Department of Mathematics and Computer Science

Software Engineering Group
June 29, 2025

Creating web-based diagram editors for specifying and executing model transformations Master Thesis

from

Florian Weidner

Submission date: June 29, 2025
First examiner: Prof. Dr. Gabriele Taentzer
Second examiner: Prof. Dr.-Ing. Christoph-Matthias Bockisch

I confirm that this master thesis is my own work and I have documented all sources and material used.

Marburg, June 29, 2025 Florian Weidner

Zusammenfassung

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

List of Acronyms

API Application Programming Interface. 5, 19, 21, 26, 29, 35

AToMPM A Tool for Multi-Paradigm Modeling. 12, 13

CSS Cascading Style Sheets. 31

DI Dependency Injection. 8

DSML Domain Specific Modeling Language. 12

ELK Eclipse Layout Kernel. 31, 32

EMF Eclipse Modeling Framework. 2, 5–7, 9, 10, 13, 15, 16, 21, 22, 29, 31–34, c

GLSP Graphical Language Server Platform. 2, 4, 7–11, 13, 15, 16, 18, 20–23, 25–35, 43, c

GUI Graphical User Interface. 7

HTML Hypertext Markup Language. 10, 35

IDE Integrated Development Environment. 3, 4, 6, 7, 12, 14, 18, 19, 25, 26, 30

JAR Java Archive. 30

JDK Java Development Kit. 30

JDT Java Development Tools. 4

LHS Left-Hand Side. 7, 33

MDE Model-Driven Engineering. 2, 12–14, 16, 26

NAC Negative Application Condition. 7

PAC Positive Application Condition. 7

PDE Plug-in Development Environment. 4

POC Proof of Concept. 29

RHS Right-Hand Side. 7, 33

RPC Remote Procedure Call. 8, 10

SDK Software Development Kit. 6, 7, 22, 29

SDV Software-Defined Vehicle. 4

SVG Scalable Vector Graphics. 10

UI User Interface. 6, 9, 10

UML Unified Modeling Language. 5, 12

URI Uniform Resource Identifier. 9

UUID Universally Unique Identifier. 34

VS Code Visual Studio Code. 2, 18, 19, 25, 26, 30

XMI XML Metadata Interchange. 5, 7, 9, 11, 14–16, 19, 21, 22, 29, 31, 32, 34

XML Extensible Markup Language. 5

1. Introduction

1.1. Background and Motivation

In software engineering, often Model-Driven Engineering (MDE) is used to increase development productivity and quality. [1] Concepts are modeled closer to the domain, so that they describe important aspects of a solution with human-friendly abstractions. The models can also be used to generate application fragments, that can be directly used as a template source code. In the process of MDE, many activities need to transform source models into different target models, while following a set of transformation rules. This model transformation process is based on algebraic graph transformations. A metamodel is used to model the structure and rules of the concept. The resulting transformation language can provide automatic model creation, development, and maintenance activities. [1] One framework to use MDE is Eclipse Modeling Framework (EMF) by the Eclipse Foundation. It provides a basis for application development, using modeling and code generation facilities. Many frameworks build upon EMF, providing various MDE tools like code generators, graphical diagramming, model transformation, or model validation. [2] One model transformation framework is Henshin. [3] It tries to provide model transformation capabilities with a high level of usability. [4] For metamodels it uses EMF Ecore files. The framework allows to create and apply model transformations on XMI instance files with a defined transformation language. It provides a graphical and textual syntax to create these transformation rules. [3] Henshin can be used as a Eclipse plugin. For new users, Eclipse IDE needs to be installed and the heavy editor makes the use of Henshin unintuitive without prior experience. Therefore, the goal exists to create a graphical option to use the Henshin model transformations without the overhead of the heavy IDE, that has to be installed. A web-based graphical editor would make the use of Henshin even more accessible and intuitive.

GLSP is a open-source framework by the Eclipse Foundation, which can be used to build a web-based Henshin graph editor. The framework is used to develop custom diagram editors for distributed web-applications. [5] It can provide graph editors for the Eclipse Desktop IDE, Eclipse Theia, Visual Studio Code (VS Code) and a stanalone version usable in any website. It brings the support of EMF models as a data source and the functionality of the Henshin SDK can be called from the Java server of GLSP. [6] With these functionalities, GLSP fits to create an easy accessible, intuitive application to create and apply Henshin model transformations called *Henshin Web*.

1.2. Problem Statement

Despite the powerful capabilities of Henshin for model transformations, its current usage presents barriers to adoption and accessibility. The framework is exclusively available as an Eclipse Integrated Development Environment (IDE) plugin, which requires users to install and configure the complete Eclipse environment before they can begin working with model transformations. This dependency limits the framework's reach and usability.

First, the requirement for Eclipse installation presents a substantial entry barrier, particularly for newcomers to MDE who wish to explore model transformation concepts without committing to a full development environment setup. For example, students or researchers who want to quickly experiment with Henshin transformation rules face unnecessary complexity in simply accessing the tool. The installation process, environment configuration, and learning the Eclipse interface adds cognitive overhead that detracts from the Henshin functionality itself.

Second, the Eclipse IDE presents usability challenges. Eclipse is a heavyweight development environment with many features and a complex interface. It can feel overwhelming when the primary goal is to create and apply model transformations. Users must navigate through multiple perspectives, views, and menus to accomplish basic transformation tasks, leading to reduced productivity and increased frustration. The standard editor for Ecore, Henshin rules and XML Metadata Interchange (XMI) instance files is a tree view, which can get unintuitive for bigger models. In order to edit the transformation rules graphically, a diagram file needs to be initialized separately. To edit the Ecore metamodels, also the diagram has to be initialized specifically. For the XMI instance files, an extension has to be installed. Especially the application of transformation rules is not supported graphically even with extensions, but only through a wizzard window.

Furthermore, the current setup limits collaborative possibilities and portability. Sharing transformation examples across different clients or collaborating on transformation development is not possible with Eclipse. Version Control System (VCS) have to be used to share files and real-time collaboration or easy access from different devices is not supported.

These accessibility and usability challenges prevent Henshin from reaching its full potential as a model transformation solution, particularly for beginners who face significant initial challenges and among users who could benefit from quick, intuitive access to transformation capabilities without the overhead of a complete IDE setup.

1.3. Research Questions

Based on the identified problems with the current Eclipse-based approach to Henshin model transformations, this thesis aims to address the following research questions that guide the development and evaluation of a web-based solution:

RQ1: How can Henshin model transformation capabilities be effectively adapted for web-based environments? This question investigates the technical feasibility and architectural considerations for translating the desktop-based Henshin functionality into a web application. It examines how the core transformation engine, metamodel handling, and rule definition capabilities can be preserved while adapting to web technologies and browser constraints.

RQ2: What are the essential functional requirements for a web-based Henshin editor that maintains usability while reducing complexity? This question focuses on identifying the minimum viable feature set that provides meaningful transformation capabilities in order to create an application that can completely handle typical use cases.

RQ3: How does a web-based approach improve accessibility and user experience compared to the traditional Eclipse plugin? This question evaluates the effectiveness of the web-based solution in addressing the identified barriers to adoption. It examines metrics such as installation complexity, learning curve, collaboration capabilities, and overall user satisfaction when working with model transformations.

RQ4: How do different deployment strategies affect the accessibility, usability, and adoption barriers for web-based model transformation tools? This question explores various deployment options for the web-based Henshin editor, such as standalone web applications, cloud-hosted services, or integration with existing platforms. It assesses how these strategies impact user access, ease of use, and the overall adoption of the tool among different stakeholder groups.

RQ5: How can the web-based editor integrate with existing EMF and Henshin ecosystems? This question explores the compatibility and interoperability requirements for ensuring that the web-based solution can work with existing metamodels, transformation rules, and instance files created in the traditional Eclipse environment, while also providing value as an independent tool.

These research questions collectively address the goal of creating an accessible, intuitive, and functionally adequate web-based alternative to the current Eclipse-dependent Henshin workflow, while ensuring that the solution provides authentic value to the identified stakeholder groups.

1.4. Scope and Limitations

This thesis focuses on developing a web-based solution for Henshin model transformations, with specific boundaries and constraints that define the research scope and acknowledge inherent limitations.

Scope of the Research: The primary scope encompasses the design, implementation, and evaluation of a web-based editor that provides core Henshin transformation capabilities with the GLSP framework. The work includes adapting the essential features of the Henshin Eclipse plugin for web environments, focusing on transformation rule creation, metamodel handling, and instance file processing. The implementation targets the fundamental workflow of loading EMF Ecore metamodels, creating transformation rules through a graphical interface, and applying these transformations to XMI instance files.

The research specifically addresses accessibility improvements to minimize the initial challenge for beginners, where users need quick access to model transformation capabilities without extensive setup requirements. The evaluation covers usability aspects, performance characteristics, and functional completeness compared to the traditional Eclipse-based approach. Integration with existing EMF and Henshin ecosystems is considered to ensure compatibility with established workflows and file formats.

Limitations and Constraints: Several limitations constrain the scope of this research. The web-based implementation does not aim to replicate every advanced feature available in the mature Eclipse Henshin plugin. Complex transformation scenarios, advanced debugging capabilities are beyond the current scope. The focus remains on core functionality that serves the primary use cases identified in the requirements analysis.

The evaluation methodology is constrained by the availability of test scenarios and user groups within the academic environment. While the research aims to demonstrate improvements over the Eclipse approach, comprehensive studies or extensive industrial validation are outside the scope of this thesis work.

Additionally, the research does not extend to developing new transformation algorithms or enhancing the underlying Henshin transformation engine itself. The focus remains on providing better accessibility and usability for existing Henshin capabilities rather than advancing the theoretical foundations of model transformation techniques.

A system constraint is that the backend has to be Java-based, to be able to directly run the Henshin SDK.

These scope definitions and limitations ensure that the research remains focused and achievable within the constraints of a master's thesis while addressing the core problems identified in current Henshin usage patterns.

1.5. Structure of the Thesis

This thesis is structured as follows, with each chapter building upon the previous ones to provide a comprehensive view of the development and evaluation of the Henshin Web application:

Chapter ?? - Theoretical Background introduces the foundational technologies and concepts essential for understanding this work. It covers the Eclipse Foundation ecosystem, EMF as the modeling framework, Henshin for model transformations, and GLSP as the web-based graphical editing platform. This chapter establishes the technical context and terminology used throughout the thesis.

Chapter 3 - Related Work surveys the landscape of model transformation tools and web-based modeling solutions. It examines scientific literature on model transformation software, analyzes existing tools and their limitations, and compares various web-based modeling environments. This analysis positions Henshin Web within the broader context of available solutions and highlights the gap it aims to fill.

Chapter ?? - Requirements defines the functional and non-functional requirements for the Henshin Web editor. It identifies potential user groups, establishes the system scope and context, and details the specific capabilities the application must provide. This chapter serves as the foundation for design and implementation decisions made in subsequent chapters.

Chapter ?? - Architecture presents the overall system architecture of Henshin Web. It describes the structural design decisions, component interactions, and architectural patterns employed to meet the identified requirements. The chapter explains how the web-based architecture integrates with existing Henshin and EMF ecosystems while providing the desired accessibility improvements.

Chapter ?? - Implementation details the concrete implementation of core components within the Henshin Web application. It covers the technical realization of key features, integration challenges, and solutions developed to adapt Henshin capabilities for web environments. This chapter provides insight into the practical aspects of translating architectural designs into working software.

Chapter 7 - Testing and Evaluation discusses the comprehensive testing strategy employed to validate the application's functionality and performance. It covers unit testing approaches, end-to-end testing methodologies, and evaluation criteria used to assess the system's effectiveness. The chapter also addresses testing limitations and their implications for the validation of research outcomes.

Chapter 8 - Deployment explores various deployment strategies and options for making Henshin Web accessible to users. It examines different hosting approaches, infrastructure requirements, and considerations for scalability and maintenance. This chapter addresses the practical aspects of delivering the solution to end users.

Chapter 9 - Usage provides comprehensive user guidance for working with Henshin Web. It includes detailed user guides for creating and editing metamodels, transformation rules, and instance files, as well as administrative guidance for user management and system configuration. This chapter serves as practical documentation for both end users and system administrators.

Chapter ?? - Discussion and Conclusion synthesizes the research findings, evaluates the success of the approach in addressing the identified problems, and reflects on the broader implications for web-based model transformation tools. It discusses limitations of the current implementation, potential future enhancements, and the contribution of this work to the field of model-driven engineering.

2. Background

In this section, the theoretical background of the project and used technologies are described. First the Eclipse Foundation is introduced, as many used frameworks are developed under the Eclipse Foundation. Then, the Eclipse Modeling Framework is described, as it is the core of the used frameworks. After that, the model transformation language Henshin is introduced. Finally, the framework GLSP is described, that is used to create web-based diagram editors.

2.1. Eclipse Foundation

The Eclipse Foundation is a not-for-profit, member-supported corporation that provides an environment for individuals and organizations for collaborative and innovative software development. [7] The Eclipse Foundation grew out of the publication of the Eclipse IDE code from IBM in 2001. The Eclipse Foundation itself was founded in 2004. The new organization was founded to continue the development of Eclipse IDE as an open source platform. Over time, the organization initiated numerous projects in the Eclipse environment, all operating under the Eclipse Public License. [8, 7] In the recent years, the key initiatives of the Eclipse Foundation are contributing to european digital sovereignty, enhancing security measures, innovating Software-Defined Vehicle (SDV), organizing community events, and improving their most popular projects. Popular projects are for example the Jakarta EE, an ecosystem for cloud-native applications with java, Eclipse Temurin, providing open source Java Development Kits and the Eclipse IDE. [9] In total, the Eclipse Foundation hosts more than 400 open source projects, supported 14 european research projects in 2024, and has 117 organizations participating in commits. [9]

The scope of this work remains within the Eclipse Foundation ecosystem. All frameworks used are projects from the Eclipse Foundation. The used frameworks are described in the sections 2.2, 2.3 and 2.4.

The Eclipse IDE is not the main project, but it is still an important part of the Eclipse infrastructure. It is divided into four main components: Equinox, the Platform, the Java Development Tools (JDT) and the Plug-in Development Environment (PDE). Together they provide everything to develop and extend Eclipse-based tools. Equinox and the Platform are the core of the Eclipse IDE. With expanding the core with the JDT or other plugins, the IDE can be used to develop different programming languages, like Java, C/C++, or PHP. [2] Eclipse provides different packages to download, depending on the use case. One package is the Eclipse Modeling Tools package by the Eclipse Modeling Project. It provides tools and runtimes to build model-based applications. It can be used to graphically design domain models and test those models by creating and editing dynamic instances. Also, Java code can be generated from the models to get a scaffold that can be used to create applications on top. [10] The base of the Eclipse Modeling Tool is EMF (section 2.2). Other modeling tools and projects that are built on top of the EMF core functionality provide capabilities for model transformation, database integration, or graphical editor generation. [2]

2.2. Eclipse Modeling Framework (EMF)

> „Eclipse Modeling Framework (EMF) is a modeling framework and code generation facility for building tools and other applications based on a structured data model.“ [11]

Eclipse Modeling Framework (EMF) is the core part of the Eclipse Modeling Project and unifies the representation of models in UML, XML and Java. You can define your model in one of these formats and use EMF to generate the other formats.

