient tests.

+ **Performant Test Execution:** The architecture of Playwright is designed for speed and efficiency. Parallel executions and utilizing modern browser features and controlling them directly reduce the overall execution times of the tests.

+ **Unique Tool Features:** Similar to Selenium, Playwright also provides a recording functionality to generate tests. Further, it provides a GUI that allows one to trace and inspect the current running tests with a time travel functionality similar to Cypress for a better debugging experience.

- **New Framework:** Playwright is a relatively new framework compared to more mature frameworks like Selenium. For this reason, it has a smaller community, and the amount of online documentation and resources is smaller. Still, Playwright is gaining popularity, and the community is growing rapidly.

- **Limited Integration:** Similary to the previous point, integrations for popular frameworks or CI/CD pipelines can be lacking or limited. For this reason, using Playwright with other frameworks or tools may require additional effort.

- **No True Cross-Browser Capability:** Playwright communicates directly with the browsers using their native API (e.g., DevTools protocol). Further, it uses patched versions of the browsers Firefox and Safari. This approach allows more interaction possibilities, but, as it is not standardized, it can break.

Playwright is relatively new but has gained traction due to its robust and reliable automation capabilities. The Playwright community works actively on extending Playwright and contributing to its growth.

3.3.5 Comparison

Selenium, Cypress, and Playwright are robust web testing frameworks, each offering unique strengths and drawbacks while automating web browsers. In the following sections, a comparative analysis of these three frameworks will be presented. This evaluation will aid in selecting the most suitable framework that aligns with the testing requirements for GLSP-based applications.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Based on the information presented thus far, the criteria outlined in Subsection 3.3.1 will now be evaluated. Table 3.1 shows an overview of the frameworks.

Table 3.1: Frameworks overview

	Selenium	**Cypress**	**Playwright**
Language/ Browser Support	Common Languages Cross Browsers	JavaScript Modern Browsers	Common Languages Modern Browsers
Automation Protocol	WebDriver	Native	DevTools
Parallelization (Performance)	Local, Distributed	Distributed	Local, Distributed
Electron Support	Possible	Integrated	Integrated
Features	+	++	+++
Popularity (Downloads)	Decreasing for JavaScript	Increasing	Increasing

Language/Browser Support

All three frameworks support all modern browsers, including Google Chrome, Mozilla Firefox, Microsoft Edge, Safari, and Electron. For programming language support, there is a difference. Selenium has support for C#, Ruby, Java, Python, and JavaScript and Playwright for JavaScript, Python, Java, and .NET. Both Selenium and Playwright allow developers flexibility in the used language. On the other hand, Cypress has support only for JavaScript. Cypress is a JavaScript-based end-to-end testing tool. The architecture limits supporting multiple languages as the test code is run directly in the browser instance which only understands JavaScript.

Parallel Execution (Performance)

Parallel execution is important for every web testing framework. Running a single test E2E takes longer compared to running a Unit test. E2E requires starting the application and different services as most of the time nothing is mocked. The required startup time and the slowness of applications generally sum up to a longer execution time. Due to this reason, it is important to have the capability to run the tests in parallel (see Chapter 5 for required time).

Selenium requires a test runner that supports parallel execution to allow it on local machines as Selenium does not include a test runner by default. Running tests in parallel is integrated into Playwright. Playwright allows the users to decide the degree of parallelism. It is possible to run different test files and test cases in parallel or sequentially on the same or a different machine and for different browsers and versions. On the other hand, by default Cypress runs test cases serially. It is complicated to run tests parallel

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

on the same machine. The easiest way to run the tests in parallel is to use cloud-based services like Cypress Cloud[14].

Electron Support

All three frameworks support Electron, but the usage details differ. While Cypress and Playwright provide clear examples of how to use Electron in their documentation, Selenium lacks such information, and consequently, it does not offer any guidance on using Electron for testing, and it is up to the testers to find the solution.

Features

This section will focus on some notable features that can impact the testing experience and the decision.

Locating and Interacting with Elements: Web elements are entities rendered on the web page. They can be seen by the users or be hidden but still available on the page. Titles, buttons, and input fields are web elements. Web elements are specified in the HTML specification and they consist of a tag name, attributes, and contents and are nodes in the DOM.

Locating elements is the capability of every automation tool. It is used to filter out elements from the DOM to allow the users to retrieve the element the user wants to interact (e.g., click, type) with. To achieve this, locators are used. A locator is an object that finds through queries web elements. This is necessary as humans interact with the page differently compared to test automation tools. Humans see the rendered entity and can scroll and click, for example, on the button and wait, but automation tools can not do that in the same way as humans. The selected frameworks do not see the rendered page usually, instead, they read the DOM and interact directly with it. That comes with a big issue. While humans can see the page and the rendered elements and, for example, make on-the-fly decisions to wait longer until the expected state (e.g., animation finished) is reached, this is harder for automation tools. Web pages are dynamic. Any interaction with any web element can change the page arbitrarily from the user's perspective. Further, it can take an unspecific amount of time to finish an action. Clicking on a button could send a request to the server and the response time and the web page being ready could vary between different executions.

In Selenium the locating and retrieving of web elements happens immediately on call. That means Selenium assumes that the web element is already loaded on the web page at the time of locating it. If the element is not available, then it will throw an exception. To overcome this problem, the users need to use different wait strategies (e.g., explicit, implicit)[15]. However, this makes the tests more fragile as the user needs to provide artificial timeouts to wait for the elements.

[14] `https://www.cypress.io/cloud/`, Accessed: 18.08.2023
[15] `https://www.selenium.dev/documentation/webdriver/waits/`, Accessed: 18.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Playwright uses auto-waiting for locating web elements and triggering actions[16]. When locating a web element, Playwright will check the current state of the web page and wait till the web element is loaded/ready and keep retrying automatically before timing out. The same applies to actions done to the retrieved web element. In this case, it does some checks depending on the action if the web element is for example attached, stable, or can receive events to ensure that the actions behave as expected. Once these checks pass, then the requested action will be performed. This approach makes tests more stable and removes the necessity to use artificial timeouts.

Cypress follows a similar approach to Playwright, it also uses auto-waiting which makes the execution more stable and decreases the flakiness during execution[17]. As Cypress has full information about the web page, it knows when the page is loaded, the network call is completed, events are fired, and if the web element is visible or any other element is covering it. This allows Cypress to pause and wait to execute any commands when any transition happens.

Flakiness: Selenium requires users to come up with solutions to overcome flakiness. Here, the flakiness arises from bad code or in the test infrastructure with different hardware. In order to overcome flakiness in Selenium, the user needs to use explicit or implicit waits to wait for a transition to finish or for the page to stabilize. Retry logic needs to be also self-implemented by the users. On the other hand, Playwright comes with solutions integrated into the framework to overcome flakiness. It will check for the state of the web element before triggering actions. Not as powerful as Cypress, Playwright can still listen to the web page's state (e.g., page loaded, navigation finished), which further improves stability. Additionally, it comes with retry logic integrated. Finally, Cypress handles flakiness very well as it has full information about the web page and has similar but more powerful features compared to Playwright.

iFrame: Web pages use iFrames to embed other web pages into the current web page. In GLSP this is used for the VS Code integration. There the diagram editor is embedded in the VS Code IDE through an iFrame. Selenium and Playwright have already built-in support for iFrames but Cypress does not. To support iFrames in Cypress, the developers need to install an extra plugin.

Popularity

This section will utilize metrics[18] retrieved from NPM and GitHub to gain insights into the popularity of the mentioned frameworks. However, it is essential to acknowledge that defining popularity solely based on these metrics can be challenging, as popularity is subjective and subject to biases [BT18, BHV16]. Hence, these metrics only provide an

[16] `https://playwright.dev/docs/actionability`, Accessed: 18.08.2023

[17] `https://learn.cypress.io/cypress-fundamentals/waiting-and-retry-ability`, Accessed: 18.08.2023

[18] The data used in this section has been collected on *31.07.2023*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

informative snapshot of the current standing of the frameworks rather than an exhaustive analysis of their actual popularity. These metrics can offer some understanding, but they by no means provide a comprehensive view of the frameworks' overall usage and impact on the development community.

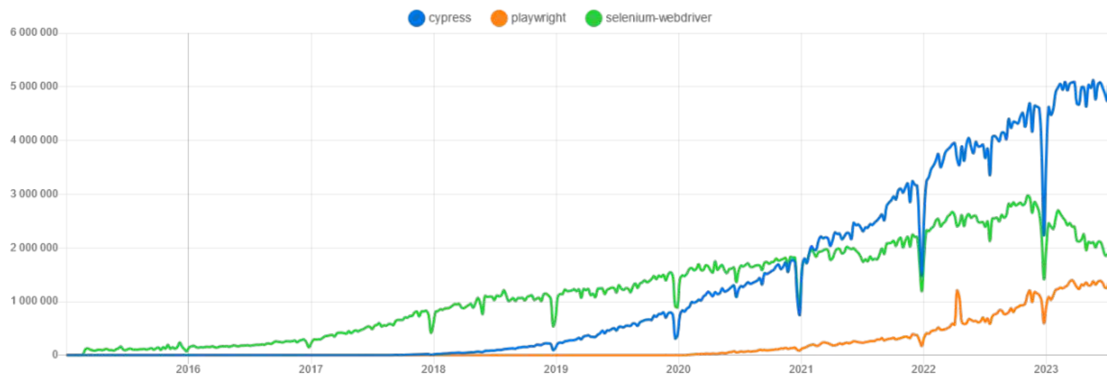

Figure 3.4: NPM weekly downloads

Figure 3.4 illustrates the weekly download trends[19] for the mentioned frameworks since their release in NPM. Notably, Selenium has a long history in the market, as visible from the graph. However, it's crucial to remember that this chart only displays the download metrics for JavaScript and omits downloads for other programming languages, such as Java and Python. Thus, the actual download count would be higher if these were included. Nevertheless, over the years, Selenium has built a stable user base and received community support in JavaScript, although in recent years, there appears to be a slight decrease in the download count. In contrast, Cypress has gained significant traction from its beginning until now, as shown by the steep download rise. This growth can be attributed to its user-friendly approach and distinct methodology, which is relatively unique for testing frameworks currently available. It is important to note that Cypress exclusively supports JavaScript; thus, the value depicted here is accurate. On the other hand, Playwright, like Cypress, has garnered attention due to its unique capabilities, leading to a steady increase in download count. While not growing as rapidly as Cypress in the early years of its release, Playwright's popularity is also rising and may overtake Selenium for JavaScript. Similar to Selenium, Playwright also supports multiple programming languages.

Table 3.2 provides an overview of the frameworks and information extracted from GitHub. The number of GitHub Stars indicates that users have marked the repository, but the exact meaning of starring is ambiguous. Users may star a repository to show appreciation or simply bookmark it for future reference. While some researchers use stars as a proxy for project popularity, it may not hold true for all projects [BT18, BHV16]. As a result, it is essential to interpret GitHub Stars cautiously when assessing a project's popularity.

[19] `https://npmtrends.com/cypress-vs-playwright-vs-selenium-webdriver`, Accessed: 31.07.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 3.2: NPM statistics

Framework	Year	Stars	Forks	Issues	Version	Weekly Downloads
Selenium	2013	27k	2.9k	209	4.10.0	1 922 555
Cypress	2017	44k	7.7k	1 400	12.17.2	5 100 196
Playwright	2020	53k	2.9k	644	1.36.2	1 406 677

Moreover, the Selenium repository contains all the other programming languages, whereas Playwright only has JavaScript in the repository. GitHub Forks can be understood as the number of users that contributed to the main repository or developers who customized the framework according to their preferences. On the other hand, GitHub Issues is also vague concerning popularity. Generally, it could mean someone requested a feature, asked a question, or anything similar. It is also used to monitor open tasks, and some projects often clean up the issues by closing them. The weekly downloads gives the numeric value of the weekly downloads available in Figure 3.4.

3.4 Discussion

Selenium, Cypress, and Playwright are all powerful testing frameworks designed to achieve the common goal of testing web applications while employing different approaches, each with unique strengths. The following discussion is subjective to some degree.

Selenium is a well-established and widely used framework favored by many legacy and newer applications. However, it can be flakier compared to Playwright and Cypress, which may complicate the process of writing test cases. The introduction of WebDriver BiDi in the future might help overcome some of these challenges. Nevertheless, working with the API currently does not feel as fluent as with the other frameworks. Additionally, handling the drivers and manually installing the test runners and assertion libraries is work the others do not require.

Cypress follows a fundamentally different approach and stands out for its uniqueness, as visible in the popularity section, benefiting from running alongside the application in the browser. This approach allows powerful features such as mocking and direct interaction with application events. Still, it comes with substantial limitations. The limitations of supporting only one browser instance or tab can be ignored in the context of GLSP. Yet, the complicated handling (i.e., or even the lack of support) for iFrames[20] and running the tests in parallel locally is a strong limitation.

On the other hand, Playwright also supports similar features and tools to Cypress. Moreover, it offers additional advantages, like built-in support for parallel execution and iFrames. By utilizing the DevTools protocol, Playwright can also work with the web page directly to some degree. Overall, it is not widely used as Cypress, and the community is

[20] https://github.com/cypress-io/cypress/issues/136, Accessed: 01.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

not as strong as the others; still, it is growing. The development is also active, and new features are introduced.

The decision falls clearly based on the comparative analysis of the three frameworks on using Playwright for GLSP. Playwright combines the strengths of Selenium and Cypress, providing a more accessible and user-friendly experience for testers. The decision to use Playwright was also supported by the GLSP developers.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

GLSP-Playwright: Automation & Testing Framework

This chapter delves into the architecture of the GLSP automation and testing framework called GLSP-Playwright by providing an in-depth exploration of the underlying structure and design principles. A well-designed architecture is crucial to enable developers to efficiently create and execute automated tests. For this reason, the testing and automation framework has been tailored to the GLSP-specific requirements. Initially, examples will demonstrate how without the use of a GLSP-specific framework automation and testing with Playwright can be done. Accordingly, it is expected from the readers, that they have a basic understanding of HTML and JavaScript. This example will be analyzed and the drawbacks will be elaborated upon and improved by the GLSP-Playwright testing framework. Subsequently, based on the insights gained from this example the challenges GLSP-Playwright faces and the concepts, design patterns, and methodologies used to overcome these challenges will be examined. Next, a more technical exploration of the key components, modules, layers, and architectural decisions that constitute the architecture will be provided. After gaining insight into the GLSP-Playwright architecture, a comparison between test scenarios with and without the application of GLSP-Playwright will be showcased. A real-world application of GLSP-Playwright will be shown in the next chapter.

Overall, this chapter aims to provide a comprehensive overview of the architecture together with the challenges it faces. By delving into the foundational design principles and technical implementations a deeper understanding of the inner workings of the framework will be acquired.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.1 Test Cases

Writing efficient test cases is critical to ensuring the quality and reliability of software applications; they are the foundation for validating the desired functionality as described in Section 2.1. For this reason, the capability and the required effort to write test cases can influence the identification of potential defects or issues.

For this reason, a collection of examples based on plain Playwright functionality is used to demonstrate how testing web applications (in particular GLSP) works. By exploring various test case examples, insights regarding the advantages and disadvantages are gathered. Each example clearly describes the test scenario, including the purpose and preconditions. Essential steps will be outlined and further elaborated. Additionally, a more technical discussion regarding the limitations will be provided where applicable. By studying and leveraging these examples, the goal is to determine the strengths and limitations of Playwright in terms of its capabilities.

For feasibility, only three examples are provided. The test cases begin with simple scenarios and progress toward more advanced situations. Before exploring the examples, a basic overview of the anatomy of a test case can be seen in Listing 4.1. Each test case consists of at least three parts:

1. Each test case (i.e., execution block) needs to be wrapped by using the function `test(...)`. It takes a description and a function that will be executed in isolation as a parameter.

2. Within the test case, actions are triggered to control and manipulate the browser. There are different actions for controlling and reading the state of the web page available.

3. Finally, an assertion is executed to validate the state of the browser. Depending on the result, the test case will either succeed or fail.

Listing 4.1: Anatomy of Playwright tests

```
1 // 1. Definition
2 test("main navigation", async ({ page }) => {
3     // 2. Actions
4     await page.goto("https://playwright.dev/");
5     // 3. Assertions
6     await expect(page).toHaveURL("https://playwright.dev/");
7 });
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.1.1 Playwright Example 1 (PE-1): Retrieving Information

The first example will discuss how to read a web page's content using Playwright. To interact with elements on the page, Playwright requires to locate the specific element first. This is done by utilizing a comprehensive API that allows testers to access and extract various types of data from the browser during test execution. Using this API, testers can retrieve text content, attributes, values, and more from the web page. The key to accessing elements is through Playwright locators. They enable the framework to find and interact with the browser's DOM elements. In essence, the locators and the respective search process use specific selectors that are CSS(-selector)[1] or XPath[2]-based strings. More detailed information regarding how the search queries are built can be taken from the documentation[3]. That means before testers can access or manipulate an element, they need to define the search query for that element on the web page.

Code

Listing 4.2 showcases a small case that reads the value of an input field to validate if it has the correct value. The CSS-based search string can be seen in line 2, where the `page.locator('...')` method processes the passed search query. In this case, Playwright will search for a web element with the ID of `'input-field'`.

Listing 4.2: Playwright: Simple locator example

```
1 test("input field should have the value hello", async ({ page
  ↪ }) => {
2     const input = page.locator('#input-field');
3     const value = await input.inputValue();
4     expect(value).toEqual('hello');
5 });
```

To demonstrate a more sophisticated case of accessing GLSP-specific elements, consider retrieving all the available options for a specific group in the tool palette. The Listing 4.3 shows this behavior. In the example, a tool palette locator is defined (line 2), followed by locating the desired group within this locator (lines 3-5). Lastly, the options locator is constructed, incorporating the previous locators along with the final locator (line 6). Now, it is possible to access all the options within the tool palette. In line 7, by using the `count()` method, it is possible to check how many options the locator would return, and in line 11, the method returns the texts of those web elements.

[1] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_selectors, Accessed: 06.08.2023

[2] https://developer.mozilla.org/en-US/docs/Web/XPath, Accessed: 06.08.2023

[3] https://playwright.dev/, Accessed: 02.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 4.3: PE-1: Retrieving information