EMF consists of three components. The EMF core part provides Ecore metamodels, runtime support for the models, and a basic API for manipulating EMF objects generically. Ecore metamodels are used to describe the structure of a model. [12] They can be serialized in XMI 2.0, as Ecore XMI, and have the file extension *.ecore*. There are several Ecore classes to represent a model, here are the most important ones:

- **EClass**: A class in the model that is identified by a name, containing attributes and references to other classes. It can also refer to a number of other classes as its supertypes to support inheritance. [2]
- **EAttribute**: An attribute of a class, that are identified by a name and have a type. [2]
- **EDataType**: A simple data type like `EString`, `EBoolean` or `EJavaClass`. [2]
- **EReference**: A reference to another class, containing a link to an instance of that class. [2]

Together, Steinberg et al. called these classes the Ecore kernel. In Figure 2.1 you can see the kernel classes and their relations. These classes are enough to define simple models. **EAttribute** and **EReference** have a lot of similarities. They both define the state of an instance of an **EClass** and have a name and a type. For that, Ecore provides a common interface for both, called **EStructuralFeature**. Ecore can also model behavioral features of classes as **EOperation** using **EParameter**. All classes have the common interface **EObject**, being the root of all modeled objects. Related classes are grouped into packages called **EPackage**. It is represented by the root element when the model is serialized. [2]

The second component of EMF is EMF.Edit. It provides generic reusable classes to build viewers and editors for EMF models. With these classes, EMF metamodels can be displayed in JFace viewers, that are part of the Eclipse UI. [12] The Eclipse IDE can display an Ecore model in a tree viewer. Eclipse accesses the data over the `ITreeContentProvider` interface to navigate the content and the `ILabelProvider` interface to provide the label

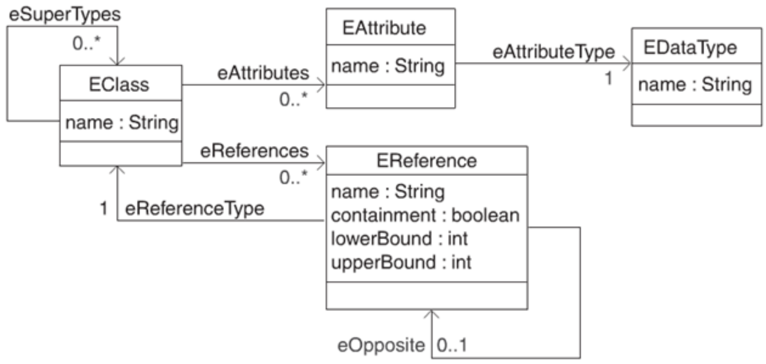

Figure 2.1.: The Ecore kernel. Image obtained from [2]

text and icons for the displayed objects. The properties of objects are displayed in a Property Sheet over the `IPropertySourceProvider`, where the user can edit the model. EMF.Edit also provides undo and redo operations when creating or editing an instance model. For that, it uses a command framework with commands like an `AddCommand`, `SetCommand` or `CopyCommand`. [2]

The third component is EMF.Codegen. It can generate Java code for a complete editor for EMF instance models of an Ecore metamodel. It provides different generation options. So, unlike EMF.Edit, that just provides generic classes for Ecore models, EMF.Codegen directly generates complete editors with a UI. [12] The generation can be done over a wizard in the Eclipse IDE or by using the command line interface. [2] The generation can be separated into three levels. The first level is to generate Java interfaces and implementations for the Ecore model classes and a factory- and package-implementation class. The second level generates specific `ItemProviders` to edit instance models based on the metamodel. The classes are structured like the EMF.Edit component for the Ecore models. The third level generates a structured editor with UI that works like the Ecore editor in the Eclipse IDE and can be a starting point for customization. [12] There are many frameworks that build on top of EMF, using these generation capabilities to create further modeling functionality. For model transformations the most popular frameworks that build upon EMF are Eclipse Acceleo, Eclipse VIATRA, Eclipse ATL, Eclipse QVT Operational, Eclipse QVT Declarative and Henshin 2.3.

2.3. Henshin

One part of the Eclipse Modeling Project for model transformations is Henshin. It can be used as a plugin in the Eclipse IDE or as an SDK. It provides a graphical and textual syntax to define model transformation rules and apply them to EMF XMI instance models. It can be used for endogenous transformations, where EMF model instances are directly transformed, and exogenous transformations, where new instances are generated from given instances using a trace model. It also brings efficient in-place execution of transformations using an interpreter with debugging support and a performance profiler.

Henshin also provides conflict and dependency analysis, and state space analysis for verification. [3]

Henshin builds on top of EMF. It uses an Ecore metamodel to define the structure of the transformation rules, resulting in a serialized XMI file with the file extension *.hensin*, that can therefore be edited in the Eclipse tree editor. [3] The metamodel of the transformation rules uses another Ecore metamodel that models the model structure of the domain, to type the nodes, egdes, and attributes of the rules. [13] In Figure 2.2 you can see the Henshin transformation rule metamodel. A rule consists of a Right-Hand Side (RHS) graph, a Left-Hand Side (LHS) graph and attribute conditions. Additionally, mappings between the LHS and RHS graph are defined between nodes. The mapping of the edges is done implicitly by the mapping of the source and target nodes. [13] Henshin uses units to control the order of rule applications. With units, control structures can be defined. Also, parameters can be passed from the previous executed rule to the next one to have a controlled object flow. Henshin's tansformation language is based on algebraic graph transformations, complying with the syntactical and semantic structure of rules and transformation units. This ensures a language usable for formal verification or validation. [13]

In the Eclipse IDE, rules can also be edited in a graphical editor. The rules are displayed as a single graph, calculated from the LHS and RHS graphs. The nodes and edges are annotated with <<*preserve*», <<*create*», <<*delete*», <<*forbid*» or <<*require*» to indicate what happens to the nodes and edges when applying the rule. These annotations can be directly edited in the graphical editor and the the LHS and RHS graphs are then adapted to the change. Also, multiple Negative Application Conditions (NACs), Positive Application Conditions (PACs) and parameters can be specified directly. [3] When a set of transformation rules are specified, they can be applied to an EMF XMI instance model, by using a wizard in the Eclipse IDE. There, the source model, the rule, and its parameters can be selected. The result of the transformation can be seen in a new XMI instance file. [3] Next to the graphical editor, Henshin also provides a textual syntax to define transformation rules and units. In a *.henshin_rule* file with the keyword **rule** a name and parameters, a new rule can be described. If you want to define a node, you can use the keyword **node** with a action keyword like **create** or **preserve** to specify the action of the node in the transformation.

The Henshin SDK consists of multiple packages oriented to the package structure of EMF. Next to a model, edit, and editor package, it provides an interpreter package, that contains a default engine to execute model transformations.

2.4. Graphical Language Server Platform (GLSP)

GLSP is a framework that provides components for the development of GUIs for web-based diagram editors. [5] It is organized within the Eclipse Cloud Development project. [6] With the framework, custom diagram editors for Eclipse Theia, Eclipse IDE, Visual Studio Code, or standalone web apps can be created. It uses a client-server architecture, where the client is implemented with TypeScript and for the server, GLSP provides

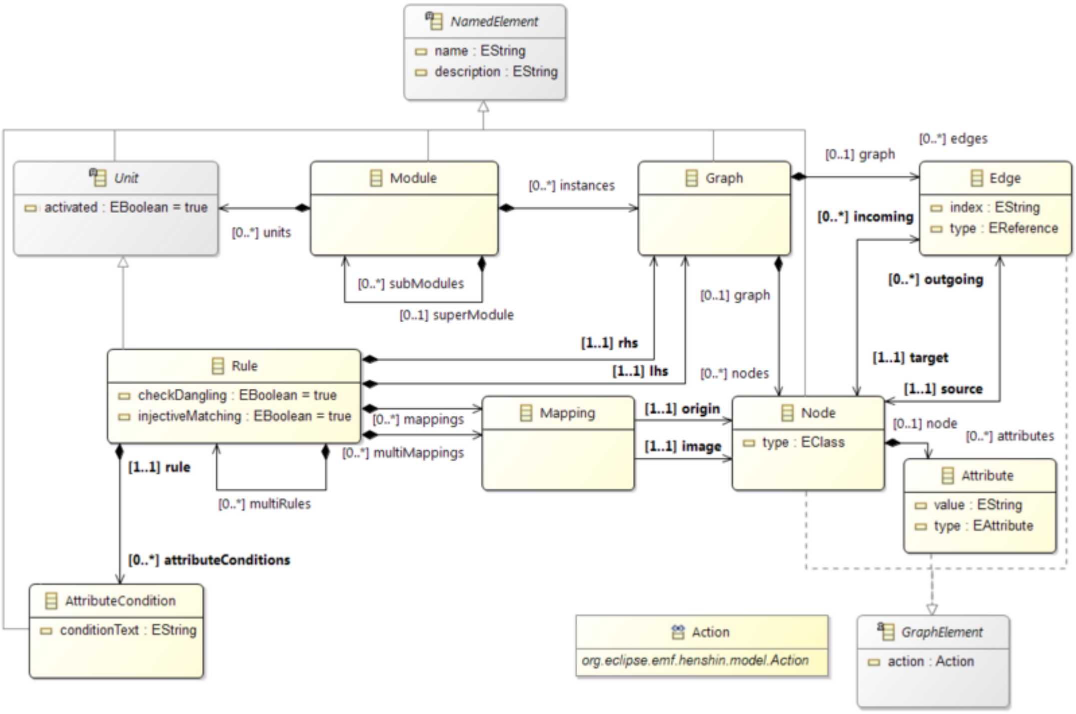

Figure 2.2.: Henshin transformation rule metamodel. Image obtained from [3]

implementations in Java and TypeScript based on nodejs, even though the server could be implemented in any programming language. As the server for this project is implemented in Java, the following discussion focuses exclusively on the Java implementation of the GLSP server. Client and server communicate over JSON-RPC with an action protocol that is similar to the Language Server Protocol [14].

The GLSP server is responsible for loading a source model and defines how to transform it into the graphical model, that should be displayed. The source model can be of any format, e.g., a database, JSON file, or an EMF model. GLSP provides dedicated modules for loading EMF models. The Java server uses Google Guice [15] for Dependency Injection (DI). The GLSP server distinguishes between DI containers. There is one server DI container to configure global components that are not related to specific sessions. For every client session, there is a diagram session DI container, that holds session specific information, handlers, and states associated with a single diagram language. In Figure 2.3 you can see that the diagram session DI container run inside the server DI container. GLSP provides some abstract base classes that have to be implemented to create a working diagram server language, that can provide a diagram to display at the client. All concrete implementations of one diagram language have to be registered in a `DiagramModule`. The server can handle multiple diagram languages by providing different diagram modules. There are some classes that have to be implemented. The interface `SourceModelStorage` defines how to load and save the source model. There is already a default abstract implementation for EMF models, that loads the XMI file into a `ResourceSet`. The interface `GModelFactory` is used to map the source model to the GLSP internal graphical model structure. Here also an abstract `EMFGModelFactory` is provided. Another important part is the `GModelState` interface, that defines the state of a client session and holds all information about the current state of the original source model. All services and handlers use the `GModelState` to obtain required information for their tasks.

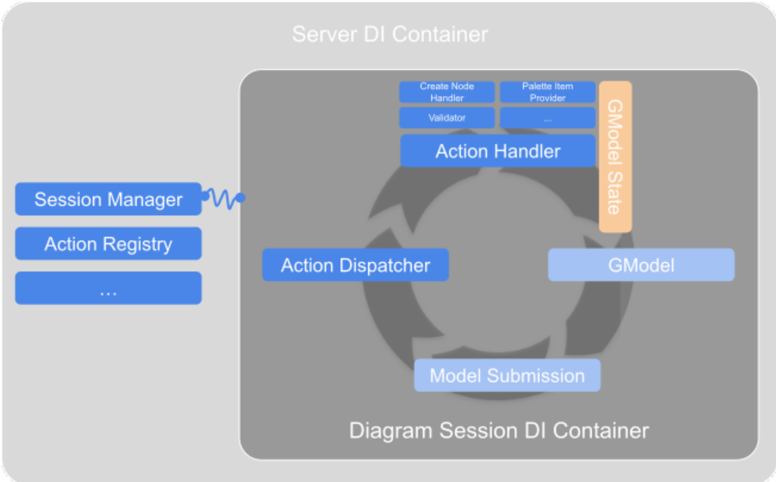

Figure 2.3.: Server DI Container vs Diagram Session DI Container. Image obtained from [6]

When the diagram should be displayed in the editor, the client sends a `RequestModelAction` with a URI of the source model to the server. The server invokes the `SourceModelStorage` to load the source model and then uses the `GModelFactory` to translate it into the graphical model, which is then sent to the client to render it. For an edit operation, the client sends the operation request to the server, where the corresponding handler is invoked. The handler modifies the source model directly. After that, the server invokes the `GModelFactory` again to map the newly modified source model into a new graphical model, which is sent to the client to re-render. The two use cases share many steps. Since a new graphical model is created every time, the format of the source model is independent and can be of any format. [6]

The GLSP client is responsible for rendering the graph and managing user interactions. The client requests all possible editing operations that can be performed on the specific model. As the client for this project is integrated into Eclipse Theia, the following discussion focuses exclusively on the Theia integration of the GLSP client. [6] GLSP provides four main UI components to apply commands or edit the graph but also allows custom User Interface (UI) extensions:

- **ToolPalette**: The ToolPalette is an expandable UI element located on the top left of the diagram editor. By default, it provides basic options to switch between selection, deletion, and marquee tools, validate the model, reset the viewport, and search in the listed operations below. Below it lists all nodes and edges that can be created in the diagram by default. It can be extended with custom actions by implementing and registering the `ToolPaletteItemProvider` interface at the server. [6, 5]

- **CommandPalette**: The CommandPalette can be invoked by pressing *Ctrl+Space*. It provides a search field to search for commands or actions that were registered. Commands can be registered by implementing and registering the `CommandPaletteActionProvider` to the server or implementing and registering the `CommandContribution` interface to the Theia frontend module. [6, 5]
- **ContextMenu**: The ContextMenu is a popup menu that can be opened by right clicking inside the diagram editor. There, any commands or actions can be structured as needed. It can be customized by implementing and registering the `ContextMenuItemProvider` to the server or implementing and registering the `MenuContribution` interface to the Theia frontend module. [6, 5]
- **EditLabelUI**: Labels of nodes and edges can be edited by double-clicking on the label. The EditLabelUI provides an input popup to edit the label text. [6, 5]
- **Custom UI Components**: Custom UI extensions have to extend `AbstractUIExtension` that provides a base HTML element and can then be registered to the client. The base class also provides functionality to show, hide, or focus the element. These UI extensions can also be enabled over a `SetUIExtensionVisibilityAction` from the server. [6, 5]

GLSP uses Sprotty [16], a web-SVG-based diagramming framework, to render the diagrams. The graphical model of GLSP called *GModel* is based on the *SModel* of Sprotty and works as a compatible extension. The graphical model is composed of shape elements and edges. They are organized in a tree, that starts with the `GModelRoot`. There are several base classes, that can be extended and also new types can be added. The `GEdge` represents an edge between two nodes or ports. Four classes inherit from `GShapeElement`, which represents an element with a certain shape, position, and size. They can also be nested inside another `GShapeElement`. The `GNode` can have `GLabel` or `GPort`, which represents a connection point for edges, as children. The `GCompartment` can be used as a generic container to group elements. The Java server uses EMF to handle the graphical model internally, to profit from the command-based editing capabilities of EMF. To send the graphical model to the client, it is serialized into JSON using GSON [17] and then sent over JSON-RPC. [6]

The layout of a graph is divided into macro and micro layouting. The macro layouting, which arranges the nodes and edges of the model, is done by the server. The client does the micro layouting by calculating the positioning and size of elements within a container element. [6] For the macro layouting, GLSP provides a notation model, that persists the position and size of the elements in a separate notation XMI file. The notation diagram can be added to the `GModelState` and then used in the `GModelFactory` to specify the layout. [5] GLSP also provides a `LayoutEngine` interface, that can be used to layout the elements of a graph that have no persisted layout yet. [6]

GLSP also provides an interface to validate the model. With the `ModelValidator` interface, specific validation rules can be defined by the server. The validation returns a list of markers that can be an info, warning, or error. The markers are then displayed in the GLSP client. The markers can also be integrated into the Theia Problems View.

3. Related Work

3.1. Scientific Literature

3.2. Existing Tools and Technologies

There are many existing tools for model transformations. Kahani et al. created a survey in 2019 of various model transformation tools. They classified 60 different tools, including Henshin. In Figure 3.1, you can see how many tools provide specific execution environments. 73% of the tools provide plugins for the Eclipse IDE, and 20% of the tools are integrated or dependent on other IDEs. 18% have no IDE support, and only two tools are web-based. In total, 89% of the tools have external dependencies such as an IDE or other tools. Dependencies often complicate the installation and usage of the tool. [18]

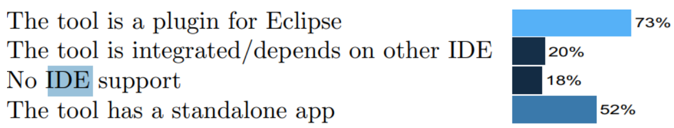

Figure 3.1.: Execution environments of model transformation tools. Image obtained from [18]

One web-based tool included in the survey is A Tool for Multi-Paradigm Modeling (AToMPM) [19]. It is a web-based modeling tool to create Domain Specific Modeling Language (DSML) environments, performing model transformations and manipulating and managing models. [19] It was created in 2013 and supports all model transformations that are based on T-Core [20], a minimal common basis that allows interoperability between different model transformation languages. [20] Metamodels can be defined with a simplified UML language. The graphical modeling environment offers debugging and the ability to collaborate and share modeling artifacts in the browser. [19]

There are also other web-based tools for MDE. WebGME [21] is a web-based modeling tool, created in 2014. It allows to collaboratively design DSMLs using model versioning and broadcasting changes to all active users. It supports prototypical inheritance, where any model can be instantiated recursively, so changes are propagated down the inheritance tree. It also provides scalability, collaborative modeling and model versioning. Metamodels and compositions can be created with WebGME, but no graph transformations can be applied to a model. Even though model transformations are not possible, the editor was one of the first solutions for web-based modeling tools. [21] The software provides extension points to customize or extend the software, but no model transformation capabilities were added by any available extension. [22] The tool is still hosted and maintained, to be used for free. [22]

WebDPF [23] is another web-based modeling tool, published in 2016. Compared to WebGME and AToMPM, it supports model navigation and element filter capabilities, a JavaScript editor for writing predicate semantics, reusability of transformation rules, partial model completion, and a termination analysis. These features try to improve the usability of the tool. [23] Even though the tool had improvments upon existing tools, the originally mentioned hosted WebDPF portal is offline by now.

There is also a GLSP-based Ecore metamodel editor, created by the GLSP development team. It was implemented with the GLSP version 0.9 but never updated further. It allows to create and edit EMF Ecore models in a Theia web editor. Even though the project cannot be used directly, due to the use of another source model format and breaking changes in major updates of the GLSP framework, it provides various classes that can be used as a template for the Henshin Web Ecore viewer. One example is the factory code that maps the EMF Ecore model to the graphical model. [24] The findings show, that there are many existing model transformation tools, but only very few web-based solutions, that provide an easy entry into MDE and model transformations. Henshin web tries to fill this gap.

3.3. Comparison and Gaps

4. Requirements Analysis

The purpose of this chapter is to systematically identify, analyze, and document the requirements of the software system developed in the context of this thesis. The chapter outlines functional and non-functional requirements as well as the system stakeholders and constraints.

4.1. Stakeholders and User Needs

- **Students**: Students who want to learn about MDE and transformation rules, want a very simple and intuitive entry into the topic. Trying out transformations in the browser is a good start, without having to install a lot of software. For Students the core functionality is sufficient, as they only want to try out transformation rules and learn how they work.
- **Researchers**: Researchers, that are researching for MDE and come arcross Henshin, want to be able to test or try out model transformations of Henshin. For Researchers the core functionality could be sufficient, but editing capabilites of the transformation rules and metamodels are pracitcal for them.
- **Software Engineers**: Software engineers, that are using MDE, want to be able to test their transformation rules. They want a powerful editor and a colaborative environment to work on their models. For Software Engineers the defined additional functionality as well as some code generation support are needed to cover their needs.

The different Stakeholders show, that for more features the application provides, more users can be reached. With more features and use cases the application can cover, it provides more value for enterprise users, that work for production systems.

4.2. System Scope and Context

The Eclipse IDE plugin of Henshin works as a template for the functionality of the application. It provides functionality to create, edit and apply Henshin transformation rules for Ecore metamodels on XMI instances. To create a full enterprise application, that will get used for projects in the industry, the application has to also provide very similar functionality. The defined core functionality is the minimum set of features that the application must provide to be useful for the users. They are only a subset of requirements to provide a full web-based copy of the Henshin Eclipse plugin.

The core functional requirements extends already existing functionality, that Eclipse Theia and GLSP already provide. Theia provides various views like the explorer for basic file management, including opening, saving, closing, creating and deleting files, a problems view, a integrated terminal, or edit operations like copy or paste. GLSP provides default functionality for each graphical editor. That includes selection of elements, moving nodes, realinging edges, zooming, or moving and reseting the viewport. Most of these features can be further configured to be able to create an editor fitting the specific needs.

4.3. Functional Requirements

The main use case that the application should support is that a user can try out transformation rules on EMF XMI instance files. In this usecase, the user already has a metamodel and transformation rules. He wants to test the transformation rules on various instances in an accessible, intuitive, and easy to use graphical editor. From that use case the following core functional requirements can be derived.

Core Functionallity:

- EMF XMI instance files should be displayed in a graphical editor. That contains the nodes, edges and attributes of the model.
- The XMI instance editor should provide editing functionality to create, update and delete nodes, edges and attributes.
- In the XMI instance editor all applicable transformation rules should be listed. When a rule is selected to get applied, all parameters have to be specifiable.
- When a rule gets appplied, the graphical editor of the XMI instance should be updated to reflect the changes made by the transformation rule. The application should also support undo and redo functionality for the applied transformation rules.
- Henshin transformation rule files should be displayed in a graphical editor. That contains the nodes, edges and attributes of the model and their action types. The user should be able to switch between all rules of a *.henshin* file.
- EMF Ecore metamodel should be displayed in a graphical editor. That contains the nodes, edges and attributes of the metamodel.

Next to the core use case, the second use case is that a user wants to create a full transformation language from scratch, that can be used to model and test the system and generate production code from it. In this use case, the user wants to create and edit metamodels and transformation rules. To support this use case, the following additional functional requirements are defined:

Additional Functionallity:

- The Henshin rule editor should provide editing functionality to create, update and delete nodes, edges, attributes and their action types.

- The Ecore metamodel editor should provide editing functionality to create, update and delete nodes, edges and attributes.
- Henshin transformation units are also listed in the XMI instance editor and can be applied.
- Show the possible transformation rule matches in the XMI instance editor, when selecting a transformation rule.
- Provide the functionality to apply a State Space analysis on a XMI instance.
- Provide the functionality to apply a conflict and dependency analysis on a XMI instance.

There exist many more use cases for model transformations and MDE in general. The application can grow to a web-based platform for MDE in th future. Additional functionality will be disscused in section 10.4 but these usecases are not scope of this thesis.

4.4. Non-Functional Requirements

In addition to the core functionality, the system must meet several non-functional requirements:

Non-Functional Requirements:

- The application should be web-based and preferably accessible via a web browser.
- The application should be responsive and work on different screen sizes. It does not have to support mobile devices and touch interactions, since GLSP is also not supporting touch interactions [5].
- The application should be user-friendly and intuitive to use. For that, the application should follow the design principles of GLSP and Eclipse Theia. That includes the use of views of theia, like the exploerer and the predefined UI contorls of GLSP, like the tool palette or the context menu.

4.5. System Constraints

One constraint is the use of Henshin as a tranformation language. Henshin is a Java-based framework, which menas that the application needs a possibility to run Java code in the backend. The easiest way for that is to use a Java-based backend, that can directly use the Henshin SDK code. The use of Henshin also brings the constraint that, metamodels and instances are based on EMF.

Another constraint is the use of web-based technologies and preferably a resulting web application. For model transformations, there exist many applications, but not many of them are web-based. This constraint is also a non functional requirement and was also motivated in previous sections. The initial version of the application will support English only.

5. System Design and Architecture

In this chapter, the architecture of the system is described. The system is designed to achieve following goals. The system should be modular and easily extensible to allow future extensions towards a full model transformation platform for production use cases. The system should also be maintainable. In the folowing sections, the high-level architecture following the GLSP architecture is described. Then the design of the components, control flow and data models is described. In the end the UI design is explained.

5.1. Design Decisions

In the section 2 the used frameworks and technologies were described. The selection of these framworks still leave some open design decisions. One open decision was which platform integration to use for the GLSP client. GLSP can be used as an extension for Eclipse Theia or VS Code, a plugin for the Eclipse IDE or as a standalone web application. They can also be used in combination, but to avoid overhead and complexity, only one platform integration is initially used. In table 5.1 the different integration options are compared. Since the integration into an existing IDE fits the graph editors, the standalone editor is not an option. For that, many additional features like a file explorer have to be implemented. The integration into the Eclipse IDE is also not an option, since it is not based on web technologies and therefore not satisfying the requirements of the project. Between the Eclipse Theia and VS Code integration, Eclipse Theia is providing more flexiblity in the usage and deployment of the application. Next to the usage as an extension that can be added during runtime, Theia also provides the option to bundle your own IDE including the GLSP graph editors. That makes it deployable as a complete application, where no additional plugins are needed. This flexibility is the main reason to choose the Eclipse Theia integration as the inital main platform for Henshin Web. With that usage and deployment flexibility, different additional platform integrations are probably not needed in the future.

Another decision was to select a edge routing style. GLSP provides two different routing algorithms, the Manhattan and Polylime stlyes. The Manhattan style was invented by Koh and Madden to achieve wire length optimization in circuit design. It only uses vertical and horizontal lines to connect nodes. [25]. The conncetion can be split into multiple segments, changing from a horizontal to a vertical line or the other way round to create a stair like connection between nodes. The Polyline style on the other hand uses straight lines to connect nodes. They line can also be split into multiple segments with arbitrary angles between them. For this use case, the main aspect is the clarity and readability of the graph. You can see the comparison of the two styles in figure 5.1. Advantages of the Manhattan style are that can prevent edge crossings, and therefore can

Table 5.1.: Comparison of GLSP Platform Integrations

Criteria	**Eclipse Theia**	**VS Code**	**Eclipse IDE**	**Standalone**
Deployment Options	Web-app, Desktop (Electron)	Desktop, Web-app	Desktop	Custom (Web or Desktop)
Extendability	Access to all Theia internal APIs	Through VS Code Extension APIs	Moderate, via Eclipse plugins (OSGi-based)	Fully customizable (with own implementations)
Provided Environment	Complete IDE	Complete IDE	Complete IDE	No other features included
Result Format	Own IDE or as a plugin	VS Code extension	Eclipse IDE plugin	javascript based web editor module
Dependencies Needed	browser	VS Code Desktop or browser	Eclipse IDE	browser

help reduce visual clutter. In complex diagrams with many nodes and edges, it is easier to trace the horizontal and vertical lines. On the other hand, it can overlap with other edges, which can make it hard to follow the edge. Especially named edges that overlaps partly with another edge can't be followed without clicking and highlighting it. You can see that in figure 5.1 between the *Bank-Account* and the *Account-Client* edge. To prevent that, the edge routing needs be stored in the notation file to be able to persist changes in the routing that remove overlapping edges. The Polyline style on the other hand is generally simpler and more compact. It can get very clutterd with many edges crossing and other nodes overlapping the diagonal lines. Metamodel, transformation rules and instances are typically not that complex, so that a simple line bewteen nodes is sufficient and additonal edge segments are not needed. Because of that, the edge placement doens't have to be stored in the notation file. The edge automatically aligns itself when a node is moved. The rotation of the edge label that it runs parallel with the line supports the simple and compact design of the Polyline style. To additionally prevent crossing lines, a option to dynammically hide the root node and its edges in XMI graphs is introduced.

One main question in the UI design was where to put the selection of the transformation rules of a *.henshin* file. There are several options to display the rule seletion. The first option is to add the list of rules to the tool palette as an additional pallete group next to the nodes and edges. Here no additional new UI element must be placed in the graph editor, but the main focus of the tool palette is to provide editing tools of the graph elements. Swapping between the rules doesn't fit into the main purpose of the tool. Another option is to create a custom UI element that is displayed in the rule graph editor. I would be easy to implement, directly integrated into the graph module and platform independent. But it would take up additional space in the graph editor view. The rule graph should be the main focus of the application and should have enough space to display the graph elements, especially for larger graphs.

For these two options, the user also has always to switch to the rule graph editor first to select a rule. It can negatively impace toe user experience. That would not be the case if the rule selection is integrated into the Theia explorer. Since the custom explorer is used anyway to select between different instance, transformation rule or

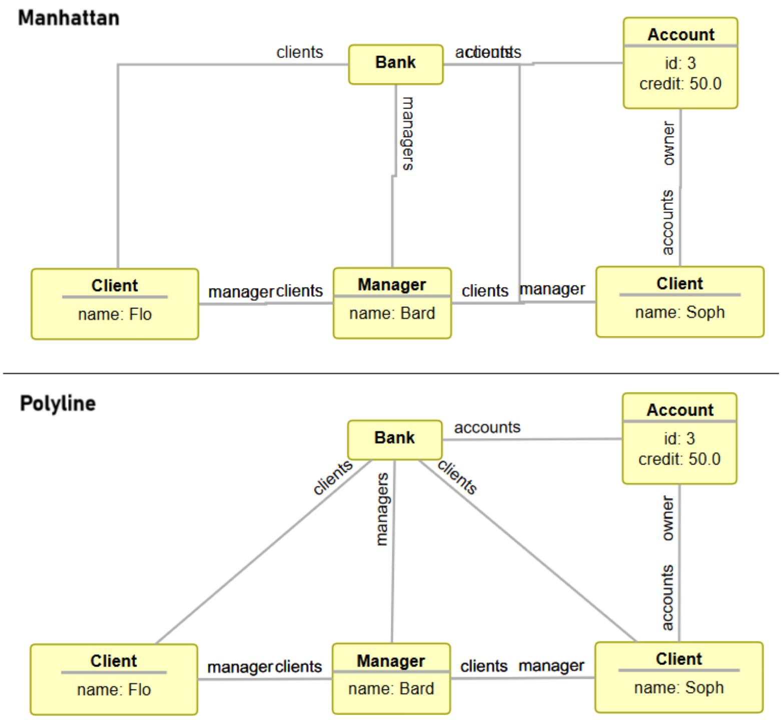

Figure 5.1.: Visual comparison of the two edge routing styles of GLSP

metamodel files, it is a good place to also select the transformation rules. The explorer can also be collapsed to save more space for the graphs and for very many rules it automatically supports scrolling. Extending theia internal API makes it more effort to add aditional GLSP platform integrations, because the custom changes need to be newly implemented for the new platform, it may even not support the same extendability. Since the theia integraion provides the most options to use and deploy the application, more platform integrations are probably not needed. Aditionally, the extension of the theia explorer prevents the occupation of additional space in the GLSP editor widget to select a rule. This improvement of the user experience and intuitiveness outweights the possible additional effort for new platform integrations. The implementaiton of the custom explorer is described in section 6.3.5.

5.2. Following the GLSP Architecture

The system is based on the GLSP architecture, that uses a client-server architecture. The client and server communicate via a websocket connection and JSON-RCP. The GLSP server can be implemented with Java or Node.js, but due to the constraint that Henshin is implemented in Java, the server is also implemented in Java. The client is implemented in TypeScript. GLSP provides a defined protocol for the communication between client and server, which can extended with custom commands and actions. The communication is performed using Action Messages, that can be sent from the client and the server to each other or also to itself. The client and the server have Action Handlers, that process the Action Messages and perform the corresponding actions. Each client connection starts its own server instance, therefore each server is only responsible for one client. [6] Since each client needs to be able to display three different graph editors for different file types, the server consists of three diagram modules. Each diagram module defines a different diagram language. The `XMIDiagramModule` is responsible for the editor of XMI instance files, the `RuleDiagramModule` is responsible for the editor of Henshin rule files and the `EcoreDiagramModule` is responsible for the editor of Ecore metamodel files. In figure 5.2 you can see the high-level architecture of a server and client instance. The architecture of the three diagram modules is quite similar. Each diagram module has a `ModelState` which is the central statefull object within a client session [6]. The `ModelState` is accesed by all other services and handler and represents the current state of the actual source model. GLSP supports the integration of EMF models as the underlying source model for the diagrams by default. For that The `EMFSourceModelStorage` can load a EMF file as a `RessourceSet`, that is then attached to the `ModelState`. That allows an simple integration of the Henshin SDK, since it based on EMF and provides a `HenshinRessourceSet` can be loaded directly over the EMF integration of GLSP into the `ModelState`.

The `ModelState` of each diagram module also contains an index and a notation model for the layout of the elements in the graphical editor. To be able to have a consistent layout of the elements, not changing after every reload or action, the position and size of each element for each model file is stored in a seperate *.notation* file. The index of the

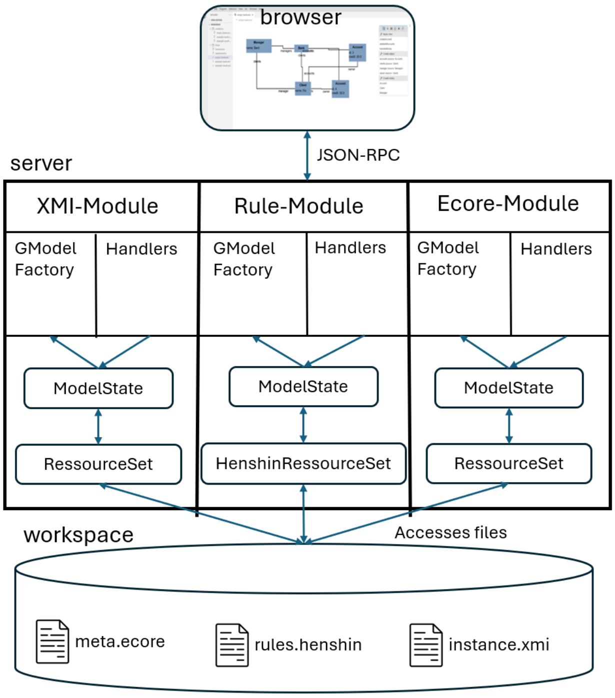

Figure 5.2.: High-Level Architecture of the System

`ModelState` is used to map the elements of the source model to the graphical model of GLSP. For each diagram module, the indexing is implemented in a different way. (see section 6.3.4 for more details).