```
1 test("returning the edge options within the tool palette",
  ↪ async ({ page }) => {
2     const toolPaletteLocator = page.locator('#sprotty
      ↪ [id$="_tool-palette"]');
3     const groupLocator =
      ↪ toolPaletteLocator.locator('.tool-group', {
4         has: page.locator('.group-header', { hasText: 'Edges'
          ↪ })
5     });
6     const optionsLocator =
      ↪ groupLocator.locator('.tool-button');
7
8     const count = await optionsLocator.count();
9     expect(count).toBe(2);
10
11    const labels = await optionsLocator.allTextContents();
12    expect(labels).toStrictEqual(['Edge', 'Weighted edge']);
13 });
```

Insights

Playwright offers powerful methods to locate specific elements on the web page using locators. However, manually writing all the search queries for commonly used elements in all test cases can be cumbersome. For instance, if the GLSP-Client modifies the tool palette to be accessible differently (by changing the selector), all test cases relying on the old selector would fail and need to be updated. Moreover, retrieving specific elements can get complicated with multiple chainings and Playwright API calls, making it necessary to abstract those away.

Another challenge is the lack of type safety. The element always has the same type when using the Playwright locator, regardless of its nature. For example, Playwright will return after accessing the tool palette or the option within always the same type. However, testers would better understand the possible behaviors and interactions of the returned object if it would be correctly typed.

4.1.2 Playwright Example 2 (PE-2): Interacting with Diagram Elements

The second example focuses on interactions. Here, the user wants to interact with the diagram editor by modifying the diagram elements. In summary, for the second example, the tester wants to achieve two goals:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

1. locate the element x that has an outgoing edge e to the element with the ID *yID*.
2. move the element x next to the bottom of the element with the ID *zID*.

The first task involves finding the specific element on the page. Playwright's locator offers basic locating options (see PE-1) but does not have built-in functionality for this particular case. Testers must implement the logic themselves to accurately locate the desired element as this is domain-specific knowledge. Moving on to the second task, Playwright allows testers to simulate different events like mouse events[4]. That enables testers to interact with the element in question using it like a user would do.

Before continuing with the detailed explanation, this test case would not be possible without the metadata (see RQ3), as mentioned earlier, already implemented in the GLSP-Client. More information regarding the metadata will be provided in Subsection 4.3.3. For now, the attributes[5] used (e.g., `'[data-svg-metadata-edge-*]'`) in the test case describe with which elements the edge is connected. The technical implementation is provided in Listing 4.4 and it is outlined as follows:

1. To identify the outgoing edge e to the diagram element with the ID *yID*, it is needed to use the metadata of the edge e provided in the SVG. In this case, the *yID* is used to search for all edges in the diagram (**reminder**, GLSP uses the term *graph* instead of *diagram* internally; see `const graphLocator`) that have a target with the same ID. This way, it is possible to find the specific outgoing edge e.
2. After obtaining the edge e and reading the source ID of the edge, it is now possible to find the diagram element x.
3. Similar to the previous step, the diagram element for z is also obtained.
4. The final step involves moving the diagram element x to the correct position. This is achieved by reading the bounding box[6] of the diagram element z, which contains information about the position, width, and height of z. Using this information, it is possible to calculate the position to which the diagram element x should be dragged.

[4] https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent, Accessed: 06.08.2023

[5] https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics, Accessed: 06.08.2023

[6] https://developer.mozilla.org/en-US/docs/Glossary/Bounding_box, Accessed: 06.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 4.4: PE-2: Interacting with diagram elements

```
1 test("moving a diagram element to a specific location", async ({ page
  ↪ }) => {
2     const graphLocator = page.locator('#sprotty svg.sprotty-graph');
3     // 1. Finding the edge e
4     const yID = '<uuid>';
5     const targetSelector =
      ↪ `[data-svg-metadata-edge-target-id="${yID}"]`;
6     const e = graphLocator.locator(`${targetSelector}`);
7     // 2. Accessing the diagram element x based on the source id
8     const xID = await
      ↪ e.getAttribute('data-svg-metadata-edge-source-id');
9     const x = graphLocator.locator(`#${xID}`);
10    expect(await x.allTextContents()).toStrictEqual(['LabelX']);
11    // 3. Accessing the diagram element z based on the id
12    const zID = '<uuid>';
13    const z = graphLocator.locator(`#${zID}`);
14    expect(await z.allTextContents()).toStrictEqual(['LabelZ']);
15    // 4. Calculating the bounding box of z and dragging x below z
16    const zBounds = (await z.boundingBox())!;
17    await x.dragTo(z, {
18        force: true,
19        sourcePosition: {
20            x: 10,
21            y: 10
22        },
23        targetPosition: {
24            x: 10,
25            y: 10 + (zBounds?.height || 0)
26        }
27    });
28
29    const xBounds = (await x.boundingBox())!;
30    expect(xBounds.x).toBe(zBounds.x);
31    expect(xBounds.y).toBe(zBounds.y + zBounds.height);
32 });
```

Insights

The first task, namely locating the diagram elements, highlights the complexity involved in accessing elements semantically. For example, in some cases, testers need to be able to use incoming and outgoing edges from nodes for their scenarios, and it is also common to read the children of a node. For this reason, there needs to be a simple way to access the diagram elements semantically without involving complex search queries by the testers.

The example also demonstrated that Playwright offers methods to address certain

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

challenges when dragging an element to a specific position. However, controlling the mouse freely (i.e., more granular) necessitates passing precise positions. Consequently, testers might need to calculate positions from diagram elements to accurately determine the required position, which can be complex. Accordingly, easing this process would be helpful.

4.1.3 Playwright Example 3 (PE-3): Creating New Elements

In this last example, the focus is on more intricate testing involving multiple components. The objective is to create a new node at a particular location. Similar to PE-1, the process involves traversing the content of the tool palette to identify the correct element to click on. Afterward, the tester can interact with the viewport to create a new element at the desired position.

Indeed, while the process may seem straightforward, a challenge arises when the tester attempts to access the newly created element. The process's asynchronous nature and the ID's randomness make it difficult to determine the new diagram element on the web page's DOM. Additionally, it is uncertain whether a single new element will be created or multiple or the element will be really placed in the precise location, as this depends on the application domain (i.e., server implementation). This unpredictability poses a challenge to correctly retrieve the newly created element.

To return back to the example, as demonstrated in PE-1 and PE-2, accessing and interacting with the web page may require multiple actions. These actions will be encapsulated within functions to simplify the test case and improve readability, as shown in Listing 4.5. For now, no concrete implementation will be provided for PE-3. The solution will be shown by using the GLSP-Playwright functionality later in this chapter. For now, only a theoretical discussion will be provided for conciseness, as implementing this functionality with Playwright would span multiple pages and would also require detailed knowledge about the Playwright API:

1. `await getOption(...)` returns the locator for the node option in the tool palette that will be used to click upon.
2. `await page.click(...)` clicks on the passed selector and the position within.
3. `await getCreatedNode(...)` is the implementation that will return the locator for the newly created node. The specific approach to identifying the new node is up to the testers. In this thesis, the algorithm involves saving a snapshot of all available diagram elements before creating a new element. Then, after triggering the creation process, the algorithm will wait until new elements become available in the diagram. Finally, the current state and the previously saved snapshot will be compared. The differences observed in both snapshots will represent the new elements. This algorithm will be further elaborated in a subsequent section of this chapter.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 4.5: PE-3: Creating new elements

```
1 test("creating a new node", async ({ page }) => {
2     // 1. [Similar to PE-1] Click on the option on the tool
        ↪ palette
3     const nodeOptionLocator = await getOption(page, 'X');
4     await nodeOptionLocator.click();
5     // 2. [Similar to PE-2] Calculate the correct position and
        ↪ click there
6     const position = ...;
7     await page.click('#sprotty svg.sprotty-graph', {
8         position
9     });
10    // 3. Retrieve the created node
11    const newNodeLocator = await getCreatedNode(page);
12 });
```

Insights

This example involves the coordination of the tool palette and the viewport. Like PE-1, the tester aims to access a particular option in the tool palette and click on it. Similar to PE-2, they will also move the mouse to a specific location within the viewport and click again. The challenge in this example lies in dealing with asynchronous behavior that occurs afterward. There is a time delay between clicking the viewport and the editor's response to show the newly created element. As a result, the test execution must wait until the editor is ready again before proceeding with the next steps. This synchronization is crucial to ensure the accuracy and reliability of the test case. Accordingly, accessing the created element is also no trivial task because the element can be placed anywhere in the diagram and, accordingly, in the DOM (i.e., more precisely in the SVG).

4.1.4 Summary

These three examples demonstrate the increasing complexity encountered by more advanced test cases. While accessing and reading content from the browser can be easily achieved, it still poses a challenge to the testers. They need to construct search queries to locate specific elements on the web page. The exact search string or content may also be required across multiple test cases. This highlights the importance of being able to reuse them. Unfortunately, the Playwright locator only offers low-level functionality for reading the DOM. As shown in PE-2, testers must implement more intricate operations themselves. Notably, the content returned by the Playwright locator lacks semantic typing, which means it does not provide information about the nature of the accessed element. This information is necessary to determine whether the underlying element is a tool palette or a diagram element. In the case of a diagram element, there could

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

also be variations. By returning a correctly typed object, it allows the testers to reuse functionalities and improve the testing experience. Also, it enables one to work with the diagram elements in more detail (i.e., reading the edges without manually querying) instead of at a generic level.

In addition to reading web page content, testers simulate user actions, as demonstrated. While Playwright offers basic support for user interactions, performing more complex scenarios with specific semantics, such as clicking on a particular option in a tool palette, often necessitates multiple steps. Providing semantic interactions tailored to the application's specific requirements enables more precise and meaningful interactions during testing instead of manually doing it. Moreover, interactions with the browser are inherently asynchronous. Triggering a change may require some time to process, as explained in PE-3. Such cases necessitate custom wait operations to enable stable test executions. Finally, algorithms specific to GLSP are necessary to handle specific circumstances, such as retrieving the last created element.

It is important to note that the examples presented here do not cover all possible scenarios, such as preparing the tool platform environment or navigating through the editor to open the correct file to edit it. They are also essential and give additional challenges in testing web applications.

4.2 Challenges

By gaining those valuable insights, it was possible to determine various challenges for a specialized GLSP testing framework. These challenges arise, especially for GLSP, from the complexity of graphical modeling tools, diverse browser and browser-like environments, and the necessity for a robust and extensible solution. There are two problem categories for GLSP-Playwright, namely inherent challenges generic frameworks face and GLSP-specific challenges. Different approaches are required to address these challenges. In the following sections, the key challenges encountered will be elaborated upon. It encompasses both technical and high-level aspects.

4.2.1 Framework-specific Challenges

Framework-specific challenges arise from the ever-evolving nature of software such as the rapid pace of technological advancements, changing requirements, and increasing features. This nature makes it mandatory to strike a balance between the flexibility and robustness of the testing framework.

Extensibility: Extensibility is the ability to incorporate new features or functionality. In practice, that means that the framework has to grow together with GLSP in such a way without fundamentally changing its core components. An extensible framework has a modular architecture, providing clear extension points and well-defined interfaces.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

This capability allows developers to adapt the framework or add new capabilities without much effort.

Maintainability: Maintainability refers to the easiness with which the framework can be understood, modified, and repaired over time due to changing requirements. A non-maintainable code is hard to work with. As with extensibility, a maintainable framework is designed to be clean and modular, adhering to best practices and following well-established architecture patterns. Maintainable code makes it easier to diagnose and fix issues. Adding new features or refactoring code without regressions is also possible. Maintainability improves the long-term stability and quality of the framework.

Scalability and Performance: Both aspects focus on resource utilization to have efficient test execution. As test cases grow in size and complexity, the framework must be able to handle a multitude of tests and their complex interactions. This can be done by utilizing parallel execution and/or distributed environments.

Stability: For testing frameworks it is essential to reach the same answer for a test case in the same context. Executing the same test should not deliver different reports. The easiest way to solve this problem is to have the testing framework validate the current state of the application before triggering new actions.

4.2.2 GLSP-specific Challenges

GLSP-specific challenges are faced due to the diversity and complexity of diagram editors. Diagram editors comprise various layers, components, integrations, and interactions. A testing framework must be capable of handling this complexity and provide mechanisms to ease the testing process.

Diagram Elements: Diagrams involve intricate and complex representations with different possible shapes, styles, and interconnections. Validating the correctness of those diagram elements and their relationships is more challenging than testing other parts of web applications. Tools perceive the diagram differently than humans. Humans see the diagram visually and can interpret it, whereas most browser automation tools and, consequently, web testing frameworks see the unrendered elements resulting in a significant challenge. In addition, usually, diagram elements do not have any information that describes them to make them understandable to automation tools.

Highly Dynamic: Diagram editors are highly interactive, allowing users to zoom, pan, drag and drop or interact with different elements and data points. Verifying the behavior and responsiveness of those features by simulating user interactions is challenging.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Tool-Platforms: GLSP runs on different platforms, which introduces subtle differences in how to start and test the editor. Testing the GLSP editor in Theia is a different experience compared to testing it in VS Code or on a web page.

4.3 Design Principles

The upcoming sections present key concepts aimed at addressing the aforementioned challenges. They draw from established practices in web testing frameworks and software engineering, as the issues related to developing maintainable and extendable frameworks are pervasive. By incorporating these concepts into the design and implementation of the GLSP-Playwright framework, it is possible to mitigate the mentioned challenges and enhance the overall quality. The sections delve into these concepts and explore how they can be applied to address the specific challenges.

4.3.1 Page Objects

The utilization of page objects as a design pattern within test automation is known for its ability to enhance the manageability, comprehensibility, and adaptability of various tests[7]. This pattern offers a systematic approach to arranging and presenting interactive elements within a web page or application. Precisely, in this thesis, a page object embodies a class equipped with essential details to locate elements within the DOM, while encapsulating the actions and functionalities of distinct components within the web application, which can be executed by the testers. Consequently, it serves as an intermediary layer, providing an abstraction between the tests and the web application. The most significant benefit is that page objects separate the test logic from the user interface implementation. By doing so, the tests focus only on the test scenarios while the respective implementation is provided in the page object. Thus, the tests only focus on high-level interactions enabling more readable tests, while the page objects focus on low-level interactions to control the browser to trigger the expected behavior.

Overall, they allow modularizing the code base by organizing related elements and interactions into separate classes allowing the test cases to interact with the page objects instead of directly manipulating the UI elements as illustrated in Figure 4.1. Further, providing an abstraction layer makes the tests more stable. Changes done to the underlying web editor primarily impact the relevant page objects. Changes like updated locators or modified behavior need to be updated in a centralized place rather than multiple test files, reducing the maintenance effort [LCRT13]. Finally, page objects also allow separation of concerns. The UI details are moved from the tests into the page object. Tests only focus on the scenario and assertions, while the page objects handle the details. Page objects are a powerful design pattern for test automation. It promotes maintainability and reusability and helps to write robust and efficient tests by decoupling the UI implementation from the test scenario [LCRS13].

[7] `https://martinfowler.com/bliki/PageObject.html`, Accessed: 07.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

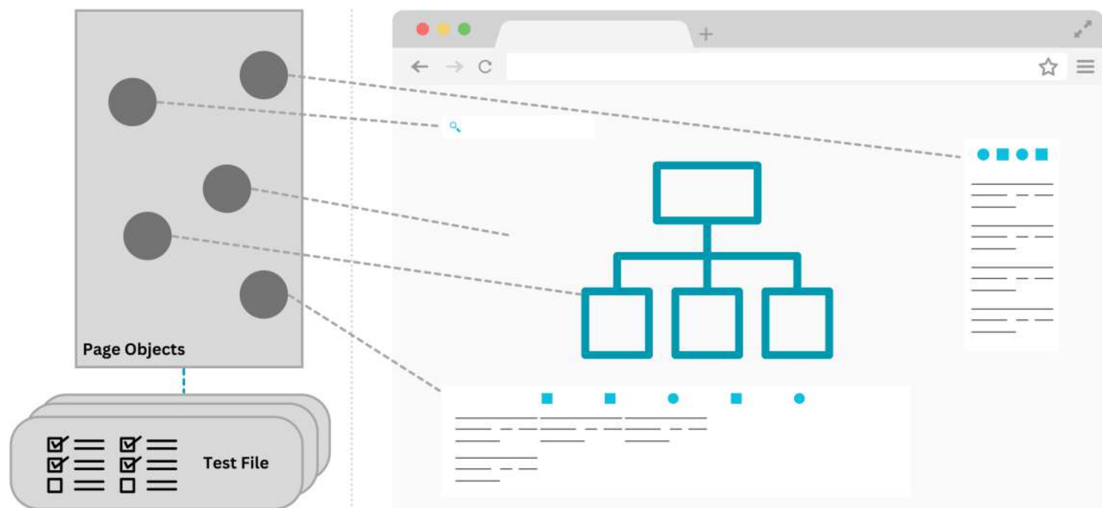

Figure 4.1: Page objects

Usage

Page objects provide a valuable technique for streamlining test cases by enabling testers to concentrate solely on the scenarios. They do not have to entangle in the intricacies of low-level framework interactions. To illustrate this improvement, the example PE-2 introduced earlier in Section 4.1 will be enhanced. The initial examples suffered from leakage of low-level actions into the test scenarios which resulted in unnecessary complexity. While it is not always possible to completely eliminate low-level functionality from test scenarios, it should be minimized whenever possible.

Listing 4.6 showcases the improved test cases. One advantage of using page objects is already visible. The test cases are now more straightforward, and the readability is also enhanced. The required search strings and logic to retrieve the data are hidden in the class `ToolPalette`. Testers can now introduce new methods specific to the tool palette within the class and also reuse them on different test cases. For example, the method `getOptionLabelsByGroupText(...)` implements the logic to return all labels in a specific options group in the tool palette. With this approach, testers can now write more reusable, robust, and maintainable scenarios by concealing the required low-level interactions with the browser.

4.3.2 Inheritance & Mixin

There are multiple ways to achieve code reuse and to promote modularity. The traditional way is inheritance. In object-oriented programming, inheritance allows reusing code from

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 4.6: Page object application for PE-2

```
1 test("returning the edge options within the tool palette",
  ↪  async ({ page }) => {
2     const toolPalette = new ToolPalette(page);
3     const labels =
      ↪  toolPalette.getOptionLabelsByGroupText("Edges")
4
5     expect(labels.length).toBe(2);
6     expect(labels).toStrictEqual(['Edge', 'Weighted edge']);
7 });
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

a base class (superclass) in multiple derived classes (subclass). Aspects such as properties, methods, and behaviors are inherited by the subclass reducing code duplication. As visible, inheritance enables the developers to think in abstraction and specializations. Common aspects are put into the base class and the more specialized functionality is provided by the subclass. The ability to override and extend the behavior of the superclass to suit the specific requirements enables the developers to establish a hierarchical view between the classes promoting a clear structure making it more organized and easier to understand. Depending on the programming language, inheritance can be also polymorph, meaning that the subclass can have multiple different superclasses.

The use of mixins[8] is not as widely adapted as inheritance. Mixins refer to a technique that allows code reusability across multiple classes without the need for inheritance. Fundamentally, mixins are classes or modules that consist of properties, methods, and behaviors that represent a specific aspect or feature encapsulating all the related information similar to superclasses in inheritance with two distinct differences. Mixins are partial classes and there is no hierarchy. The former means, that they only describe an aspect and not the whole. Whereas the latter defines that mixins can be applied to unrelated target classes. In contrast, inheritance introduces relatedness between classes (i.e., hierarchy).

Inheritance and mixins seem similar, but they follow a different approach.

- **Relationship:** Inheritance establishes an *"is-a"* relationship. Fundamentally introducing a parent-child relationship as different aspects are inherited by the child. On the other hand, mixins do not introduce an *"is-a"* relationship. The specific functionality of a mixin can be incorporated into multiple unrelated classes.
- **Hierarchy:** Traditional inheritance hierarchies tend to become complex and deep with increasing subclasses. This introduces issues like tightly coupled classes which makes refactoring hard. Mixins do not require a hierarchy. They are flexible and

[8] `https://www.patterns.dev/posts/mixin-pattern`, Accessed: 07.08.2023

modular classes that apply the functionality to the target. However, that does not mean that they are the solution for everything. If wrongly used, they can also make the code more complex than necessary.

- **Code Reuse:** It is possible to reuse code within the hierarchy, however, all of the properties and behaviors are inherited by the children. That means if a superclass in the hierarchy is changed, all of the subclasses will have the change. This can be an advantage and disadvantage at the same time. If the hierarchy is not well designed or deep, this behavior can introduce bugs or unexpected behavior. In contrast, mixins enable a more fine-grained way. They can encapsulate specific functionalities and mix them into the different classes independently allowing more modular and targeted code reuse.

Both inheritance and mixins are powerful concepts and are used widely. They have both distinct advantages and disadvantages. Inheritance establishes a hierarchy between classes where the parent-child and the *"is-a"* relationship is important. Mixins provide modularized functionality to the classes by applying them directly to **unrelated** classes providing greater flexibility.

Technical Outlook

This section will elaborate on how mixins are applied in JavaScript[9]. As previously defined, mixins enable the developers to apply behavior to classes without relying on traditional inheritance. In JavaScript, it is possible to modify class definitions at runtime, which allows for arbitrary changes to the class hierarchy and definition. A simple way to apply mixins is by using the `Object.assign(...)` method, which allows copying method definitions from one object to another. In the provided Listing 4.7, an object called `sayWelcomeMixin` with methods and a class `User` with only a name property is defined. By using `Object.assign(...)`, it is now possible to modify the `User` class at runtime and add the methods from the `sayWelcomeMixin` object to it. This way, the `User` class retrieves the functionality defined in `sayWelcomeMixin` as shown in line 12. Overall, this approach only works because JavaScript has no type safety and the access is not validated (i.e., it throws an error to runtime if something does not work as intended). This huge issue will be improved later in Subsection 4.5.2.

Usage

Page objects provided a way to conceal the implementation for the behavior that should be triggered from the test cases. On the other hand, mixins focus on providing the same behavior to a diverse set of classes (i.e., page objects). For better understanding, assume the existence of two different diagram elements x and y in a diagram on the web page with the respective page objects PX and PY. Three conditions are applied for the sake of this example:

[9] `https://www.w3docs.com/learn-javascript/mixins.html`, Accessed: 03.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 4.7: Mixins in JavaScript

```
let sayWelcomeMixin = {
  sayWelcome() {
    console.log(`Welcome ${this.userName}`);
  }
};
class User {
  constructor(userName) {
    this.userName = userName;
  }
}
Object.assign(User.prototype, sayWelcomeMixin);
new User("Haydar").sayWelcome(); // Welcome Haydar!
```

- First, the diagram elements x and y have a disjoint set of functionalities.
- Second, the page objects PX and PY exposes methods to trigger those functionalities.
- Third, the diagram elements x and y (and also page objects PX and PY) have nothing in common except that they are diagram elements (i.e., no relatedness).

Now assume, a new functionality f is introduced for the diagram elements. The necessary logic to trigger this new functionality need to be implemented in the page objects. The simple approach would be to use inheritance for the page objects and to implement the new functionality f in the superclass. However, that would introduce a relationship between x and y, which should not happen. In those cases where a diverse set of classes exists and behavior needs to be added to them, mixins can be helpful.

In this thesis, mixins serve as a means to implement various behaviors, ranging from simple actions like dragging, renaming, and deleting, to more advanced functionalities offered by GLSP. By applying the already implemented mixins to the page objects, testers can significantly reduce the amount of required work to implement these common behaviors without having to develop them from scratch and it also streamlines the implementation process and behavior of a diverse set of page objects.

4.3.3 Metadata

Machines can not, in the same way, test as a tester would manually. Testers have knowledge about the application tested and domain-specific information that machines currently lack. For example, testers can recognize diagram elements and their structure and then interact with them. The same can not be easily said for machines. For this

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

reason, metadata can help. Metadata is extra information provided to give more relevant knowledge regarding the underlying data. This extra information provides knowledge like what the rendered element is or with which elements the edge is connected. With that extra information, the testing framework can understand the structure of the diagram.

GLSP-Client

The illustration in Figure 4.2 demonstrates how metadata is implemented in the GLSP-Client. On the left side, the SVG representation of a diagram element without the metadata is visible. However, it is not straightforward to comprehend the meaning or purpose of the SVG from this representation alone. On the right side of the illustration, the same SVG with the addition of metadata can be observed. The metadata is integrated into the DOM using attributes that reveal the underlying model structure of the SVG, providing a clearer understanding of its purpose and components. For instance, the depicted SVG is now marked that it is of type `"task:manual"` and that it consists of two child elements, namely of an `"icon"` and `"label:heading"`.

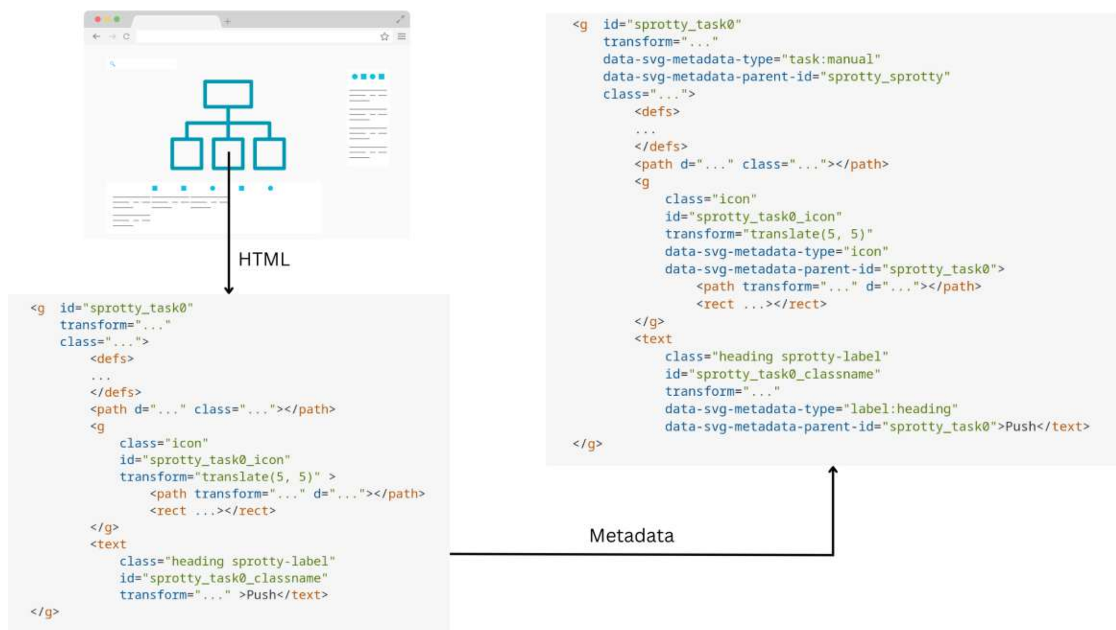

Figure 4.2: Metadata for diagram elements

Overall, five new attributes with different objectives were introduced to accomplish this:

- **data-svg-metadata-api:** The model root has this attribute to emphasize that the current SVG supports the implemented SVG metadata. Browser automation frameworks can check the existence of this tag to either continue or fail in its absence. Moreover, it enables locating the starting point of the diagram.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- **data-svg-metadata-type:** Every diagram element has a type defined by the GLSP-Server. Based on the type, the diagram element is rendered in the client. This type is not accessible by third parties, but it is essential to understand the structure of the model. For this reason, the used type is also now exposed in the diagram.
- **data-svg-metadata-parent-id:** The ID of the diagram elements was already accessible in the DOM, but the parent ID was not. The parent ID of the diagram element is necessary for cases when the diagram element consists of nested elements.
- **data-svg-metadata-edge-source-id and data-svg-metadata-edge-target-id:** The relationship between parent and child diagram elements can be easily determined by only checking if some element is nested. For edges, this is more challenging. The created edge can be placed anywhere in the SVG and be rendered as connected to a diagram element. Both attributes provide secure information regarding which elements the edge is connected to.

With those attributes, the GLSP testing framework has sufficient metadata on how the diagram is structured and what the diagram element depicts.

Usage

PE-2 has highlighted the challenges in identifying diagram elements within a diagram. While using the ID of a diagram element for lookup operations may be feasible for some cases, it alone is insufficient usually. Accordingly, the ID and the SVG alone are not enough to understand the relationships between elements and their boundaries, especially for edges. This lack of information hampers decision-making and makes tests more prone to fragility or even impossible (see PE-2).

To address this issue, metadata is employed to support the search process. By leveraging it, testers can obtain more precise and concrete search criteria. This allows for easier identification and differentiation between diagram elements. With this approach, the search process becomes more straightforward and enables more advanced queries.

4.4 GLSP-Playwright

In the following sections, the thesis will delve into the architecture of the testing framework called GLSP-Playwright. Understanding the architecture is crucial for comprehending how the framework operates and how it addresses the challenges, mentioned earlier. Accordingly, by exploring the various modules and interactions within the framework, it will be possible to gain insights into how the design principles are applied and the rationale behind the implementation choices. Theoretical aspects and also more technical information will be provided. Moreover, the examination will showcase the extensibility and scalability of the framework as well as its ability to cater different tool platforms

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

and testing scenarios. Overall, the inner workings will be elaborated upon by going into the details.

4.4.1 TypeScript

The implemented testing framework has been developed using TypeScript[10]. TypeScript is a statically typed superset of JavaScript. That means, that it introduces type safety and compile-time checks to JavaScript code. This aspect is of importance. The type safety introduced by TypeScript prevents most bugs at compile time and provides better code navigation and refactoring capabilities. The absence of type safety in JavaScript results in potential errors and bugs that might not be immediately apparent during development but can cause issues during runtime or in the code's functionality. For this reason, by utilizing TypeScript the development experience is enhanced.

In more detail, the decision to use TypeScript brings several advantages. Firstly, the already mentioned static typing feature. Static typing means the developers can define which type variables, function parameters, return values, and more should be. Type safety allows TypeScript to catch errors and bugs during development by enabling better tool support and code analysis while resulting in the same code as JavaScript. This aspect is significant for GLSP-Playwright. The application of type safety in GLSP-Playwright enables developers to know what the current worked-on element is. For example, accessing the diagram element returns always the correctly typed page object and not a generic Playwright locator (see Section 4.1). Accordingly, to prevent invalid states, the testing framework validates that the page object can be applied to the web element accessed where possible. In cases where there is a mismatch between the page object and the element on the web page particularly for accessing diagram elements, then the GLSP-Playwright framework will notify the inconsistency of the developers and fail the test.

Secondly, the TypeScripts tooling and language services facilitate better code organization and documentation. Finally, new proposed features to the JavaScript language are faster incorporated compared to plain JavaScript. The reason for that is that JavaScript is standardized and browser engines need to implement those standards before it can be used. Yet, TypeScript can provide support for proposed and still not implemented language features in browsers as it can implement all stable language feature proposals as soon as possible internally by providing an implementation independently of the browsers. This means that the TypeScript compiler will, depending on the target JavaScript version and browser version, enable developers to use those new language features without having to wait that the browser engines implement it.

Overall, by embracing TypeScript, the testing framework commits to leveraging modern development practices and enhancing the development experience.

[10] `https://www.typescriptlang.org/`, Accessed: 07.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.4.2 Open Source

GLSP-Playwright has already been released as an open-source project under the eclipse-glsp umbrella[11]. The decision to release the testing framework as an open-source project brings numerous benefits. Firstly, the source code has been reviewed and approved by the GLSP authors. Secondly, it encourages community engagement and participation. In accordance, a broader range of perspectives contributes to a healthy framework evolution. Moreover, the overall enhancement by faster bug fixes and new features by the collective effort of the community is possible. Finally, the increased peer review effort also improves code quality and reliability.

In summary, by embracing an open-source approach, the testing framework promotes collaboration and knowledge sharing. It also empowers users to customize and adapt the framework according to their specific needs by forking the public repository. By open-sourcing the testing framework, the GLSP ecosystem and community growth are endorsed.

4.5 Architecture

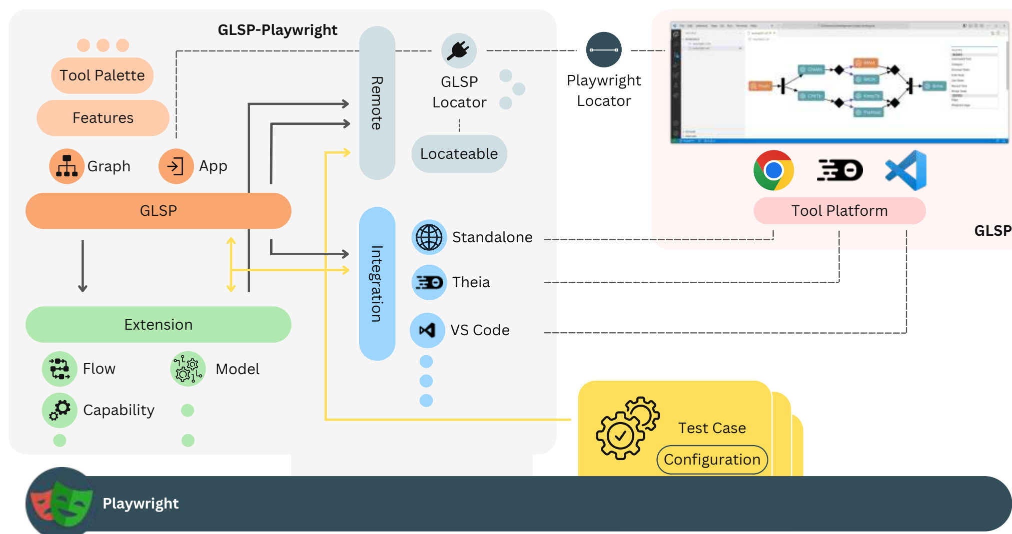

Figure 4.3: GLSP-Playwright architecture

The architecture of a software system plays a crucial role in its extensibility and maintainability. In the context of this thesis, the architecture encompasses a set of carefully designed modules. It also adopts approaches that work harmoniously with the defined

[11] https://github.com/eclipse-glsp/glsp-playwright, Accessed: 02.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

design principles to achieve the framework's goal. An overview of the architecture is illustrated in Figure 4.3

The testing framework embraces modularity and promotes a clear separation of concerns and the encapsulation of functionalities. Structure-wise, it is organized into distinct modules, each responsible for specific aspects. Within each module, code components are prepared to fulfill their responsibilities by following the single responsibility principle. Concept-wise, the concerns vary such as page object management, tool-platform handling, and diagram (i.e., graph) accessing. Moreover, the components are designed to be reusable to enable developers and testers to extend the functionality according to their preferences, and they use TypeScript features to leverage the strengths of these technologies to enhance the framework's performance and developer experience. Lastly, technical details will be provided where feasible in the respective section.

The subsequent sections will explore and provide a comprehensive understanding of the utilized modules, approaches, and key decisions made throughout the framework's development.

4.5.1 Remote Module

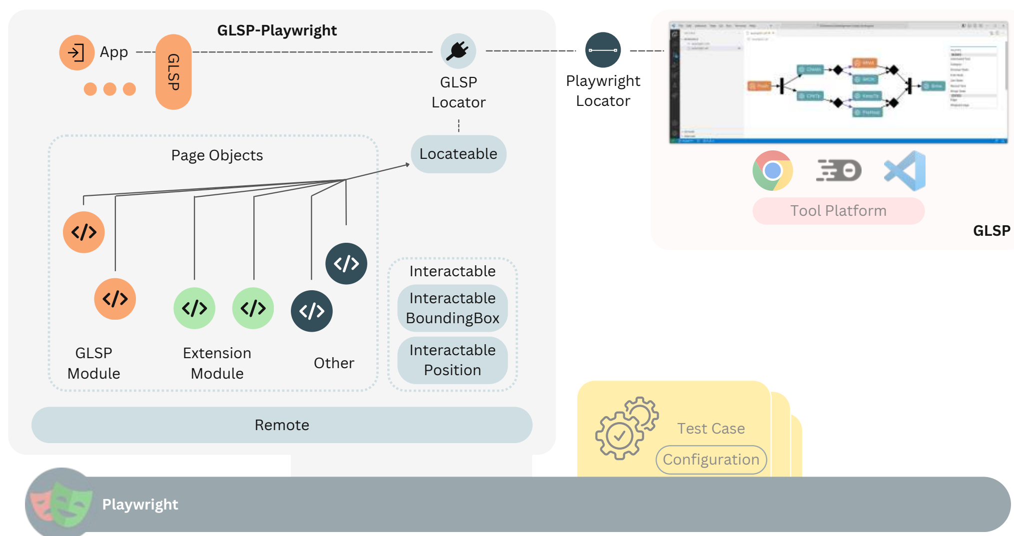

Figure 4.4: Remote module architecture

The first module presented in this thesis is known as the remote module. It mainly fulfills two purposes: handling tasks specific to interacting with the web page, and extending the Playwright locator. Additionally, the remote module provides the base class for all the page objects available in the framework. Aside, page objects for common browser functionalities, such as input fields, are also provided. Finally, the module also offers

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the means to calculate and interact with the bounding box, which the page object is responsible for. Through this module, the framework simplifies the process of accessing and manipulating elements within the **remote** browser environment. An overview of this module can be seen in Figure 4.4.

Components

Technical and background information will be provided for each core component exported from the remote module in the subsequent paragraphs.

GLSPLocator: The first component focuses on the Playwright locator. The Playwright locator is a powerful object designed to locate specific element(s) on a web page, as showcased in the examples in Section 4.1. It allows the testers to use various methods for searching and identifying elements on the web page. In more detail, the locator utilizes lazy loading, which means it references the element but does not immediately resolve it. Instead, it resolves to the actual element only when an action on the page should be performed.

In the GLSP-Playwright framework, the *GLSPLocator* is a wrapper for the aforementioned Playwright locator to provide additional functionalities currently unavailable. It allows the testers to manipulate the already passed search criteria to the locator, which is usually not possible easily. More importantly, the *GLSPLocator* bridges the Playwright and GLSP-Playwright environments. In this context, bridging means that the places that use the *GLSPLocator* gain access to the Playwright locator and, thus, to the Playwright functionalities and to the GLSP-Playwright environment called the *GLSPApp* (more on it later). This approach provides the architecture with a centralized component to interact with both frameworks. Finally, it is utilized in all page objects to ensure seamless integration and interaction between the framework functionalities.

Locateable: The base class for all the page objects, known as the *Locatable* class, acts as a basis for all page objects in the framework. Each page object inherits from the *Locatable* class, requiring the *GLSPLocator* as a constructor parameter to function properly.

Technically, the *Locatable* class defines a set of utility methods that can check the visibility state of the referenced element on the web page and allows to access its interactable bounds or positions on the web page. Further, it also offers support for specific events, such as waiting for the element to become hidden or visible. This is often necessary to ensure synchronization with the web page's state after interacting with it.