Another important part of each diagram module is the `GModelFactory`, which is responsible for creating the graphical model that is sent to the client from the source model. Since metamodel, transformation and instance EMF model are strucutred differently, each `GModelFactory` of each diagram module is implementing its own mappings.

5.3. Data Models and Structures

All three diagram modules have an EMF based source model. For the Ecore metamodel and the XMI instances, The standard data model of EMF is used. As described in section 2.2 different implementations of `EObject` are used, representing all used elements like nodes, attribues or references. For the Hensin transfomations model, the data model of the Henshin Software Development Kit (SDK), that builds upon the EMF data model, are used. No additional data structures are needed, since every created domain model is based on the EMF data model. The data model of the graphical representation is provided by GLSP. It can be extended with custom elements, but the default elements are sufficient for the current use cases.

The user has to select or create a workspace in the UI, where all the source models are located. Each workspace for Henshin Web has to be in a specific structure. It should contain one *.ecore* metamodel and one *.henshin* transformations file. Additionally, arbitrary *.xmi* instance files can be added. All of these files should be stored in the root folder of the workspace. When creating a new model file or opening it for the first time, a new notation file is generated and stored in the *.notation* subfolder of the workspace. These notation files are not displayed in the theia explorer.

5.4. GLSP Client Structure

The GLSP client is divided in different main modules. The *henshin-glsp* module is responsible for platform independent code. It contains client side action handlers, custom UI extensions and custom graph elements. This module is used by the three theia specific modules, that are responsible for the integration of the GLSP client into the Eclipse Theia framework. There is one module for each diagram type. The target specific code is loacted here. One example is the customization of the Theia explorer view, that it also displays all rules of a *.henshin* file and hides the notation files. These three theia extensions are then combined in the *henshin-browser-app* module, that has no additional code, but only combines the three theia modules into one application.

5.5. User Interface Design

The design of the user interface is based on the design principles of GLSP and Eclipse Theia. For editing the graphs, the main UI element is the tool palette on the right side of the graph editor. It lists all available nodes and edges, that can be added to the graph as well as the transformation rules that can be applied. It also contains a set of predefined GLSP actions, which are switching between selection mode, deletion mode and marquee mode, as well asreseting the viewport and search for tool palette entries. For keyboard usage, ther is also the command palette, that can be opened with the [Ctrl + Space] shortcut. It opens a searchbar with a list of options below. Here all editing operations are registered listed and can be performed by searching or navigating through the list and selecting the desired operation.

The design of the custom UI elements like the parameter selection form or the display of the transformation rule information follows the design of GLSP. The design of Theia uses a flat design with minimal gradients, shadwos or 3D elements. Compared to that uses GLSP a more 3D-like design with shadows and gradients, because the UI elements need to be on top of the main graph plane. That shows that the UI elements are not part of the graph, but are additoinal elements to interact. The tool palette, comamnd pallete and context menu all use shadows and rounded eges. New custom UI elements like the parameter selection also use the same shadow and rounded corners to also show that they are not part of the graph, but elements to interact with.

The colors of the custom UI elements follows the color theme of Theia. Important is that Theia uses the dark mode as a default theme. Graph editors typically use a light backgroung. All custom UI elements are designed to adjust to the dark mode and use the according colors as the default dark Theia elements. The final UI can be seen in appendix A.1, A.2 and A.3.

6. Implementation

This chapter describes the development process and shows the solution and implementation of specific problems, that appeared while implementing the application. The first challenge was to integrate the Henshin SDK into the GLSP project. Another challenge was to index the elements of the different EMF models and formats. One big UI desicion was where to place the selection of transformation rules in the application.

6.1. Development Process

The development of the Henshin Web GLSP editor was done by one person in a time span of about 6 months. Derived from the functional requirements (see section 4.3) the project was split into 7 milestones. The milestones were defined as follows:

- **Milestone 1:** Setup the project and create a diagram editor that can display *.xmi* files.
- **Milestone 2:** Create editing capabilities for XMI instance files.
- **Milestone 3:** Henshin transformation rules can be displayed and applied to the instance model.
- **Milestone 4:** Create an additional diagram editor that can display Henshin rules.
- **Milestone 5:** Create an additional diagram editor that can display Ecore metamodels.
- **Milestone 6:** Create editing capabilities for Henshin rules.
- **Milestone 7:** Create editing capabilities for Ecore metamodels.

Each milestone was split into smaller issues. The first milestone was used to create Proof of Concept (POC) to test the integration of Henshin into a GLSP project. In this phase, the focus was to get to now how the frameworks EMF, GLSP, Henshin and Eclipse Theia work. Even though GLSP provides a well structured documentation and project templates, they didn't cover many use cases for the development of Henshin Web. Henshin also doesn't provide an documentation of their API. Therefore for these frameworks a lot of source code reading and understanding was needed. Git was used as a version control system. The development of a milestone was don in a seperate development branch. When all features of the milestone were implemented, the state of the application was additionally tested and then merged into the main branch.

6.2. Tooling and Environment

For the development of Henshin Web, VS Code [26] was used to develop the client and InteliJ IDEA [27] was used to develop the Java server. For understanding the source code of not well documented frameworks, the use of Chatbot Agents was very helpful. I used Github Copilot in VS Code with the model Claude Sonnet 4 [28] in agent mode. Because it has access to the source code of the dependend frameworks, it can search for specific classes or methods or explain certain concepts. Git was used as version control system and GitLab was used as a remote repository. It was also used for the project management, where the milestones were defined and the issues were created. A issue board was used to show the current sate and progress of the project. The GitLab package registry was also used to store the Henshin maven packages, to be able to access them from the GLSP project. These packages are then available to every contributor of the GitLab, that want to develop on the project. More about the creation on Maven packages will be shown in the next section.

6.3. Code Examples...

This section shows the solution and implementation of specific problems, that occured during the development process.

6.3.1. Integration of Henshin into a GLSP project

The Henshin source code provides both the Eclipse IDE plugin and a Java SDK for using the Henshin interpreter. The project of Henshin is structured as an Eclipse project and is available as a set of Eclipse plugins and features. [3] On the other hand, GLSP projects typically use a Maven project structure. [5] To add dependencies to a Maven project, the dependencies should ideally be available as Maven artifacts. However, Henshin doesn't provide a Maven artifact, since that is not needed for an Eclipse plugin. The Henshin version 1.8.0 is compatible with JDK 11 and higher. GLSP version 2.3.0 has the prerequisite of JDK 17. Therefore, the versions are compatible to run together. The Henshin code consists of 45 plugins, of which 22 are contained in the Henshin SDK, that we need as a dependency in our Henshin Web GLSP project. Each plugin can be downloaded as a JAR file. To create Maven packages from the JARs, a PowerShell script is used. It reads all JARs files from a folder, renames them to the correct Maven artifact name, creates a basic `pom.xml` file for them, deploys them to the GitLab package repository, and creates a list that needs to be included in the Maven `pom.xml` file of the GLSP project. A package of each plugin is created, because for the Henshin Web editor, only some parts of the Henshin SDK are needed. To use the Henshin model package, the additional dependency of the Nashorn JavaScript engine [29] is needed. The Nashorn engine is used to execute calculation expressions of transformation rules. [13]

6.3.2. GModelFactory

The heart of a GLSP server diagram module is the `GModelFactory`. It is responsible for creating the graphical model from the source model. In listing **??** you can see the implementation of parts of the creation of the graphical nodes. The method `fillRootElement(GmodelRoot newRoot)` gets called when a new graphical model should be created. It fetches the source model elements from the `ModelState`, iterates over them, and creates `GNode` elements using a builder pattern. In the method `createNode(DynamicEObjectImpl eObject)`, it is configured how the node should look in the editor. It sets the id, adds CSS classes and configures rounded corners. It also sets the type of the node. The type can be a default type, or custom types, that have their own customized client implementation. In listing **??** you can see for example that the root node of the XMI instance model gets a different type. The type is used to configure that only one root node can exist and that it cannot be deleted, if other child nodes exist. The method `applyShapeData(eObject)` adds the layout information from the notation model. In the `GNodeBuilder` also the builded child elements like the header or the attributes are added. This creates a tree strucutre of graphical elements, that are all atached to the `GModelRoot`. The `RuleGModelFactory` and the `EcoreGModelFactory` work similar to the `XMIGModelFactory`, but they create different node types and handle the source model elements differently.

6.3.3. Layouting

EMF Ecore metamodel files (*.ecore*), Henshin rule files (*.henshin*) and EMF instance files (*.xmi*), don't contain information about the position or size of elements in a graph. [2, 3] To provide a good user experience, the graphical editors need to provide a consistent macro layout for nodes and edges. Newly created nodes should not overlap with existing nodes, and the nodes should stay in the same place after reloading the editor. In general, the GLSP server is responsible for the macro layouting. [6] GLSP provides multiple options to layout the graph. The interface `LayoutEngine` can be used to create a custom layout algorithm, that is applied after the creation of the graphical model from the source model. GLSP provides the `ElkLayoutEngine` implementation, that uses the Eclipse Layout Kernel (ELK) to layout the graphical model. [30] With ELK, different layout algorithms can be used and additionally configured. Even though ELK provides much flexibility for the layout, the layout is newly created after every change to the source model. This means that the layout is not consistent and nodes can move around after every change. To provide a consistent layout, the position of nodes need to be stored in addition to the source model. The GLSP server provides a notation model, that can be used to store the position and size of nodes and edges. [5] This brings the overhead of updating the notation model every time when the source model is updated. GLSP provides classes to make the synchronization of the notation model easier. The notation model is stored in an additional *.notation* file, that is loaded together with the source model and applied to the graphical model in the `GModelFactory` using the `NotationUtil.applyShapeData(shape, builder)` method. To capture changes of position and size of nodes, the GLSP client sends the `ChangeRoutingPointsOperation` and `ChangeBoundsOperation` operations automatically when moving or resizing a node or edge. At the server, the corresponding handlers are updating the notation model using commands to provide undo and redo functionalities.

To achieve layouting in the Henshin Web editor, notation models for the metamodel, Henshin rules, and instances are used. The *.notation* file is created when the source model is loaded for the first time. Here, ELK can be used to create a fitting initial layout. For the XMI instance models, when the graphical model gets created in the `GModelFactory`, the shape data from the notation model is added to the EMF elements over an EMF `Adapter`. Each EMF `EObject` has a list of adapters, that can be used to store additional information. [2] To connect the notation to an element, the `NotationAdapter.getOrAssignNotation()` method checks if the element already has a notation, either returning the existing notation or appending a new Adapter with the notation information. For the Henshin rules and the Ecore metamodels, the notation element mapping is stored in the model index that is contained in the `ModelState`. The reason for the different indexing approaches and their implementations will be explaine in the next section.

6.3.4. Indexing EMF models

Like the layout information, EMF Ecore metamodels and EMF XMI instance models don't by default contain unique identifiers for nodes, edges, or attributes. [2, 11] The graphical model of GLSP on the other hand uses identifiers for each element that is displayed. If no identifiers are specified when creating the graphical elements, GLSP generates its own internal unique identifiers. These identifiers are used in edit operations like renaming or deleting a node, where the graphical element needs to be mapped back to the source model element and only the identifier of the graphical element is sent from the client to the server. To be able to map the graphical element back to the source model element, custom identifiers need to be stored. Additionally, during the transformation of the source model into the graphical model, elements need to be accessed multiple times. For example, a source node is accessed over the EMF package when it is mapped into a `GNode` and then again for all its connected edges and attributes. An indexing of the elements avoid multiple lookups in the EMF source model. To be able to support any created domain meta and instance models and to prevent prerequisites for the use EMF models in Henshin Web, the GLSP server needs to create own inexes for the elements of the source model.

The idexing of the three different source model types is implemented in differnt ways, due to the different internal structures and stored informations. The simplest approach is used for the Henshin rule model. Henshin already creates identifiers for each node and edge of a transformation rule. These identifiers are also stored in the *.henshin* file. When building the graphical model, the identifiers can be accessed over the method `getURIFragment(element)` of the EMF resource. When a new element is created, the index is stored in a bidirectional hash map in the `RuleModelIndex` that is accesible over the `ModelState`. This index can also be used for the notation model, where the semantic element id needs to be stored to be able to map the layout information back to the source model element. One problem of storing the Henshin identifiers is that a transformation rule is stored as a LHS and RHS part. Each part hs its own identifier, even though it is only one element in the graph. For that Henshin also stores mappings of the LHS and RHS elements in the *.henshin* file. To be able to correctly map the source model elements to the graphical model elements, these mappings are also stored in the `RuleModelIndex`. In listing 6.1 you can see the implementation of the methods `getRuleElement(id)` and `getRuleElementId(element)` that are used to get the element from the index or get the index of an element. You can see that before searching the index, the mapping list is checked to ensure that the LHS element is preferably returned, if it exists. That is for example needed for setting the source and target nodes of an edge. If the edge only appears in the RHS part and it should get deleted when applying the rule the `getSource()` method returns the RHS node element, but the source node was initially created from the LHS element. Without the mapping, the source node would not be found in the index and therefore creating an new index, that results in a invalid route in the graphical model.

```java
public void indexRuleElement(String id, GraphElement element) {
    if(ruleElementIndex.containsKey(id))
        return;
    ruleElementIndex.put(id, element);
}

public GraphElement getRuleElement(String id) {
    if (rhsToLhs.containsKey(id)) {
        String lhsId = rhsToLhs.inverseMap().get(id);
        if (ruleElementIndex.containsKey(lhsId)) {
            return ruleElementIndex.get(lhsId);
        }
    }
    return ruleElementIndex.get(id);
}

public String getRuleElementId(GraphElement element) {
    String id = element.eResource().getURIFragment(element);
    if(rhsToLhs.inverseMap().containsKey(id)){
        return rhsToLhs.inverseMap().get(id);
    }
    if(lhrToRhs.inverseMap().containsKey(id)){
        return lhrToRhs.inverseMap().get(id);
    }

    return ruleElementIndex.inverse().get(element);
}
```

Listing 6.1: Parts of `RuleModelIndex`

This problem doesn't appear for the Ecore metamodel indexing because no content independent indexes are stored in the EMF model. Here the indexing is used from the existing GLSP Ecore editor [24]. The `EcoreModelIndex` stores an index for the semantic elements, the notation elements and an additional index for inheritance edges. For the semantic index, random Universally Unique Identifiers (UUIDs) are created. They are used until the client session is closed. During this time, operations on the source model can access EMF elements by their UUIDs over the stored `HashMap` and then apply the operation on the EMF element. The indentifiers are content-independent, which has the advantage, that the identifiers are not changing when nodes are updated. The problem with temporary identifiers on the other hand is, that they cannot be mapped to the source elements after the client session is closed. Therefore, the UUIDs cannot be used in the notation model, because the same notation model needs to be loaded across client sessions. Here, the name of the EMF class is used, since it is unique for each element in the Ecore metamodel. This index has to be updated if a class is renamed. The inheritance index for the Ecore metamodel is used to find already created inheritance edges and retrieve their bend points. With that information, the edges can be connected at bend points to create the typical inheritance arrow structure.

For the notation models of XMI instance models, also content hashes are used as identifiers. Here the name of a object is not unique, because multiple objects of one class can exist. Therefore the content hash, is created from the class name and the names and values of all its attributes. A hash for the class *Client* can look like this: *Client:DynamicEObjectImpl-name:EString=Alice* This content hashes is generated every time the graphical model is created for the first time in a session. It needs to be updated when a attribute value is changed. Content hashes for edges would be even more complex, because they need to include the source and target node hashes combined with the edge type. This is also a reason, why the edge layout information is not stored in the notation model, since the hashes need to be changed for many edit operations to the source model. For XMI instance elements, the additional use of adapters are used. The `NotationAdapter` and the `UUIDAdapter` store the index in the Adapter, which is then directly attaced to the EMF element. This has the advantage, especially for the `NotationAdapter`, that when the content hash has to be updated, the notation model can be directly fetched from the EMF element. It also contains the hashing algorithm for nodes. You can see the implementation of the `NotationAdapter` in listing **??**. These content hashes need to be used for session independent identifiers, but using them as the only identifier would need a lot of overhead to update the hashes. Therefore, the indexing of the semantic elements works like the metamodel indexing, where UUIDs are used. The combination of the UUIDs and content hashes allows flexibility for editing the source model, while maintaining the connection to the notation model.