Interactable: The last component focuses on interactions. In order to simplify the process of interacting with specific positions on the web page, the framework provides a mechanism for it. The necessity for this arises due to the complexity seen in PE-2.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Depending on the usage, there are two implementations: *InteractableBoundingBox* and *InteractablePosition*. The *InteractableBoundingBox* is responsible for retrieving the bounds of an element in the browser and provides methods to access specific positions, such as its top-left or center-right coordinates. It only serves as a convenient interface for accessing and manipulating these coordinates that the testers usually use in the test scenarios. After accessing the position, it returns the positions as *InteractablePosition* objects. The *InteractablePosition* object represents the x and y coordinates of the positions defined by the *InteractableBoundingBox*. It allows testers to perform various interactions, such as clicking on the precise coordinates or relative to it. The goal is to provide a straightforward way to access and utilize the coordinates.

By incorporating these functionalities into the framework, testers can easily work with positions on the web page, improve test scenarios' readability, and reduce the effort required to calculate and interact with specific coordinates retrieved from page objects.

Discussion

The introduced remote module serves the purpose to abstract the concerns related to the remote browser and browser-like environments and addresses general issues related to the browser that are independently of the web application running on the web page.

The *GLSPLocator* is the bridge between the Playwright and the GLSP-Playwright framework. This simplifies the development and maintenance of page objects, as testers can access the necessary features and resources through the *GLSPLocator*. The Figure 4.4 illustrates this behavior. All page objects have their own *GLSPLocator* that has access to the GLSP-Playwright environment and to the Playwright locator.

The base class for page objects (i.e., *Locatable* class), enhances the maintainability of the testing framework. It provides utility methods for checking visibility states, accessing element bounds, and waiting for specific events. These built-in capabilities save testers from writing repetitive code and improve the efficiency of test case development.

The *interactable* functionalities introduced in this section further enhance the framework's usability and readability. They simplify the process of interacting with specific positions relative to elements on the web page. This is done by abstracting the complexities of position calculations. With this functionality, the testers can focus on defining their test scenarios more intuitively.

Overall, the remote module and its associated functionalities are not complex but still vital for the testing framework. They address the initial challenges faced in remote browser automation and provide the means of working with page objects in the architecture. Which page objects exist and their responsibility will be listed in the respective module.

4.5.2 Extension Module

The extension module in the framework plays a crucial role in facilitating the application of reusable behavior to page objects. The enabling concept for this is mixins, as already

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

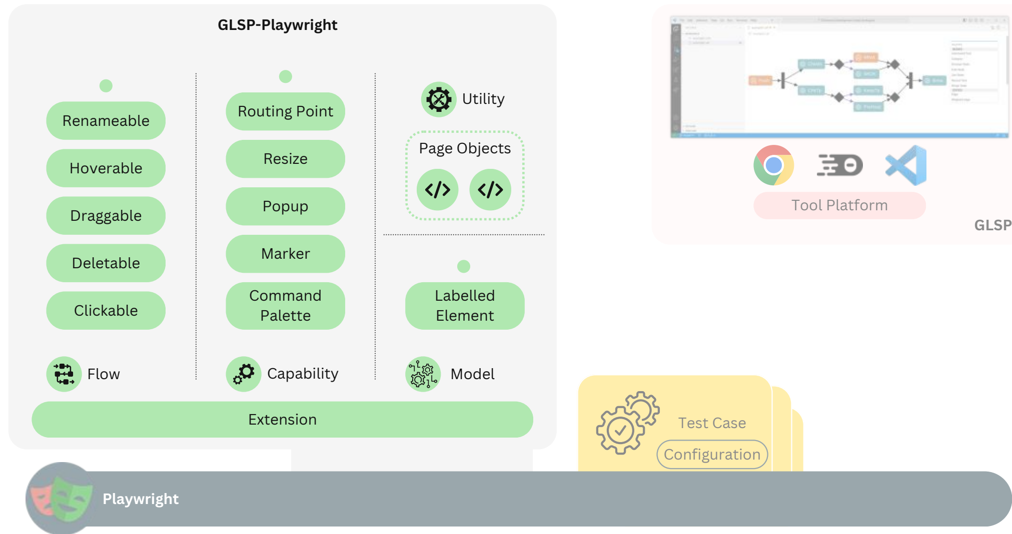

Figure 4.5: Extension module architecture

discussed in Subsection 4.3.2.

Implementing maintainable and extendable mixins can be challenging. In the context of the testing framework, ensuring type safety is one of the main factors that complicate the development process of the testing framework. TypeScript is a statically typed language but sometimes it has limitations regarding inferring types. Unlike strongly typed languages like Java, TypeScript may encounter situations where it becomes impossible to deduce the type, which results in a loss of type information which can easily happen in the context of mixins as mixins modify the type of classes by introducing new functionalities. To highlight the problem in more detail, mixins are applied on the runtime to the target class; however, it is necessary to have the type information already to compile time to work in TypeScript. Unfortunately, there is little difference between TypeScript and JavaScript when the type is lost. Therefore, it is crucial to preserve the type information when applying mixins to avoid any loss of type safety. The key factors to ensure it are as follows:

1. **Be Explicit:** The TypeScript compiler has the ability to infer most types automatically when they are deducible. However, explicitly specifying types helps the developers to prevent errors that could lead to the failure of type inference. By being explicit with types, the compiler gains access to clear and precise information and that enables it to ensure accurate type checking. For this reason, all mixins (and also the rest of the architecture) are explicitly typed to prevent any invalid deduction. Moreover, by being explicit, the code gets also documented and the developers can be sure what to expect.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

2. **Declaration and Definition Separation:** By separating the implementation from the type definition (i.e., by using interfaces), it is possible to reuse the type information in different contexts, e.g., return or method parameter types. This approach of decoupling the type definition from the implementation allows for greater flexibility and modularity in the codebase. Developers can make changes to the implementation (i.e., the behavior) of the mixin without affecting the type information of the applied target class. By following this, a more maintainable and adaptable solution is reached for mixins.

3. **Use a Builder:** Applying mixins to the target class while having type safety can be cumbersome in TypeScript and involve writing a lot of code. To simplify this process and ensure type safety, a strongly typed builder can be utilized. The builder provides a convenient way to apply mixins to the target class and accordingly, it improves the readability and type safety. By leveraging the builder, developers can apply mixins to target classes while maintaining the integrity of the type system. This approach enhances code readability and makes it easier to work with mixins in a safe and efficient manner.

The following sections will delve into the various types of mixins illustrated in Figure 4.5 and offered by the framework while providing a more detailed exploration of the technical aspects involved. Each type of mixin serves a specific purpose and brings unique functionality to the target class. By understanding the technical intricacies of these mixins, developers can effectively leverage them to enhance their page objects and extend them according to their preferences. The discussion will cover the implementation details, guidelines followed, and potential benefits of each type of mixin. Overall providing a comprehensive understanding of their role within the framework.

Flows

Flows introduce specific behavior such as deletion and renaming to the target class. They are self-contained functionalities that encompass the necessary steps required to perform in the browser. As highlighted in the previous section, flows consist of an interface (declaration) and an implementation (definition).

To illustrate this concept, consider the example of the *Renameable* flow in Listing 4.8. Initially, the interface declares a single method, `rename(...)` (line 2), which takes a new name as a parameter. The implementation can be seen in the `useRenameableFlow` function (line 5). It applies the implementation to the provided target class and consists of three key elements.

1. The generic part of the function constrains the possible base classes that can be used. This ensures that the function can only be called by classes that support both *Locateable* and *Clickable* (also a *flow*) interfaces. The TypeScript compiler will validate this constraint and prevent any invalid classes to be passed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 4.8: Renameable flow

```
1 export interface Renameable {
2     rename(newName: string): Promise<void>;
3 }
4 // 1. Renameable implementation provider
5 export function useRenameableFlow<TBase extends
  ↪ ConstructorA<Locateable & Clickable>>(Base: TBase):
  ↪ Flow<TBase, Renameable> {
6     // 2. Renameable mixin
7     abstract class Mixin extends Base implements Renameable {
8         async rename(newName: string): Promise<void> {
9             // 3. Implementation for renaming
10            const keyboard = this.page.keyboard;
11            const labelEditor = this.app.labelEditor;
12            await this.dblclick();
13            await labelEditor.waitForVisible();
14            await keyboard.type(newName);
15            await keyboard.press('Enter');
16            await labelEditor.waitForHidden();
17        }
18    }
19    // 4. Returning an explicitly typed mixin "Flow<TBase,
      ↪ Renameable>" (line 5)
20    return Mixin;
21 }
```

2. The body of the function creates a new mixin by dynamically generating a new class that inherits from the passed base class. Further, it also implements the *Renameable* interface. This approach allows the newly class to inherit all the functionality of the base class while implementing the necessary behavior for renaming the element. Since the base class is constrained to be both *Locateable* and *Clickable*, it is secured that the method can safely access the methods provided by these interfaces (e.g., `this.dblclick()`, `this.page`).

3. Here, all the required steps and the respective low-level actions are defined. The provided implementation performs a double-click on itself and then waits until `const labelEditor` (a component handling label input in SVGs) becomes visible. After that, the method utilizes the keyboard to type the new name and submits it by triggering the `'Enter'` key. Finally, there is a wait step to ensure the label editor disappears.

4. The returned mixins are explicitly defined as a `Flow<...>` typing, which is an alias

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

for the union type of the base class and the `Renameable` interface. Technically, it means that the returned class will possess all the functionality from the base class and the interface.

This approach of modifying class definitions at runtime to apply mixins differs from how mixins are used in JavaScript (see Section 4.3.2). These additional steps are necessary to ensure that the type information is preserved and no data is lost during the process[12]. Other implemented flows, such as those listed in Table 4.1, follow a similar approach and provide additional functionalities to the target classes.

Table 4.1: Available flows

Flow	API	Description
Clickable	click, dblclick	Allows clicking on page objects
Deletable	delete	Allows deleting diagram elements
Draggable	dragToAbsolutePosition, dragToRelativePosition, dragTo	Allows dragging page objects to specific positions
Hoverable	hover	Allows hovering on page objects
Renameable	rename	Allows renaming diagram elements

Capabilities

While flows only provide behavior by implementing a sequence of actions, capabilities enable access to more complicated functionalities available in GLSP. There is no difference between capabilities to flows concerning the type definition and the provider as shown in the *ResizeHandle* capability in Listing 4.9.

The main difference lies in the implemented method in the interface. Flows execute step-by-step actions and then stop. On the other hand, capabilities return primarily new page objects. Those returned page objects have the necessary logic to provide the expected functionality.

The returned page object for the capability in line 10 of Listing 4.9 can be seen in Listing 4.10 and is outlined as follows:

1. The returned utility class is not a typical page object. It is designed this way because of how resize handles work. When a user clicks on a diagram element that supports resize handles, four resize handles become visible. The users can decide on which corner the resizing should happen and this utility class allows testers to specify which corner they want to interact with.

[12] `https://www.typescriptlang.org/docs/handbook/mixins.html`, Accessed: 03.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 4.9: Resize Handle capability

```
1 export interface ResizeHandleCapability<TResizeHandles extends
  ↪ ResizeHandles = ResizeHandles> {
2     resizeHandles(): TResizeHandles;
3 }
4
5 export function useResizeHandleCapability<TBase extends
  ↪ ConstructorA<PNode & Clickable>>(
6     Base: TBase
7 ): Capability<TBase, ResizeHandleCapability> {
8     abstract class Mixin extends Base implements
      ↪ ResizeHandleCapability {
9         resizeHandles(): ResizeHandles {
10            return new ResizeHandles(this);
11        }
12    }
13
14    return Mixin;
15 }
```

2. The method creates the concrete `ResizeHandle` page object by passing the *GLSPLocator* (`this.element.locator.child(...)`) with the correct search string to determine the correct resize handle. In essence, the search string indicates that the search should be done on the child elements of the currently active element locator, and it should search for a type (see Subsection 4.3.3) in the SVG metadata that matches the specified value. The method can also handle auto-waiting, ensuring the resize handle is visible before interacting with it.

3. An example of how mixins are applied through a builder to extend the base class is presented here. In this case, only the `Draggable` mixin is applied to the target class `Locateable`. The resulting `ResizeHandleMixin` is then used as the base class for the `ResizeHandle` page object, which now inherits all the defined behaviors.

4. Finally, the dragging behavior is adjusted for the resize handle case. In essence, the page object allows testers to automatically perform necessary steps (e.g., making the resize handles accessible) if requested before dragging itself. This helps to streamline the testing process.

All of the implemented capabilities are highlighted in Table 4.2.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 4.10: Resize handle page object

```
1 // 1. Helper class
2 export class ResizeHandles {
3     ...
4     async ofKind(kind: ResizeHandleKind, options?:
      ↪ AutoWaitOptions): Promise<ResizeHandle> {
5         // 2. Locating the resize handle
6         const resizeHandle = new ResizeHandle(
7             this.element.locator.child(
8             `[${SVGMetadata.type}="..."]
9              [data-kind="${kind}"]`),
10            this,
11            kind
12        );
13        await this.autoWait(resizeHandle, options);
14        return resizeHandle;
15    }
16    ...
17 }
18 // 3. Using mixins through the builder
19 const ResizeHandleMixin =
   ↪ Mix(Locateable).flow(useDraggableFlow).build();
20 ...
21 export class ResizeHandle extends ResizeHandleMixin {
22     ...
23     // 4. Overriding the drag behaviour
24     override async dragToRelativePosition(position: Position,
      ↪ options?: AutoPrepareOptions): Promise<void> {
25         await this.autoPrepare(options);
26         await super.dragToRelativePosition(position);
27         await this.app.graph.deselect();
28     }
29     ...
30 }
```

Model

Flows and capabilities can be applied to the target class either through the `Mix` function or directly. They come with pre-defined implementations and can be readily used. On the other hand, model extensions do not have a default implementation. They are interfaces that must be applied directly to the target class, and the tester needs to provide the implementation themselves. GLSP-Playwright exposes the *PLabelledElement* extension,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Table 4.2: Available capabilities

Capability	Description
CommandPaletteCapability	Allows diagram elements to interact with the command palette
MarkerCapability	Allows diagram elements to access the marker, which will be shown on specific cases
PopupCapability	Allows diagram elements to interact with the popup
ResizeHandleCapability	Allows diagram elements (nodes) to resize
RoutingPointCapability	Allows diagram elements (edges) to modify the routing points

which allows the diagram element to expose a label property. The specific implementation of how the label is retrieved depends on the target class itself. Model extensions play a vital role in simplifying test scenarios. They allow diagram elements to have semantic information, such as labels, which allows, for example, GLSP-Playwright to search for elements based on their labels rather than their IDs. This enhances the ease and flexibility of testing diagram elements in a web application.

Discussion

The incorporation of mixins in the architecture of GLSP-Playwright brings several benefits to the testing framework.

First and foremost, mixins enable the implementation of reusable behavior for page objects. Testers can define specific functionalities, such as dragging, renaming, or deleting, as mixins, which can be easily applied to multiple page objects. This promotes code reuse and eliminates the need to duplicate code across different test cases. By encapsulating common behaviors in mixins, testers can define more maintainable tests.

Additionally, mixins enhance the flexibility and extensibility of the framework. Testers can create their own mixins to cater to specific testing requirements or integrate existing mixins provided by the framework. This modular approach allows for the selective application of desired behaviors to target classes, enabling testers to customize the functionality of page objects based on their testing needs.

Moreover, mixins separate the implementation of behavior from the type definition, ensuring that type information is preserved and maintained. This is especially important in a statically typed language like TypeScript, where type safety is crucial. By keeping the type information intact, mixins enable the TypeScript compiler to provide accurate type checking and catch potential errors during the compilation process.

Overall, by leveraging mixins, GLSP-Playwright reduces the effort required for testers to implement complex behaviors and promotes a more efficient and streamlined testing process. Testers can focus on writing high-level test scenarios without getting bogged

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

down in the details of low-level interactions. This improves productivity, enhances code maintainability, and ultimately leads to more robust and reliable testing.

4.5.3 Integrations

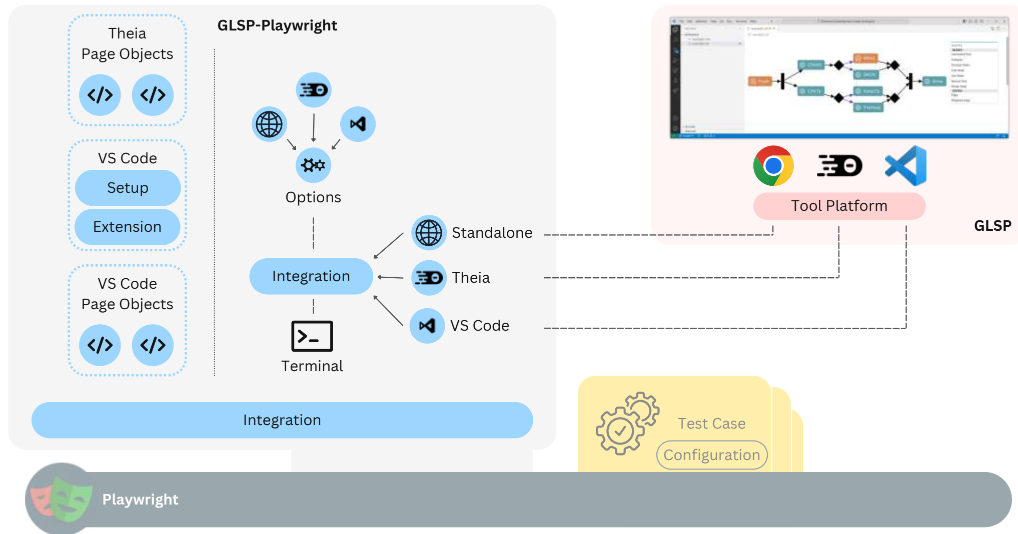

Figure 4.6: Integration module architecture

The next module of the framework focuses on different tool platforms. As already mentioned, the GLSP-Client is designed to be executed in various environments, including browsers and browser-like environments like Electron. Due to this reason, different integrations are tailored according to the specific requirements and configurations of these platforms to manage those. Integrations are objects that can control and manage the different tool platforms. This is required as Playwright, being the underlying technology, seamlessly interacts with the DOM and handles elements in browser environments. However, certain tool platforms may require additional preparations or modifications to the default behavior of Playwright. For instance, executing the test cases against Electron based environments, which is the case for VS Code, requires preparations before starting the application and also before running each test case.

Concretely, to ensure common functionality across integrations, a base class is provided. The specific integrations tailored to the respective tool platform then inherit from this base class and leverage its shared functionalities. For example, the base integration class provides methods to assert SVG metadata's existence and to modify the locating capabilities of the GLSPLocator, which can be required due to some constraints. On the other side, the specialized integration is responsible for starting and stopping the tool platform and managing other requirements regarding the tool platform.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

The GLSP-Playwright provides support for three tool platforms, namely, Standalone, Theia, and VS Code that are executed before each test case. The following sections will go into detail about what their responsibilities are and what they do. A depiction of the module can be seen in Figure 4.6.