6.3.5. Custom UI extensions

This section demonstrates the creation of custom UI extensions by two different examples. GLSP provides a predefined interface for creating custom UI elements, that could be used in all platoform integrations. For that the abstract class `AbstractUIExtension` must be extended and added to the *henshin-glsp* application module. One simple example is the transformation rule name with its parameters that is displayed in the top left of the rule editor. The extension needs a defined id and a parent container id. With the `SetUIExtensionVisibilityAction`, the UI element can be made visible from external over the id. With the method `initializeContents(containerElement)`, the HTML elements can be created and added to the container. After the model initialization and over a public updated method, the class requests the rule name and its parameters over the `IActionDispatcher` and updates the UI. This update method can be called from any other class, when the `RuleNameUIExtension` is registered and injected over the dependency injection. One example is the explorer view, where the rule can be opened and therefore the rule name must be updated.

This custom explorer is a Theia exclusive extension, accessing the Theia internal APIs. It cannot be used for other GLSP platform integrations. To use a custom theia explorer was already discussed in section 5.1. To implement a custom explorer, the classes `FileNavigatorModel`, `FileNavigatorTree`, and `FileNavigatorWidget` are extended and registered in the theia specific *rules-theia* module via dependency injection. To add additional virtual elements in the explorer tree, the two new tree nodes `HenshinRootNode`, that contains a list of children, and `HenshinRuleNode`, that contains information like the rule name, are created. The method `resolveChildren(parent)` is overwritten in the `FileNavigatorTree`. Here, if it iterates over a *.hensin* file node, it requests the transformation rules from the server and creates the corresponding `HenshinRuleNode` for each rule. It creates also an additional node that works as a „add rule“ button. In the `HenshinNavigatorWidget`, the method `onSelectionChanged` event is subscribed. It checks if a virtual `HenshinRuleNode` was selected. If that is the case, it tries to find the GLSP rule wdiget and opens it. It also sends the selected rule name to the server, that is then selecting the rule in the `RuleGModelFactory`, where the graphical model is created. To provide a fitting look to the new tree nodes, the `HenshinNavigatorWidget` implements the methods `toNodeName(node)` and `toNodeIcon(node)`. Here, fitting icons are selected and the displayed names are configured.

7. Testing and Evaluation

This chapter discusses the testing strategy of the Henshin Web application. First the general strategy is described. Then the structure and results of the unit tests and end-to-end tests are seperatly presented. Finally, the limitations of the testing are discussed.

7.1. Testing Strategy

For the Java backend, unit tests wer implemented using JUnit. For every milestone of the development process, unit tests were added to ensure the added functionality works as expected. The tests cover the core functionality of the backend. Mocking was used to simulate the behaviour of some components, like the `ModelState`.

To test the UI, automated UI tests were created using Playwright.

7.2. Unit Tests

7.3. E2E Tests

For end-to-end testing, different frameworks were considered to use, including Cypress, Playwrite, Selenium and some more.

7.4. Limitations

8. Deployment

This chapter shows different deployment and usage options for the Henshin Web editor. The options are evaluated and the best one is selected. Finally, the implementation of the deployment is described.

8.1. GLSP Integration Options

A GLSP editor can be deployed and used in production in various ways. GLSP provides platform integrations for the Eclipse Desktop IDE, Eclipse Theia, VS Code, and as a standalone web application. Each integration brings different integration possibilities, deployment, and usage options for the editor. [6] The main considerations for the deployment and usage are:

- The user should need as few dependencies as possible. Dependencies are a browser runtime, an IDE to install, or an extension to install.
- The app should be easy to access. Possible barriers are the creation of an account or the installation of dependencies.
- Using a self-hosted server or a cloud service. With a self-hosted server, the user has full access of local files to open and edit. With a cloud service, the user has to upload and download files to the server.

To use GLSP as a standalone web application, a dependency injection container with the custom GLSP client is added to a TypeScript browser application. Like that the editor of a certain file as a data source can be displayed. When the app is hosted, no other dependency than a browser runtime is needed to use the standalone diagram editor. [31] This option provides the most flexibility, as it can be used in any web application, but also requires the most effort to implement, when developing a complete editor. All features, like file management, window management, or other features a IDE brings, need to be implemented by the developer. [31] For our use case, the standalone web application is not an option, as these additional features are needed.

The other GLSP integrations are IDE integrations and therefore provide many features out of the box. For the Eclipse IDE integration, Eclipse has to be installed, and the GLSP plugin has to be added to the Eclipse installation. The plugin can be installed from the Eclipse Marketplace or manually by downloading the plugin jar file. [32] The VS Code integration also provides this option. The IDE can be installed and the GLSP editor can be added as an extension. The extension can be installed from the Marketplace or manually using a *.vsix* file. [33] The GLSP VS Code integration can provide a *.vsix* file.

[5] VS Code is the most used IDE. 73.6% of developers use VS Code due to the survey of Stack Overflow In 2024 [34]. An advantage to Eclipse is that VS Code provides a browser version, which brings the same capabilites as the desktop IDE. [33] So this integration provides the advantage that no IDE has to be installed to be able to use Henshin Web. The user can open VS Code, add the extension, and directly open a metamodel, rule, or instance model file and start editing.

The Eclipse Theia IDE is not as widely popular as VS Code [34], but its focus is not to provide a ready IDE but to provide tools to create custom IDEs. The Eclipse Theia project is part of the Eclipse Foundation and is used as a basis to create your own IDEs based on web technologies. [35] They provide the Theia IDE that acts as a template editor and can be downloaded and used on all common operating systems or used in as a web editor in the browser. Due to the focus on providing a framework to build custom IDEs, Theia provides more options to use extensions and plugins to extend the functionality. You can see the options and their architectural integration into Theia in figure 8.1.

- **VS Code extensions** Theia provides the VS Code extension API, so that existing VS Code extensions can be used in Theia. They only interact with the API and therefore can be installed at runtime.
- **Theia plugins** are working like VS Code extensions. They interact with the Theia plugin API and can also access the VS Code extension API. They can access some Theia specific features, that VS Code extensions cannot access, like directly contributing to the frontend. They can also be installed at runtime, or be pre-installed at compile time.
- **Headless plugins** are also working like VS Code extensions. They can also be installed at runtime and can access custom extended Theia backend services.
- **Theia extensions** are the core architecture parts of Theia. Theia is fully built using Theia extensions in a modular way. The template Theia IDE contains Theia extensions, including the core. Custom Theia extensions can be developed and added to Theia with full access to all Theia functionality via dependency injection. They need to be installed at compile time. [35]

The GLSP Theia integration is creating a Theia extension, that is packed into a custom Theia IDE. It is also possible to use the GLSP VS Code integration that provides a VS Code extension, that can also be added to a Theia IDE at runtime. [5] The option to use the diagram editor in the browser makes the GLSP Eclipse integration not interesting for Hensin Web. VS Code has the advantage of popularity and simplicity to use the editor without any registration or installation. Eclipse Theia has the advantage of modularity and further extensibility. Further features can be added in the future to provide a web-based environment for MDE. Theia also provides different ways to deploy a Theia IDE. These considerations show that the Theia integration is the best option for deploying the Henshin Web editor. Theia combines the advantages of browser-based access, modularity, and extensibility.

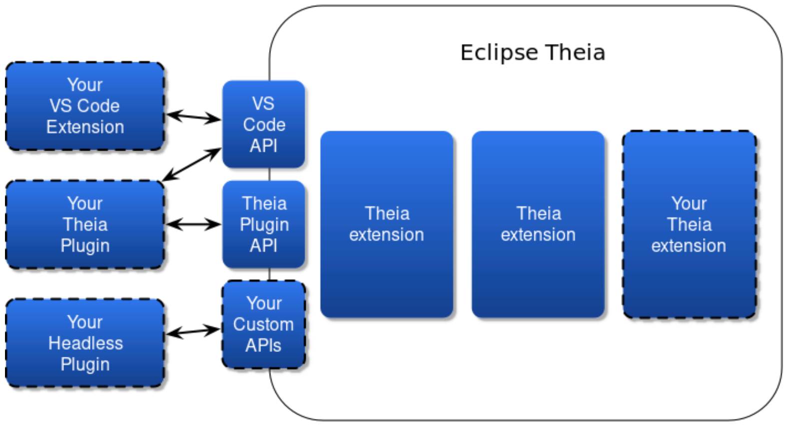

Figure 8.1.: Theia high level extensions and plugins architecture. Image obtained from [35]

8.2. Deployment Options and Evaluation

There are different options to provide a GLSP Theia application. The Theia editor, consisting of the TypeScript client and the Java server, can be hosted in the cloud and accessed via a web browser. The Eclipse Foundation provides the Theia Cloud project [36] to deploy Theia based products on Kubernetes clusters [37]. Theia Cloud introduces three custom Kubernetes resource types. *App Definitions* contain all necessary information about the Theia based product. *Workspaces* define persistent storage solutions, where metamodel, rule, or instance model files can be stored for each user. *Sessions* are acting as a runtime representaions. Theia Cloud includes components like a landing page, authentication, authorization, a cloud monitor, and a cloud operator, that deploys sessions and manages workspaces. You can see the different components and their interactions in figure 8.2. The service provides two preconfigured configurations for quickly trying out Theia Cloud on a cluster. [36]

Because of the limited file access of the browser, the user has to upload and download all files to the server to use them. To be able to access the local file system of the user directly, the server needs to be hosted locally. For that, GLSP Theia application can be hosted in a Docker container. [38] The Docker container can contain the Java server and the TypeScript client, that are started together. The user can then access the editor via a web browser. On a machine with a Docker environment, this solution can be started locally in an easy way and has the access to the file system. The Docker container can also be used to deploy the application on a server so that it can be accessed by multiple users. The single Docker container solution doesn't provide as much scalability as using a cluster with Theia Cloud.

The GLSP Theia application can also be used as a desktop application. Theia uses Electron [39] to bundle the application into a desktop application, that can be installed via an installer. This approach also provides access to the local file system, since the electron application works like a self-hosted web application, and therefore the GLSP

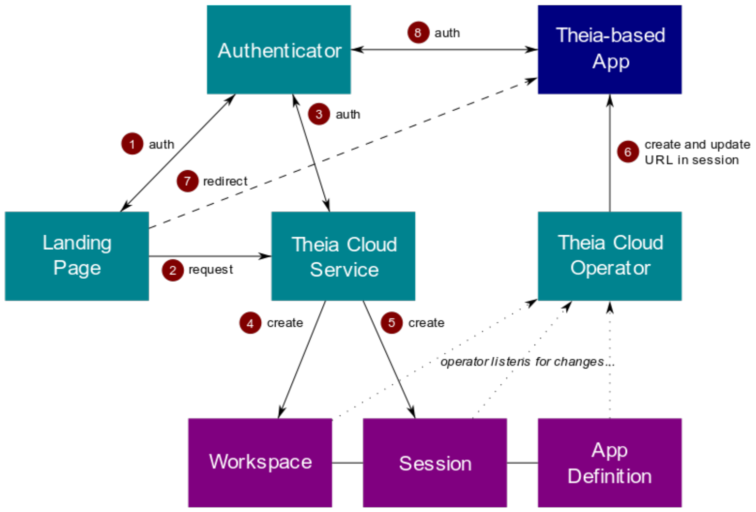

Figure 8.2.: Interaction between Theia Cloud components. Image obtained from [36]

Java server is started locally. All in all, the GLSP Theia integration provides all different options to use the Henshin Web editor. Further clients can always be added later if needed.

Table 8.1.: Comparison of GLSP Theia deployment options

Option	**Self-hosted Container**	**Cloud Hosted Container**	**Theia Cloud**	**Desktop (Electron)**
Installation Effort	Local Docker Setup required	None (access via browser)	None (creating an account)	Installing over a standard installer
Dependencies	Docker runtime, web browser	Web browser only	Web browser only	Application installer
Multi-user Support	Single user	Multi-user possible but no shared editor	Built-in shared workspaces, but no shared editor	Single user per installation
Hosting Requirements	Local	Cloud service (Container hosting)	Kubernetes cluster	Local machine
Cross-platform	Yes (via browser)	Yes (via browser)	Yes (via browser)	Platform-specific builds
Offline Usage	Yes	No	No	Yes
File System Access	Full local access	Upload/download required	Upload/download required	Full local access
Costs	No Costs	Cloud server costs	Cost of Google Cloud Kubernetes cluster	No Costs (maybe provisioning of installer)

In table 8.1 the different deployment options are listed. Each pair of options is similar.The self-hosted Docker and Desktop Electron bring no costs to provide the application, can be used offline and provide full local file access. Here the Desktop option is easier to install and use, as no external dependencies need to be installed before. And even thought it doesn't use the browser, it is based on web technologies. The other two similar options are host a container in the cloud and using Theia Cloud. A cloud hosted container would be a self implemented and configured solution. That would bring more flexibility, but also more effort to implement and maintain. Especially when the user management and workspace management should be implemented. Theia Cloud on the other hand provides these features out of the box. Both options bring costs for hosting the server and need an internet connection to be used. They also don't provide access to the local file system, as they are not self-hosted. Since Theia Cloud also leaves configuration options open and it is made for hosting Theia based products, it is the better option to host Henshin Web in the cloud. Now the decision is between the Electron desktop application and Theia Cloud. The desktop application has the advantages, that it can be used offline and provides full access to the local file system. This would be no bid difference to use Henshin in the Eclipse IDE. Theia Cloud on the other hand provides the advantage of easy access via a web browser without any installation. It also provides the possibility to use Henshin Web on different devices, like tablets or smartphones. Therefore Theia Cloud is the best fitting option to deploy Henshin Web.

8.3. Deployment Implementation

This chapter describes the implementation of the deployment of Henshin Web using Theia Cloud. Theia Cloud provides two different deployment options to host the application on. It is possible to host the application on a Kubernetes cluster in the Google Cloud using Google Kubernetes Engine (GKE) or on a local Kubernetes cluster using minicube. The following implementation shows the deployment using GKE. The implementation using minicube is similar, only the terraform code and the prerequisites of the local machine are different.

8.3.1. Docker Container Architecture

The first thing that needs to be done is to create a docker image of the application. The docker image contains the Java server and the TypeScript client. The resulting docker file is shown in listing A.1. The Dockerfile implements a multi-stage build approach. This approach separates the build environment from the runtime environment, resulting in smaller and more secure production containers. The build process consists of three distinct stages, each serving a specific purpose in the containerization pipeline. This separation allows for parallel execution of backend and frontend builds, improving overall build performance.

The first stage, labeled as *build*, establishes the development environment using a Debian Bullseye image with Node.js preinstalled. This stage installs all necessary dependencies for building both the Java backend and TypeScript frontend components. The environment includes Maven for Java compilation, OpenJDK 17 for runtime compatibility, and various system libraries required by Theia. The second stage, *backend*, focuses on compiling the GLSP Java server. It copies the server source code and uses Maven to clean, compile, and verify the Java components. The Maven settings file is copied to handle GitLab authentication for private repositories, ensuring that all dependencies can be resolved during the build process. The third stage, *frontend*, handles the compilation of all Theia components and plugins. The Yarn autoclean feature is configured to remove unnecessary files like TypeScript sources and test files from the final image, reducing the container size. The final production stage uses a slim Debian image to minimize the attack surface and container size. It installs only the runtime dependencies necessary for Theia operation, including Java runtime, system libraries, and development tools. A non-root user with ID 200 is created, following Theia Cloud standards for user management. The home directory is properly configured with appropriate permissions for file operations. The container exposes port 3000 for web access and configures the entry point to start the Theia backend server with specific parameters for workspace management and plugin loading.

8.3.2. Terraform Configuration

When the docker image is built, it can be used to deploy the application on GKE to make it accessible via the internet for everyone. Theia Cloud provides a modular terraform [40] configuration to deploy the Kubernetes cluster to the Google Cloud. This Infrastructure as Code (IaC) approach provides the advantage that the infrastructure can be defined as code and therefore can be easily adapted, versioned, and reused.

The Terraform configuration follows a modular architecture with three main components: cluster creation, Helm chart deployment, and Keycloak [**keycloak-repo**] authentication setup. The main configuration file orchestrates the deployment by calling specialized modules that handle specific infrastructure concerns.

The cluster creation module provisions a production-ready GKE cluster with auto-scaling node pools. The configuration removes the default node pool to enable custom machine types (`e2-standard-2`) and scaling parameters (1-2 nodes), providing cost optimization while ensuring adequate resources:

```
resource "google_container_cluster" "primary" {
  name                     = var.cluster_name
  location                 = var.location
  remove_default_node_pool = true
  initial_node_count       = 1
  deletion_protection      = false
}
```

Listing 8.1: GKE Cluster Configuration

Network connectivity is established through a reserved static IP address that supports both custom domains and automatic DNS resolution via sslip.io. The configuration integrates multiple providers (Google Cloud, Helm, Kubectl, Keycloak [**keycloak-repo**]) with dynamic authentication using cluster credentials obtained from the GKE module.

The Helm module manages the deployment of essential Kubernetes applications including NGINX Ingress Controller, Cert-Manager for SSL certificates, Keycloak [**keycloak-repo**] for authentication, PostgreSQL database, and the Theia Cloud application components. Variable management separates configuration from sensitive data, with security-sensitive variables marked appropriately:

```
variable "keycloak_admin_password" {
  description = "Keycloak Admin Password"
  type        = string
  sensitive   = true
}

variable "theia_docker_image" {
  description = "Docker image for your Theia application"
  type        = string
  default     =
    "gcr.io/henshin-web/henshin-web-model-transformation:latest"
}
```

Listing 8.2: Variable Configuration

8.3.3. Deployment Execution

The deployment of the Henshin Web application follows a systematic process that combines containerization, cloud infrastructure provisioning, and configuration management. The deployment workflow is automated through PowerShell scripts that orchestrate the entire process from local development to production deployment on Google Cloud Platform.

Before executing the deployment, several prerequisites must be satisfied:

- Google Cloud Platform account with billing enabled
- Google Cloud SDK (gcloud) installed and authenticated
- Docker Desktop or Docker Engine installed
- Terraform CLI installed (version >= 1.4.0)
- Access to the project's Google Container Registry

The first step involves building the Docker container and pushing it to Google Container Registry. This process is automated through the `build-and-push.ps1` script:

```
docker build -t "henshin-web-model-transformation:latest" .

$FullImageName =
  "gcr.io/henshin-web/henshin-web-model-transformation:latest"

docker tag "henshin-web-model-transformation:latest"
  $FullImageName
docker push $FullImageName
```

Listing 8.3: Container Build Process

Once the container image is available in the registry, the infrastructure deployment is executed through the `deploy.ps1` script. This script starts the creation of the resources in the Google Cloud using Terraform:

```
Set-Location
    "$PSScriptRoot\..\terraform\configurations\henshin-web-app"

terraform init

$env:GOOGLE_OAUTH_ACCESS_TOKEN = (gcloud auth print-access-token)
terraform apply
```

Listing 8.4: Infrastructure Deployment

The deployment script automatically handles authentication by obtaining fresh OAuth tokens from the gcloud CLI. This ensures that Terraform has the necessary permissions to create and manage Google Cloud resources without requiring manual token management.

The Terraform configuration executes the deployment in a specific sequence to ensure proper dependency resolution:

1. **GKE Cluster Creation**: Provisions the Kubernetes cluster with auto-scaling node pools
2. **Network Configuration**: Reserves static IP addresses and configures ingress rules
3. **Core Services Deployment**: Installs NGINX Ingress Controller and Cert-Manager
4. **Database Setup**: Deploys PostgreSQL for Keycloak authentication services
5. **Authentication Configuration**: Installs and configures Keycloak with realm settings
6. **Application Deployment**: Deploys the Henshin Web application and landing page
7. **SSL Certificate Provisioning**: Automatically obtains Let's Encrypt certificates

The deployment process uses environment-specific configuration through the `terraform.tfvars` file, which contains:

- Project identification and resource naming conventions
- Container image references with specific tags
- Authentication credentials for Keycloak and database services
- Regional deployment settings and cluster specifications

Because this includes sensitive configuration values, they are managed separately from the codebase to maintain security best practices while enabling automated deployment workflows.

Upon successful completion, the deployment script provides the application URL, typically in the format `https://<static-ip>.sslip.io`, where the application becomes immediately accessible. The automated SSL certificate provisioning through Let's Encrypt ensures secure HTTPS connectivity without manual certificate management.

The deployment can be reversed using the `destroy.ps1` script, which safely removes all provisioned resources while preserving any persistent data that needs to be retained for future deployments.

9. Usage

This chapter provides a user guide with the necessary informations to use Henshin Web. Furthermore, it describes how to set up a development environment for Henshin Web.

9.1. User Guide

This chapter provides a user guide how to use Henshin Web. After starting the application at the landing page, users have to create an account and login to access an instance of Henshin Web. After registration, the email address has to be verified in order to access Henshin Web. Each account has its own workspace that is persistent across sessions. The workspace is the space where all important files for Henshin are stored. Different models or projects can be structured in folders and subfolders. A Henshin project consists of at least a *.ecore* metamodel file, a *.henshin* transformation rules file, and optionally one or more *.xmi* model files. In an empty folder, a new Henshin project can be created by clicking on *New Project* in the explorer folder. It creates a starting point with the minimal content needed. Existing projects can also be imported by uploading the necessary files by right-clicking on a folder and selecting *Upload Files*.

All files are shown in the explorer of theia. *.henshin* Rule files can be extended to see all containing rules as child nodes in the explorer. Here also new rules can be added by clicking on the *+ Add* button in the explorer. When opening one of the three file types in the explorer, the graphical editor will be opened, allowing users to edit the file content dirctly on a graph. With *Open With* in the context menu, the files can also be opened with a text editor. The handling of the graphs of the different file types works very similar. The opened graph can be moved and zoomed with the mouse by dragging the background with the mouse and scrolling the mouse wheel or a touchpad. Zooming and reseting the viewport can be also accesed over the toolbar of Theia. Nodes and edges can be selected by clicking on them. The selection of elements is necessary to move or delete elements. Nodes and edges can be moved by dragging and dropping them to a new position. Edges follow the position of their connected nodes, but can be rerouted temporarily. The position of the nodes are persisted in a notation file, to keep the layout across sessions. Notation files are stored in the *.notation* folder in each project folder. They can be deleted to completely reset the layout. All editors support undo and redo operations, which can be accesed via the toolbar or the keyboard shortcuts *Ctrl + Z* and *Ctrl + Y*. Instance files provide the additional option to hide the root node. By toggling the *Hide Root Node* button on the bottom right, the root node gets nearly invisible to give a better overview of big instance models.

On the right side of the editor is the tool palette, which contains all available elements that can be added to the diagram. To create a node you can click on the element in the palette and then click in the diagram to place it. To create an edge, you can click on the edge element in the palette and then the on the source node. After that a drawn edge line will follow the mouse cursor until you click on the target node to finish the edge creation. If a specific edge was drawn on a wrong source or target node, it wont be created. In the Instance editor, also the transformation rules can be applied over the tool palette. When clicking on a rule in the palette, a dialog opens where the parameters of the rule can be specified. The tool palette also contains icons to perform common actions. The editing mode can be switched between *Selection Mode*, *Deletion Mode* and *Marque Mode*, where a square can be drawn to select multiple elements within. Next to that, a searchbar can be opened to filter and search the options of the tool palette.

Now elements of the graph can be created and deleted. To edit the properties of an element, different elements have different editor windows. They can be opened by double-clicking on an element or element part or by right clicking an element and selecteing the *Open Properties* option in the context menu. All editors work similar. They contain a specific form where the properties of the selected element can be modified. The changes can be persisted by clicking outside of the editor window. That also closes the editor window. Here is a list of all available editor windows:

- **Ecore editor**
  - **Attribute Editor**: In this window the attributes of nodes can be modified. It can be opened by double clicking on the attribute part of a class node. You can see the window in figure A.4. It allows to add, update and delete the name, type, multiplicity, and if the attribute is required.
  - **Operation Editor**: In this window the operations of nodes can be modified. It can be opened by double clicking on the operations part of a class node. You can see the window in figure A.5. It allows to add, update and delete operations. Thatincludes the name, return type, parameters, visibility, and if the operation is abstract.
  - **Reference Editor**: In this window the references of nodes can be modified. It can be opened by double clicking on an edge or its labels. You can see the window in figure A.6. It allows to edit the name and multiplicity of the reference and its opositie. You can also specify if the reference is a containment reference.
  - **Enum Editor**: In this window the enums of nodes can be modified. It can be opened by double clicking on bottom part of a enum node. You can see the window in figure A.7. It allows to add, update and delete enum literals with their name, value and literal.
  - **Datatype Editor**: In this window the datatypes of nodes can be modified. It can be opened by double clicking on a datatype node. It allows to specify the name, the instance class name of the datatype and if the data type is serializable.
- **Henshin rule editor**

- **Action Type Editor**: In this window the action types of nodes can be modified. It can be opened by double clicking on the action type part of a node or an edge. It allows to change the action type of the node between *Preserve*, *Create*, *Delete*, *Forbid*.
- **Rule Editor**: In this window the name and parameters of a rule can be modified. It can be opened by clicking on the edit icon of the rule name box on the top left of the editor. You can see the window in figure A.8. This window can also be moved by dragging it. With this window, the rule can also be deleted. Here the changes have to be confirmed with the check icon on the bottom right or discarded with the cross icon on the top right.
- **Attribute Parameter Mapping Editor**: In this window the attribute parameter mappings of nodes can be modified. It can be opened by double clicking on the attribute parameter mapping part of a node. You can see the window in figure **??**. It allows to map the parameters of the rule to an attribute of the selected node.

- **Instance editor**
  - **Attribute value editor**: In this window the attribute values of nodes can be modified It opens when double clicking on an attribute label of a node. It allows to change the value of the attribute according to its type.

9.2. User Administration Guide

The deployment of Henhin Web with Theia Cloud provides a user management. Here users can be created, deleted and managed. The administration is done over the Keycloak admin console, which is available at `http://<domain>/keycloak/` after the deployment of Henshin Web. Keycloak is an open-source identity and access management solution [**keycloak-repo**].

After logging in with the admin credentials, specified in the *terraform.tfvars* file, the admin console provides different management options. You can see the UI in figure A.9. In order to perform changes, the dropdown on the top right should be changed to *TheiaCloud* realm. After a completly new deployment of Henshin Web, following configurations should be done in the Realm Settings:

- Under General, change the *display name* and the *Hypertext Markup Language (HTML) display name* to Henshin Web.
- Under Login, enable user registration, forgot password, remember me and verify email features.
- Under Email, configure the SMTP settings to enable email functionalities like verification emails and password reset emails. New, unverified accounts cannot access Henshin Web.

- Under Localization, enable internationalization and select the desired supported locales.
- Under Sessions, adjust the session timeouts according to the desired security level.

Additionally, many other configurations can be done in the admin console, like adding identity providers for the login, configure session timeouts, roles, groups, or authentication flows. Keycloak provides a linked documentation that contains information about all possible settings.

9.3. Development Setup

This section describes how to set up a complete development environment for the Henshin Web project. The project is built using Eclipse GLSP with a Java server backend and a Theia-based client frontend.

Before setting up the development environment, ensure that the following software components are installed on your system:

- **Node.js** (version $\geq$ 18): Required for building and running the client application
- **Yarn** (version $\geq$ 1.7.0, $<$ 2.x.x): Package manager for JavaScript dependencies
- **Java** (version $\geq$ 17): Required for the GLSP server backend
- **Maven** (version $\geq$ 3.6.0): Build tool for the Java server component

First, get access to the gitlab repository, clone it and navigate to the project directory:

```
git clone
    https://gitlab.uni-marburg.de/weidnerf/henshin-web-model-transformation.git
cd henshin-web-model-transformation
```

Before you can build the GLSP backend project, you need to specify maven authentication settings, to be able to access the private maven repository with the Henshin artefacts. To do that, create a `settings.xml` file in the `.m2` folder in your home directory (e.g., `C:\Users\<username>\.m2\settings.xml` on Windows). The file should contain the following content with your gitlab username and a personal access token as password:

```
    <settings>
  <servers>
    <server>
      <id>gitlab-maven</id>
      <username>YOUR_USERNAME</username>
      <password>YOUR_PASSWORD</password>
      <configuration>
        <authenticationInfo>
          <userName>YOUR_USERNAME</userName>
          <password>YOUR_PASSWORD</password>
        </authenticationInfo>
      </configuration>
```

```
    </server>
  </servers>
</settings>
```

Listing 9.1: Maven Settings

Navigate to the source directory and build both client and server components:

```
cd src
yarn build
```

This command will build the GLSP server using Maven, install and build all client dependencies, and prepare the application for execution. Once the build is complete, start the application:

```
cd glsp-client
yarn start
```

The application will be available at `http://localhost:3000`.

If you want to debug the Java server, you have to start the client separately. So start the Java server in debug mode (e.g., with IntelliJ). You can then start the client with the following command in the `glsp-client` folder:

```
yarn start:external
```

You can additionally use `yarn watch` in the `glsp-client` folder to automatically rebuild the client on code changes. That makes the development much easier, because you do not have to rebuild the client manually after each change.

To have a better overview of the project structure, here is a brief description of the main components of the Henshin Web project:

- **Client Components**
  - `henshin-browser-app/`: Main browser application integrating Theia with the GLSP extensions
  - `henshin-glsp/`: Core GLSP client module containing code for custom UI elements and logic. The module is used in all three following diagram editor modules.
  - `ecore-theia/`: Client module for the Ecore diagram editor. Contains code for context menu and command contributions.
  - `xmi-theia/`: Client module for the XMI diagram editor. Contains code for context menu and command contributions.
  - `rules-theia/`: Client module for the Henshin rule editor. Contains code for context menu and command contributions. It also contains the code for the Theia explorer extensions.
- **Server Components**: The server project is structured in following parts for each diagram module and a base module:
  - `actions/`: Custom actions for diagram operations

  - `handler/`: Action handlers for diagram operations
  - `model/`: Defines source model storage, graphical model factory, and state management
  - `provider/`: Custom providers for the tool palette

- **Henshin SDK Packager**
  - Pre-packaged Henshin SDK JAR files and dependencies
  - Maven configuration for Henshin integration
  - Deployment scripts for package registry

10. Discussion

10.1. Interpretation of Results

This section evaluates the research findings and addresses the five research questions that guided the development of Henshin Web. The evaluation is based on the successful implementation of a fully functional web-based model transformation editor that demonstrates significant improvements in accessibility and usability.

RQ1: Web-based Adaptation of Henshin Capabilities - The implementation demonstrates that adapting Henshin for web environments is both technically feasible and practically achievable through the GLSP framework. The successful integration of the Henshin Java SDK into the GLSP server architecture preserves complete transformation functionality while enabling web delivery. Converting Eclipse plugins into Maven artifacts enabled deployment in standard web application architecture. The three-editor architecture (`XMIDiagramModule`, `RuleDiagramModule`, `EcoreDiagramModule`) successfully handles workflow complexity while maintaining modular design and seamless integration between metamodels, transformation rules, and instance models.

RQ2: Essential Functional Requirements - The implementation successfully addresses the core use case of applying transformation rules to EMF XMI instance files. The identified requirements prove both necessary and sufficient for meaningful model transformation work across stakeholder groups. The parameter specification system through custom UI extensions maintains transformation flexibility while simplifying application. Integration of rule selection into the Theia explorer enhances workflow efficiency, and undo/redo functionality supports iterative development. The architectural foundation demonstrates scalability for planned enhancements including transformation units and analysis capabilities.

RQ3: Accessibility and User Experience Improvements - The web-based approach demonstrates significant accessibility improvements. Eliminating Eclipse installation requirements enables browser-based access without complex environment setup, particularly benefiting students and researchers. The graphical interface based on GLSP and Theia principles provides more intuitive experiences than traditional tree-based editors. Cross-platform accessibility and consistent visual representation reduce learning curves. Direct integration of transformation application into the instance editor improves workflow efficiency compared to Eclipse wizard-based approaches.

RQ4: Deployment Strategy Impact - The evaluation of multiple deployment options reveals that Theia Cloud proves most effective for maximizing accessibility. The cloud-based approach eliminates installation requirements while providing consistent performance and enterprise-ready capabilities. Trade-offs between Docker containers (local file access vs. installation complexity) and Electron applications (no browser dependency vs. installation requirements) highlight the advantages of the cloud approach. The Kubernetes infrastructure with automatic scaling addresses performance concerns while infrastructure-as-code ensures reproducible deployments.

RQ5: EMF and Henshin Ecosystem Integration - The implementation demonstrates comprehensive compatibility with existing ecosystems. Direct use of EMF data structures ensures complete compatibility with existing files, while `EMFSourceModelStorage` integration preserves metadata and relationships for seamless round-trip editing. Although notation files use different formats, semantic content remains fully compatible. The Henshin SDK integration preserves complete transformation semantics, ensuring identical results between web and Eclipse environments. The modular architecture supports future integration while providing value as an independent tool.

10.2. Challenges and Limitations

While Henshin Web demonstrates significant progress toward accessible model transformation tools, several challenges emerged during development and certain limitations constrain the system's capabilities compared to the mature Eclipse Henshin plugin.

Technical Integration Challenges - The integration of Henshin into GLSP presented substantial challenges. Bridging the architectural gap between Eclipse plugin-based and Maven-based development required creating a custom packaging system to convert 45 Eclipse plugins into Maven artifacts. This introduces maintenance overhead for future Henshin versions. The indexing of EMF models proved complex, requiring different strategies for each model type: Henshin rules (existing identifiers), Ecore metamodels (UUIDs and content hashes), and XMI instances (adapters and content hashes). Notation model synchronization presents ongoing challenges, particularly the content-hash approach for XMI instances requiring frequent updates.

Functional Limitations - The current implementation focuses on core functionality, leaving advanced features unavailable compared to the full Eclipse plugin. Missing capabilities include transformation units for complex scenarios, state space analysis, conflict detection, and comprehensive debugging tools. The web environment constraints prevent integration of step-through debugging capabilities that Eclipse provides. Collaborative editing features, natural for web environments, are not yet implemented, limiting team-based development scenarios.

Performance and Scalability Constraints - The web architecture introduces performance considerations absent in desktop applications. Client-server communication overhead becomes noticeable with large models, and JSON-RPC protocol serialization creates additional overhead. Current indexing strategies may not scale to very large models, particularly the content-hash approach requiring regeneration for attribute changes. Cloud deployment introduces network dependency, preventing offline work and potentially impacting responsiveness for users with limited connectivity.

Ecosystem and User Experience Limitations - While maintaining file format compatibility, the system lacks seamless integration with the broader Eclipse modeling ecosystem. Tools depending on Eclipse platform services cannot be directly integrated.

Notation model format differences require conversion when moving between environments. The web interface, though more intuitive, cannot provide the full range of keyboard shortcuts and interaction patterns experienced Eclipse users expect. Cloud deployment requires file upload/download, creating workflow friction compared to direct file system access. Current English-only interface limits international adoption.

10.3. Summary of Contributions

This thesis presents several significant contributions to model-driven engineering and web-based modeling tools, successfully demonstrating that sophisticated model transformation capabilities can be made accessible through modern web technologies while maintaining compatibility with established modeling ecosystems.

Web-based Model Transformation Architecture - The primary contribution is a comprehensive web-based architecture for model transformation editing that bridges traditional desktop-based modeling tools and modern web applications. The three-module architecture (`XMIDiagramModule`, `RuleDiagramModule`, `EcoreDiagramModule`) demonstrates how complex modeling workflows can be decomposed into manageable, specialized components while maintaining integration. The successful integration of the Henshin Java SDK into web application architecture shows how existing modeling frameworks can be preserved in new deployment contexts without sacrificing functionality.

Novel GLSP Integration - The thesis contributes a novel application of the GLSP framework to model transformation scenarios. While GLSP has been used for various diagrammatic editing applications, this work demonstrates its applicability to the complex domain of model transformation with multiple interrelated model types. The custom indexing strategies for different EMF model types represent a significant technical contribution, using hybrid approaches with UUIDs for session-based editing and content hashes for persistent notation. The integration of transformation rule application directly into the graphical interface provides a more intuitive workflow compared to traditional wizard-based approaches.

Accessibility and Barrier Reduction - A major contribution is the demonstrated reduction in barriers to entry for model transformation technology. By eliminating Eclipse installation and configuration requirements, the web-based approach makes model transformation accessible to broader audiences, particularly students and researchers exploring concepts without extensive tool setup. The user interface design provides a more approachable alternative to traditional tree-based editors, and flexible deployment options demonstrate how model transformation tools can be delivered as accessible services.

Migration Methodology and Open Source Contribution - The thesis contributes a practical methodology for migrating Eclipse plugin-based tools to web-based architectures. The process of converting Eclipse plugins to Maven artifacts provides a template for similar migration projects. The complete implementation serves as an open-source contribution providing a working example of web-based model transformation editing. The documentation of deployment strategies, including Kubernetes-based infrastructure and Docker containerization, contributes practical knowledge for hosting and scaling web-based modeling tools.

Validation and Foundation for Future Development - The successful implementation validates the feasibility of web-based approaches for complex modeling tasks traditionally requiring desktop applications. The performance characteristics and scalability analysis contribute empirical data about trade-offs in web-based modeling tool development. The modular architecture and compatibility with existing EMF and Henshin ecosystems provide a solid foundation for future enhancements while ensuring migration paths for existing users.

10.4. Suggestions for Future Development

The foundation established by Henshin Web provides numerous opportunities for enhancement and extension. Future development efforts should focus on expanding transformation capabilities, improving collaborative features, and integrating additional modeling tools to establish Henshin Web as a comprehensive platform for model-driven engineering.

Edapt Integration for Model Evolution - The integration of Edapt (EMF Adaptation) represents a natural and valuable extension to Henshin Web's capabilities. Edapt provides systematic support for model evolution and migration, addressing the common challenge of maintaining instance models when metamodels change. This integration would involve extending the `EcoreDiagramModule` to include Edapt's difference detection algorithms and migration rule generation. The web-based environment would provide unique advantages, allowing users to visualize the impact of metamodel changes across multiple instance files and show proposed migration paths graphically. Users could review and customize migration rules before application, positioning Henshin Web as a complete solution for model evolution scenarios.

Collaborative Modeling Capabilities - The web-based architecture provides an ideal foundation for implementing collaborative modeling features impossible in traditional desktop applications. Real-time collaborative editing would transform Henshin Web into a platform for team-based model transformation development. Implementation should include operational transformation algorithms for concurrent edits, user presence indicators, visual highlighting of elements being modified by others, and commenting systems for asynchronous collaboration. Version control integration could provide Git-like branching and merging specifically designed for model artifacts, enabling parallel development with systematic integration capabilities. The cloud deployment model facilitates collaboration across organizations and time zones through workspace sharing mechanisms.

Complete Henshin Functionality Implementation - The current implementation should be systematically extended with the complete Henshin feature set. Transformation units would support sequential, conditional, and iterative rule applications through flowchart-like graphical interfaces for designing complex transformation workflows. State space analysis functionality would enable interactive visualization of transformation paths and resulting models. Conflict and dependency analysis would highlight conflicting rules and visualize dependency relationships. Advanced debugging capabilities should include step-through execution, inspection of intermediate states, and detailed logging with debugging views showing current transformation state, matched elements, and parameter bindings.

Integration with Model-Driven Engineering Ecosystem - Future development should integrate Henshin Web with the broader MDE ecosystem. Code generation capabilities would extend utility beyond transformation to complete development workflows, allowing users to define generation templates and apply them to transformed models. Integration with model validation frameworks would enable constraint checking, completeness analysis, and semantic validation directly within the web interface. Model repository integration would enable sharing and reuse of metamodels, transformation rules, and instances across projects with version control and dependency management for model artifacts.

Performance and Domain-Specific Extensions - Performance improvements should address current limitations through optimized indexing with sophisticated caching and incremental updates, lazy loading and virtualization for large models, more efficient serialization protocols, and database integration for scalable storage. Domain-specific extensions could provide specialized modeling notations: business process modeling with BPMN-style notations for transformation workflows, software architecture modeling with UML-style notations and architecture-specific patterns, and educational extensions with guided tutorials, interactive examples, and assessment capabilities including gamification elements and progressive complexity levels for learning support.

Bibliography

[1] S. Sendall and W. Kozaczynski. "Model transformation: the heart and soul of model-driven software development". In: *IEEE Software* 20.5 (2003), pp. 42–45. DOI: 10.1109/MS.2003.1231150.

[2] David Steinberg et al. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009. ISBN: 0321331885.

[3] Daniel Strueber. *Henshin*. https://github.com/eclipse-henshin/henshin/wiki. 2025.

[4] Daniel Strüber et al. "Henshin: A Usability-Focused Framework for EMF Model Transformation Development". In: *Graph Transformation*. Ed. by Juan de Lara and Detlef Plump. Cham: Springer International Publishing, 2017, pp. 196–208. ISBN: 978-3-319-61470-0.

[5] Philip Langer and Tobias Ortmayr. *Eclipse GLSP*. https://github.com/eclipse-glsp/glsp. 2025.

[6] Eclipse Foundation. *Eclipse GLSP™ – Documentation*. Accessed: 2025-05-20. 2025. URL: https://eclipse.dev/glsp/documentation.

[7] Michael Jastram. *Die Eclipse Foundation wird 20 Jahre alt: Die interessantesten SE-Projekte*. de. https://www.se-trends.de/eclipse-foundation-wird-20-jahre-alt/. Accessed: 2025-6-2. Dec. 2024.

[8] Alexandra Kleijn. *Eclipse und die Eclipse Foundation*. Accessed: 2025-6-2. 2006. URL: https://www.heise.de/-221997.

[9] Eclipse Foundation. *2024 Annual Community Report*. en. https://www.eclipse.org/org/foundation/reports/annual_report.php. Accessed: 2025-6-2. 2024.

[10] Eclipse Foundation. *Eclipse Modeling Project*. Accessed: 2025-06-02. 2025. URL: https://eclipse.dev/modeling/.

[11] Eclipse Foundation. *Eclipse Modeling Framework*. https://github.com/eclipse-emf/org.eclipse.emf. 2025.

[12] Eclipse Foundation. *Eclipse Modeling Framework*. Accessed: 2025-06-02. 2025. URL: https://eclipse.dev/emf/.

[13] Thorsten Arendt et al. "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations". In: *Model Driven Engineering Languages and Systems*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 121–135. ISBN: 978-3-642-16145-2.

[14] Microsoft. *Language Server Protocol*. https://github.com/microsoft/language-server-protocol. 2025.

[15] Google. *Google Guice*. https://github.com/google/guice. 2025.

[16] Eclipse Foundation. *Sprotty*. https://github.com/eclipse-sprotty/sprotty. 2025.

[17] Google. *Gson*. https://github.com/google/gson. 2025.

[18] Nafiseh Kahani et al. "Survey and classification of model transformation tools". In: *Software & Systems Modeling* 18 (2019), pp. 2361–2397.

[19] Eugene Syriani et al. "AToMPM: A web-based modeling environment". In: *Joint proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013): September 29-October 4, 2013, Miami, USA*. 2013, pp. 21–25.

[20] Eugene Syriani, Hans Vangheluwe, and Brian Lashomb. "T-Core: a framework for custom-built model transformation engines". In: *Softw. Syst. Model.* 14.3 (July 2015), pp. 1215–1243. ISSN: 1619-1366. DOI: 10.1007/s10270-013-0370-4. URL: https://doi.org/10.1007/s10270-013-0370-4.

[21] Miklós Maróti et al. *Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure*. White paper / Technical report. White paper available at https://webgme.org/WebGMEWhitePaper.pdf. 2014.

[22] *WebGME: Web-based Generic Modeling Environment*. Accessed: 2025-06-18. 2025. URL: https://webgme.org/.

[23] Fazle Rabbi et al. "WebDPF: A web-based metamodelling and model transformation environment". In: *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2016, pp. 87–98.

[24] Eclipse Foundation. *ecore-glsp*. https://github.com/eclipsesource/ecore-glsp. 2025.

[25] Cheng-Kok Koh and Patrick H. Madden. "Manhattan or non-Manhattan? a study of alternative VLSI routing architectures". In: *Proceedings of the 10th Great Lakes Symposium on VLSI*. GLSVLSI '00. Chicago, Illinois, USA: Association for Computing Machinery, 2000, pp. 47–52. ISBN: 1581132514. DOI: 10.1145/330855.330961. URL: https://doi.org/10.1145/330855.330961.

[26] Microsoft. *Visual Studio Code*. Accessed: 2025-07-23. 2024. URL: https://code.visualstudio.com/.

[27] JetBrains. *IntelliJ IDEA*. Accessed: 2025-07-23. 2024. URL: https://www.jetbrains.com/idea/.

[28] Anthropic. *Claude 3 Sonnet*. Accessed: 2025-07-23. 2024. URL: https://www.anthropic.com/claude/sonnet.

[29] OpenJDK. *Nashorn Engine*. https://github.com/openjdk/nashorn. 2025.

[30] Sören Domrös et al. *The Eclipse Layout Kernel*. 2023. arXiv: 2311.00533 [cs.DS]. URL: https://arxiv.org/abs/2311.00533.

[31] Philip Langer and Tobias Ortmayr. *Eclipse GLSP - Client*. https://github.com/eclipse-glsp/glsp-client. 2025.

[32] Eclipse Foundation. *Eclipse Documentation - Current Release*. Accessed: 2025-06-28. 2025. URL: https://help.eclipse.org/latest/index.jsp.

[33] Microsoft Corporation. *Visual Studio Code Documentation*. Accessed: 2025-06-28. 2025. URL: https://code.visualstudio.com/docs.

[34] Stack Overflow. *Stack Overflow Developer Survey 2024: Technology*. Accessed: 2025-06-28. May 2024. URL: %5Curl%7Bhttps://survey.stackoverflow.co/2024/technology#most-popular-technologies-new-collab-tools%7D.

[35] *Eclipse Theia Documentation*. Accessed: 2025-06-28. Eclipse Foundation, 2025. URL: %5Curl%7Bhttps://theia-ide.org/docs/%7D.

[36] *Theia Cloud*. Accessed: 2025-06-28. Eclipse Foundation, 2025. URL: %5Curl%7Bhttps://theia-cloud.io/documentation/%7D.

[37] *Kubernetes Documentation*. Accessed: 2025-06-28. Linux Foundation, 2025. URL: %5Curl%7Bhttps://kubernetes.io/docs/home/%7D.

[38] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014.239 (2014), p. 2.

[39] OpenJS Foundation and Electron contributors. *Electron*. https://github.com/electron/electron. 2025.

[40] HashiCorp. *Terraform*. https://github.com/hashicorp/terraform. 2025.

A. Appendix

A.0.1. Figures

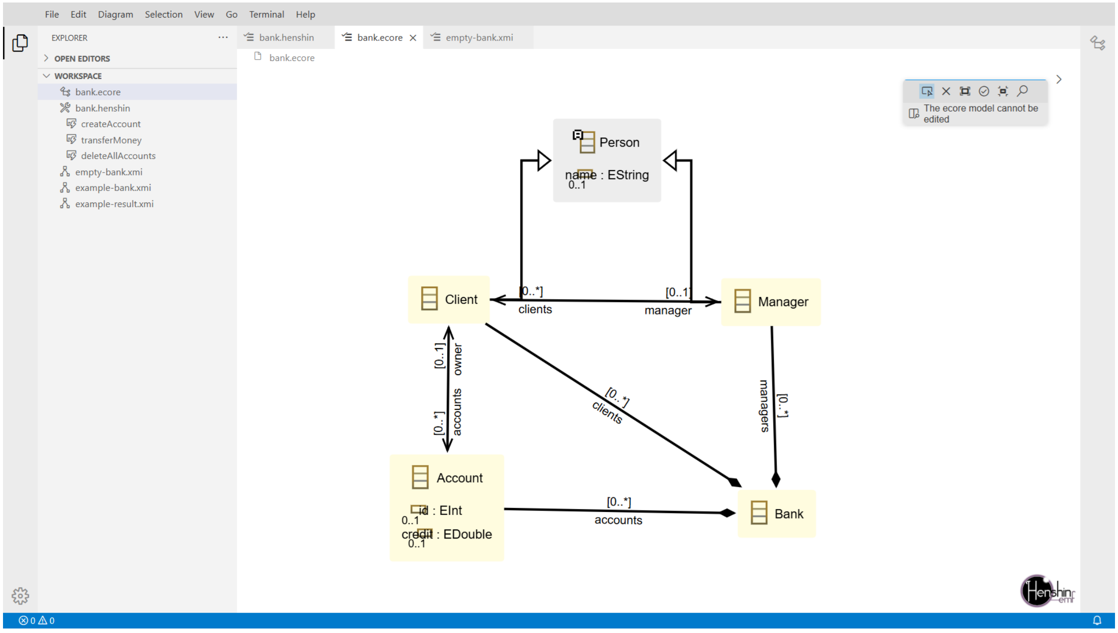

Figure A.1.: Henshin Web Ecore graph editor

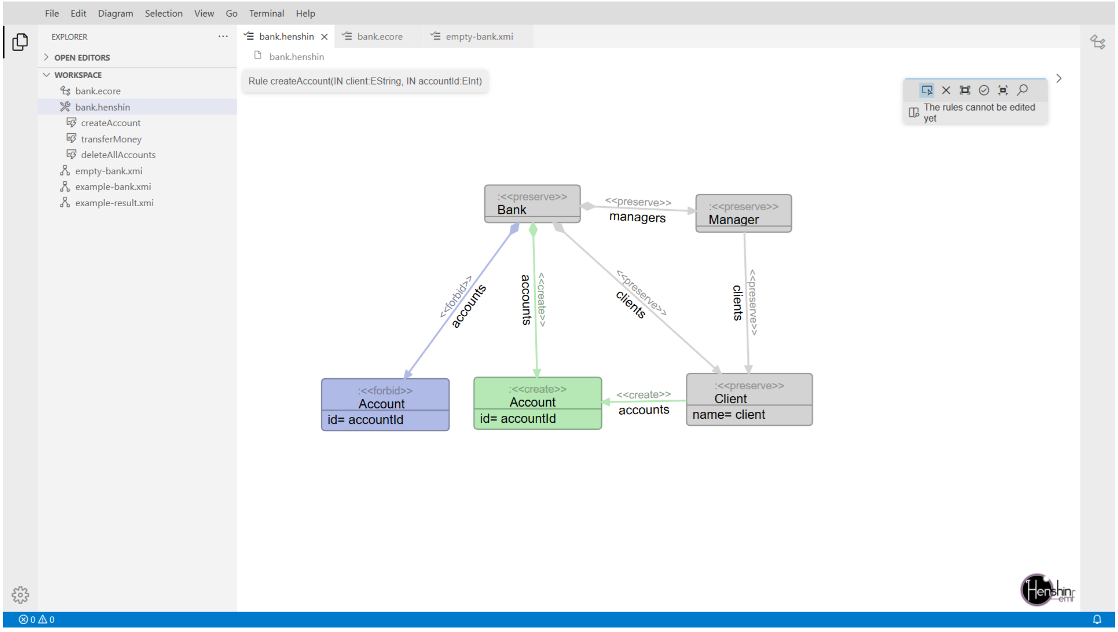

Figure A.2.: Henshin Web Rules graph editor

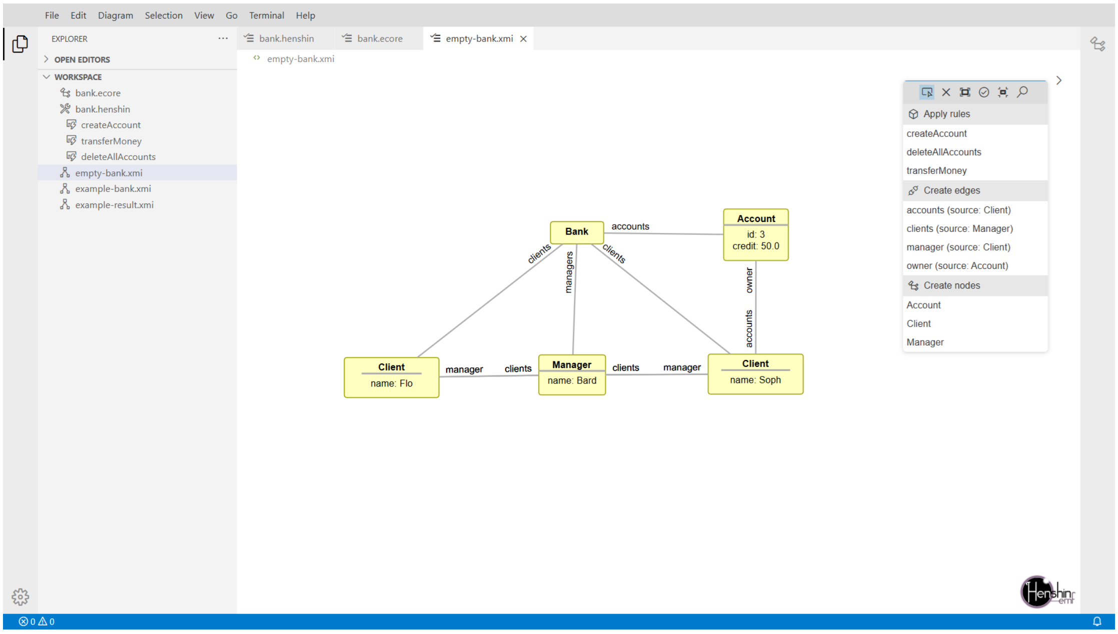

Figure A.3.: Henshin Web Instance graph editor

A.0.2. Code Listings

```
# Setup dev environment
```

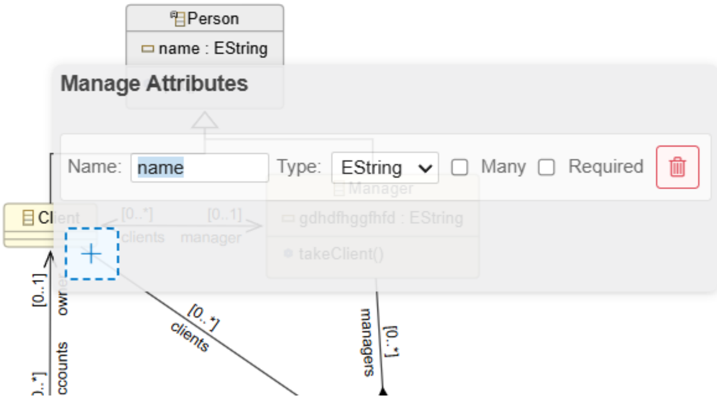

Figure A.4.: Attribute Editor Window

```
FROM node:18-bullseye AS build

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install -y --no-install-recommends \
    git \
    bash \
    maven \
    openjdk-17-jdk \
    software-properties-common \
    libxkbfile-dev \
    libsecret-1-dev \
    build-essential \
    libssl-dev \
    && rm -rf /var/lib/apt/lists/*


# Build the Java backend
FROM build AS backend

WORKDIR /henshin-editor

COPY ./src/glsp-server ./glsp-server

WORKDIR /henshin-editor/glsp-server

# Copy Maven settings for GitLab authentication
COPY ./src/glsp-server/.m2/settings.xml /root/.m2/settings.xml

# Verify settings file exists and run Maven with explicit settings
RUN mvn clean verify -s /root/.m2/settings.xml

# Build frontend
FROM build AS frontend

```

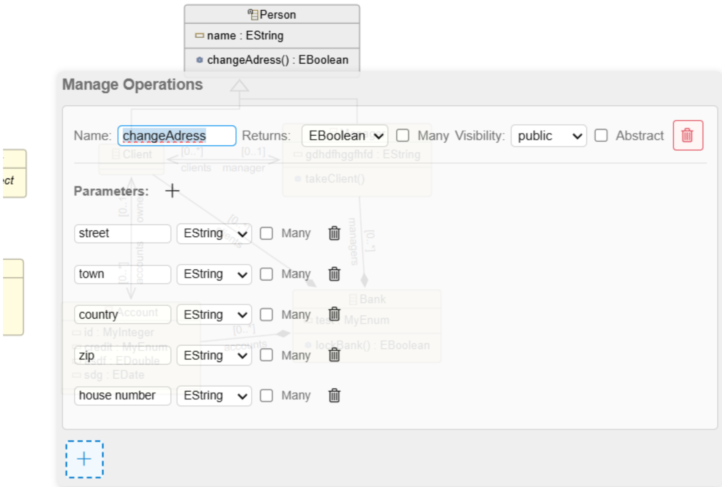

Figure A.5.: Operation Editor Window

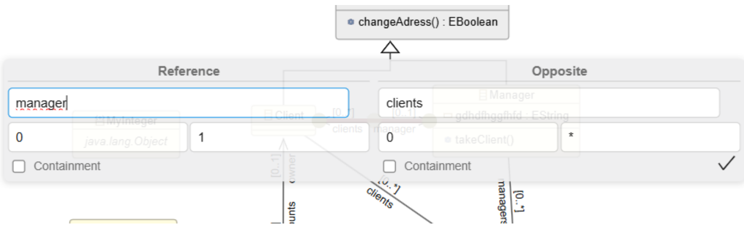

Figure A.6.: Reference Editor Window

```
WORKDIR /henshin-editor

COPY ./src/glsp-client ./glsp-client

WORKDIR /henshin-editor/glsp-client

RUN yarn install && \
    yarn build

WORKDIR /henshin-editor
COPY --from=backend /henshin-editor/glsp-server ./glsp-server
WORKDIR /henshin-editor/glsp-client

# Create plugins directory for Theia (even if empty)
RUN mkdir -p henshin-browser-app/plugins

RUN yarn autoclean --init && \
    echo *.ts >> .yarnclean && \
    echo *.ts.map >> .yarnclean && \
    echo *.spec.* >> .yarnclean && \
```

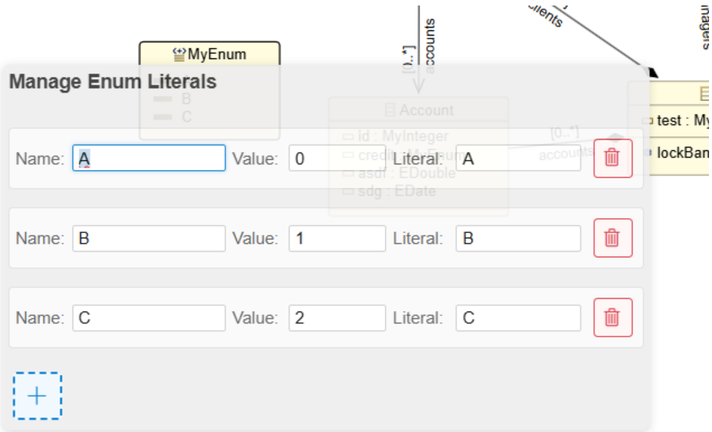

Figure A.7.: Enum Editor Window

```
    yarn autoclean --force && \
    yarn cache clean


# Build production image
FROM node:18-bullseye-slim AS production
ENV DEBIAN_FRONTEND=noninteractive

# Theia dependencies/Java
RUN apt-get update && apt-get install -y --no-install-recommends \
    software-properties-common \
    libxkbfile-dev \
    libsecret-1-dev \
    ca-certificates-java \
    openjdk-17-jdk \
    build-essential \
    libssl-dev \
    wget \
    gnupg \
    git \
    gdb \
    && rm -rf /var/lib/apt/lists/*

# C/C++ dependencies
RUN add-apt-repository 'deb http://apt.llvm.org/bullseye/
   llvm-toolchain-bullseye-14 main'
RUN wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key | apt-key add -
RUN apt-get update && \
    apt-get -y install clangd-14 cmake && \
    apt-get purge -y && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
RUN update-alternatives --install /usr/bin/clangd clangd
   /usr/bin/clangd-14 100
```

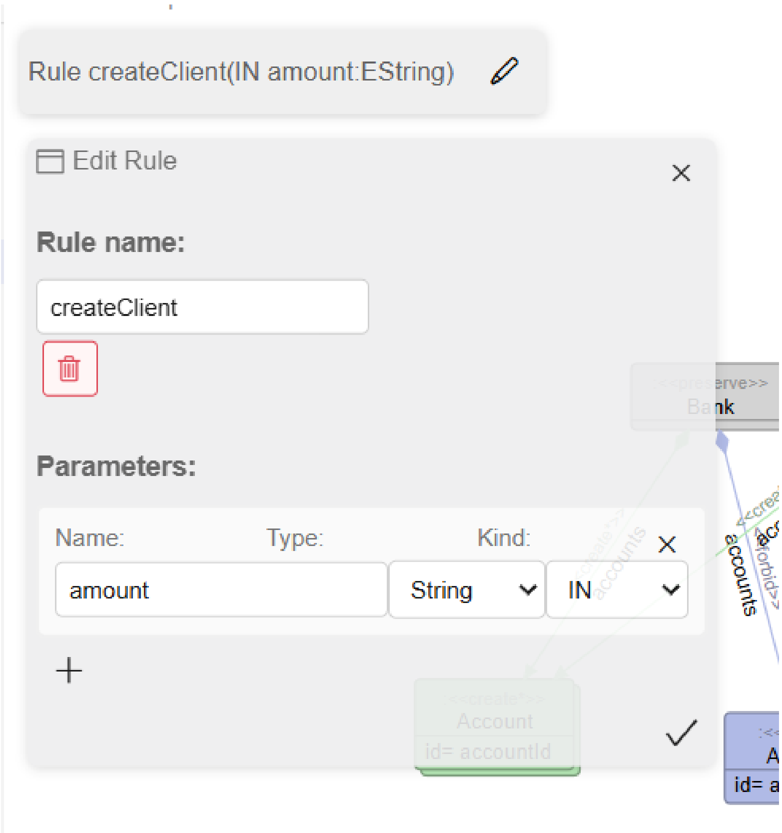

Figure A.8.: Rule Editor Window

```

# Make readable for root only
RUN chmod -R 750 /var/run/

RUN rm -f /root/.m2/settings.xml

# Create a non-root user with a fixed user id and setup the environment
    (following Theia Cloud standard)
RUN adduser --system --group --uid 200 theia && \
    chmod g+rw /home && \
    mkdir -p /home/theia && \
    chown -R theia:theia /home/theia
ENV HOME=/home/theia

# Copy frontend from build-stage
WORKDIR /home/theia
COPY --chown=theia:theia --from=frontend /henshin-editor/glsp-client
    ./glsp-client

# Copy model to production stage (for model comparison)
COPY --from=backend /henshin-editor/glsp-server/target/*.jar \
```

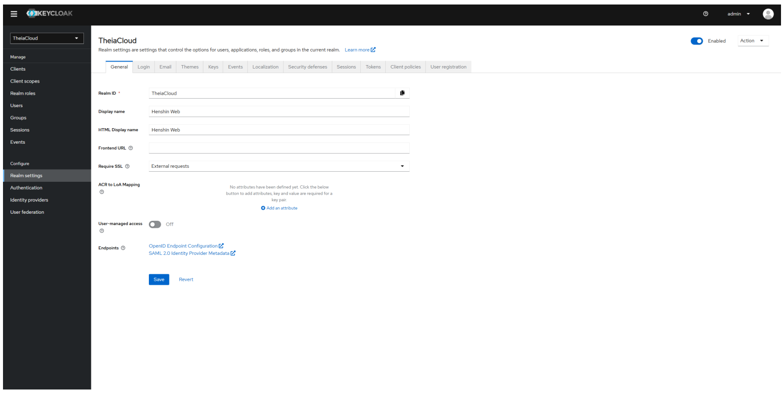

Figure A.9.: Keycloak Admin Console

```
    /home/theia/glsp-server/target/

# Copy favicon
RUN cp ./glsp-client/henshin-browser-app/favicon.ico
    ./glsp-client/henshin-browser-app/src-gen/frontend/
RUN sed -i 's/<\/head>/<link rel="icon" href="favicon.ico"
    \/><\/head>/g'
    glsp-client/henshin-browser-app/src-gen/frontend/index.html

# Create and setup the persisted project directory
RUN mkdir -p /home/project/persisted/workspace && \
    chown -R theia:theia /home/project && \
    chmod -R 755 /home/project

# Copy workspace content to the persisted location as well
COPY --chown=theia:theia ./src/glsp-client/workspace/
    /home/project/persisted/workspace

EXPOSE 3000
USER theia
WORKDIR /home/theia/glsp-client/henshin-browser-app/

ENTRYPOINT [ "node",
    "/home/theia/glsp-client/henshin-browser-app/src-gen/backend/main.js"
    ]
CMD [ "--root-dir=/home/project/persisted/workspace",
    "--hostname=0.0.0.0", "--port=3000",
    "--plugins=local-dir:/home/theia/glsp-client/henshin-browser-app/plugins",
    "--build-id=2025-08-11-v2" ]
```

Listing A.1: Dockerfile for Henshin Web Model Transformation Application

List of Figures

List of Tables