Standalone (Page)

The first integration within the framework is designed for browsers such as Google Chrome and Mozilla Firefox. Since Playwright already offers robust capabilities for browser control, this integration does only provide quality-of-life methods. The main distinction between the Page and Standalone integration lies in what happens after the browser has been launched.

In the Page integration, there is an optional parameter called options that allows testers to specify a specific URL. That URL will be used to set the initial page of the browser by loading the web page. Simply, the options parameter is a flexible way for the testers to define the initial state of the testing environment and is defined outside the test scenarios. The reason is that different tool platforms can have other initial states independently of the test scenarios. For the Standalone integration, this options parameter is necessary.

Theia

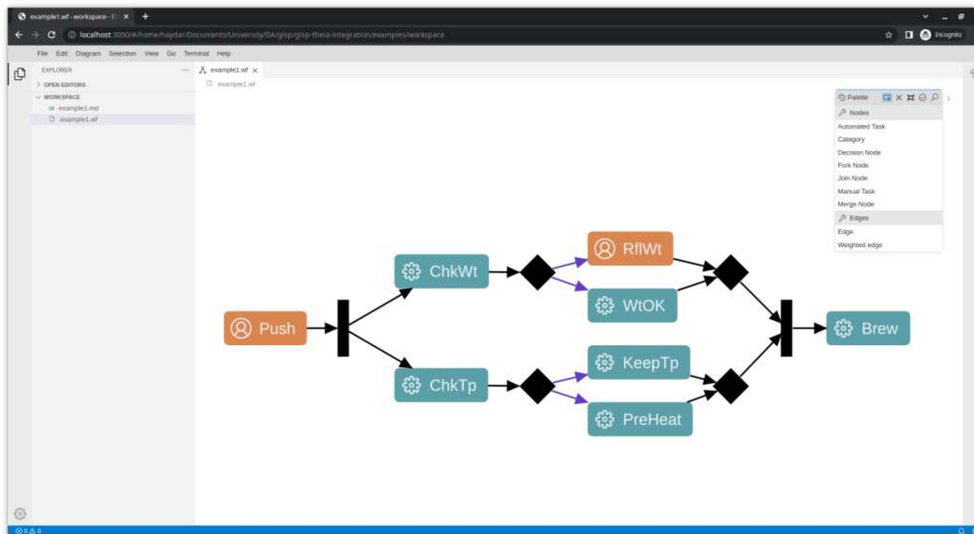

Figure 4.7: Theia Workflow example

The second integration in the framework is specifically designed for Theia. Theia is an editor similar to VS Code but running in the browser. Unfortunately, setting up the initial testing environment for Theia requires more effort compared to the previous integration.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In the previous integration, it was possible to directly open the URL and access the GLSP-Client. That is not possible in the same way for Theia as it requires additional steps. The tester needs to first open the workspace and then the specific file from the explorer within the editor, similar to other known editors. To address this challenge, the Theia integration provides Playwright-based page objects [13] that include the necessary steps to navigate and interact with workspaces and files within Theia. It is also possible to pass additional parameters containing specific information to properly configure Theia and set it to the desired initial state for testing purposes similar to the previous integration.

VS Code

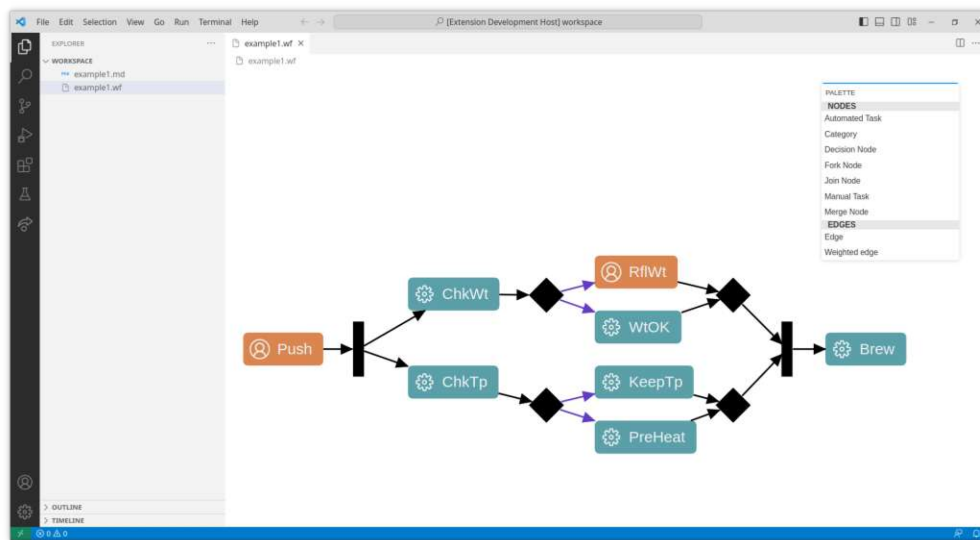

Figure 4.8: VS Code Workflow example

The final integration in the framework is designed for the VS Code tool platform. This integration operates differently compared to the previous ones. The other integrations had full support from Playwright itself, but this is not the case for VS Code. While the earlier integrations run within the browser environment, the VS Code instance is based on Electron. Playwright, by default, supports browser instances directly and Electron applications through experimental ways. For this reason, additional steps are required in Electron-based applications like VS Code.

Unlike browser instances, where Playwright automatically handles the downloading and installing of different browser versions, Electron-based applications do not follow the same process. Developers are responsible for managing the Electron application installation and configuration themselves, as Playwright does not handle it directly.

[13] `https://www.npmjs.com/package/@theia/playwright`, Accessed: 06.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Extension: One notable distinction between the Theia and VS Code tool platforms lies in how the GLSP-Client is distributed and integrated. In Theia, the developers have the flexibility to directly integrate the GLSP-Client into the existing Theia editor. Yet, in the case of VS Code, additional steps are required. The developers need to package the GLSP-Client functionality that contributes new features and capabilities to the editor into a format compatible with the editor, which is referred to as an extension[14].

This requirement poses challenges for this thesis and the purpose of working with the diagram in the VS Code tool platform. For this reason, it is necessary to install the specific extension that leverages the GLSP-Client before running the test cases.

Setup: One possible way to overcome the challenges mentioned above of installing the VS Code instance locally and the extension can be solved by manually setting up the necessary environment on the machine. However, this approach is error-prone. Having a stable testing experience is critical. For this reason, the VS Code integration provides ways to locally prepare the environment without much effort.

The integration for VS Code exposes two functionalities to prepare the local testing environment for the testers:

1. **Downloading VS Code:** The integration can detect whether a local installation of the VS Code instance is present or not. Based on this detection, it can proceed with different actions. In cases where a local VS Code installation is not found, it will download and extract an executable VS Code instance by utilizing a library[15] developed by Microsoft to a local folder in the project and make it accessible to the framework. Otherwise, it will skip the download process.
2. **Installing Extensions:** The integration can also detect whether an extension is installed or not. In cases where it is not, it will install the extension before continuing with the test execution. Further, it will also determine the installation date and the extension's creation date. If the extension has a newer creation date, then the current extension will be replaced. The same process also applies if the extension has a newer version.

Those tasks need to be done only once and before the tests are executed. For this reason, it is possible to prepare the local environment and then to execute all the test cases. Accordingly, the setup process for the local environment is demonstrated in Listing 4.11. In the test scenarios, testers utilize the `vscodeSetup` object, which manages the local environment and facilitates the download and installation of VS Code, as well as the installation of the extension. The first setup function handles the installation of the VS Code instance on the local machine and saves the path to the executable. Later, this executable path is then used to launch the VS Code instance. The second setup function

[14] `https://code.visualstudio.com/api`, Accessed: 07.08.2023
[15] `https://www.npmjs.com/package/@vscode/test-electron`, Accessed: 06.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

is responsible for installing the extension if it is not already installed. Overall, both setup functions are executed only once during the entire test run to ensure the proper configuration of the local environment.

Listing 4.11: VS Code setup

```
1  setup.describe('Setup VSCode', () => {
2      setup('Download VSCode', async ({ vscodeSetup,
       ↪  integrationOptions }) => {
3          assertVSCodeOptions(integrationOptions);
4          expect(vscodeSetup).toBeDefined();
5
6          const vscodeExecutablePath = await
           ↪  vscodeSetup!.downloadVSCode('stable');
7
8          await
           ↪  VSCodeStorage.write(integrationOptions.storagePath,
           ↪  { vscodeExecutablePath });
9      });
10
11     setup('Install extension', async ({ vscodeSetup,
       ↪  integrationOptions }) => {
12         assertVSCodeOptions(integrationOptions);
13         expect(vscodeSetup).toBeDefined();
14
15         const vscodeExecutablePath =
16             (await VSCodeStorage.read(
17                 integrationOptions.storagePath))
18             .vscodeExecutablePath;
19
20         await vscodeSetup!.install({
21             vscodeExecutablePath
22         });
23     });
24 });
```

Parallel Execution: Another important difference relates to the parallel execution of test cases. Playwright is capable of running test files and individual test cases in parallel. Usually, each test is running in a separate browser instance. The same principle applies to the VS Code tool platform, but parallel execution is more complex in this case. Ordinarily, VS Code instances cannot easily open the same folder from the command

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

line interface (CLI)[16]. That limitation presents a challenge for parallel execution, and to overcome this limitation, a workaround has been implemented by using the *–user-data-dir* argument. This argument specifies the directory where user-specific data is stored, such as settings and keybindings. By providing unique user directories for each instance, the framework tricks VS Code into starting separate instances for the same folder. Moreover, this approach ensures that each VS Code instance begins with a clean state, without any personalized modifications or custom settings.

From a technical standpoint, the VS Code integration follows a specific procedure to ensure smooth operation. First, it generates a unique *run configuration* that includes the path of the user data folder, which is stored in the temporary folders of the operating system. Afterward, when launching the VS Code instance, this run configuration is used to pass the unique data directory to the VS Code CLI. Once the VS Code instance is up and running, the integration utilizes custom-implemented page objects, similar to the approach used in Theia, to verify if the extension has started successfully. This step is crucial because the extension requires some time to become fully operational. Once the extension is confirmed to be active, the integration proceeds to open the desired file within the workspace, allowing the usual testing process to take place.

iFrame: In the context of the VS Code tool platform, there is a notable distinction in how the GLSP-Client is accessed. Unlike the other tool platforms where the GLSP-Client is readily available within the web application itself, in the case of VS Code, it is located behind two iFrames. This is a security measure enforced by VS Code. To ensure proper access to the GLSP-Client, the Playwright selectors need to be modified with a prefix that specifies the correct path to the iFrames. This prefixing of the *GLSPLocator* is done automatically before running any tests in the VS Code Integration. By applying this modification, the framework ensures that it can interact with the GLSP-Client within the VS Code environment seamlessly and in the same way as the other tool platforms.

Discussion

The integrations in the repository play a crucial role in enabling effective testing of the GLSP framework. Each integration is specifically designed to cater to the unique requirements and environments of different tool platforms such as browsers, Theia, and VS Code and glue the GLSP-Playwright environment with the testing process. Using those integrations allows the testers to seamlessly interact with the GLSP-Playwright framework and effectively validate the behavior of GLSP-based editors independently from the tool platform as the integrations mainly execute before the test cases.

Having dedicated integrations in the framework is necessary for several reasons. Firstly, it ensures that the testing framework is compatible with different tool platforms, and it can also be easily adapted for other tool platforms. This enables testers to write their test

[16] `https://code.visualstudio.com/docs/editor/command-line`, Accessed: 06.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

scenarios independently of the underlying tool platform as the integrations are concerned with the tool-platform setup, and the test cases only focus on testing the GLSP-Client.

Secondly, the integrations provide a standardized and consistent approach to setting up the necessary dependencies and configurations required for testing regarding the tool platform. They handle the installation of required software components, such as browsers or VS Code instances, and provide the necessary functionality to interact with the GLSP-Client by influencing the behavior of the GLSP-Playwright framework.

Likewise, the integrations in the framework act as reference implementations and serve as a starting point for testers to test their own distinct tool platforms. Testers can leverage the existing integrations as templates and customize them to suit their specific testing needs. They can, for example, also start their own servers or other necessary runtime requirements there. This saves time and effort in setting up the testing environment and allows testers to focus on writing test cases.

Overall, the integrations in the framework are essential components of the testing framework for GLSP. They provide the necessary glue code between the various tool platforms and the GLSP-Client to effectively test GLSP-based editors across different tool platforms by ensuring consistent and reliable testing outcomes.

4.5.4 GLSP

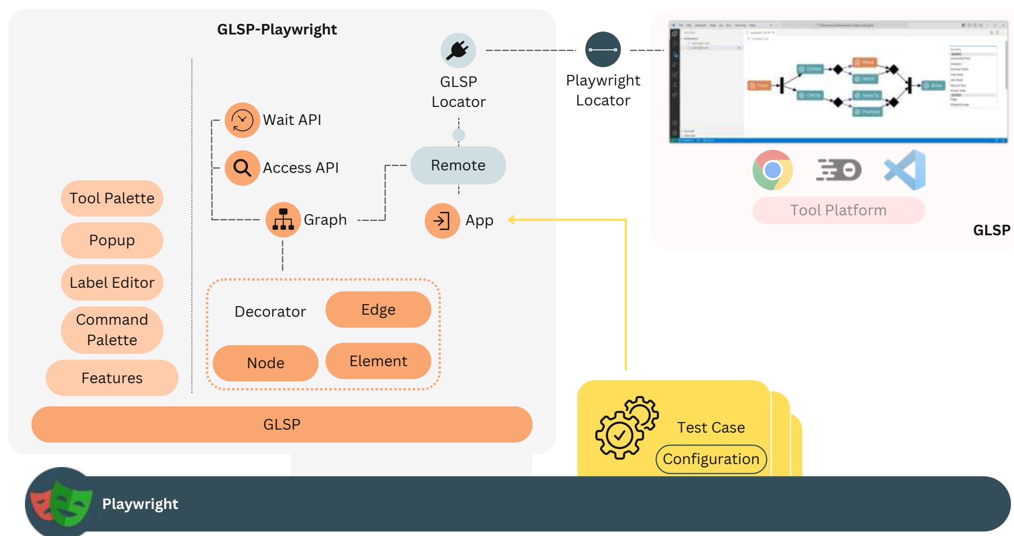

Figure 4.9: GLSP module architecture

The final module illustrated in Figure 4.9 that connects everything is called the GLSP module. It is responsible for providing the necessary functionality to test the GLSP

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

user interface based on the means already discussed and provides further the means to work with the GLSP-Playwright framework in the test scenarios. Accordingly, it includes page objects corresponding to the features discussed in Section 2.2. The GLSP module consists of three sub-modules, namely *app*, *features*, and *graph*, each with its own set of responsibilities. Since testing the *graph* part of the editor efficiently requires additional explanation, it will be covered in a dedicated section.

App

The app sub-module is responsible for providing the class *GLSPApp*. It is responsible for managing everything necessary in the GLSP-Playwright framework to enable test cases to work with the diagram editor. This is achieved by being the entry point and the central hub for creating necessary dependencies, such as the page objects (e.g., tool palette) before running the test cases. The *GLSPApp* class encapsulates various functionalities, including:

1. **Entry Point to GLSP-Playwright:** The *GLSPApp* object is created for each test execution, serving as a bridge between the test scenarios and the GLSP-Playwright environment. It provides the means to access the different aspects of the GLSP-Playwright framework in those test cases.

2. **Root *GLSPLocator* Provision:** The *GLSPApp* object also generates the root *GLSPLocator* instance, which all page objects use (see *Locatable* class). This *GLSPLocator* serves as a starting point for all those instances.

Example: The in Listing 4.12 demonstrates the process of creating a new *GLSPApp* object. The code block inside the `beforeEach(...)` method will be executed before each individual test case, which ensures that a new instance of the *GLSPApp* is used. This approach allows runs to have a clean state before each execution and to reuse the page objects, such as the tool palette, in the actual test. By leveraging this approach, testers can conveniently access and utilize the necessary page objects provided by the GLSP-Playwright framework throughout their test scenarios.

Features

The GLSP framework offers various built-in editor interface extensions, including the command palette and tool palette, and more. Depending on the functionality, those features can be triggered from various locations. For instance, the command palette can be activated from both the diagram element and the diagram, which results in different behaviors. Accordingly, to facilitate testing with those different functionalities various page objects are implemented. Each page object is dedicated to a respective functionality integrated within the GLSP framework. The following page objects are exported by the GLSP-Playwright framework can be seen in Table 4.3.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 4.12: Creating a *GLSPApp* instance

```
1 test.beforeEach(async ({ integration }) => {
2     app = await GLSPApp.load({
3         type: 'integration',
4         integration
5     });
6     toolPalette = app.toolPalette;
7 });
8 test("tool palette should return the options for [...]", async
   ↪ ({ page }) => {
9     const options = toolPalette.getOptions();
10    ...
11 });
```

All of those listed page objects have been crafted with extensibility in mind. That means, developers can customize or introduce new behaviors to the existing implementations, which have been utilized for instance in the evaluation chapter.

4.5.5 GLSP-Graph

This section of the thesis will address the primary challenge at hand: automating diagrams, specifically the automation of diagram elements within them. While the metadata offers a way to comprehend the semantics of these elements, the question of how to automate them and the algorithm of PE-3 still remains unanswered. The following subsections will delve into this question and explore the necessary tools and techniques to empower testers in effectively interacting with diagram elements. Also, the role of mixins in the overall solution and how they practically reduce the required effort will be discussed.

Diagram Elements

GLSP-Playwright classifies diagram elements into three types: *PModelElement*, *PNode*, and *PEdge*. The *PModelElement* is the base page object for all diagram elements. It provides common default behaviors for all diagram elements, for instance, it offers utility methods to access the ID and other attributes of the corresponding object on the web page. The *PNode* class, a child of *PModelElement*, represents node-based diagram elements and allows access to child elements. On the other hand, the *PEdge* class represents edges and is also a child of *PModelElements*, and it provides methods to read the source and target elements of an edge. All these operations are carefully typed to enhance the development experience. For example, if the current context for an edge permits, then the correct type will be used for the source and target node. These classes play a significant role throughout the framework and serve as base classes for testers to build their custom solutions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 4.3: Exported GLSP-UI page objects

	Description
Command Palette	The command palette, as mentioned earlier, can be accessed from two different locations within the framework. For this reason, a non-instantiable base class is utilized to implement the common behavior. Subclasses for each use case inherit from this base class and provide their specific implementation to activate the command palette from their respective locations.
Label Editor	The label editor is a specialized input field that appears in certain cases above the label of diagram elements. It enables users to modify the label text, as SVG itself does not offer the capability. The label editor can be triggered by various ways such as double-clicking on the label or by utilizing keyboard shortcuts.
Popup	The popup functionality is triggered by specific user actions, such as hovering over diagram elements. It appears as an overlay next to the diagram element and provides contextual information. For this reason, the content displayed in the popup can vary depending on the implementation and the specific use case. To overcome this problem, the implemented page object for the popup primarily focuses on providing the underlying mechanisms to interact with it rather than specifying the exact content or behavior of the popup.
Tool Palette	The page object for the tool palette enables testers to interact with the options available within it. The tool palette is divided into two sections: the toolbar and the content. The toolbar provides functionality to control the state of the editor, such as triggering validation. On the other hand, the content section contains the options for creating new nodes or edges in the diagram. Testers can click on these options to perform the desired actions.

Decorator

In order to tackle the challenge of retrieving the correct typed diagram element, decorators are employed as a solution. Decorators, in theory, wrap existing code and modify its behavior at runtime without altering the underlying functionality of objects. In essence, they extend the capabilities of an object by wrapping it with additional functionalities (i.e., function in a function). Decorators are widely used in various programming languages, either through built-in functionality like in Python, Java, and C#, or by leveraging object-oriented programming principles.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In TypeScript, experimental support for decorators[17] has been available for some time. Technically, in JavaScript, a decorator is a function that allows for customizing classes and their members in a reusable manner by acting as a higher-order function that modifies the behavior of a function, method, or class. Comparatively, mixins or inheritance introduce behavior by adding or overriding functionality, whereas decorators introduce new behavior without modifying the underlying code. Instead, they do it by wrapping it.

In this thesis, decorators are utilized to provide important metadata to various target objects, such as page objects. This metadata is crucial for the framework's architecture and serves as a counterpart to the SVG metadata introduced in the GLSP-Client in Subsection 4.3.3. The framework includes three decorators: *ModelElementMetadata*, *NodeMetadata*, and *EdgeMetadata*. These decorators share similarities and add additional properties to the page object. The most important property applied is the specific type of diagram element represented in the diagram. This information will be utilized in subsequent stages of the framework. The next section will showcase how decorators are used.

Decorators, Mixins, and Diagram Elements

This section presents a combination and demonstration of three concepts: *decorators*, *mixins*, and *diagram elements* (e.g., page objects). In practical scenarios (explained later in Chapter 5), testers won't directly use the exposed diagram elements (e.g., *PNode*) in their test cases. Instead, they will create their own custom page objects for each specific diagram element present in their unique diagram. For instance, a rectangle would have its own page object named Rectangle, while a triangle would have a separate class, and accordingly, these custom classes would extend the diagram elements exposed by the architecture (e.g., *PNode*). Yet, in the framework, the exposed base classes are intentionally kept lightweight, only containing the essential functionalities for their respective representations. To address the missing behaviors, mixins are employed. Testers can now apply pre-implemented behaviors to their custom diagram elements individually without the need to apply them to the exposed base classes beforehand. This approach prevents unused behavior from leaking into the sub-classes and allows lightweight page objects in the GLSP-Playwright framework.

The Listing 4.13, is split into three sections. First, we have the mixin definition that extends the exposed base class `PNode` with the flows to enable clicking and hovering. The capability to work with the popup is also added. The resulting base class with all the functionalities is then used as the superclass for the Rectangle page object in section three. In section 2, the `NodeMetadata` decorator is used with the `'my-custom-rectangle'` type. This means that all objects in this class will always have the `'my-custom-rectangle'` type associated with them. However, it is important to note that, at this stage, this information doesn't have significant implications

[17] `https://www.typescriptlang.org/docs/handbook/decorators.html`, Accessed: 03.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

on its own. It simply adds metadata to the class definition and will be utilized in a subsequent section for further functionality.

Listing 4.13: Decorators, mixins and diagram elements

```
// 1. Creating the base class
export const RectangleMixin = Mix(PNode)
    .flow(useClickableFlow)
    .flow(useHoverableFlow)
    .capability(usePopupCapability)
    .build();

// 2. Applying the metadata
@NodeMetadata({
    type: 'my-custom-rectangle'
})
// 3. Extending the base class
export class Rectangle extends RectangleMixin {...}
```

Access API

The next component in focus is responsible for accessing and retrieving diagram elements from the web page by returning typed page object of the diagram elements. The whole process is illustrated in Figure 4.10. This component, called *GLSPGraph*, is exposed by the graph module and contains the necessary logic to handle these tasks effectively. To achieve this, it exposes an API aligned to finding different nodes, edges, or elements in the diagram. Usually, the tester needs to provide information to correctly identify the diagram element. This information can be a selector, an expected page object (e.g., *Rectangle*, *TaskManual*), or semantic knowledge like a label (see *PLabelledElement* in Subsection 4.5.2) or any other information that can help identify diagram elements. With the available metadata and structure implemented in the architecture, testers can employ various implementations to locate their elements. Using this information, the *GLSPGraph* searches for the specified diagram element in the diagram and returns the elements that align with the expected page object (e.g., *Rectangle*, *TaskManual*). Besides, the *GLSPGraph* offers an advanced API that allows querying more complex cases, such as searching for edges with specific source types and other common query examples. Once the elements are retrieved from the diagram, the *GLSPGraph* ensures that the page object of the passed diagram element can be properly assigned to the returned element from the web page. This verification is performed by comparing the metadata available in the class with the metadata defined on the web page. In the final step, a new object that is responsible for the element on the web page based on the passed page object is created and returned to be used in the test scenarios.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

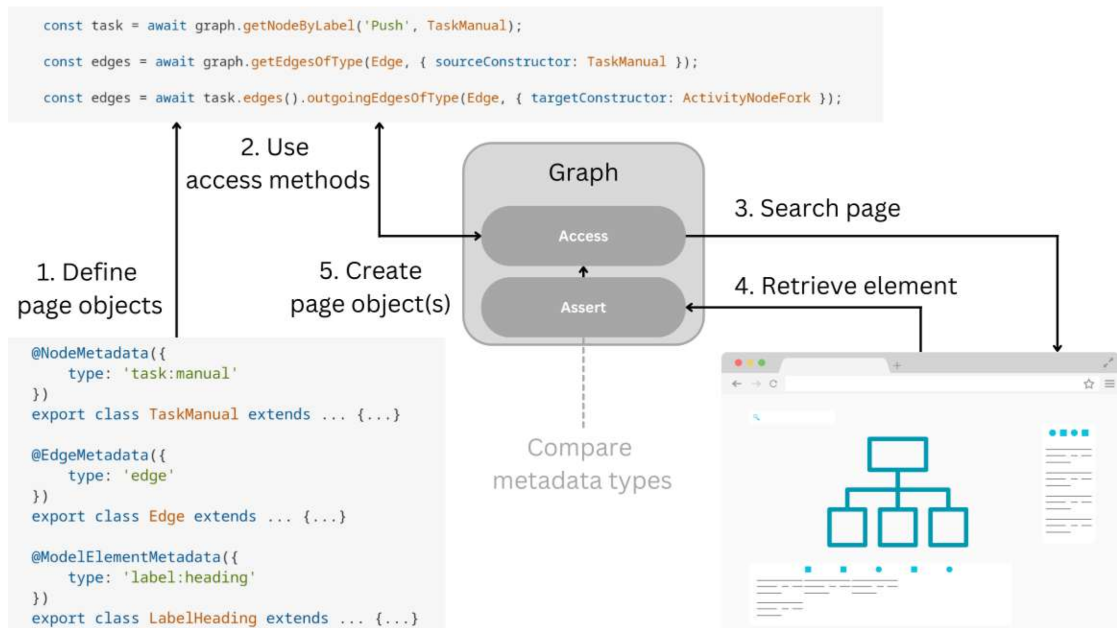

Figure 4.10: Accessing diagram elements

Wait API

In most cases, testers have prior knowledge of the expected outcome. For instance, they can wait for certain elements to become visible after navigation, indicating that the navigation is complete before proceeding with the test. In the context of GLSP, it depends on the operation. When it comes to deleting diagram elements, testers can verify whether the element's specific ID is no longer present in the SVG to confirm a successful deletion. However, in the scenario of creating new diagram elements, there are no such clear indicators. The ID of the new element cannot be determined beforehand, and the operation may only add a new entry somewhere in the SVG or create a child element, making it challenging to confirm the specific result. This raises the question of how the testing framework can determine if a new diagram element has been created and which element that is.

In this thesis, a poll and compare approach is used. Polling means repeatedly calling a function until it either succeeds or reaches a timeout. The function that will be polled is the compare function, and it is responsible for determining if there have been any changes in the diagram. To make this work, testers need to utilize specific methods provided by the *GLSPGraph* API. Now, for any scenario that wants to create a new element, it needs to go through the *GLSPGraph* API as shown in Listing 4.14. Internally, the *GLSPGraph* will determine all existing diagram elements, perform the desired steps passed as a callback, and then wait for a specific condition to be met. In the case of creating new diagram elements, the condition is an increase in the number of diagram elements, which is checked using polling. After a change has been determined, the framework will retrieve

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

all diagram elements again and compare them with each other. Hence, the elements that did not exist before are the newly created ones that will be returned. This approach ensures that the test waits for the necessary changes to occur before proceeding further.

In practice, the tester needs to pass two parameters: the page object of the diagram element that should be created and a callback that will create the element. The Listing 4.14 shows the available methods for this use case. If the user does not pass a page object, then the methods will return the IDs of the new elements.

Listing 4.14: Wait API for creation

```
1  const rectangles = await
   ↪ graph.waitForCreationOfNodeType(Rectangle, async () => {
2      // Steps to create rectangles
3  });
4  const lines = await graph.waitForCreationOfEdgeType(Line, async
   ↪ () => {
5      // Steps to create lines
6  });
7  const triangles = await graph.waitForCreationOfType(Triangle,
   ↪ async () => {
8      // Steps to create triangles
9  });
10 const circleIds = await graph.waitForCreationOfType("circle",
   ↪ async () => {
11     // Steps to create circles
12 });
```

4.6 Examples

Using the insights gathered, the test cases presented in Section 4.1 will be re-implemented with the GLSP-Playwright framework. The Playwright examples did not detail the tool platforms and only showcased small examples. To have a better overview, a new example called *GE-0: Preparations* is introduced.

4.6.1 GLSP Example 0 (GE-0): Preparations

The example in Listing 4.15 demonstrates how the GLSP-Playwright environment is initialized. The `beforeEach` block runs before each test execution. Within this block, a new `GLSPApp` instance is created, and the appropriate integration responsible for setting up the tool platform is passed as a parameter. The specific integration used is determined in a configuration file, which will be discussed in detail in Subsection 4.7.1. Now, testers

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

can access various page objects using the new app instance (lines 12-13). This structure is followed for all test cases.

Listing 4.15: GE-0: Preparations

```
// Variables
let app: GLSPApp;
let graph: GLSPGraph;
let toolPalette: GLSPToolPalette;

// Preparations
test.beforeEach(async ({ integration }) => {
    app = await GLSPApp.loadApp({
        type: 'integration',
        integration
    });
    graph = app.graph;
    toolPalette = app.toolPalette;
});

// Test case
test('...', async () => {...});

```

4.6.2 GLSP Example 1 (GE-1): Retrieving Information

The first example PE-1 faces issues with complex locator definitions and a lack of type safety. The test scenarios become harder to read due to writing low-level locating information there. These challenges are addressed in the new example presented in Listing 4.16. The new test case utilizes the page object for the tool palette, making it unnecessary for testers to have knowledge of its internal access. Even if adjustments are required, they can easily be made by overriding the page object in the GLSPApp, as GLSP-Playwright exposes all internal implementations and allows customization of every part. The exposed page object for the tool palette provides different methods for accessing various options within. For instance, testers can access specific option groups using their labels or directly access individual options within by specifying both the group and the option label. In line 5, the option group labeled 'Edges' is accessed, and subsequently, all the items in that group can be retrieved, as the returned object is also a page object.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 4.16: GE-1: Retrieving information

```
...
let toolPalette: GLSPToolPalette;
...
test("returning the edge options within the tool palette",
  ↪ async () => {
    const group = await
     ↪ toolPalette.content.toolGroupByHeaderText('Edges');
    const items = await group.items();

    const count = items.length();
    expect(count).toBe(2);

    const labels = await Promise.all(items.map(i => i.text()));
    expect(labels).toStrictEqual(['Edge', 'Weighted edge']);
});
```

Discussion

Two significant improvements have been achieved by adopting page objects in the GLSP-Playwright framework. First, the internal implementation details, such as how the tool palette is accessed, are now hidden from the test scenarios; this information is usually without importance in the test scenarios. This abstraction simplifies the test cases, making them more straightforward. This applies to simple cases like the tool palette and more complex scenarios involving components like the command palette or resize handles. By abstracting away these implementation details, the test cases become more focused on their intended functionality. Secondly, the use of page objects in the test scenarios introduces type safety. Previously, when working with locators, the test cases lacked clarity about the type of web element being accessed. With page objects, testers now have an explicit knowledge of the specific web elements they are working with. The example provided in the test case demonstrates this feature by returning page objects when accessing options in the tool palette. This allows testers to differentiate between different web elements and apply custom behaviors to specific page objects, enhancing the overall flexibility and extensibility of the framework. Overall, the adoption of page objects in the GLSP-Playwright framework improves the clarity, maintainability, and extensibility of the test cases, making them more robust and easier to manage in the long term.

4.6.3 GLSP Example 2 (GE-2): Interacting with diagram elements

In this example, two possible implementations for the test case PE-2 will be explored. The first implementation, shown in Listing 4.17, directly maps the PE-2 example. However,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

the second implementation, presented in Listing 4.18, leverages semantic knowledge to enhance the test case.

Listing 4.17: GE-2: Interacting with diagram elements

```
1 ...
2 let graph: GLSPGraph;
3 ...
4 test("moving a diagram element to a specific location", async
  ↪ () => {
5     // 1. Finding the edge e
6     const edges = await graph.getEdgesOfType(PE, {
      ↪ sourceConstructor: PX, targetSelector: '#<yID>' });
7     const e = edges[0];
8     // 2. Retrieving x
9     const x = await e.source();
10    expect(await x.label).toBe('LabelX');
11    // 3. Retrieving z
12    const z = await graph.getNodeBySelector('#<zID>', PNode);
13    expect(await z.label).toBe('LabelZ');
14    // 4. Dragging x below z
15    const zBounds = await z.bounds();
16    await x.dragToAbsolutePosition(
17        zBounds.position('bottom_left').data);
18    ...
19 });
```

In the direct mapping (Listing 4.17), the following simplifications have been made:

1. Accessing edges can be done through the `GLSPGraph` page object, which now exposes different methods to assist testers in identifying the desired edge. Testers no longer need to understand the intricacies of the metadata or how to access diagram elements with locators. In this example, the `graph` object searches the diagram by utilizing the edge type `PE` with a source page object of type `PX` and the target node with the ID `'<yID>'`. This approach allows for various options to influence the search process.

2. Once the edge `e` is retrieved, the element `x` can be directly accessed, and the framework automatically returns the correct typed page object, as it now has enough knowledge.

3. The `graph` object also allows for accessing diagram elements using selectors. The difference here is that the returned object is typed, providing better clarity.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

4. Common calculations are now abstracted away from testers. They can access specific locations of the bounding box, and by utilizing mixins applied to the page object PX they can easily drag the element x to a specific position, such as the bottom left corner of element z.

Listing 4.18: GE-2: Interacting with diagram elements (improved)

```
...
let graph: GLSPGraph;
...
test("moving a diagram element to a specific location", async
 ↪ () => {
    // 1. Finding the edge e
    const y2 = await graph.getNodeByLabel('LabelY', PNode);
    const edges2 = await y2.edges().incomingEdgesOfType(PEdge,
     ↪ {
        sourceConstructor: PNode
    });
    // 2. Retrieving x
    const x2 = await edges2[0].source();
    // 3. Retrieving z
    const z2 = await graph.getNodeByLabel('LabelZ', PNode);
    // 4. Dragging x below z
    const z2Bounds = await z2.bounds();
    await x2.dragToAbsolutePosition(
        z2Bounds.position('bottom_left').data);
    ...
});
```

The previous test case can be further improved if testers utilize semantic knowledge in their page objects as shown in Listing 4.18. For example, if they know the label instead of the ID, they can search for that element based on the label (if the page object supports it). Moreover, they can access all the incoming and outgoing edges of the node. Similar to the graph object used in the examples, those edges can also be further constrained to find a specific element. In accordance, the framework will also type the resulting objects.

Discussion

In this example, how diagram elements are accessed and interacted with was the focus. Playwright, by itself, provides low-level methods to interact with these elements. However, it lacks direct support for semantic knowledge, as demonstrated in the example PE-2. By using the GLSP-Playwright framework, testers can now access diagram elements more naturally. It is possible to directly access children or edges from the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

object without the necessity to write complicated locators or search queries. This higher-level abstraction makes the test scenarios more readable and understandable, as testers can now interact with the diagram elements using a more domain-specific approach. The same also applies to calculations and interactions. Common calculations of the bounding box are easily accessible and by utilizing mixins the testers know what the diagram element is capable of. In this example, only the drag action was demonstrated. However, if a more complex scenario is tested like utilizing the resizing handle, then the Playwright-only solution would require multitude of low-level steps in the test scenario. While with GLSP-Playwright those would be hidden away behind a method like `x.resizeHandles().ofKind('top-left').dragTo*(...)`. Overall, by employing semantic knowledge and enhancing the capabilities of the page objects, the test case becomes more intuitive and less dependent on low-level implementation details. This approach simplifies test development and makes the test scenarios more robust and maintainable.

4.6.4 GLSP Example 3 (GE-3): Creating new elements

PE-3 did not delve into the technical details of how creating a new element would be implemented with Playwright alone. Only a pseudo-code and a theoretical explanation were provided to maintain clarity and brevity. In Listing 4.19, an actual implementation using GLSP-Playwright is presented. In this example, the creation of a new node of a page object `PX` is expected to be created. The process begins with calling the `graph.waitForCreationOfNodeType(...)` method in line 6. This method will only complete once a new web element that the page object PX can be assigned to is created. The steps required to create the new element are outlined in lines 7 to 12, where the test clicks on the option `'x'` in the `'Nodes'` options group within the tool palette. Next, the test clicks to the left of the node with the label `'LabelZ'`. Following these steps, the graph will then wait for a new node on the web page for the page object `PX` to be created. Upon completion, the test case verifies the correctness of the label for the newly created node. This approach with GLSP-Playwright allows testers to wait for the creation of specific diagram elements, enabling smoother and more reliable test execution.

Discussion

The GLSP-Playwright framework simplifies detecting new elements, which would otherwise be complex and time-consuming for the testers to handle. Testers can now focus on writing straightforward test cases without worrying about the asynchronous nature of the editors. In the example provided above, it doesn't matter whether the user creates one new node or multiple nodes or uses different components like a tool palette or a command palette for the creation. Additionally, the example reuses parts of GE-1 and GE-2, showcasing the reusability and flexibility of GLSP-Playwright. In contrast, the Playwright-only solution would require testers to deeply understand the GLSP-Client and write complex selectors to find the correct web elements. This complexity is abstracted

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Listing 4.19: GE-3: Creating new elements

```
1 ...
2 let graph: GLSPGraph;
3 let toolPalette: GLSPToolPalette;
4 ...
5 test("creating a new node", async () => {
6     const nodes = await graph.waitForCreationOfNodeType(PX,
       ↪  async () => {
7         const item = await
           ↪  toolPalette.content.toolElement('Nodes', 'x');
8         await item.click();
9
10        const pz = await graph.getNodeByLabel('LabelZ', PZ);
11        const bounds = await pz.bounds();
12        await bounds.position('top_left').moveRelative(-50,
           ↪  0).click();
13    });
14    expect(nodes.length).toBe(1);
15
16    const px = nodes[0];
17    const label = await px.label;
18    expect(label).toBe('LabelX');
19 });
```

away with GLSP-Playwright, allowing testers to write test cases in a more natural and user-friendly language. This approach enhances test case maintainability and readability, contributing to a more efficient and enjoyable testing process.

4.7 Technical Outlook

This section will delve into technical aspects that haven't been covered yet, but are helpful for gaining a deeper understanding of the subject matter.

4.7.1 Configuration

The first topic will be about exploring the configuration of Playwright. In Playwright, each run is accompanied by a configuration file that is used by the test runner to determine how the tests should be executed. This configuration file contains various options, such as setting the timeout for test cases or enabling parallel execution with a specified number of threads. Additionally, the configuration can define projects, which allow for specific run configurations that apply only when explicitly called.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

GLSP-Playwright makes use of those projects to provide different settings based on the tool platform, as visible in Listing 4.20.

Listing 4.20: Playwright configuration

```
1  ...
2  const theiaIntegrationOptions: TheiaIntegrationOptions = {
3      type: 'Theia',
4      url: getDefined(process.env.THEIA_URL),
5      ...
6      workspace: '../workspace', file: 'example1.wf'
7  };
8  const vscodeIntegrationOptions: VSCodeIntegrationOptions = {
9      type: 'VSCode',
10     workspace: '../workspace', file: 'example1.wf',
11     vsixId: getDefined(process.env.VSCODE_VSIX_ID),
12     vsixPath: getDefined(process.env.VSCODE_VSIX_PATH),
13     ...
14 };
15 const config: PlaywrightTestConfig<GLSPPlaywrightOptions> = {
16     ...
17     timeout: 30 * 1000,
18     workers: process.env.CI ? 1 : undefined,
19     ...
20     projects: [
21         ...
22         {
23             name: 'theia',
24             testMatch: ['**/*.spec.js'],
25             use: {
26                 ...devices['Desktop Chrome'],
27                 integrationOptions: theiaIntegrationOptions
28             }
29         },
30         {
31             name: 'vscode',
32             testMatch: ['**/*.spec.js'],
33             dependencies: ['vscode-setup'],
34             use: {
35                 integrationOptions: vscodeIntegrationOptions
36             }
37         }
38     ]
39 };
```

Each integration, such as Theia and VS Code, has its respective option variables containing the information required for all test runs. For instance, in the case of Theia (lines 2-7), the options specify the URL for accessing Theia and the folder and file to open automatically before each test. Similarly, for VS Code (lines 8-14), the options define the workspace and file to open, as well as details about the extension to install (e.g., `vsixId`, `vsixPath`).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

These options are utilized later in the project definitions. Between lines 22-29, a project for Theia is defined, using the previously set options and specifying to use the Chrome browser for running the test cases. In the case of VS Code, the project is defined between lines 30-37, which also references the setup project responsible for locally installing the VS Code instance and the provided extension. Internally, the GLSP-Playright framework will read those passed options and, depending on them, load the different integrations to manage the tool platforms. In summary, these configurations allow for setting up and customizing the test environment for different integrations, ensuring consistency and convenience during testing.

4.7.2 Parallelization

This section will provide an overview of how parallelization is achieved. Playwright uses the concept of workers. A worker is a process that runs in parallel to other workers. By default, each test file is assigned to a worker, and each test case in this file is run sequentially by the same worker process (this can be customized).

Technically, each worker is an OS process run independently and orchestrated by the test runner. Each of those workers has identical environments and controls its own browser. It is not possible to communicate between the workers, and they are shut down on the failure of a test case.

In this thesis context, those workers' behavior is not influenced. Only the VS Code setup process requires that it is run once before any test case. This is solved by defining a project that is run before the real VS Code test cases.

TU Wien Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 5

Evaluation

In this chapter, the evaluation of the GLSP-Playwright framework based on the Workflow diagram presented in Section 2.2 will be done. The main goal of this evaluation is to assess how well the GLSP-Playwright framework fulfills the requirements and objectives set for an efficient and user-friendly web testing solution for GLSP-based editors to automate it. Yet, the intention was not to develop an exhaustive test suite with complex test scenarios that cover every aspect of it.

The evaluation process will encompass a series of key aspects, including support for GLSP-specific functionalities, and integration with different tool platforms. It will also examine how the framework handles complex interactions, asynchronous behavior, and dynamic elements within the GLSP editor environment. To carry out this evaluation, test cases will be run against the Workflow editor to identify the current ability of the GLSP-Playwright framework.

By conducting a comprehensive evaluation of the GLSP-Playwright framework, it is aimed to identify areas of improvement and potential optimizations, thus facilitating its future development and enhancing its overall usability.

5.1 Preparations

In order to enhance the testing process and align it with the Workflow diagram, several classes (see Table 5.1) have been developed as extensions of the existing GLSP-Playwright implementation. These classes are designed to customize the behavior of the testing framework, making it more closely aligned with the Workflow diagram. However, it is important to note that these extensions are not mandatory and have been only implemented to showcase the capability to extend the default behavior. It would still work without those. In essence, by employing these customized classes, testers can tailor their test scenarios to better match testing the Workflow diagram, which is the case for:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 5.1: Workflow classes

Class	Description
WorkflowApp	Overrides *GLSPApp* to return custom tool palette and the graph implementations
WorkflowToolPalette	Overrides *GLSPToolPalette* to enable a custom *ToolPaletteContent* implementation
WorkflowToolPaletteContent	Constraints which values are allowed to be passed as labels
WorkflowGraph	Overrides *GLSPGraph* to introduce custom wait behavior after creating a node

- **WorkflowToolPaletteContent**: Constraints all the methods exposed in the Page Object to only allow correct option labels to be passed as a parameter, as any string can be given to the *ToolPaletteContent* Page Object. Simply by doing this, the testers can not request an option from the tool palette that does not exist in the test scenarios. Moreover, this is done in a type safe manner.
- **WorkflowGraph**: The Workflow diagram has a custom behavior after creating a new node. It will select the node after some time. That means the default implementation is not sufficient. For this reason, the Workflow-specific implementation for the graph will additionally wait until the created node is selected before continuing the test case making it less fragile.

This can generally lead to a more seamless testing experience, improved test reliability, and simplified interactions with the GLSP editor environment.

Table 5.2: Workflow diagram elements

<table>
<tr><th></th><th>Activity-Node-Fork</th><th>Edge</th><th>Label-Heading</th><th>Task-Autmated</th><th>Task-Manual</th></tr>
<tr><td>Representation</td><td>Node</td><td>Edge</td><td>Label</td><td colspan="2">Node</td></tr>
<tr><td>Flows</td><td>-</td><td>Clickable</td><td>Clickable, Renameable</td><td colspan="2">Clickable, Draggable, Deletable, Hoverable</td></tr>
<tr><td>Capabilities</td><td>-</td><td>RoutingPoint</td><td>-</td><td colspan="2">ResizeHandle., Popup., CommandPallete, Marker</td></tr>
<tr><td>Semantic</td><td>-</td><td>-</td><td>-</td><td>Label</td><td>-</td></tr>
</table>

An excerpt of the Page Objects for the existing diagram elements in the Workflow diagram can be seen in Table 5.2. The flows and capabilities of each element are also listed there.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Those Page Objects are not optional, they are required. Every diagram has to implement a respective Page Object for the diagram element with at least the correct metadata passed by the decorator. The metadata is crucial to determining if the Page Object can be used as a representative for the diagram element on the web page. For demonstration purposes, only the *TaskManual* Page Object exposes that it has a label. This semantic information helps to identify it easier in the diagram. An example with comments of the Page Object for the *TaskManual* can be seen in Listing 5.1.

Listing 5.1: *TaskManual* definition

```
1 // Flows and capabilities of the diagram element
2 export const TaskManualMixin = Mix(PNode)
3     .flow(useClickableFlow)
4     .flow(useHoverableFlow)
5     .flow(useDeletableFlow)
6     .flow(useDraggableFlow)
7     .capability(useResizeHandleCapability)
8     .capability(usePopupCapability)
9     .capability(useCommandPaletteCapability)
10    .capability(useMarkerCapability)
11    .build();
12 // Decorator to define the type that the diagram element has
13 @NodeMetadata({
14     type: 'task:manual'
15 })
16 export class TaskManual extends TaskManualMixin implements PLabelledElement {
17     // Overriding how children are accessed
18     override readonly children = new TaskManualChildren(this);
19
20     // Semantic information for the GLSP-Playwright framework
21     get label(): Promise<string> {
22         return this.children.label().then(label => label.textContent());
23     }
24 }
25 // Extending the default ChildrenAccessor to allow easier access to the label
26 export class TaskManualChildren extends ChildrenAccessor {
27     // Return a Page Object for the label
28     async label(): Promise<LabelHeading> {
29         return this.ofType(LabelHeading, { selector:
       ↪  SVGMetadataUtils.typeAttrOf(LabelHeading) });
30     }
31 }
```

With the necessary preparations completed, writing the test cases is now possible.

5.2 Test Cases

The test cases are categorized to encompass various aspects and areas of testing. This division allows the evaluation to systematically cover different functionalities exposed by

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the GLSP-Playwright framework. Some categories may have fewer test cases than others because they target specific functionalities that require fewer scenarios. Accordingly, they also have various lengths and complexities. The goal is to have a balanced representation of test scenarios across all relevant areas implemented in the framework to showcase the ability of the framework.

C1: Common

The first category comprises essential test cases that are fundamental to interacting with diagram elements. These test cases cover various aspects, including:

- **Handling diagram elements:** Evaluating the GLSP-Playwright API for accessing nodes and edges in the diagram.
- **Handling incoming and outgoing edges:** Testing the capability to handle incoming and outgoing edges from nodes.
- **Type deduction:** Verifying whether the diagram can correctly deduce the types for edges and nodes after accessing it.
- **Node capabilities:** Assessing the ability of nodes to access their children's elements.
- **Simple flows:** Testing basic operations like element deletion.
- **Shortcuts:** Evaluating the functionality of shortcuts to trigger specific actions in the application.

C2: Command Palette

The *Command Palette* category comprises test cases focused on evaluating the behavior of the command palette. This includes both the global and element-specific command palettes. The test cases in this category cover the following aspects:

- **Reading the entries:** Ensuring that the Page Objects can accurately read and access the entries present in the command palette.
- **Searching:** Verifying the capability of the command palette to search for specific entries based on user input.
- **Triggering:** Evaluating advanced scenarios where the command palette is tested for its ability to trigger specific entries by confirming them. Those actions are asynchronous in nature and usually modify the underlying diagram.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

C3: Tool Palette

Similar to the *Command Palette* category, the *Tool Palette* category centers around testing the framework's capabilities to interact with the tool palette. This category encompasses several aspects, including:

1. **Retrieving Options:** Evaluates the framework's API for accessing different option groups and individual options within the tool palette.
2. **Retrieving Toolbar Items:** Tests the API's ability to retrieve and interact with various toolbar items available in the tool palette.
3. **Triggering:** Verifies if nodes and edges can be created using the tool palette. Additionally, tests are conducted to ensure that the framework can effectively trigger actions associated with the toolbar items. Accordingly, those actions modify the diagram.

C4: Resize Handle & Routing Point

The *Resize Handles and Routing Points* category in evaluating the GLSP-Playwright framework focuses on these components' visibility and interaction capabilities. The tests within this category verify whether the resize handles and routing points become visible when certain conditions are met. Subsequently, the tests assess the framework's ability to interact with these components to perform actions like resizing a diagram element or adjusting the routing of edges. By evaluating the presence and functionality of resize handles and routing points, this category ensures that the framework adequately handles these important aspects of graphical applications during testing.

C5: Popup & Marker

The *Popup* category focuses on the ability of the framework to hover on elements and afterward to access the content that appears.

The *Marker* category focuses on evaluating the behavior of and the tests serve two main purposes: Firstly, they verify the ability to trigger markers through the tool palette like a user would do. Secondly, the tests evaluate the framework's capability to read and access markers by hovering over them, similar to popups, ensuring that the markers can be correctly identified and interacted with during the testing process.

5.3 Report

In this section, the results of the tests will be analyzed. A total of 37 tests have been defined, covering various test categories as visible in Table 5.3. The majority of the tests fall under the *C1: Common* category, as it deals with fundamental interactions with diagram elements, which is more complex compared to other categories, and they

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

serve as the foundation for all other tests. If they fail, it may affect the success of other tests. In addition, the tests were executed on different tool platforms to identify any potential differences between the integrations and to ensure comprehensive testing coverage. Throughout different stages of the development of the testing framework, two tests failed on Theia and two in the VS Code integration, resulting in 4 failing of 37 available tests. On the other hand, the Standalone integration had no issues. At the end, the testing framework could identify 2 bugs in Theia and 1 discrepancy in VS Code. These issues for Theia have been reported and subsequently fixed by the authors. While the failures for VS Code made it necessary to adjust the test cases. More about it later in Subsection 5.3.2.

Table 5.3: Test case report

	C1	**C2**	**C3**	**C4**	**C5**	**Total**
Test Cases	18	6	6	4	3	37
Success	**16**	**6**	**4**	**4**	**3**	**33**
Failures	**2**	-	**2**	-	-	**4**
Standalone	**18**	**6**	**6**	**4**	**3**	**37**
Theia	**16**, **2**	**6**	**6**	**4**	**3**	**35**, **2**
VS Code	**18**	**6**	**4**, **2**	**4**	**3**	**35**, **2**

The test report excerpt in Figure 5.1 exported by Playwright shows the result concerning the resize handle tests. The complete set of executed test cases contains 37 tests for the Standalone platform, 37 tests for Theia, and 37 tests for VS Code (including an additional 2 tests for setup), totaling 113 tests. These 37 test cases are only defined once but executed for each tool platform individually. The entire process took around 210 seconds (i.e., parallel execution). Following the fix of all identified errors and issues, all tests now run successfully without any failures. However, it is noteworthy that the test execution can be time-consuming due to the complexity and thoroughness of the testing framework.

Table 5.4 provides more detailed and accurate runtime values for each test category and also showcases various metrics related to the execution time of each category. Those metrics are measured in seconds and parallel execution did not influence the values.

The test results indicate a significant time difference between the Standalone tool platform and Theia and VS Code integrations in category C1. The Standalone tool platform is approximately 10 to 18 times faster than the other integrations. This discrepancy may be attributed to the time required for Theia and VS Code to become ready before the test cases are executed which is not necessary for the Standalone tool platform. It raises important considerations on how to enhance the performance of these integrations to reduce the time taken to prepare them for testing. This topic could be explored further to optimize the testing experience and efficiency for all tool platforms. Another factor contributing to slower tests in VS Code is that it does not operate in headless mode. Headless mode involves running a browser without a graphical user interface, leading

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

resize-handle

All 113 | Passed 113 | Failed 0 | Flaky 0 | Skipped 0

Total time: 3.5m

../../tests/features/resize-handle.spec.ts

✓ The resizing handle › should allow resizing standalone 696ms
../../tests/features/resize-handle.spec.ts:35

✓ The resizing handle › should show 4 handles standalone 692ms
../../tests/features/resize-handle.spec.ts:54

✓ The resizing handle › should allow resizing theia 6.1s
../../tests/features/resize-handle.spec.ts:35

✓ The resizing handle › should show 4 handles theia 5.5s
../../tests/features/resize-handle.spec.ts:54

✓ The resizing handle › should allow resizing vscode 13.0s
../../tests/features/resize-handle.spec.ts:35

✓ The resizing handle › should show 4 handles vscode 12.7s
../../tests/features/resize-handle.spec.ts:54

Figure 5.1: Excerpt of the Playwright test report

Table 5.4: Sequential runtime in seconds for each category

Category	Test Cases	Tool Platform	Sum	Min	Max	Avg
C1	18	**Standalone**	14,2	0,5	1,5	0,7
		Theia	112,1	5,7	6,7	6,2
		VS Code	241,8	12,2	15,5	13,4
C2	6	**Standalone**	13,7	1,5	3,1	2,2
		Theia	43,4	6,2	8,2	7,2
		VS Code	96,0	14,3	18,4	16,0
C3	6	**Standalone**	6,5	0,6	1,6	1,0
		Theia	38,2	4,8	7,3	6,3
		VS Code	79,6	7,9	16,3	13,2
C4	4	**Standalone**	2,3	0,5	0,6	0,5
		Theia	23,9	5,5	6,3	5,9
		VS Code	52,8	12,7	13,6	13,2
C5	3	**Standalone**	6,1	1,8	2,3	2,0
		Theia	22,7	7,4	7,7	7,5
		VS Code	46,5	14,5	17,0	15,5

to faster execution. In this mode, the browser can perform various functions, such as navigating pages, clicking links, and downloading content, just like a regular browser, but without the need to display the graphical user interface. The absence of headless mode in VS Code is a reason for its relatively slower test execution compared to the Standalone tool platform. In addition, running all those tests sequentially would take about 810 seconds, which is about 4 times slower than in parallel. Due to this reason,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

running the tests in parallel is crucial.

5.3.1 Diagram Automation

The final topic of the report concerns the capability of GLSP-Playwright to automate diagram elements, one of the main challenges of this thesis. Table 5.5 It shows an excerpt of the features of a diagram element in the GLSP framework and if GLSP-Playwright can test and automate it.

Table 5.5: Automation report for the diagram elements

Diagram Element	Test Available	Automation Possible
Locating		
- Semantic	✓	✓
- Query	✓	✓
Operations		
- Create	✓	✓
- Delete	✓	✓
- Move	✓	✓
- Rename	✓	✓
Node		
- Loc. Children	✓	✓
- Loc. In/Out Edges	✓	✓
- Resizing	✓	✓
Edge		
- Loc. Source/Target	✓	✓
- Rerouting	✓	✓
Other		
- Hover	✓	✓
- Select	✓	✓
- Popup	✓	✓
- Command Palette	✓	✓
Verifying Visually	X	X

The table is divided into six sections, each representing different aspects of the GLSP-Playwright framework's capabilities regarding diagram elements.

- The first section, **Locating**, deals with the framework's ability to access diagram elements semantically or through search queries. The advantage of the framework is that diagram elements are always typed, and testers can use semantic information like labels to identify elements without complex search queries. Additionally, if no semantic information is available or implemented, the framework allows testers to use CSS or XPath-based search queries with the benefit of type safety.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- The **Operations** section focuses on the framework's ability to perform operations that modify the underlying diagram, such as creating, deleting, moving, and renaming diagram elements. The framework provides an API that allows testers to work with elements in a more natural language. As all operations are asynchronous, the framework will wait until their completion.

- The **Nodes** section highlights specific features of nodes, such as their children and incoming/outgoing edges. The framework enables testers to access these elements in a type safe manner and navigate through the diagram in a more semantic way, eliminating the need for complex logic to locate elements. Node-specific actions, like resizing, are also supported.

- Similarly, the **Edges** section covers specific functionalities of edges. The framework allows testers to access the source and target of edges securely, while maintaining type safety. Additionally, edges can be rerouted easily.

- In the **Other** section, various actions like hovering, selecting, and triggering custom user interfaces, such as the command palette and popups, can be performed using the testing framework.

- However, the framework falls short of understanding the diagram visually. It does not provide any means to comprehend the rendered elements beyond the SVG code. Consequently, scenarios that involve the rendering or visual aspects of any component in the editor cannot be verified using the GLSP-Playwright framework.

5.3.2 Found Bugs

During the evaluation of the testing framework, three issues were identified, consisting of two bugs and one discrepancy.

The first bug was discovered during the framework's development and related to the Theia editor's failure to clean up after closing the GLSP-Client. Some parts of the GLSP-Client were left on the web page, leading to test case failures. This bug was promptly reported to the authors of GLSP and subsequently fixed by them.

The authors introduced the second bug after they reworked how Theia uses the GLSP-Server. This resulted in server crashes under specific scenarios triggered by the test cases. The bug was reported on GitHub[1] and successfully fixed.

The third issue arose with VS Code, where a discrepancy in the tool palette's creation was observed. VS Code used uppercase labels, while the other implementation utilized sentence case. Consequently, two test cases failed due to the differences between the integrations. Rather than fixing the issue directly in the VS Code integration, the GLSP authors asked for a solution where the testing framework could differentiate between the tool platforms during execution. As a result, the failing test cases now determine which

[1] `https://github.com/eclipse-glsp/glsp/issues/1030`, Accessed: 05.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

integration is active and adapt the expected label on-the-fly to ensure successful test runs.

5.4 Discussion

The evaluation of the GLSP-Playwright framework has proven valuable in assessing its capabilities and effectiveness in testing GLSP-based editors. The framework demonstrated its ability to provide a more natural and intuitive language than Playwright-alone for writing test cases, making it easier for testers to interact with diagram elements and access different functionalities. Throughout the evaluation, a set of test cases was designed and executed across different test categories. Notably, by no means was the goal to test/implement all aspects of the GLSP framework in this thesis. GLSP is under development and retrieves frequent updates. For this reason, the importance lies in developing a testing framework and a set of tests for the core elements of the GLSP framework with more weight on the diagram elements.

The results indicated that the GLSP-Playwright framework successfully handled various aspects of testing, including diagram operations, access to nodes and edges, and interaction with UI extensions such as the tool palette and command palette. In accordance, the framework provided a type safe way to access diagram elements, making it easier to perform interactions and verify results. Moreover, separating concerns in the implementations and encapsulating complex behaviors in Page Objects allowed for writing straightforward test cases without worrying about handling asynchronous processes or the underlying details. The evaluation also uncovered some discrepancies, such as differences in tool palette behavior between different integrations (Theia and VS Code). However, the framework's flexibility and extensibility allowed for dynamic adaptation based on the integration in use, enabling test cases to handle varying behaviors gracefully.

Furthermore, the evaluation highlighted the significant performance advantage of the Standalone tool platform over Theia and VS Code. The headless mode and the possibility to directly use the Standalone tool platform contributed to faster execution times, resulting in overall more efficient testing. The Standalone version is just the GLSP-Client directly used in the browser without any integration or extras, such as Theia or VS Code. This raises the question of whether it is more efficient to test the GLSP-Client-specific use cases only on the Standalone version and not parallelly on the other tool platforms. This would improve the execution time significantly and allows testers to retrieve faster feedback. The tool platform-specific behaviors would then be tested separately. Nevertheless, a combination would probably be the best solution here, where locally, the tests run on the Standalone version and online on all tool platforms. In the end, it depends on the project and if the GLSP-Client can be cleanly tested alone.

On the other hand, the evaluation also identified a limitation in the framework's inability to understand the diagram visually, which could impact scenarios involving rendering or visual aspects of elements. This includes all aspects where the rendering is of importance.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Currently, this topic is unattended as it is a complex topic in itself, and implementing a correct solution on top of GLSP-Playwright would be a separate research area.

In conclusion, the GLSP-Playwright framework proved to be a powerful tool for testing GLSP implementations, enabling testers to write more expressive, readable, and maintainable test cases. While some challenges and limitations were discovered, the framework's flexibility and ease of use make it a valuable asset for ensuring the reliability and functionality of GLSP-based web modeling tools.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

Future Work

The testing framework has already been released as an open-source framework[1]. It has the means to interact with and test the functional requirements of GLSP-based editors. Still, there are ways to improve it.

Functionality-wise, the focus was clearly on working with diagram elements. This challenge has been solved with this thesis. However, there are still open challenges that are not solved. A new challenge with the most benefit would be implementing a way to validate the diagram elements visually. Currently, there is no efficient way to do it. There is a way to do screenshot comparisons called visual diffing. Here the rendered element is compared against an expected rendering. Nevertheless, this approach is error-prone. Running the same tests on different machines could produce false results due to the different screen resolutions and rendering engines. Researching this topic and providing a prototype for the GLSP testing framework would be beneficial.

Implementation-wise, there are some improvemenets possible. Decorators are an upcoming ECMAScript (i.e., JavaScript) feature that has also been implemented in TypeScript 5.0. Decorators enable developers to customize classes and their members in a reusable way. At the time of this writing, the proposal is still not final. Still, there are discussions on how they can influence the type system of TypeScript. One of those discussions highlights the possibility to allow decorators to modify the type of the applied class. Currently, decorators can not influence the type of the class. That means, even if the decorator would introduce new members to the class, the TypeScript compiler would not be able to detect it. However, if this behavior changes in the future and the type could be influenced by the decorator, then it would greatly benefit the mixins introduced in the GLSP testing framework. They could be directly applied to the target class without needing to use the builder first.

[1] `https://github.com/eclipse-glsp/glsp-playwright`, Accessed: 18.08.2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Integration-wise, currently, the Eclipse IDE has no support. The Eclipse IDE is neither a browser nor a browser-like application. Testing it would require a different approach compared to the already supported tool platforms.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

CHAPTER 7

Conclusion

The thesis presented an in-depth exploration of testing GLSP-based web modeling tools using the GLSP-Playwright framework. Initially, we defined in Section 2.1 what software testing means and connected it with the goal to develop a solution that could efficiently test diagram elements and GLSP-based editors with ease in Section 2.3. To arrive at such a solution, we introduced three research questions at the beginning of this thesis, guiding our research and development efforts. Now, we will provide answers to these questions.

RQ-1 **What do most GLSP-based diagrams and editors have in common regarding possible user interaction possibilities, respective user interfaces that support those interactions, and tool platforms?**

In Section 2.2, we conducted a comprehensive analysis of the Workflow example used by the GLSP authors in their projects. We aimed to address the first research question by thoroughly examining the example, which subsequently identified common characteristics, diagram functionalities, UI extensions, and integrations typically present in such editors. Our findings led us to conclude that users can engage with diagram elements through various means, including mouse interactions and keyboard shortcuts (e.g., clicking, dragging, using the delete key). Additionally, we observed that the diagram is complemented by diverse UI extensions, such as the tool palette and the command palette, which provide supplementary functionalities. Lastly, we established that these tools are commonly integrated into browser and browser-like platforms such as Theia, VS Code, or the Standalone platform.

RQ-2 **How can the new testing framework be implemented by respecting extendability and maintainability so that different GLSP-based diagrams and editors can be tested?**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

To address this question, we initially conducted a thorough analysis of browser automation techniques in Chapter 3. We delved into the WebDriver, DevTools, and the Native protocols to gain a deep understanding of these mechanisms. This groundwork enabled us to establish our expectations for an effective testing framework for GLSP, as outlined in Section 3.2. In alignment with these expectations, we proceeded to compare three known web testing frameworks: Selenium, Cypress, and Playwright, as detailed in Section 3.3. Our comparison highlighted the suitability of Playwright as the foundational choice for developing the GLSP testing framework.

This brought us to the question of Playwright's existing capabilities in testing GLSP-based editors in Chapter 4. We formulated three testing scenarios that exclusively employed Playwright and observed that while Playwright could interact with the diagrams and the editor, it lacked the domain-specific knowledge required for creating easily understandable test scenarios. Additionally, we identified that many user interactions could be shared across various scenarios. With these findings, we addressed the design principles that would guide our goal of extensibility and maintainability, as explored in Section 4.3. There, we introduced the concept of page objects and the application of mixins. Page objects allow testers to encapsulate specific web page behaviors in classes by exposing an API to interact with it. By doing that, the specific logic to handle the web application is only defined on the page object, which improves maintainability. Additionally, this approach also allows us to write easier-to-understand test scenarios, as the concrete low-level implementations are hidden. Beside, the application of mixins allows testers to add different implemented behaviors to page objects without the necessity to implement them themselves. The mixins can be added to various page objects directly without issues. Moreover, all of the exposed classes and page objects from the GLSP-Playright framework are designed with extensibility and type safety in mind. With this approach, extensibility is secured. Furthermore, as detailed in the Section 4.5, we outlined the architectural framework, emphasizing the separation of concerns and adherence to the single responsibility principle, which further improves extensibility and maintainability.

Finally, we also redefined the initially defined testing scenarios with the GLSP-Playwright framework, which resulted in easier to write and more comprehensible tests.

RQ-3 **What is the necessary metadata the testing framework needs from the GLSP-based diagram editor to process the diagram?**

In Subsection 4.3.3, we established the essential metadata necessary for processing diagram elements. Our investigation revealed the significance of obtaining data about the interconnections among distinct elements such as nodes, edges, and their child components. Equally important was the determination of the semantic context carried by each diagram element. With this metadata, external libraries can proficiently handle the diagram's intricacies, enabling streamlined processing.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

In conclusion, the thesis demonstrated that the GLSP-Playwright framework is a powerful and efficient tool for testing web-based diagram editors in Chapter 5. Its use of page objects and mixins significantly improved test case readability and maintainability. The exposed API simplified complex scenarios, empowering testers to focus on testing functionalities without being burdened by low-level implementation details. Overall, the thesis presented a thorough investigation into testing web-based diagram editors and showcased the effectiveness of the GLSP-Playwright framework in providing a convenient and efficient approach for interacting with and verifying diagram elements. As the framework continues to evolve and receive further refinements, it holds great promise as an integral tool for testing GLSP-based editors.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

2.1 Test pyramid . . . 13
2.2 Overview of GLSP components and their interplay [BLO23] . . . 16
2.3 Overview of the UI (Standalone Version) . . . 19
2.4 Workflow example in Theia . . . 21
2.5 Workflow example in VS Code . . . 21
2.6 Visual representation of the SVG example . . . 24

3.1 WebDriver protocol . . . 28
3.2 DevTools protocol . . . 29
3.3 Native protocol . . . 30
3.4 NPM weekly downloads . . . 45

4.1 Page objects . . . 60
4.2 Metadata for diagram elements . . . 64
4.3 GLSP-Playwright architecture . . . 67
4.4 Remote module architecture . . . 68
4.5 Extension module architecture . . . 71
4.6 Integration module architecture . . . 78
4.7 Theia Workflow example . . . 79
4.8 VS Code Workflow example . . . 80
4.9 GLSP module architecture . . . 84
4.10 Accessing diagram elements . . . 90

5.1 Excerpt of the Playwright test report . . . 107

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

2.1 (Software) Quality definition 10
2.2 (Software) Quality assurance definition 11
2.3 Testing definition 12
2.4 Unit testing definition 14
2.5 Integration testing definition 14
2.6 System testing definition 15
2.7 Acceptance testing definition 16
2.8 Tool platform differences 22

3.1 Frameworks overview 42
3.2 NPM statistics 46

4.1 Available flows 74
4.2 Available capabilities 77
4.3 Exported GLSP-UI page objects 87

5.1 Workflow classes 102
5.2 Workflow diagram elements 102
5.3 Test case report 106
5.4 Sequential runtime in seconds for each category 107
5.5 Automation report for the diagram elements 108

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Listings

2.1 SVG Example . . . 24

4.1 Anatomy of Playwright tests . . . 50
4.2 Playwright: Simple locator example . . . 51
4.3 PE-1: Retrieving information . . . 52
4.4 PE-2: Interacting with diagram elements . . . 54
4.5 PE-3: Creating new elements . . . 56
4.6 Page object application for PE-2 . . . 61
4.7 Mixins in JavaScript . . . 63
4.8 Renameable flow . . . 73
4.9 Resize Handle capability . . . 75
4.10 Resize handle page object . . . 76
4.11 VS Code setup . . . 82
4.12 Creating a *GLSPApp* instance . . . 86
4.13 Decorators, mixins and diagram elements . . . 89
4.14 Wait API for creation . . . 91
4.15 GE-0: Preparations . . . 92
4.16 GE-1: Retrieving information . . . 93
4.17 GE-2: Interacting with diagram elements . . . 94
4.18 GE-2: Interacting with diagram elements (improved) . . . 95
4.19 GE-3: Creating new elements . . . 97
4.20 Playwright configuration . . . 98

5.1 *TaskManual* definition . . . 103

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[Aha23] Irshad Ahamed. What is browser automation: A complete tutorial to web browser automation with examples and best practices. `https://www.lambdatest.com/learning-hub/browser-automation`, 2023. Accessed: 06.08.2023.

[bac17] Iso/iec/ieee international standard - systems and software engineering–vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, pages 1–541, 2017.

[BHV16] Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of GitHub repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, 2016.

[BLO23] Dominik Bork, Philip Langer, and Tobias Ortmayr. A vision for flexibile GLSP-based web modeling tools. *CoRR*, abs/2307.01352, 2023.

[BP06] Luciano Baresi and Mauro Pezzè. An introduction to software testing. *Electronic Notes in Theoretical Computer Science*, 148(1):89–111, 2006.

[BT18] Hudson Borges and Marco Tulio Valente. What's in a GitHub star? Understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018.

[BZZ20] Mario Bernhart, Markus Zoffi, and Christina Zoffi. Lecture: 183.290 software testing, 2020. TU Wien, PESO.

[CLB22] Giuliano De Carlo, Philip Langer, and Dominik Bork. Advanced visualization and interaction in GLSP-based web modeling: Realizing semantic zoom and off-screen elements. In Eugene Syriani, Houari A. Sahraoui, Nelly Bencomo, and Manuel Wimmer, editors, *ACM / IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MoDELS'22)*, pages 221–231. ACM, 2022.

[CLR20] Maura Cerioli, Maurizio Leotta, and Filippo Ricca. What 5 million job advertisements tell us about testing: a preliminary empirical investigation. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1586–1594, 2020.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[DLF06] Giuseppe A Di Lucca and Anna Rita Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology*, 48(12):1172–1186, 2006.

[Gag23] Luc Gagan. Comparing automated testing tools: Cypress, Selenium, Playwright, and Puppeteer. `https://ray.run/blog/comparing-automated-testing-tools-cypress-selenium-playwright-and-puppeteer`, 2023. Accessed: 06.08.2023.

[GB21] Philipp-Lorenz Glaser and Dominik Bork. The bigER tool - hybrid textual and graphical modeling of entity relationships in VS Code. In *2021 IEEE 25th International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 337–340, 2021.

[GGGMO20] Boni García, Micael Gallego, Francisco Gortázar, and Mario Munoz-Organero. A survey of the Selenium ecosystem. *Electronics*, 9(7), 2020.

[Goo] Google. Chrome DevTools protocol specification. `https://chromedevtools.github.io/devtools-protocol/`. Accessed: 06.08.2023.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

[IST18] ISTQB. Standard glossary of terms used in software testing, 2018. version 3.2.

[IST23] ISTQB. Certified test foundation level syllabus, 2023. version 4.0.

[JAAA16] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. Software testing techniques: A literature review. In *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, pages 177–182, 2016.

[Jor13] Paul C. Jorgensen. *Software Testing: a Craftsman's Approach*. Auerbach Publications, fourth edition, 2013.

[LCRS13] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. Improving test suites maintainability with the page object pattern: An industrial case study. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 108–113, 2013.

[LCRT13] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 272–281, 2013.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[LGRW23] Maurizio Leotta, Boni García, Filippo Ricca, and Jim Whitehead. Challenges of end-to-end testing with Selenium WebDriver and how to face them: A survey. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 339–350, 2023.

[LT93] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[MB23a] Haydar Metin and Dominik Bork. Introducing bigUML: A flexible open-source GLSP-based web modeling tool for UML. In *Proceedings of the 26th International Conference on Model Driven Engineering Languages and Systems, MODELS 2023*, page in press. IEEE, 2023.

[MB23b] 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*, page in press. IEEE, 2023.

[Mica] Microsoft. Language server protocol implementations. `https://microsoft.github.io/language-server-protocol/implementors/servers/`. Accessed: 13.04.2023.

[Micb] Microsoft. Language server protocol specification. `https://microsoft.github.io/language-server-protocol/specifications/specification-current/`. Accessed: 13.04.2023.

[Mic16] John Micco. Flaky tests at Google and how we mitigate them. `https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html`, 2016. Accessed: 19.08.2023.

[Pat23] Kailash Pathak. Playwright vs Selenium vs Cypress: A detailed comparison. `https://www.lambdatest.com/blog/playwright-vs-selenium-vs-cypress/`, 2023. Accessed: 06.08.2023.

[Phi] Philip Langer. Diagram editors with GLSP: Why flexibility is key. `https://www.youtube.com/watch?v=mSTXgUZCBVE`. Accessed: 19.08.2023.

[Pla20] Wolfgang Platz. As test automation matures, so do false positives. `https://www.stickyminds.com/article/test-automation-matures-so-do-false-positives`, 2020. Accessed: 19.08.2023.

[Rag20] Giovanni Rago. Puppeteer vs Selenium vs Playwright, a speed comparison. `https://www.checklyhq.com/blog/puppeteer-vs-selenium-vs-playwright-speed-comparison/`, 2020. Accessed: 06.08.2023.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[REIWC18] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. Towards a language server protocol infrastructure for graphical modeling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, page 370–380, New York, NY, USA, 2018. Association for Computing Machinery.

[Saw22] Sawjan. Different approaches (protocols) to automate the browser. `https://dev.to/jankaritech/different-approaches-protocols-to-automate-the-browser-39f1`, 2022. Accessed: 06.08.2023.

[SMB23] Aylin Sarioglu, Haydar Metin, and Dominik Bork. How inclusive is conceptual modeling? A systematic review of literature and tools for disability-aware conceptual modeling. In *Proceedings of the 42nd International Conference on Conceptual Modeling (ER 2023)*, page in press. Springer, 2023.

[W3Ca] W3C. Webdriver BiDi specification. `https://w3c.github.io/webdriver-bidi/`. Accessed: 06.08.2023.

[W3Cb] W3C. Webdriver specification. `https://www.w3.org/TR/webdriver/`. Accessed: 06.08.2023.

[YS21] Jecelyn Yeen and Maksim Sadym. Webdriver BiDi - the future of cross-browser automation. `https://developer.chrome.com/articles/webdriver-bidi/`, 2021. Accessed: 06.08.2023.

[YS23] Jecelyn Yeen and Maksim Sadym. A look back in time: the evolution of test automation. `https://developer.chrome.com/blog/test-automation-evolution/`, 2023. Accessed: 06.08.2023.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub