Accepted for publication in Software and Systems Modeling (SoSyM)
https://www.springer.com/journal/10270

A Reference Architecture for the Development of GLSP-based Web Modeling Tools

Haydar Metin[1,2] and Dominik Bork[2*]

[1*]EclipseSource, Schwindgasse 20 / 2-3, Vienna, 1040, Austria.
[2*]Business Informatics Group, TU Wien, Favoritenstrasse 9-11, Vienna, 1040, Austria.

*Corresponding author(s). E-mail(s): dominik.bork@tuwien.ac.at;
Contributing authors: hmetin@eclipsesource.com;

Abstract

Web-based modeling tools provide unprecedented opportunities for the realization of modern, powerful, and usable diagram editors running in the cloud. The development of such tools, however, still poses significant challenges for developers. The Graphical Language Server Platform (GLSP) aims to reduce some of these challenges by providing the necessary frameworks to efficiently create web modeling tools. However, realizing modeling tools with GLSP remains challenging and not much support for interested tool developers is provided yet. This paper discusses these challenges and lessons learned after working with GLSP and realizing several GLSP-based modeling tools. We present experiences, concepts, and a reusable reference architecture to develop and operate GLSP-based web modeling tools. As a proof of concept, we report on the realization of a GLSP-based UML editor called BIGUML. Through BIGUML, we show that our procedure and the reference architecture we developed resulted in a scalable and flexible GLSP-based web modeling tool for the UML. The lessons learned, the procedural approach, the reference architecture, and the critical reflection on the challenges and opportunities of using GLSP provide valuable insights to the community and shall ease the decision of whether or not to use GLSP for future tool development projects. With this paper, we publicly release a reference implementation of our architecture.

Keywords: UML, Software modeling, GLSP, Modeling tool, Web modeling, LSP

1 Introduction

The development of modeling tools has a long tradition in modeling research [13, 18, 20, 36, 47] and is acknowledged as a scientific contribution [38]. The availability of web technologies and frameworks like the Graphical Language Server Platform (GLSP), which are built on these technologies, enable new avenues for the modeling community to develop and deploy custom, flexible, and highly usable web-based modeling tools.

Moreover, frameworks like GLSP enable flexibility by allowing developers to customize modeling tools on all architectural layers. While this provides more freedom and power compared to traditional metamodeling platforms where customizations were mostly only possible at well-defined places, this freedom also raises the entry barriers for new developers. However, as of now, there is a gap in lessons learned, best practices, and reusable architectural patterns that foster the development of this new breed of tools. The paper at hand aims to tackle this gap.

Modeling tools assist users in efficiently creating models of high quality by following standards like UML or ER, or by supporting custom domain-specific languages [53]. Today, model engineering has many different tools at its disposal. Most of these tools are mature applications that have been actively worked on over a relatively long period but have barely evolved in recent years [15]. Their functionalities are often built on older technology stacks, i.e., they are not compatible with state-of-the-art web technologies [8, 9]. Aside from a tool's functionalities, a well-designed, modern, and responsive graphical user interface is crucial for efficient and enjoyable use [48]. However, current tools are often labeled as not very useful [15, 41]. Tool development is, therefore, still denoted as a crucial part of modeling research and a valuable research contribution to the community [32, 38].

Historically, Integrated Development Environments (IDEs), similar to established modeling tools, were developed as rich clients with built-in functionality for all the necessary language support. Recently, a trend toward separating the client from the language-specific part using the Language Server Protocol (LSP) [6] to realize more flexible and modular architectures [4] can be recognized. This change allows thin clients focusing on responsive and modern UIs to be hosted on the web and connected to a language server as the backbone, which does the heavy lifting and provides the language smarts. As more editors utilize web technologies, we now see similar possibilities arising for modeling tools, i.e., *web modeling tools* [7, 44]. However, the development of web-based modeling tools still poses significant challenges for developers.

The Graphical Language Server Platform (GLSP) [10] aims to mitigate these challenges by allowing users to develop modeling tools similar to other LSP-based editors [43, 44]. Yet, GLSP is relatively new. Documentation and a few examples already exist but as with every new framework, there is a lack of reported lessons learned, experiences, best practices, and discussions about using those. Consequently, researchers and developers aiming to create new web modeling tools face the challenge of making an informed decision about whether or not to adopt new frameworks like GLSP. Relevant information with respect to such decisions is missing, e.g., what features such technology provides, which prerequisites need to be fulfilled, what effort is attached to the development, how the development should be conducted, and what its limitations are. Moreover, developers who already chose GLSP for tool development face the challenge of navigating the existing overarching code base with several frameworks and components to collaboratively realize the GLSP-based modeling tools.

This paper is an extended version of our previous conference publication [30]. Several extensions are reported in this paper, primarily, we want to steer the reader to: *i*) a more comprehensive discussion of related works (see Section 2.1); *ii*) a richer elaboration on the history of the reference architecture development (see Section 3.5); *iii*) major extensions to the whole reference architecture itself (see Section 4) by e.g., more details on technical implementation with examples (see Section 5), the addition of several diagrams to visually illustrate the reference architecture concepts and design patterns; and *iv*) an extended discussion now featuring recommendations for developers and a vision for our reference architecture (see Section 6).

This paper shares our experience of realizing several web modeling tools with GLSP and provides a reference architecture that aims to foster tool development by providing a higher abstraction layer for modeling tool developers composed of generic implementations of recurring functionality. We share our lessons learned and reflect on the strengths and weaknesses of GLSP as well as the prerequisites for realizing modern web modeling tools with GLSP. As the modeling community is increasingly interested in the development of web modeling tools (cf. [8, 32, 43, 44]), we believe this paper makes an original contribution that is of value for researchers and software engineers. For researchers interested in establishing similar reference architectures for other tool development platforms, this paper holds relevant experience, lessons learned, and practical hands-on patterns. For developers who consider developing such a tool or migrating an existing standalone tool (e.g., EMF-based) to a web modeling tool, we publicly release a reference implementation of our architecture to foster adoption and reuse [28, 31].

In the remainder of this paper, Section 2 first introduces GLSP, relevant foundations, and related works. Afterward, we move the attention

to the development and deployment of GLSP-based web modeling tools in Section 3. A reference architecture is presented in Section 4 and its instantiation in the BIGUML proof-of-concept modeling tool is described in Section 5. We close this paper with a comprehensive discussion of our lessons learned, a critical reflection, and some recommendations for prospective GLSP tool developers in Section 6 and concluding remarks in Section 7.

2 Graphical Language Server Platform

The Graphical Language Server Platform (GLSP) is an extensible open-source framework for building custom diagram editors based on web technologies [10]. The realized editors can be integrated into plain web applications but also into tool platforms such as Eclipse Theia[1] and VS Code[2], and even in traditional Rich Client Application platforms like Eclipse RCP[3]. GLSP is an open-source project hosted at the Eclipse Foundation on GitHub[4]. GLSP is under active development by the community; the current major release, version 2.2.1, was announced in July 2024.

Generally, GLSP adopts the basic protocol structure and way of working as introduced by the Language Server Protocol (LSP) [33, 34]. GLSP extends LSP to account for the specific challenges of working with graphical models (compared to textual documents). These challenges include e.g., moving from a two-dimensional (document row, character position) to a three-dimensional space (elements occupy a geographical area and can graphically and semantically compose child elements); moving from plain editing operations that boil down to character edits to complex editing operations like creating a relationship between two nodes in a diagram, constraining the allowed connections, moving elements geographically inside another one, etc. In its current version, GLSP-based web modeling tools are built on the following core components (cf. Fig. 1):

Fig. 1: GLSP Core Architecture [10]

- **Server framework.** Used to build particular GLSP diagram servers for e.g., UML or a domain-specific graphical modeling language;
- **Client framework.** Used to build a particular GLSP graphical modeling language client including e.g., rendering styles for the concrete syntax and user interaction;
- **Protocol.** The messages that can be exchanged between the GLSP-Clients and servers are specified in a flexible and extensible GLSP protocol which handles graphical diagram editing operations similar to how LSP handles textual editing operations;
- **Platform integration.** Reusable platform integration components that take an implemented GLSP diagram client and integrate it seamlessly into e.g., Eclipse RCP, Atom, or VSCode.
- **Source Model.** The source containing the model e.g., a UML model.

With these components, GLSP enables the development of web-based diagram clients, whereas the front-end is focused on rendering the model reacting to user interactions, all the language smarts like language implementation, model management, model validation, model manipulation, etc., are encapsulated in a diagram server. This separation of concerns, which is already seeing great adoption and success in LSP, enables high flexibility and interoperability [7]. Similarly to the idea behind LSP, GLSP allows the implementation of the language smarts in a client-agnostic way which fosters reuse and flexible integration of the editor in arbitrary client frameworks as long as they 'speak' the same language, i.e., they communicate via the standardized and extensible protocol. What is left to be done to achieve such multiple-client support

[1] https://theia-ide.org/, last visited: 03.10.2024
[2] https://code.visualstudio.com/, last visited: 03.10.2024
[3] https://wiki.eclipse.org/Rich_Client_Platform/, last visited: 03.10.2024
[4] https://github.com/eclipse-glsp, last visited: 15.09.2024

is to customize the user interaction and the look-and-feel using the client's API and the platform integrations offered by GLSP [7].

Accordingly, the GLSP framework is fundamentally language-independent, containing no inherent knowledge of the specific modeling language. It provides a reusable foundation that supports graphical editing operations universally across any modeling language like the creation/deletion of nodes and edges. GLSP offers designated contact points that framework users are required to implement. These include aspects such as the source model and its management, i.e., how the model is saved, edited, and otherwise manipulated, which constitute the language-dependent components of GLSP. These implementations are tailored to the specific requirements of the graphical language in use, such as UML, and provide the necessary customization to handle unique model syntax, semantics, and validation rules. This architecture aligns with the principles of LSP, which abstracts language-specific complexities to enhance focus on building effective client-side applications.

Action & Action Handlers

In the context of GLSP, 'Actions' are structured messages exchanged between client and server. These are crucial for implementing the Messaging and Command patterns within GLSP, facilitating the necessary interactions for operations such as model updates, element creations, and save actions. 'ActionHandlers' correspond to Actions by processing these messages. Each ActionHandler is specific to an Action type, interpreting its data and executing the relevant operations to modify the system state as requested by the Action. For example, an ActionHandler for a CreateOperation[5] will add a new element to the model based on the details provided in the Action. With these patterns, GLSP inherently supports multiple undo/redo capabilities. That means that GLSP manages operations through a command stack for reversal or replay. This architecture enables a clear separation of concerns, allowing for the modular addition or modification of functionalities through new Actions and corresponding Handlers. For more detailed information on implementing Actions and ActionHandlers, the GLSP documentation[6] provides extensive resources and examples.

Built-in Functionality

There is already a list of functionalities that GLSP offers out of the box. For example, GLSP supports the creation of nested nodes within diagrams. It also facilitates the customization of edge routing points, allowing for precise control over node connections. Additional functionalities include copy-paste capabilities for model editing and experimental accessibility features such as keyboard-only navigation, further detailed by Sarioglu et al. [45]. At the same time, GLSP integrates keyboard shortcuts for streamlined operations and experimental helper lines for alignment and positioning. Essential operations like undo and redo are natively supported, alongside the use of ghost elements for previewing changes before they are applied. For a comprehensive list of all features, we refer to the GLSP repository on GitHub[7].

2.1 Related Work

In the following, we briefly report on related frameworks and platforms that focus on the development of web-based modeling tools. Note, that we do not cover a comprehensive systematic assessment of other approaches as this would not fit within this paper. The aim is to shed light on other platforms that provide similar means to support the development of web-based modeling tools as GLSP does.

2.1.1 Related Frameworks and Platforms

Other tools also enable the construction of graphical editors by utilizing web technologies such as SiriusWeb, CINCO Cloud, WebGME, AToMPM, PictoWeb, and Kieler. In the following, we briefly elaborate on our experience of exploring these tools and the extent to which they support the development of new web-based modeling tools.

[5] Operations are special Actions that modify the model source

[6] https://eclipse.dev/glsp/documentation/actionhandler/, last visited 15.11.2024

[7] https://github.com/eclipse-glsp/glsp, last visited 24.11.2024

SiriusWeb[8] is a low-code platform to create and deploy diagram editors on the web. It is a further development of the long-lasting Eclipse project Sirius[9] that allowed the development of graphical concrete syntaxes for Ecore metamodels developed with the Eclipse Modeling Framework. SiriusWeb focuses on providing the modeling interface through the web while the language implementation is not natively supported by web technologies. SiriusWeb is an open-source project under the Eclipse Foundation and publicly hosted in a Github repository[10] with an active developer community. Modeling projects realized with SiriusWeb can be easily shared with collaborators using the project's URL. The platform then supports real-time collaborative modeling. It moreover comes with automated layout algorithms to support the appropriate rendering of large models. SiriusWeb is also ready for integration into other web applications via webhooks[11]. Such webhooks allow third-party applications to register to certain platform events, similar to the actions in GLSP. Once a subscribed event is emitted, SiriusWeb sends an HTTP POST message to the registered URL.

AToMPM [49][12] is an open-source framework for designing domain-specific modeling environments, performing model transformations, and manipulating and managing models. AToMPM runs fully in the cloud and allows online collaborative modeling. The framework is hosted as a public repository on Github[13] with extensive documentation for users[14]. AToMPM, similarly to GLSP, also separates a client from a server. The client provides a GUI that modelers and method engineers use to create models and domain-specific modeling languages, respectively. The framework is highly flexible and comes with a Plugin Manager that allows swift extension of the core functionality for modeling language- and functionality-specific requirements [49]. One focus and strength of AToMPM is the efficient definition and execution of model transformations. AToMPM is a research framework for the development of new domain-specific languages with additional features like model transformation. It follows a traditional metamodeling framework paradigm where new modeling languages are defined by means of specializing in a very generic class diagram language along well-defined extension paths. Integrations into other platforms are not intuitively supported.

Web-based Generic Modeling Environment (WebGME) [24][15] is a well-known tool in the realm of graphical language engineering on the web, emerging as a successor to the original Eclipse Generic Modeling Environment (GME) [23]. WebGME is hosted as a public Github repository and still under development albeit its latest release in 2021[16] A strength of WebGME is that the language engineering and language use components are tightly integrated, enabling the modeling editor to reflect on the effects of language (i.e., metamodel) changes on the fly. WebGME, similarly to SiriusWeb and AToMPM is focusing on providing a rich and responsive, web-based user interface to a model. Unique features of WebGME are its prototypical inheritance, multi-paradigm modeling, extensibility, and version control [24]. The architecture of WebGME also allows the extension of the core tool with plugins that further enrich the functionality of the developed tools. WebGME supports this extensibility by offering a REST web services API that enables language and technology-independent access to a model via a Model API [24].

CINCO Cloud [3][17], further development of the CINCO [35] application, is a powerful client-server and language server-based architecture for the development and deployment of modeling tools in the cloud. CINCO Cloud can be accessed via a public Gitlab repository[18] and it is under active development. For the realization of graphical modeling editors and the rendering of the diagrams, CINCO Cloud also uses GLSP and

[8] https://www.eclipse.org/sirius/sirius-web.html, last visited: 08.10.2024
[9] https://eclipse.dev/sirius/, last visited: 09.11.2024
[10] https://github.com/eclipse-sirius/sirius-web, last visited: 10.11.2024
[11] http://docs.obeostudio.com/, last visited: 09.12.2024
[12] https://atompm.github.io/, last visited: 09.11.2024
[13] https://github.com/AToMPM/atompm, last visited: 09.10.2024
[14] https://atompm.readthedocs.io/en/latest/, last visited: 08.10.2024
[15] https://webgme.org/, last visited: 09.10.2024
[16] https://github.com/webgme/webgme, last visited: 09.10.2024
[17] https://scce.gitlab.io/cinco-cloud/, last visited: 09.10.2024
[18] https://gitlab.com/scce/cinco-cloud, last visited: 09.10.2024

Sprotty, respectively. Given its roots in the Eclipse CINCO project, developers can now configure their language server for a new modeling language using Xtext and Xtend and then use this language server within CINCO Cloud. The platform comes with a clear architecture that separates the modeling language-dependent logic from the core logic of model editors and the logic in which the client and server communicate.

Picto Web[19] is a platform, that generates web-based views of conceptual models. Models of different modeling languages can be imported and are then transformed into HTML, Graphviz, and PlantUML formats using rule-based model-to-text transformations [55]. The focus of Picto Web is to enable the efficient generation of responsive and interactive model visualizations on the web. The development of new modeling languages or full-fledged web-based model editors is not in the scope of the current version of Picto Web.

One recently proposed platform that is close to GLSP is KIELER [19]. KIELER also utilizes the LSP-extended version of Sprotty for rendering diagrams. It moreover also uses similar communication paradigms between the client and the server and how model updates are treated (i.e., model edits are first applied to the course model, then an updated model is sent to the client for rendering). In contrast to GLSP, which follows a diagram-first approach, KIELER emphasizes a text-first approach, assuming a given metamodel and a textual concrete syntax from which a diagrammatic representation is automatically generated and kept consistent during the modeling process. Another feature of KIELER is that it comes with advanced visualization and layout algorithms and a set of default implementations for rendering models.

In 2023, jjodel[20] was introduced [42]. jjodel is a cloud-based, reflective modeling software realized with web technologies (javascript libraries and typescript) [50]. One interesting aspect of jjodel is its aim to support students and the general user basis by limiting the efforts for configuration, installation, and deployment that naturally establishes an entry barrier for most other metamodeling platforms [42]. jjodel follows the traditional metamodeling platform way of supporting language development with a graphical language definition editor and a default graphical concrete syntax. The coarse-grained architecture of jjodel [50] also follows a separation of concerns where a *Model API* component encapsulates the access to the model to a model externalization component on the one side and to the *View Layer* and *Rendering* components on the other. Communication between the components is, like in the case of GLSP, handled by triggered events which are mapped, with optional constraints, to actions.

Gentleman [22] was also recently introduced. Gentleman is a lightweight web-based projectional editor that allows users to create models with simple structures and manipulate them with user-friendly projections [22]. The projectional architecture realized in Gentleman enables users to efficiently define projections for tailoring the rendering, creation, and manipulation of models in graphical, tabular, or textual views, or even in interactive widgets. The architecture of Gentleman supports separation of concerns, by separating the *Concept Component API* with basic model CRUD operations from the *Projection Component* that supports the definition and management of model projections and the *Editor Component* that handles e.g., the user interaction, import/export functionality, and that offers extension points to the developers.

Dandelion [25] is another recently released cloud-based language engineering workbench. It is based on a technology stack that primarily uses web technologies like React and vis.js on the front-end and Node.js and Typescript on the back-end. At the core, Dandelion integrates technologies from the EMF work, like the Eclipse Layout Kernel (ELK), for the automated layout of the models. Metamodeling in Dandelion is supported by a graphical editor while the language engineering itself is supported by the Droid [2] recommender, which recommends metamodel properties to the engineer during the metamodeling process itself. While currently offering an approach that is independent of GLSP, the authors stress that they are interested in aligning to the working scheme of GLSP [2].

[19] https://github.com/epsilonlabs/picto-web, last visited: 10.10.2024

[20] https://github.com/MDEGroup/jjodel, last visited: 14.10.2024

2.1.2 Related Experience with GLSP

Given the rise in popularity of GLSP, an increasing number of academic articles report on their respective experience of realizing web-based modeling tools with GLSP. In the following, we briefly reflect on them.

Walker et al. [51] report on their experience in realizing a domain-specific language for the development of distributed real-time systems. The authors report on their success in combining an Xtext-based textual concrete syntax with a GLSP-based graphical concrete syntax. Their tool uses the validation features of GLSP and the platform integrations to VS Code as an IDE for running the tool.

Walsh et al. [52] introduce an architecture of hybrid language servers. The architecture combines LSP for the textual language server, GLSP for the graphical one, and a client that is capable of communicating with both servers. The fundamental idea put forward by the authors is that both language servers are based on the same abstract syntax, which eases their integration. Their architecture is successfully applied in the development of a hybrid modeling tool for UML-RT [46].

Hölzl and Barner [17] report on their efforts to implement the model-based engineering tool AutoFOCUS 3[21]. Given that tool support for AutoFOCUS 3 was already available within the Eclipse Rich-Client Platform, the authors report on their efforts of migrating part of the tool's functionality using a web technology stack. The new tool uses Theia[1] for the front-end and back-end components, both extended with the GLSP components for rendering the model and processing the model editing operations, respectively. An additional EMF-based model server handles validation and is connected to the GLSP-Server for (de-)serialization of the graph model which is required for the Sprotty-based rendering of the model in the front-end. The authors report the successful implementation of their tool while they also share challenges they faced which are related to our reference architecture. Primarily, the authors mention two challenges: the separation of concerns, i.e., that server and client are implemented and deployed separately, and the dependency injection, i.e., that the injection and, therefore, overriding of default behavior can lead to inconsistency and increased complexity. The authors propose the creation of comprehensive code documentations that also interrelate the server and the client. Moreover, they ask for extended testing support to prevent injecting invalid code.

Ali et al. [1] report on the evaluation of several frameworks for the development of web-based modeling tools, including GLSP. The authors conclude that GSLP servers satisfy most of their requirements and, subsequently, report on the successful implementation of a GLSP-based modeling tool called CaMCOA Cloud. In their architecture, they combine Eclipse GLSP with the Theia integration and a dedicated model server. While the authors report on the success of adopting GLSP, they also stress challenges related to the evolution of the GLSP codebase itself, and the lack of documentation and best practices. The paper at hand should help developers who face similar challenges when using GLSP for the first time.

An interesting practical evaluation of GLSP is reported by McLeod and Cox [26]. The authors stress the code-focused nature of GLSP-based tool development and present different alternatives for separating the concerns of the server and the client which are informed by a previous development experience, realizing a graphical modeling tool with Smalltalk. We believe some of the criticism raised (like the required lines of code) can be mitigated by adopting our reference architecture.

In an effort to compare different Low Code Development Platforms with respect to their extensibility, Popov et al. [40] conclude that Langium is the best tool for language engineering while GLSP is the best diagram editing framework among the platforms they analyzed.

2.1.3 Synopsis

While a systematic and comprehensive comparison of related frameworks and platforms for the development of web-based modeling tools is not within the scope of this paper, we believe it is noteworthy to state that there are many other approaches for the development of web-based modeling tools available with an active community pushing the further development of

[21] https://www.fortiss.org/en/results/software/autofocus-3, last visited: 15.10.2024

these approaches. Section 2.1.1 provided an initial overview of recent and mature platforms.

When analyzing the papers that introduce these related approaches and exploring their source code repositories and documentation, it becomes apparent that there are neither reported reference architectures nor reference implementations. Most approaches provide documentation, a short hands-on tutorial on a simplified metamodel, and maybe a high-level architecture of the few core components and their interplay. A detailed conceptual reference architecture and its technical implementation are in great need.

Section 2.1.2 sheds light on several already publicly documented experience reports of using GLSP. The tools developed come from diverse backgrounds and have different levels of complexity when considering their metamodel, their tool functions, and their tool integrations into IDEs. From the analysis of these experiences, a twofold synopsis can be derived. On the one hand, GLSP has been successfully applied in diverse academic groups for the development of web-based modeling tools. All authors report that they are satisfied with the results, often emphasizing the flexibility of GLSP and its rich client-side model rendering and user interaction. On the other hand, challenges are reported which also relate to the flexibility of GSLP (e.g., the separate development of client and server components), the development effort (e.g., necessitating redeployment of components once changes have been performed, the end-to-end testing of the tool), and the lack of documentation, best practices, and reference implementations.

The literature underpins that GLSP offers a flexible framework designed to integrate seamlessly with modern development environments. GLSP focuses on providing a robust back-end for diagram editors which can be easily connected to various front-ends, promoting a highly modular and flexible approach. The feasibility of the approach taken by GLSP is also visible in the implemented solutions industrially[22] and in the open-source community[23] having different business domains.

With the publication of this paper, we hope to start a reflection and comparison process that might trigger the development of a generic reference architecture for web-based modeling tools to which several platform and tool vendors could relate their own architecture. As a manageable first step, we propose a concrete reference architecture for the development of GLSP-based web modeling tools in the following.

3 Developing GLSP-based Web Modeling Tools

GLSP provides an extensible client-server framework to develop web modeling tools[24]. This extensible framework offers developers different implementation options. Currently, the following options for the implementation can be used [7]:

1. **GLSP-Server.** It is possible to implement the server with Java or TypeScript with NodeJS.
2. **Source Model.** The means to save the source models can also be chosen. GLSP allows accessing the models in different formats or even remotely. The framework provides base modules for common choices like EMF, EMF.cloud, or saving the GModel (i.e., graphical elements) directly.
3. **Tool Platform.** GLSP allows developers to employ any web-based client, and use the editor in a web app or as a standalone application. Client integrations exist for easier usage for platforms such as Eclipse Theia, VS Code, and Eclipse RCP.

These options allow for different combinations. Fortunately, GLSP offers flexible getting-started templates for quickly setting up the development environment for common combinations[25]. Nevertheless, developers are not constrained to these combinations but can create their solutions without using the provided templates just by using the framework directly. Notably, these options and the freedom provided by GLSP require the developers to determine the modeling tool's scope and usage/integration scenarios before starting the development.

[22] https://blogs.eclipse.org/post/paul-buck/theia-adopter-story-logicloud-modern-engineering-platform-industrial-automation, last visited: 13.10.2024

[23] https://github.com/imixs/open-bpmn, last visited: 13.10.2024

[24] https://www.eclipse.org/glsp/documentation, last visited: 15.11.2024

[25] https://eclipse.dev/glsp/documentation/gettingstarted/, last visited: 04.12.2024

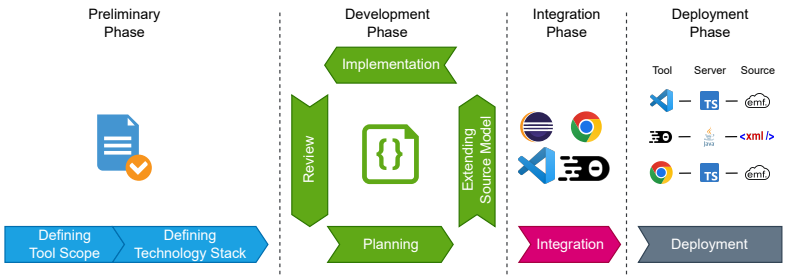

Fig. 2: Development and operation process for GLSP-based web modeling tools.

In the following, we will focus on those open questions developers need to answer and provide a GLSP development and operation process (illustrated in Fig. 2) to structure the realization of GLSP-based web modeling tools. The process consists of four phases. The *Preliminary* phase focuses on the tasks necessary before developing the modeling tool. Important questions related to the scope and the technology stack of the modeling tool need to be answered here. Afterward, the *Development* phase follows. Here, the modeling tool is iteratively realized. After reaching a stable version, integrations into the targeted (optional) tool platforms are required, which is realized in the *Integration* phase. Finally, the *Deployment* phase is concerned with deploying the modeling tool. The phases will be explained in greater detail in the following.

3.1 Preliminary Phase

The preliminary phase sets the scope and the technology stack for the modeling tool project. Consequently, its decisions should be stable over time as changes to these decisions likely have far-reaching effects on all subsequent phases.

3.1.1 Defining Tool Scope

Before deciding on the technology stack and starting to develop, knowing which goal the tool should fulfill is crucial. GLSP is language-agnostic and only provides the foundation to abstract from the protocol and the interactions. Consequently, the developers need to decide the language-specific parts, like which elements (e.g., nodes, edges) the diagram consists of or how to interact with the elements. Moreover, GLSP provides only the basic features (e.g., CRUD operations, tool palette). Still, the extensibility of GLSP allows for providing custom features (e.g., a property palette, a minimap, or a diagram outline). Therefore, the resources required to implement those custom features must also be considered. Additionally, tools can support single or multiple diagram types. A modeling tool that only interacts with a single diagram type requires a different approach than one that supports various diagram types.

3.1.2 Defining Technology Stack

The technology stack must be determined depending on the developers' experience and the tool's scope for the GLSP-Client and GLSP-Server.

- **Client.** The GLSP-Client is developed with TypeScript and can be extended or modified depending on the tool's scope. Necessary knowledge of SVG to render the diagram elements is required. Knowledge about browsers or tool platforms (e.g., VS Code, Eclipse Theia) is helpful for more complex custom features.
- **Server.** The server can be implemented in Java or TypeScript. Custom support for other languages would be possible due to the open protocol. It can be beneficial to have the full stack in the same language (e.g., TypeScript) as the TypeScript version of the GLSP-Server is aligned with the Java version.

The GLSP-Client and GLSP-Server can be reused for all of the tool platforms, which enables cross-platform interoperability for the same diagram-specific features. Only the platform-specific features (e.g., UI elements like menus) must be implemented for each platform separately. Aside from the programming language and tool platforms, how the source model should be

managed is essential. As the EMF.cloud has integrations for GLSP, it would be possible to reuse all of the Ecore functionality for the GLSP-based web modeling tool. However, GLSP also supports other formats for the source model like XML and JSON.

3.2 Development Phase

This phase focuses on the realization of the modeling tool. Here, new features are iteratively developed and tested. Integrations to the different tool platforms are separate from this phase because GLSP works outside of tool platform-specific features the same way for all platforms. Generally, two ways to realize new features can be distinguished: *feature-oriented* and *architecture-oriented*.

- In **feature-oriented development**, the goal is to develop a single feature through all components of the GLSP architecture before starting another feature. Each feature is implemented from the source model to the GLSP-Client or the other way around. This approach allows the incremental and independent implementation of new features by different developers. However, the developers need experience and need to be aware of all coding guidelines in all components for this approach to work properly.
- In **architecture-oriented development**, multiple features are developed for a single component of the GLSP architecture. This approach is better suited for an organization with multiple teams. Different teams can be responsible for different components and provide the necessary functionality. This approach allows better isolation between the components and the teams but requires more organizational overhead as specific teams are responsible for everything on a specific component.

Both approaches have advantages and disadvantages. It depends on the experience and organizational structure of the project at hand to select the best suitable option or a combination of them. Feature-oriented development is more suited for smaller teams and for less complex modeling tools, i.e., tools supporting small metamodels of less than ~ 30 elements and offering none or only selected additional functionality on top of basic modeling support like CRUD operations for all metamodel elements. In contrast, architecture-oriented development better aligns with more complex modeling tools, i.e., tools supporting large metamodels and offering a lot of advanced features like model transformation, code generation, AI assistants, etc. The latter is also significant if, besides GLSP, different other services are used in the architecture. We see strong parallels to the development of larger software systems, where development teams naturally separate into, e.g., specialized teams for front-end and back-end development. The GLSP architecture allows a similar separation of concerns within the tool development team. Independently of the chosen approach, the following steps should be followed.

3.2.1 Planning

After declaring the tool's scope and the technology stack, the tool should be iteratively extended. Every iteration should have clear goals and features that should be introduced or extended. The planning also includes definitions of how the tasks should be reviewed and tested. Moreover, depending on the tasks, the client and/or the server parts could be affected. For this reason, a bottom-up approach is recommended. The source model is available through all the components in the back-end, and it is essential to update it first to access it correctly. Afterward, depending on the tasks, the GLSP-Server and the GLSP-Client must be updated.

3.2.2 Extending the Source Model

Adding new nodes or edges to the editor requires updating the source model. If the source model management is outsourced, for example, to a model server, then necessary changes to those services are needed.

3.2.3 Implementation

This phase focuses on providing the functionality. The developers will implement the features either in a feature-oriented or architecture-oriented approach in the different components. Regardless of the approach, the expected result is the functionality implemented for all components of the GLSP architecture.

3.2.4 Review

Every iteration ends with the review step. An iteration can affect multiple components. Thus, the changes should be adequately tested as defined in the planning step. Testing in GLSP can be performed at various layers, including unit testing, integration testing, and end-to-end testing. For comprehensive end-to-end testing, GLSP offers a specialized framework known as GLSP-Playwright[26]. This framework facilitates the writing of test cases for graphical elements by minimizing the need for boilerplate code. While developers are free to utilize any testing framework that suits their needs, GLSP-Playwright provides tailored support that streamlines the testing process specifically for GLSP applications. Additional details and best practices for using GLSP-Playwright can be found in the documentation and are extensively discussed in [27].

3.3 Integration Phase

The GLSP-Client works cross-platform. Any web-based platform can utilize it. However, if tool platform-specific features (e.g., I/O, Context Menu) will be used, then the GLSP-Client cannot use those independently. In that case, additional integrations are necessary to connect the GLSP-Client with those. Those integrations are per tool platform. Thus, a custom integration will be required for every aimed platform. Consequently, utilizing tool platform-specific features requires additional work.

It is also possible to move the integration phase into the development phase. However, not all modeling tools support different tool platforms. The tools also only sometimes use platform-specific features. For this reason, this phase is optional for most features and is, as a consequence, separated from the core editor development phase.

3.4 Deployment Phase

After reaching a stable version, the modeling tool can be released. Different steps are necessary depending on the scope and supported tool platforms. For feasibility, we assume that the repository uses a continuous integration (CI) / continuous delivery (CD) system to support DevOps. We further assume that the CI is triggered after a merge and that all the components are built, tested, and used by the CD system to deploy it. Depending on the organization, different environments (e.g., registry, production, staging) can exist as a deployment target. Due to the flexibility of GLSP, there exist multiple deployment options [7]. Every part of the architecture can be deployed independently. The servers can also be deployed in containers (e.g., Docker) on different machines. In the following, we list some of the common GLSP deployment options [7, 39]:

- **Integrated Server.** The GLSP-Client and the GLSP-Server are deployed together on the same machine.
- **Separated Server.** The GLSP-Client and the GLSP-Server are deployed and run on different machines.
- **Multiple Servers.** In the case of multiple different servers, they can be hosted on different machines.
- **No Server.** It is possible that the GLSP-Client has no necessity for a GLSP-Server and the GLSP-Client has all the necessary knowledge.

There is no best option. The servers' deployment depends on the developers and the tool's scope and needs to be individually decided. The following list describes the most common deployment scenarios, which can also be combined:

- **Registry Scenario.** Framework developers can release the sources and builds of their modeling tool to a registry (e.g., NPM, Maven), to make it publicly accessible. This approach allows other developers to reuse the released code in their modeling tools.
- **Standalone Scenario.** In this case, a web application should utilize the GLSP-Client part of the modeling tool. The GLSP-Client can be released to any internal or online registry (e.g., NPM) and can be loaded from there like any other library in the web application. The server can be hosted like any other server instance (e.g., container, locally).
- **Eclipse Theia Scenario.** The GLSP integration for Theia is used, and the Eclipse Theia instance is afterward hosted. In this case, utilizing a container (e.g., Docker) is recommended. The previously built integration can be started

[26] https://github.com/eclipse-glsp/glsp-playwright, last visited 15.11.2024

with the other servers in the container and accessed from the browser. This approach has the benefit that it is possible to create new clean instances for every user dynamically, which is especially useful for staging and testing environments.
- **VS Code Scenario.** The VS Code integration cannot be used or hosted directly after building it. It needs to be first packaged into a .vsix file. Afterward, it can be installed on any VS Code instance locally or uploaded to the marketplace. Moreover, packaging the servers together with the extension and starting them when the extension starts is recommended.

3.5 Framework Development History

The journey of our framework for GLSP, particularly focusing on the development of BIGUML, demonstrates a clear trajectory of growth, maturation, and refinement over several years. Throughout this journey, the framework was exposed to feedback from more than 50 software developers (Master students at TU Wien who already work as software developers in the industry on a part-time basis). Considering the European Credit Transfer System (ECTS) of the course, the student efforts amount to at least 6,250 hours of working with our framework. Admittedly, the ECTS are a rough estimate. However, it is one means to quantify the extent to which our framework was exposed to developers using it, and, thereby, evaluating it.

We used and thereby evaluated our framework on a twofold basis: First, we used it in an elected Master-level course on Advanced Model Engineering where students heavily worked on extending GLSP and the BIGUML modeling tool. Second, we supervised nine finished master theses and six finished bachelor theses, adding up to an ECTS effort of 7,575 hours or approx. 50 person months that pushed the boundaries of GLSP and our framework to realize advanced features, e.g., for the visualization of models [5, 8], for improving the accessibility of GLSP-based modeling tools [45], and for developing means to support collaborative modeling [16].

- **2021: Initial Development Stages.** During the initial years, BIGUML was developed alongside the early stages of GLSP and a basic model server. These years were crucial for laying the foundational elements of what BIGUML was intended to become. The architecture was still developing, and, as such, it was a period of intense learning and adaptation, responding to the primary needs of handling complex diagrams within a web-based environment.
- **2022: Major Milestones and Architectural Refinement.** The year 2022 was marked by noteworthy advancements. GLSP version 1.0.0 was released at the end of June 2022, bringing with it a more stable architecture that was better suited to support a variety of modeling requirements. In response, BIGUML underwent a major rewrite to not only better accommodate multiple types of diagrams but also to align more closely with the evolving capabilities of GLSP. This year also saw the beginning of a fundamental shift in BIGUML's architecture to a more streamlined and flexible design, incorporating the initial versions of the concepts discussed in this paper.
- **2023-2024: Towards a Generic Framework.** From 2022 to today, the developmental focus shifted towards abstracting the general components of BIGUML into a more generic framework. This evolution was aimed at broadening the applicability of the framework beyond just UML to potentially include other types of modeling paradigms. This step represented a maturation of BIGUML from a specific tool into a more versatile framework capable of supporting a wide array of modeling needs. At the same time, GLSP 2.0.0 was also released, which was incorporated into BIGUML without any issues.
- **2024: Independence from Eclipse IDE.** In 2024, a significant transition was made by switching the build system of BIGUML to Gradle. This change enabled the development of the application to be independent of the Eclipse IDE, aligning with one of the most requested features by our student developers. As a result, developers can now use VSCode or any other preferred IDE to develop.

Each year, the feedback gained from the students in the Advanced Model Engineering course and in the course of their thesis projects played

a pivotal role in shaping the architecture. This feedback not only guided the iterative improvements but also led to a gradual reduction in the recommendations for further improvements as the architecture matured. The consistent enhancement of the framework over these years reflects a responsive and adaptive development process, where practical user feedback directly influences the trajectory and effectiveness of the framework enhancements.

Based on the insights gained over the years, we will now introduce the reference architecture with a focus on its underlying concepts and the lessons we have learned during its development and extensive use for BIGUML.

4 Reference Architecture

The GLSP platform provides the flexibility to design the tool's architecture as the developers wish. GLSP uses Dependency Injection with slim abstractions and direct access to the underlying technologies to allow developers the same power as the GLSP authors. Yet, ignoring some patterns could negatively affect code maintainability, stability, and scalability. Consequently, to overcome those problems, the reference architecture which is derived from our experience of developing several GLSP and Sprotty[27]-based modeling tools [14, 29] as well as the extensive collaboration and knowledge exchange with EclipseSource[28] utilizes the following patterns:

- **Separation of Concerns (SoC).** The GLSP platform comprises multiple components such as the GLSP-Client and GLSP-Server (cf. Fig. 1), and the complexity increases with the addition of services like a model server and ECore. To effectively manage this complexity, the architecture is divided into distinct components, each addressing specific concerns. Concerns range from general, such as *"the model server manages the source models"*, to specific, like *"model mappers create the graphical models"*. SoC is applied not only at the architectural level but also within the code itself, where functionality is encapsulated into discrete units. This structural approach enhances modularity, simplifies the codebase, facilitates maintenance, and improves reusability. Moreover, it supports independent iteration of modules, allowing teams to focus on specific areas without impacting others.
- **Single Source of Truth (SSoT).** The model should only be modifiable and readable from a single place to prevent invalid modifications that could corrupt the source model or result in services (i.e., services that require the model) operating with outdated data. Employing a model server can significantly enhance this aspect by decoupling the source models from GLSP, thus allowing multiple services to connect to the same model server for model updates. However, it is important to acknowledge that a model server introduces additional overhead regarding system complexity and resource requirements such as maintenance, which needs to be managed carefully to maintain system efficiency. For small applications or teams, this overhead could be an obstacle. Besides a model server, organizing modification-related code into distinct code packages or modules can also reinforce the system's integrity by isolating functional responsibilities.
- **Single Responsibility Principle (SRP).** Every component should only focus on one single responsibility because testing and maintaining a component with multiple responsibilities is cumbersome and prone to error. This principle can be applied to different architectural levels, from the implementation level to the server operation or to services.

Building on these foundational principles, we have developed and released a framework[29] [28] that enhances the GLSP-Server, tailored for community reuse. This framework encapsulates all the best practices and architectural patterns discussed in this paper and already includes support for the advanced features detailed in the subsequent sections. By offering this framework, we aim

[27] Eclipse Sprotty is an open source project enabling the creation of powerful diagramming tools and graphical visualizations available at https://sprotty.org/

[28] EclipseSource (https://eclipsesource.com/) is specialized in the industrial development of GLSP-based modeling tools and one of the driving forces behind the further development of GLSP.

[29] https://github.com/glsp-extensions/bigGLSP-framework

to streamline the development process for GLSP-based modeling tools, making these powerful capabilities readily accessible to developers. In the following sections, we will go deeper into the theoretical parts and provide examples in Section 5 based on the framework we have developed.

4.1 Concepts

In this section, we introduce the fundamental concepts that underpin our architecture, focusing on the mechanisms that facilitate modularity and flexibility. A central element in this reference architecture is *Dependency Injection*, which is pivotal for enabling developers to modularize and loosely couple their code, thereby enhancing the separation of concerns [12]. Dependency Injection is employed extensively across all facets of the GLSP platform, serving as a foundational concept that supports the overall system design.

Building upon the flexibility afforded by Dependency Injection, we further categorize the implementation features into distinct modules to improve loose coupling: *core features*, *tool features*, and *diagram features*. This categorization allows for clearer interaction between different components of the architecture, facilitating easier maintenance and scalability.

4.1.1 Modules and Interactions

In the architecture, features are categorized based on their functionalities and interaction with the underlying server framework (e.g., GLSP, Modelserver, and others). This structuring not only organizes the features into logical units but also outlines clear interaction patterns among them, enhancing the modularity and extensibility of the system as illustrated in Fig. 3. Building on the discussion about GLSP (see Section 2), our framework splits functionality into language-independent *Core* and *Tool features* that handle general modeling capabilities, and language-dependent *Diagram features* that manage language-specific operations. This clear division makes modularity more explicit, allowing precise control over language-specific interactions while supporting broad tool functionalities.

Fig. 3 can be read as follows: diagram modules make use of the APIs provided by the tool modules and the core module. In contrast, tool modules can interact with other tool modules and the core module by leveraging their respective APIs to implement extended functionalities. The core module, however, remains isolated in terms of dependencies, only interfacing with the underlying foundation framework (e.g., GLSP, Modelserver) without relying on any tool or diagram modules. The only exception to this rule is when the core module registers or installs (i.e., *binds*) the other modules. This is a necessity due to how Dependency Injection works, as the core module is the only one that can exist independently. The other modules must be referenced to become available in the Dependency Injection container. The subsequent figures will reuse the respective colors to maintain comprehensibility. Diagram modules will be represented in *orange*, tool modules in *green*, and the core module in *blue*. The arrows indicating interactions will consistently originate from these modules, following the same color scheme.

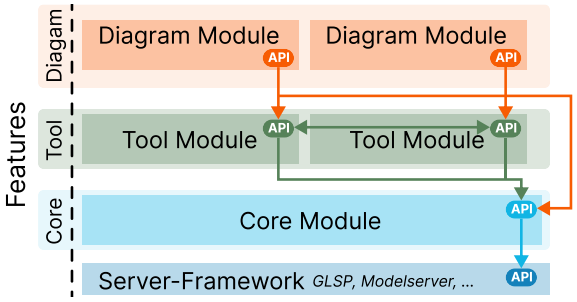

Fig. 3: Features and their Modules

The *Core feature* directly interacts with the underlying server framework (e.g., GLSP-Server, Modelserver) and provides essential server functionalities without incorporating any language-specific information. The primary role of the core feature is to act as the glue code between the underlying server framework and other modules (e.g., tool, diagram), facilitating the loading of these modules and managing the application. This centralized approach enables developers to efficiently respond to changes in the underlying server framework.

Tool features introduce tool specific functionalities. They implement custom behaviors by utilizing interfaces exposed by core features, adhering to the principles of Dependency Injection. Tool features are designed to provide specific functionalities not inherently supported by the GLSP

platform, such as custom user interfaces (e.g., outline view, property palette, minimap), copy-paste capabilities, auto-complete functions, or specific import/export functionalities (e.g., PlantUML). Additionally, tool features expose their own API, allowing other tool and diagram features to enhance or modify behaviors based on the defined contracts.

Diagram features specialize in providing language-specific functionalities and directly accessing the source model. They rely on core and selective tool features to implement CRUD (Create, Read, Update, Delete) operations necessary for other features to interact with the source model. Diagram features ensure that modifications to the source model are controlled and coordinated, preventing direct modifications that could lead to inconsistencies or errors.

4.1.2 Manifests & Contributions

The architecture needs to define clear structural boundaries between features based on the previously defined categorization to ensure effective separation of concerns. Technically, each feature module isolates specific functionality and interacts with other modules via *Manifests* and *Contributions* through Dependency Injection.

- **Manifests.** Every feature module has a Manifest that defines all contributions a feature module aims to make. Manifests serve as the glue code that connects Contributions to their implementations. This is achieved by using the operations (i.e., methods) exposed in the Contribution to register the implementation within the Dependency Injection container. Additionally, Manifests can install other Manifests, enhancing the modularity and reusability of code by allowing complex dependencies and functionalities to be structured hierarchically.
- **Contributions.** Core features and tool features can provide Contributions. These Contributions are used within a feature to delegate execution logic to other features. For instance, delegating GLSP-create operations or providing content for the property palette from diagram modules. The core feature employs Contributions to permit other features to extend or override the default functionality of the underlying server framework (e.g., GLSP platform). Conversely, tool features utilize these core Contributions to enhance specific GLSP platform functionalities or to develop new capabilities. As tool features are language-agnostic, they also offer Contributions that diagram features can implement, thus enabling access to source models from the tool features. Contributions foster a system where different features are loosely coupled, communicating exclusively through well-defined interfaces/APIs. Technically, Contributions provide operations that allow other modules to register their implementations for exposed interfaces/APIs, adhering to the Dependency Injection pattern.

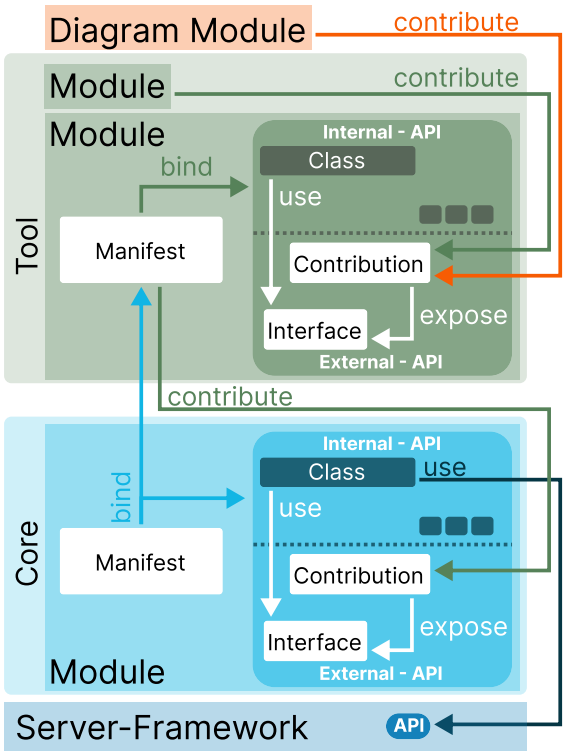

Fig. 4: Manifests and Contributions Overview

Together, Manifests and Contributions facilitate the loose coupling of feature modules (cf. Fig. 4). Manifests define the contributions they wish to implement, and by doing so, they fulfill the requirements set out by Contributions. Thus, modules that provide contribution points can be confident that their needs will be met if utilized. This methodology allows the application to load separate, maintainable, extensible, and modular functionalities.

Section 5.1.1 illustrates the use of Manifests and Contributions in the development of the BIGUML modeling tool.

Limitations

The Manifest and Contribution architecture, while promoting modularity and loose coupling through inversion of control, also has limitations. A key drawback is its reliance on third-party frameworks for Dependency Injection, which can vary in support and configuration complexity. Depending on the programming language, developers can use Guice[30] or InversifyJS[31]. This reliance on frameworks and Dependency Injection can increase development time, steepen learning curves, and introduce potential errors (e.g., invalid configurations), particularly for teams with limited experience in Dependency Injection. These challenges highlight the need for careful framework selection to balance ease of use, flexibility, and architectural goals.

4.1.3 Source Model Representation Separation

A *source model* refers to the underlying data structure that a diagram visually represents, such as UML or ER diagrams. Each source model can be viewed through different *diagram representations*, which are the specific visual interpretations or views presented to the user. For example, in a UML source model, the diagram representation could be a Class diagram, showing classes and their relationships, or a Sequence diagram, focusing on the flow of messages.

The interactions permitted within these representations can vary. A node element in a Communication diagram might show fewer details, while the same node in a Sequence diagram might provide complete information on how and in what order the elements interact with each other, reflecting its usage in the representation's context. Furthermore, certain representations may be set to a read-only mode, imposing further restrictions on what interactions are permitted. To effectively manage these variations, feature modules must always be aware of the currently active diagram representation of the source model. This knowledge is crucial for enforcing interaction constraints specific to each representation.

Section 5.1.2 illustrates the use of the source model representation separation during the development of the BIGUML modeling tool.

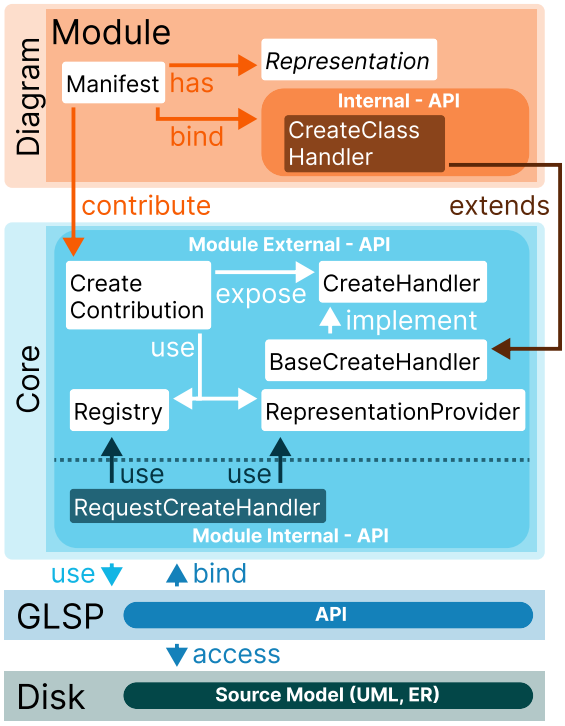

Fig. 5: Representation Separation and Flow Similarity using the Class Creation as an Example

4.1.4 Flow Similarity

In many cases, the flow for operating on different elements in a source model by a modeling tool remains largely identical up until the point of modification. The elements might be semantically distinct, but the process leading up to modifying the source model is similar. Typically, the user initiates an operation, which is processed by the GLSP-Server; subsequently, the source models are updated either by the GLSP-Server or a model server. This similarity in the workflow allows for the implementation of generic functionalities that vary only in the specifics of the model modifications.

[30] https://github.com/google/guice, last visited 15.09.2024
[31] https://github.com/inversify/InversifyJS, last visited 15.09.2024

To streamline the implementation of new operations such as node and relation creation, developers can leverage programming concepts like inheritance or composition. These patterns reduce the effort required by enabling code reuse and providing a structured way to handle common behaviors across different model elements (cf. Fig. 5). In this case, the *BaseCreateHandler* implements all the generic parts, and the *CreateClassHandler* can solely focus on the semantics. Moreover, operating at a meta-level—defining operations in a generic manner rather than individually for each diagram element—can decrease the development overhead. For instance, deletion operations are typically uniform across most elements; implementing these at a higher genericity level rather than for each element individually can reduce complexity and maintenance burden. For example, the core module could implement a generic class that can handle all delete operations without requiring any bindings through the diagram modules. Still, individual diagram elements could rebind/replace the generic implementation if customized solutions are necessary for specific elements.

Adopting a hybrid approach that combines both design patterns and meta-level operations can enhance code stability and reduce the workload involved in interacting with elements. Such strategies not only help maintain cleaner, more maintainable, and extensible code but also facilitate easier troubleshooting and updates since the interactions across different parts of the system follow a predictable and uniform pattern.

Section 5.1.3 illustrates the use of the flow similarity concept during the development of the BIGUML modeling tool.

Limitations

While generic workflows, inheritance, and meta-level operations reduce development effort for modeling tools, they have limitations. Generic approaches may not fully address specific requirements of unique model elements, necessitating custom solutions that disrupt consistency. Over-reliance on inheritance can lead to tightly coupled, hard-to-manage systems, making changes complex and impacting scalability. While simplifying common operations, meta-level abstractions can obscure specific element behaviors, complicating debugging and understanding of the code. Additionally, generalized operations may introduce performance inefficiencies due to insufficient optimization for specific elements. Consequently, balancing generic strategies with tailored implementations is crucial to maintaining flexibility and performance.

4.2 Integrated Architectural Concept

Fig. 6 illustrates an integrated conceptual view of our reference architecture, integrating the previously introduced conceptual components and architectural patterns. At the foundational level of this reference architecture, we have the core module, which wraps GLSP-specific functionalities. This core feature is crucial as it simplifies access to various GLSP components and establishes a base for the application's broader functionality. It defines essential base classes and interfaces that are utilized throughout the application and implements default handlers or customizes GLSP-specific operations to suit unique requirements.

- **Source Model Representation Services.** Within the core module, services are implemented to determine and communicate the context of the active source model. This ensures that the application remains context-aware. Contributions and other files can make use of those services through Dependency Injection.
- **Delegation of GLSP Requests.** The core module also handles the delegation of requests triggered in the context of the GLSP platform to appropriate source model representations. This is pivotal in managing how different user interactions are processed depending on the active diagram or source model context.
- **Contribution Points Exposure.** It exposes various contribution points designed to respect and adapt to the active source model representation. This allows for a flexible extension and customization of the GLSP functionalities, enabling developers to tailor the tool's behavior to specific needs, contexts, or diagrams. At the same time, it also allows the provision of default implementations for specific GLSP functionalities with, if necessary, diagram-level customizations.
- **Meta Level Operations.** Furthermore, the core module facilitates meta-level operations

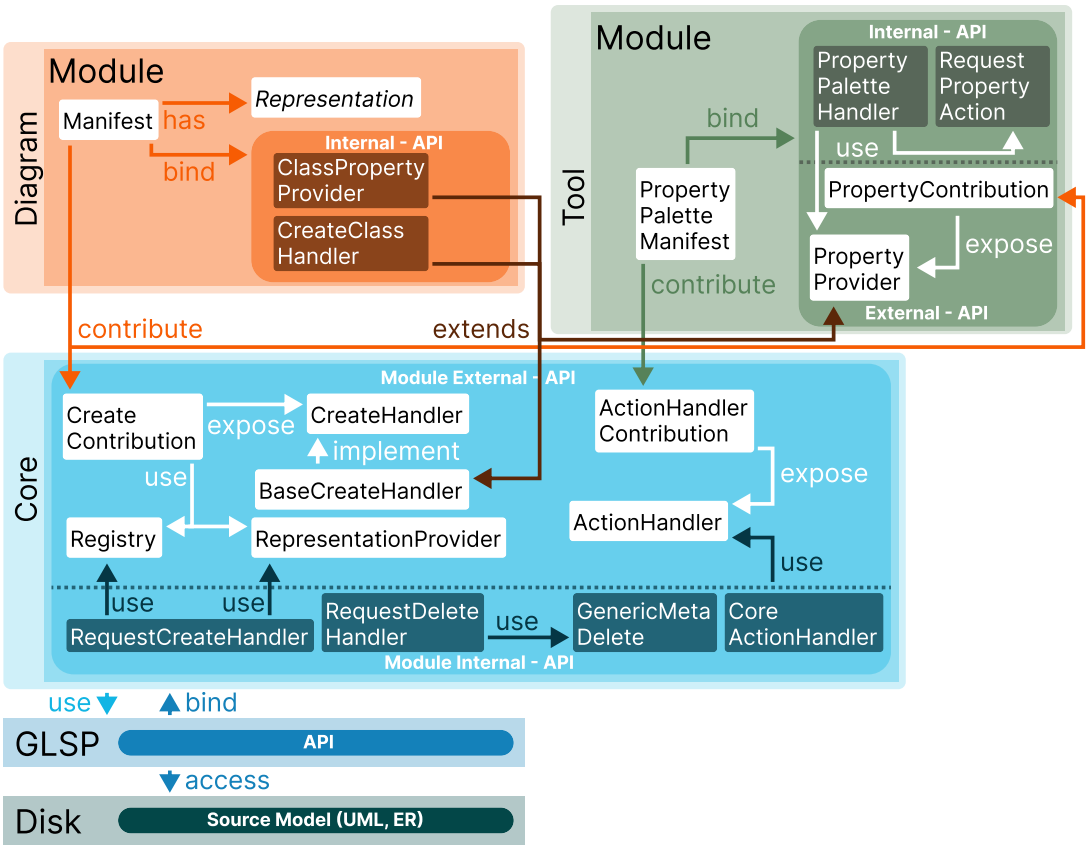

Fig. 6: Integrated Conceptual View on our Reference Architecture (the figure shows illustrative examples of Manifests, Contributions, Actions, and Handlers)

applicable across all diagram elements, such as deletion, reconnecting, or other specific manipulations. These mechanisms are critical for maintaining a consistent and robust manipulation capability across the tool, regardless of the element or the context in which it is used.

Moving up from the core features, the architecture incorporates tool features, which are isolated modules designed to implement specific functionalities not inherently provided by the GLSP platform. Examples of such tool features include a property palette, an outline view, a minimap, or a code generator, which enhance the user interface and interaction capabilities of the modeling tool. These tool features make extensive use of the contribution points defined by the core module. By binding their implementations to these predefined points, tool features can effectively handle specific requests and interactions within the framework. This approach ensures that the tool features are both modular and replaceable. Furthermore, tool features are not limited to utilizing existing contribution points; they also have the capability to define their own. By exposing new contribution points, tool features can offer functionalities that other modules, including diagram modules, can leverage. This enhances the extensibility and scalability of the tool.

Diagram modules play a crucial role in both the representation and manipulation of source models. These modules are tasked with two primary functions: modifying source models and mapping these source model elements to the specific graphical models required by the GLSP platform. Firstly, the diagram modules are responsible for implementing any necessary modifications to the source models. This includes creating, updating, or deleting elements based on user interactions or programmatic requirements. Secondly,

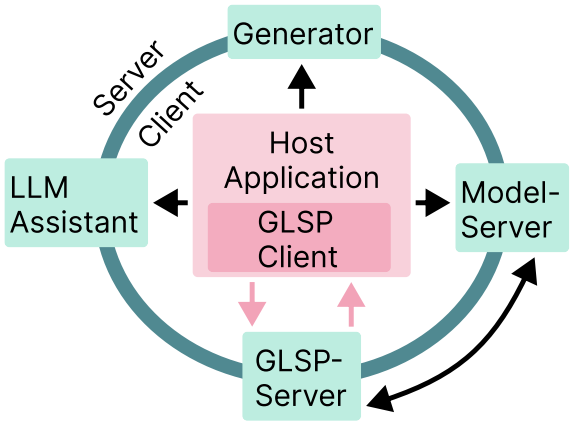

Fig. 7: Application Architecture

these modules map the managed source model elements to their corresponding graphical representations. This mapping is essential for visualizing the elements within the GLSP environment, allowing users to interact with a graphical interface that accurately represents the underlying data structure. Consequently, the diagram modules ensure that each element of the source model is appropriately represented in the graphical model.

4.3 Application Architecture

The application architecture of the GLSP framework and the broader vision for our reference architecture allows for the incorporation of already existing solutions (cf. Fig. 7). The arrows in the figure represent the direction in which communication is usually done. For example, the host application triggers communication with the respective server instances. In contrast, the communication between the GLSP-Client and server is bi-directional, allowing both to exchange messages and respond to changes dynamically.

1. **Client Deployments** As previously mentioned, the primary user interface is deployable across multiple host applications such as Theia, VSCode, and web applications. We chose VSCode for our platform due to its widespread use and the simplicity of installing extensions. Additionally, VSCode extensions can also be installed in Theia, which supports our needs without requiring the additional functionalities that Theia offers[32]. At the same time, VSCode supports seamless background initiation of additional servers. However, running the application on the user's machine introduces security risks since sensitive data like tokens cannot be protected, and the diversity in user environments might cause compatibility issues. To address these issues, it is possible to serve server components on controlled servers while the client remains on the user's machine. This hybrid approach balances security and dependency management but requires careful evaluation of the inherent trade-offs.
2. **Client-Server Interaction** Within the host application's context (e.g., Theia, VSCode), the GLSP-Client renders and facilitates interactions with diagrams, typically interacting mostly with the GLSP-Server. However, it is important not to view this interaction in isolation. We should see the GLSP platform as a backbone that enhances our diagramming capabilities. Nevertheless, we should not forget that we can use additional services besides GLSP, as GLSP can be integrated seamlessly with additional servers or other technologies like Sprotty[27] or Langium[33]. The flexibility of the GLSP-Client allows the host application, whether Theia or VSCode, to utilize these servers, thereby enriching the functionality and extending the capabilities of the overall system.
3. **Integration of Additional Services** Building on the previous point, we can enhance the core functionality of the modeling tool by integrating additional services beyond GLSP. For example, incorporating an LLM-Assistant server enables the use of large language models (LLMs) for data processing and interactions, enhancing the modeling tool's capabilities. Additionally, we can extend the system to include essential features like code generators, which are necessary for comprehensive modeling tools but are not inherently supported by the GLSP platform.
4. **Overall Architecture Model** While each component of the system fundamentally operates on a client-server model, the overarching

[32] https://theia-ide.org/docs/user_install_vscode_extensions, last visited 15.09.2024
[33] https://langium.org/, last visited: 04.10.2024

structure can be described as a star architecture. This model centralizes the GLSP platform and the application host at the core, with the ability to connect various clients and additional service servers. This configuration not only simplifies management and scaling but also allows for the flexible integration or removal of services and servers as required by the deployment environment or specific project needs (see Fig. 7).

5 Proof-of-Concept: The bigUML tool

By following the development and operation process described in Fig. 2 and the reference architecture in Section 4, a GLSP-based UML modeling tool called BIGUML [29] was developed[34]. Initially started as a comprehensive tool for UML diagramming, BIGUML has evolved, with its more generic components now abstracted into a separate reference architecture. This shift has allowed BIGUML to specialize further and focus solely on the diagram-specific aspects of UML.

Software engineers, architects, business users, and more utilize UML regularly. The UML specification comprises multiple diagrams, each presenting unique challenges in terms of tool support. A particular challenge is the need to visually render the same source model element differently across various UML diagram representations, in addition to varying constraints on relationships between nodes depending on the diagram.

Given these specific challenges and the community's experience in how difficult it is to provide rich modeling support for UML [11, 21, 37], we believe the UML case is an excellent candidate to thoroughly test the strengths and weaknesses of the GLSP platform on the one side, and our proposed development and operation process, as well as our reference architecture, on the other. In the following, we will discuss the use of our reference architecture during the realization of BIGUML.

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

5.1 Realization

We followed the feature-driven development approach to implement BIGUML. In every iteration, one feature was implemented throughout the architecture. This approach allowed us to see faster results, detect limits of the current architecture, and rework it accordingly to exploit the improved architecture for the next features. As UML consists of multiple diagram types and representations, it was crucial to have a scalable, maintainable, and extensible architecture. Consequently, we used the introduced concepts and instantiated the reference architecture.

After extracting our framework for GLSP, the focus during these iterations was primarily on the diagram modules. These modules were responsible for implementing the modifications to the source model and for mapping from the source model to the graphical model. To enhance modularity and reusability, each individual diagram element was encapsulated within its own module and installed by the diagram modules. Within these modules, all operations related to that element—such as creation, deletion, and updates—were implemented. This modular approach not only simplified maintenance and scaling but also allowed for more straightforward enhancements and refinements to each element's functionality as needed.

5.1.1 Manifest & Contributions

To demonstrate the modularity and extensibility of our framework, it is important to understand the roles of Manifests and Contributions within the GLSP platform. Manifests act as configuration points within the Dependency Injection framework, specifying which bindings should be used. Contributions, on the other hand, are mechanisms that allow other modules to provide additional functionality, thereby facilitating the customization. For a more thorough introduction to these concepts, please see Section 4.1. Table 1 shows examples of available manifests in BIGUML.

Each tool and diagram feature within the GLSP ecosystem typically utilizes different contributions based on their specific functionalities. Tool features generally extend the base functionalities of GLSP, providing additional features and, as a result, often contribute new actions (e.g., *ActionContribution*). These actions might include anything from new ways to interact with models

Table 1: Manifests excerpt

Manifest Name	Feature	Contributions Used (e.g.)	Description
Autocomplete	Tool	ActionContribution	Exposes custom actions to the application.
Outline	Tool	ActionContribution	Exposes custom actions to the application.
PropertyPalette	Tool	ActionContribution	Exposes custom actions to the application.
EnumerationElement	Diagram	CreateOperationContribution, DirectEditingContribution, PropertyPaletteContribution	Defines the necessary implementations for working with enumerations.
AssociationElement	Diagram	CreateOperationContribution, ReconnectEdgeContribution, PropertyPaletteContribution	Defines the necessary implementations for working with associations.
ClassDiagram	Diagram	ToolPaletteContribution, OutlineContribution	Customizes the tool palette and outline view according to the class diagram.

Table 2: Contributions excerpt

Contribution Name	Provided By	Used By	Description
Action	Core	Tool	Allows to define custom Actions and Action Handlers.
CreateOperation	Core	Diagram	Allows the contribution of custom create operations.
DeleteOperation	Core	Diagram	Allows the contribution of custom delete operations.
DiagramConfiguration	Core	Diagram	Allows diagram modules to configure their node / edge element.
DirectEditing	Core	Diagram	Allows the customization of label editing.
Popup	Core	Diagram	Allows the customization of the hover/popup functionality.
ReconnectEdge	Core	Diagram	Allows the customization of the behavior for reconnecting edges.
ToolPalette	Core	Diagram	Allows diagram modules to contribute custom tool palettes.
Autocomplete	Tool	Tool / Diagram	Allows the customization of the autocomplete feature.
Outline	Tool	Diagram	Allows the customization of the outline feature.
PlantUML	Tool	Diagram	Allows the customization of the import and export PlantUML feature.
PropertyPalette	Tool	Diagram	Allows the customization of the property palette feature.

to user interface enhancements. Therefore, mainly core contributions are used.

On the other hand, diagram features focus more on the semantic representation of the models and typically make use of contributions that are more aligned with the creation, accessing, and manipulation of diagram elements, such as *CreateOperationContribution* or *PropertyPaletteContribution*. That means the diagram features will provide the functionality the core and tool modules require concerning the specific diagram elements. Table 2 provides an excerpt of the Contributions realized in BIGUML.

It is important to note that while the core Contributions like *ActionContribution* and *CreateOperationContribution* can be seen across various tools due to their generic nature, while Contributions such as *ReconnectEdgeContribution*, *OutlineContribution*, *PlantUMLContribution*, and *PropertyPaletteContribution* might vary depending on the specific requirements and context of the diagram they are intended to support.

We will now demonstrate these concepts through a practical example based on the *Property Palette* tool feature (see Fig. 8). The Property Palette is an essential component that provides a user interface for editing properties of selected elements within a diagram.

- **Tool - PropertyPaletteManifest.** The Manifest utilizes the contribution points exposed in the core module to bind custom handlers for specific actions within the property palette. This process involves leveraging existing functionalities of the GLSP platform to enhance the customizability of the property palette.
- **Tool - Property(Palette)Contribution.** In this class, we define methods that allow other modules to provide implementations, such as the next item in this list.
- **Tool - PropertyProvider.** This interface allows individual diagram elements to specify the properties they wish to display in the property palette. By implementing this interface, diagram elements contribute to the dynamic composition of the property palette, with the list of PropertyProviders being used to populate the palette accordingly.
- **Diagram - ClassManifest.** The Manifest for class diagram elements outlines the contributions to the property palette by providing the necessary implementation for the previously defined interface.
- **Diagram - ClassPropertyProvider.** This provider implements the defined interface and specifies which properties of the class diagram elements should be visible in the property palette.

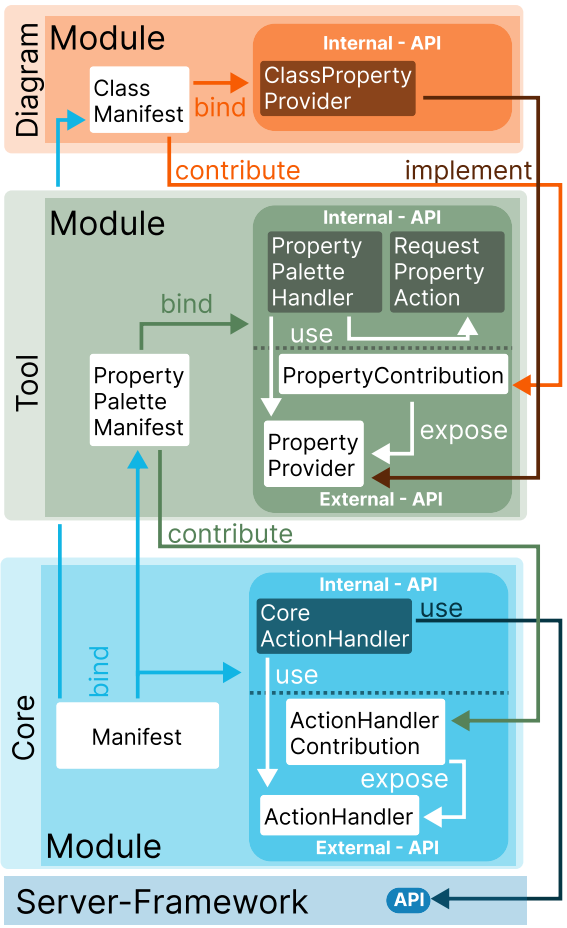

Fig. 8: Property Palette Manifest & Contribution Example

This example not only demonstrates the modularity and flexibility of our reference architecture but also showcases the effective use of Dependency Injection in the GLSP platform. A fundamental principle of GLSP is empowering developers to customize components as if they were the original authors. Following this principle, we introduced an abstraction layer above GLSP that aims to further simplify the tool development process.

In technical terms, as previously mentioned, Manifests serve as containers within a Dependency Injection framework, and Contributions are mechanisms for binding functionality to specific target points. The specific implementation largely depends on the Dependency Injection framework in use. GLSP employs Google Guice for its Java framework; consequently, BIGUML is also required to use it. For guidance on effectively utilizing Google Guice, we refer interested readers to its documentation[35]. In the case of BIGUML, Contributions are classes that take an implementation and bind it within the framework's Dependency Injection system. The sole purpose of this process is to facilitate a more maintainable approach for integrating functionality, thereby simplifying and streamlining the binding process.

5.1.2 Source Model Representation

The core module of our reference architecture is crucial for managing the context of user sessions, particularly in identifying the active diagram representation. Technically, it provides services that detect which diagram representation is active to provide contextual information. Each contribution can now include a registry that catalogs implementation details, using the diagram representation as a key (see Fig. 5). This setup allows for the dynamic querying and retrieval of relevant contributions based on the active diagram.

Likewise, diagram modules are designed to offer specific functionalities related to their representations, activated only when their respective diagrams are in use. For example, contributions pertinent to a Sequence diagram are loaded only when that diagram is being viewed. The core and tool modules selectively load these appropriate contributions from the registries depending on the active diagram representation. This selective loading ensures that the functionalities are not only segregated by context but are also appropriately activated.

By maintaining a registry that links contributions to diagram representations, the system can swiftly adapt to user needs, providing relevant tools and functionalities dynamically. It must be highlighted that the GLSP platform also supports loading dependency modules based on the active diagram, similar to the description provided; in this case, the client must initiate this process. To ensure our solution is future-proof and independently changeable, we have developed a mechanism that allows on-the-fly loading of contributions without requiring the client to provide information about the active diagram.

5.1.3 Flow Similarity

When implementing an editor with the GLSP platform, developers must handle specific actions like *create* and *delete* (of nodes and relations). The GLSP platform provides the necessary meta-information, such as identifying which elements need to be created or deleted, but does not prescribe how to implement these handlers. Instead of creating a handler that can handle all elements, we chose a different approach. Drawing on the strategies outlined in the "Source Model Representation" section, our action handler for the create request can identify the active representation and appropriately delegate the request to the corresponding diagram module by utilizing the respective registries (see Fig. 5). The handle in the specific diagram module is then responsible for modifying the source models. In the diagram module, we apply programming patterns like inheritance and composition to structure our operations efficiently. This design allows us to define a custom API between the GLSP action handler and the source model modifications, using contributions to streamline the process.

5.2 Application

BIGUML is influenced by Eclipse Papyrus [21] and incorporates modern technologies to improve usability and user experience. As of version 0.5.0, BIGUML supports a variety of UML diagrams, including Activity, Class, Communication, Deployment, Information Flow, Package, Sequence, State Machine, and Use Case diagrams. However, it is important to note that some of these diagrams are still under development. Key functionalities such as the Property Palette and Outline view have been extracted to the reference architecture and are available for use without further modifications. This approach simplifies the development process and ensures the availability of these features across different GLSP-based modeling tools. We openly released BIGUML as

[35] https://github.com/google/guice, 10.12.2024

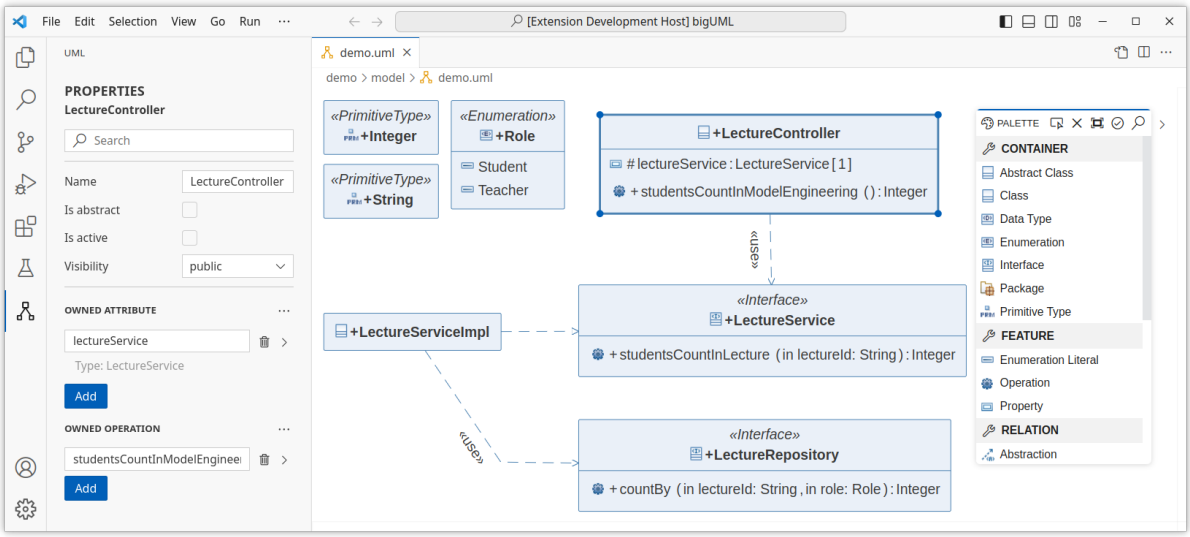

Fig. 9: BIGUML VS Code Extension (available via: https://marketplace.visualstudio.com/items?itemName=BIGModelingTools.umldiagram)

an extension to the VS Code marketplace[34]. As of December 10, 2024, BIGUML has more than 2.600 downloads with approximately ten new downloads daily.

6 Discussion

We now discuss our experience in developing several GLSP-based modeling tools, including BIGUML. The discussion covers the *effort* involved, the *lessons learned*, a *critical reflection*, and finally some *recommendations* for potential future GLSP developers. We conclude the discussion with a vision for our reference architecture.

6.1 Development Effort

Now, we will discuss the effort required to work with GLSP. There are different getting-started templates provided. They only differ in the used technologies like Java or Node and the usage of a model server and the tool platform. Those templates enable a fully running instance that can be customized according to the developer's preferences and goals. Accordingly, the necessary project structure, dependencies, and execution are already addressed. With those templates, the developers can focus on implementing their editor based on the existing structure. The time spent and, consequently, the effort required can be split into the area that will be extended. The developers will mainly focus on the following tasks:

- **Extending the source model.** Defining the source models directly in GLSP or outsourcing the model is also possible. The templates already provide ways to save the models as JSON or to use an EMF-based approach. That means it is possible to define the models from scratch, reuse some existing Ecore models, or use a different server to manage the data. The selected approach determines the initial effort needed. Afterward, extending the source models and introducing the means to modify them can be done without any issues.
- **Defining the graphical model.** To visually display source model elements, they need to be mapped to corresponding elements in the graphical model. The graphical model serves as a description that can be easily transmitted and understood by the client. The level of effort needed for this mapping depends on the complexity of the graphical element. Representing a basic node is simpler compared to an element with multiple parent-child connections.
- **Customizing the rendering.** The client renders the respective graphical models using SVGs and CSS. For this reason, implementing the correct design can be time-consuming.

GLSP already provides basic graphical elements, for example, labels and nodes; regardless, more unique representations require customized implementation from the developers.

GLSP only provides a small set of user-facing editor tool features (e.g., tool palette). It is possible to customize those features, but implementing new editor features from scratch is something potential tool developers need to be aware of. GLSP has been designed to be fully customizable and extensible. Introducing new features can be done on the client and the server side with the help of the low-level means offered by GLSP. The GLSP-Client allows to add new user interfaces (i.e., views). Unfortunately, no commonly known front-end framework (e.g., React, Lit) is used for this part yet. As a result, plain JavaScript (particularly TypeScript) functionality is employed to manage user interactions and build user interfaces. Still, it is possible to generically use such front-end frameworks to ease the development, but it requires initial work from the developers. Hence, it is crucial to acknowledge the importance of implementing customized tool functionality and the associated effort it entails.

Lastly, the discussion of how the reference architecture aligns with the aforementioned points. GLSP offers the essential features required for constructing editors. Yet, it grants developers the flexibility to design the architecture according to their specific needs and requirements. It is possible to develop a functional diagram editor without adhering to a specific schema, but as the complexity increases, maintaining the code becomes more challenging. The reference architecture addresses this by initially separating the tool features, diagrams, and the core functionality of the editor. While this separation introduces additional overhead, such as defining manifests and contribution points, it also provides clear definitions of how the various modules interact through well-defined interfaces. By adopting this approach, the responsibilities of different modules are clearly defined, allowing for independent extension and customization. This leads to a codebase that is both extensible and maintainable, which ultimately should outweigh the initial effort required.

Now we will provide metrics concerning the required effort to extend the source model or features with the reference architecture. We are using Java for the GLSP-Server and model server. It needs to be noted that some programming languages distort the required Lines of Code metric due to the overhead (e.g., headers, imports, formatting). For this reason, we will provide two different Lines of Code metrics. The File Lines of Code (FLoC) provides the number of lines in a file. The Effective Lines of Code (ELoC) describes the lines that need to be updated to provide similar functionality, e.g., creating a different node or edge. Further, we will split the study also into two categories. The first category focuses on the necessary preparations needed to be done once. The second category analyses the required work to extend the functionality with new source model elements.

6.1.1 Preparations

Supporting the reference architecture to delegate functionality to other feature modules requires, on average, four new files, which must be done once. The four files are as follows. First, the class with the implementation logic that wants to delegate a part of the functionality is required. Afterward, the part of the functionality we want to delegate needs to be defined as a (Java-) interface. A Contribution class must also be defined to allow gluing in the module Manifests with the implementation and optionally a registry that allows categorizing the provided contributions (see Fig. 6). Those preparations need to be done only once. Then any number of feature modules can be supported for this specific use case (i.e., functionality).

For implementing the feature module, only two files are necessary. First, defining the class that implements the delegated (Java-) interface and provides part of the required functionality is required. Afterward, it can be contributed in the Manifest of the feature module to the required place using the previously defined Contribution class.

6.1.2 Extending the Source Model

We will analyze the required Lines of Code and files to extend the source model and use it in the modeling tool. We assume that the source element is already available (e.g., ECore, model, or class)

and that the source model provides a method to add the new element. Further, we assume that the necessary modules (i.e., respective Contributions) are already available and correctly used. Only providing the functionality to create and render the element is missing. For this reason, we will only analyze the new files created to support a new diagram element server-side.

Table 3 compares the UML elements *Enumeration* with *Interface* concerning the lines of code required. The UML elements *Enumeration* and *Interface* are similar in rendering and configuration. Yet, multiple files are required, but the LoC is small. In the FLoC, the headers and imports are also included; thus, the LoC that the developers are required to write is even smaller. The ELoC also includes lines where the class name or variable type has changed. Hence, for writing, the LoC would also be smaller. Each **ElementManifest* outlines necessary contributions such as node creation, choice of *GModelMapper* for graphical mapping, and property display in the property palette. **Configuration* details source model information for the GLSP framework. **OperationHandler* uses the command pattern to manage node creation in the source model, focusing exclusively on this process. **GModelMapper* translates the source model into a graphical format that GLSP can interpret. *G*Builder* requires the most lines of code as it dictates the graphical representation of elements. The most ELoC lies in the *GEnumerationBuilder* and *GInterfaceBuilder* as those classes need to describe how the source model should be rendered. This example illustrates that our architecture enables the efficient extension of a modeling tool with further metamodel elements by requiring only a few lines of additional code.

Table 3: Comparison LoC required between supporting *Enumerations* and *Interfaces* in the UML source model.

File	FLoC	ELoC
EnumerationElementManifest	43	10
EnumerationConfiguration	49	4-6
EnumerationOperationHandler	49	6
EnumerationGModelMapper	39	2
GEnumerationBuilder	62	18-20
InterfaceElementManifest	45	10-11
InterfaceConfiguration	50	4-7
InterfaceOperationHandler	48	6
InterfaceGModelMapper	38	2
GInterfaceBuilder	63	17-20

6.2 Lessons Learned

The BIGUML modeling tool has already gone through multiple iterations and architectural changes to accomplish the requirements better. Initially, the most significant problems were the Separation of Concerns, the Single Responsibility Principle, and the Source Model Representation Separation. They were not respected. This caused different unexpected behavior while using the modeling tool. Consequently, it was necessary to re-design the whole architecture. Clear architecture patterns and introducing coding guidelines made extending the modeling tool faster and easier and reduced unexpected behavior.

The lessons learned from using initial versions of BIGUML also in university Master courses and Master theses projects helped us further to improve the genericity and extensibility of the architecture. While the initial architecture was feature-wise working mostly stable, the feedback gained from the students—who mostly already have several years of industrial software engineering experience—and monitoring their progress showed the flaws with respect to clarity. This is why we abstracted and introduced the architectural concepts into our reference architecture. Using this new architecture clearly showed huge improvements in the effectiveness and quality of the student GLSP development projects.

Another aspect we learned is that the different technologies and the deployment also have side effects with respect to the runtime requirements for running the GLSP-based modeling tools. In cases where the GLSP-Server or the model server is realized with Java, a JRE dependency materializes to run the tool. This also applies to the VS Code-based integration of the tool. As one cannot expect a JRE in a specific version to be installed on the client, this imposes some minimal requirements on the runtime environment which should be taken into account in the preliminary phase of the development and operation process. With the release of the purely TypeScript-based GLSP-server, this issue, and the respective JRE runtime requirement, is already mitigated.

Eventually, it needs to be stated that GLSP is still under active development by the community. This is good and bad at the same time. When using earlier versions of GLSP, we faced several bugs and instability issues. The feedback from our students and other developers helped to increase the maturity and stability of GLSP. When developing a GLSP-based modeling tool one should always monitor the development of the base frameworks and make sure to develop the language-specific components in a way that base framework updates can be easily integrated. This is another reason why we developed our architectural concepts.

Model Server

Initially, in the development of our BIGUML modeling tool, we implemented the model server to separate concerns effectively. This decision aimed to improve flexibility and maintain a clear separation between the GLSP-Server and the source model management. The model server we used, was the EMF.cloud model server[36], facilitating the Eclipse Modeling Framework (EMF). By this arrangement, the model server was solely responsible for modifying the model files, while external services had only read access. The source models were based on the EMF implementation of the UML metamodel. This setup allowed the GLSP-Server to access the UML metamodel and the respective notation to focus only on managing the graphical elements, thereby controlling user interactions more effectively.

However, over time, we observed that maintaining a separate model server required substantial resources. The complexity and the resources involved in managing a dedicated model server became a significant burden. Additionally, this approach introduced multiple layers of abstraction, complicating the system architecture more than anticipated.

Given these challenges, we decided to reintegrate the model management functionalities back into the GLSP-Server. This shift aimed to streamline operations by reducing the overhead associated with a separate model server and eliminating unnecessary layers of abstraction. By consolidating the management of source models back into the GLSP-Server and implementing separation of concerns at the code level rather than at the application level, we have simplified the architecture. This code-level separation allows for more granular control and modular maintenance without sacrificing the system's effectiveness and flexibility, making the architecture hopefully more manageable and resource-efficient—in the case of BIGUML it did that.

6.3 Critical Reflection

Having multiple programming languages in the technology stack makes it also necessary to know about DevOps for those. Depending on the experience, that knowledge can vary. Thus, having the same programming language (e.g., TypeScript) for the client and server can help the development and deployment experience. Also, not all programming languages work efficiently with the Contribution and Manifests system. The system allows flexibility but introduces some overhead if used with Java, but not so much with TypeScript.

Consequently, the decisions on the technology stack need to carefully balance the experience of the development team. Moreover, experts in, e.g., Java could focus on the model server and the GLSP-Server while TypeScript experts could focus on, e.g., the GLSP-Client. Obviously, still, the technology does not only add flexibility and richness in creating modern web modeling tools with advanced user interaction and model representation functionality [8, 9] (a gallery of examples is provided online[37]) it also introduces challenges for the development team. This is in contrast to e.g., pure EMF-based modeling tool development where one can solely utilize Java.

From our point of view, the flexibility of the GLSP platform and the modern, feature-rich, cross-platform web modeling tools that one can develop with it clearly outperform the challenges discussed at the outset. Our experience is that modelers intuitively enjoy working with a GLSP-based editor, given its appealing look and feel and its UI responsiveness. The responsiveness of these new breeds of modeling tools is a huge improvement when compared to traditional modeling tools like those developed purely with

[36] https://github.com/eclipse-emfcloud/emfcloud-modelserver

[37] https://www.eclipse.org/glsp/gallery/

traditional, non-web-based, metamodeling platforms. We hope such web-based modeling tools, including those developed with GLSP, will help elevate modeling tools to the level users are used to working within other web applications.

6.4 Recommendations

The GLSP platform fundamentally changes the development of modeling tools by bringing them into the web. GLSP is powerful and flexible, but knowing the modeling tool's scope is crucial before deciding which technologies should be used. GLSP runs on the browser and browser-like applications (e.g., Electron) which constrains its use. Currently, it has no direct support for using it natively on a platform (e.g., Android, iOS, Windows). However, this constraint can be overcome easily as most platforms already provide web views or panels where the GLSP-Client can run, like in the case of the Eclipse IDE integration. We thus recommend really paying attention to the *Preliminary* phase of our development and operation process (see Fig. 2). Aside from the browser constraint, GLSP works for modeling tools of varying complexity and is also actively customized for industrial solutions[28]. Yet, depending on the tool's complexity, the architecture needs to be minded to scale efficiently. The overhead of using a model server benefits the architecture in the long term, but for modeling tools that will never use additional services, using it can cause more drawbacks.

Implementing our comprehensive architecture might be considered overblown for modeling tools that require only basic functionalities. In such cases, employing plain GLSP could be a more suitable option. Additionally, our solution is designed to support multiple diagram types, an aspect that may not be necessary for simpler applications. Moreover, the extracted framework still requires further iterations to fully mature. In the future, our focus will be on refining this framework to reduce the effort required for setting up new projects, aiming to streamline the development process and enhance usability for a broader range of applications.

Finally, we recommend GLSP for tool practitioners who want to prototype their first modeling tools. GLSP's streamlined setup and web-based architecture make it an ideal choice for rapidly developing and testing initial concepts without the complexities of a more elaborate framework. For those starting out in tool development, or for projects where simplicity and quick deployment are key, the GLSP platform provides the necessary functionality to get started with minimal overhead. This enables developers to focus on core features and usability without being bogged down by the more intricate aspects of advanced architectural setups. For more intricate solutions, planning the architecture is of vital importance, and using our initial generic framework for the GLSP platform aims to fill that gap.

6.5 Vision

GLSP, as a language server, is specifically designed to support the functionality of working with diagrams. It manages source models and the necessary interactions for diagrammatic representation, adhering to its goal of enabling diagram operations. This specialization makes GLSP highly effective and efficient within its scope, primarily focusing on diagrams.

In the evolution of our reference architecture, we initially abstracted the GLSP-Server to leverage its robust capabilities. Moving forward, our plan involves expanding this solution to incorporate the client side, aiming to establish a common foundation with GLSP as the base. However, our vision extends beyond supporting diagram editors. We aspire to develop a comprehensive reference architecture for a variety of modeling tools. While diagrams remain a crucial component addressed by GLSP, our reference architecture seeks to encompass a broader range of modeling functionalities, such as code generation, versioning, model validation, and hybrid (e.g., textual and graphical) editing capabilities. While GLSP does not inherently provide hybrid editing capabilities, its flexible design allows for such implementations. Developers are responsible for creating the UI and API and integrating them to manage edits across different model representations.

This shift marks the main distinction between GLSP and our vision. We want to cater to diverse modeling requirements, incorporating built-in solutions for various challenges. It is important to note that this transition towards a more generalized modeling framework is relatively recent, and

achieving a fully satisfactory solution will require time and continued development.

Imagining a potential future, it seems clear that GLSP will continue to serve as both the backbone and the heart of our reference architecture. Our aim is to incrementally develop a comprehensive framework that not only relies on GLSP as its foundation but also enriches the tool with more generic functionalities. The goal is to develop a versatile framework to be applied across various modeling tools spanning different business domains.

Until now, our main focus has been on generalizing the GLSP-Server part to extract common functionalities. Moving forward, we also plan to extend this generalization to the client components of GLSP. By developing more adaptable and universally applicable client-side elements, we aim to ensure that our framework can support a wide range of user interfaces and interaction models (cf. [5] for a taxonomy of advanced representation and interaction features in conceptual modeling).

Our vision extends to creating a framework equipped with a standard API that supports the seamless integration of additional services on the client and server sides. The standard API is intended to facilitate the development of independent components that can be easily installed or removed, such as allowing visual diffs to be shown to the user. This modularity will allow users to simply install new packages and utilize the isolated functionality without the need to modify existing code, functioning much like an extension system. By adopting this approach, we hope to provide a scalable and adaptable solution that meets the evolving needs of users and empowers innovation in modeling tool technology.

6.6 Limitations

This research also comes with limitations. The most important ones shall be discussed in relation to the taxonomy proposed in [54]. First and foremost, we need to limit the generalizability of our results. We have developed several modeling tools with GLSP but the number is still below 10. Moreover, we can report on the extensive experience of many student developers we closely supervised while working with GLSP in general and our reference architecture and implementation in particular. Still, we cannot generalize our lessons learned with constraints to other software developers and to other modeling languages.

Secondly, there can be a bias based on the selection of modeling tools (and their underlying modeling languages and supported features) we currently have implemented. We believe using the example of UML is a good choice because UML is well known and, at the same time, it is a very complex modeling language. However, of course, each modeling language comes with its particularities and specific requirements. GLSP, as well as our reference framework, require further instantiations in the future to mitigate that potential bias. With the publication of this paper and the public release of the reference architecture [28], we look forward to seeing a good adoption by the community and expect feedback that will help further improve the reference architecture.

A final threat to validity relates to the fact that the authors of this article were also involved in supervising the students, designing the reference architecture, and developing the reference implementation. We believe this threat is, to some extent, mitigated by our strong collaboration with EclipseSource, who played a very active role in that process and were able to validate the usefulness of our results. The fact that ideas and concepts developed in the course of the reference architecture are now integrated into the general GLSP repository underpins the knowledge exchange and the value of our research.

7 Conclusion

The development of modeling tools has a long tradition in modeling research. Still, it is considered a current issue [32] and acknowledged as a valuable scientific contribution [38]. The availability of web technologies and frameworks like the Graphical Language Server Platform (GLSP), which are built on them, enables new avenues for the modeling community to develop and deploy custom, flexible, and highly usable web-based modeling tools.

However, the development of such web-based modeling tools still poses significant challenges for developers. In this paper, we reported our experience in developing web modeling tools with GLSP. We propose a development and operation process, a set of architectural principles, and a reference architecture for GLSP-based web modeling tools.

As a proof of concept, we reported on our endeavors toward realizing a GLSP-based UML editor called BIGUML [31]. BIGUML is released as a VS Code extension[38]. We showed that GLSP is a powerful framework that provides a foundation developers can use to implement modern web modeling tools.

We believe this paper is of interest to all researchers and software engineers interested in the development of modern web modeling tools. Our critical reflection and lessons learned should help developers make an informed decision about whether or not to use GLSP. Moreover, the development and operation process, as well as the reference architecture, should facilitate knowledge transfer and enable others to benefit from our lessons learned during their tool development. To foster the reuse of our reference framework, we publicly release a reference implementation, which is available here: https://github.com/glsp-extensions/bigGLSP-framework [28].

In the future, we hope to see more successful GLSP tool developments to form a repository of GLSP tools. The community could clearly learn from each other and the technology stack of GLSP also allows a much easier integration of generic solutions that were provided by others.

Acknowledgments

Part of this research was funded through the FFG Innovationsscheck entitled 'Automatisiertes End-to-End-Testen von Cloud-basierten Modellierungswerkzeugen' (No. 903552). We further thank EclipseSource Vienna for the close collaboration regarding GLSP-based tool development in general and the development of the BIGUML tool in particular. Finally, we want to thank all students who contributed to the development of BIGUML and provided us with their feedback.

References

[1] Ali Qua, Kolovos DS, García-Domínguez A, et al (2024) Advancing domain-specific high-integrity model-based tools: Insights and future pathways. In: Egyed A, Wimmer M, Chechik M, et al (eds) Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, MODELS 2024, Linz, Austria, September 22-27, 2024. ACM, pp 104–113, https://doi.org/10.1145/3640310.3674094

[2] Almonte L, Guerra E, Cantador I, et al (2022) Building recommenders for modelling languages with droid. In: 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022. ACM, pp 155:1–155:4, https://doi.org/10.1145/3551349.3559521

[3] Bainczyk A, Busch D, Krumrey M, et al (2022) Cinco cloud: A holistic approach for web-based language-driven engineering. In: Margaria T, Steffen B (eds) Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part II, Lecture Notes in Computer Science, vol 13702. Springer, pp 407–425, https://doi.org/10.1007/978-3-031-19756-7_23

[4] Belafia R, Jeanjean P, Barais O, et al (2021) From monolithic to microservice architecture: The case of extensible and domain-specific ides. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS 2021 Companion, Fukuoka, Japan, October 10-15, 2021. IEEE, pp 454–463, https://doi.org/10.1109/MODELS-C53483.2021.00070

[5] Bork D, Carlo GD (2023) An extended taxonomy of advanced information visualization and interaction in conceptual modeling. Data Knowl Eng 147:102209. https://doi.org/10.1016/J.DATAK.2023.102209

[6] Bork D, Langer P (2023) Language server protocol: An introduction to the protocol, its use, and adoption for web modeling tools. Enterp Model Inf Syst Archit Int J Concept Model 18:9:1–16. https://doi.org/10.18417/EMISA.18.9

[7] Bork D, Langer P, Ortmayr T (2023) A vision for flexible glsp-based web modeling

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

tools. In: Almeida JPA, Kaczmarek-Heß M, Koschmider A, et al (eds) The Practice of Enterprise Modeling - 16th IFIP Working Conference, PoEM 2023, Vienna, Austria, November 28 - December 1, 2023, Proceedings, Lecture Notes in Business Information Processing, vol 497. Springer, pp 109–124, https://doi.org/10.1007/978-3-031-48583-1_7

[8] Carlo GD, Langer P, Bork D (2022) Advanced visualization and interaction in GLSP-based web modeling: realizing semantic zoom and off-screen elements. In: Wasowski A, Paige RF, Haugen Ø (eds) 25th International Conference on Model Driven Engineering Languages and Systems. ACM, pp 221–231, https://doi.org/10.1145/3550355.3552412

[9] Carlo GD, Langer P, Bork D (2022) Rethinking model representation - A taxonomy of advanced information visualization in conceptual modeling. In: 41st International Conference on Conceptual Modeling. Springer, pp 35–51, https://doi.org/10.1007/978-3-031-17995-2_3

[10] Eclipse Foundation (2024) Eclipse graphical language server platform. https://github.com/eclipse-glsp/glsp, accessed: 13.05.2024

[11] Eichelberger H, Eldogan Y, Schmid K (2009) A comprehensive survey of UML compliance in current modelling tools. In: Liggesmeyer P, Engels G, Münch J, et al (eds) Software Engineering 2009: Fachtagung des GI-Fachbereichs Softwaretechnik, LNI, vol P-143. GI, pp 39–50, URL https://dl.gi.de/20.500.12116/23336

[12] Fowler M (2008) Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html

[13] Frank U, Strecker S, Fettke P, et al (2014) The research field "modeling business information systems" - current challenges and elements of a future research agenda. Bus Inf Syst Eng 6(1):39–43. https://doi.org/10.1007/S12599-013-0301-5

[14] Glaser P, Bork D (2021) The biger tool - hybrid textual and graphical modeling of entity relationships in VS code. In: 25th International Enterprise Distributed Object Computing Workshop, EDOC Workshop 2021, Gold Coast, Australia, October 25-29, 2021. IEEE, pp 337–340, https://doi.org/10.1109/EDOCW52865.2021.00066

[15] Gulden J, Reijers HA (2015) Toward advanced visualization techniques for conceptual modeling. In: Grabis J, Sandkuhl K (eds) Proceedings of the CAiSE 2015 Forum, CEUR Workshop Proceedings, vol 1367. CEUR-WS.org, pp 33–40, URL https://ceur-ws.org/Vol-1367/paper-05.pdf

[16] Hegedüs M (2023) Real-time Collaborative Modeling with Eclipse GLSP. URL https://repositum.tuwien.at/handle/20.500.12708/192848, Master thesis at TU Wien

[17] Hölzl F, Barner S (2023) Implementing a model-based engineering tool as web application. CoRR abs/2302.14091. https://doi.org/10.48550/ARXIV.2302.14091, 2302.14091

[18] Jarke M, Gallersdörfer R, Jeusfeld MA, et al (1995) Conceptbase - A deductive object base for meta data management. J Intell Inf Syst 4(2):167–192. https://doi.org/10.1007/BF00961873

[19] Kasperowski M, Rentz N, Domrös S, et al (2024) KIELER: A text-first framework for automatic diagramming of complex systems. In: Lemanski J, Johansen MW, Manalo E, et al (eds) Diagrammatic Representation and Inference - 14th International Conference, Diagrams 2024. Springer, pp 402–418, https://doi.org/10.1007/978-3-031-71291-3_33

[20] Kelly S, Lyytinen K, Rossi M (1996) Metaedit+: A fully configurable multi-user and multi-tool CASE and CAME environment. In: Constantopoulos P, Mylopoulos J, Vassiliou Y (eds) Advances Information System Engineering, 8th International Conference, CAiSE'96, Heraklion, Crete, Greece, May 20-24, 1996, Proceedings, Lecture Notes in Computer Science, vol 1080. Springer, pp 1–21, https://doi.org/10.1007/3-540-61292-0

_1

[21] Lanusse A, Tanguy Y, Espinoza H, et al (2009) Papyrus UML: an open source toolset for MDA. In: 5th European Conference on Model-Driven Architecture Foundations and Applications, pp 1–4

[22] Louis-Edouard L, Syriani E (2024) Modeling with gentleman: a web-based projectional editor. Software and Systems Modeling https://doi.org/10.1007/s10270-024-01219-4

[23] Manders E, Biswas G, Mahadevan N, et al (2006) Component-oriented modeling of hybrid dynamic systems using the generic modeling environment. In: Machado RJ, Fernandes JM, Riebisch M, et al (eds) Proceedings of the Joint Meeting of The Fourth Workshop on Model-Based Development of Computrer-Based Systems and The Third International Workshop on Model-based Methodologies for Pervasive and Embedded Software, MBD/MOMPES 2006, Potsdam, Germany, March 30, 2006, Proceedings. IEEE Computer Society, pp 159–168, https://doi.org/10.1109/MBD-MOMPES.2006.6

[24] Maróti M, Kecskés T, Kereskényi R, et al (2014) Next generation (meta)modeling: Web- and cloud-based collaborative tool infrastructure. In: Balasubramanian D, Jacquet C, Gorp PV, et al (eds) Proceedings of the 8th Workshop on Multi-Paradigm Modeling co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, MPM@MODELS 2014, Valencia, Spain, September 30, 2014, CEUR Workshop Proceedings, vol 1237. CEUR-WS.org, pp 41–60, URL https://ceur-ws.org/Vol-1237/paper5.pdf

[25] Martínez-Lasaca F, Díez P, Guerra E, et al (2023) Dandelion: A scalable, cloud-based graphical language workbench for industrial low-code development. J Comput Lang 76:101217. https://doi.org/10.1016/J.COLA.2023.101217

[26] McLeod G, Cox G (2024) Gloss - a graphical language server on the smalltalk platform. https://www.inspired.org/s/IWST-GLSP-in-Smalltalk-Paper.pdf, last visited: 15.10.2024

[27] Metin H (2023) Testing of glsp-based web modeling tools. Master's thesis, Technische Universität Wien, https://doi.org/10.34726/hss.2023.106767

[28] Metin H (2024) glsp-extensions/bigGLSP-framework: SoSyM 24. https://doi.org/10.5281/zenodo.14316697

[29] Metin H, Bork D (2023) Introducing BIGUML: A flexible open-source glsp-based web modeling tool for UML. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023 Companion. IEEE, pp 40–44, https://doi.org/10.1109/MODELS-C59198.2023.00016

[30] Metin H, Bork D (2023) On developing and operating glsp-based web modeling tools: Lessons learned from BIGUML. In: 26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023. IEEE, pp 129–139, https://doi.org/10.1109/MODELS58315.2023.00031

[31] Metin H, Weiß J, Bork D (2024) borkdominik/bigUML: SoSyM 24. https://doi.org/10.5281/zenodo.14316667

[32] Michael J, Bork D, Wimmer M, et al (2024) Quo vadis modeling? Softw Syst Model 23(1):7–28. https://doi.org/10.1007/S10270-023-01128-Y

[33] Microsoft (2024) Language Server Protocol Implementations. https://microsoft.github.io/language-server-protocol/implementors/servers/, accessed: 13.04.2024

[34] Microsoft (2024) Language Server Protocol Specification. https://microsoft.github.io/language-server-protocol/specifications/specification-current/, accessed: 13.04.2024

[35] Naujokat S, Lybecait M, Kopetzki D, et al (2018) CINCO: a simplicity-driven approach to full generation of domain-specific graphical modeling tools. Int J Softw Tools Technol Transf 20(3):327–354. https://doi.org/10.1007/S10009-017-0453-6

[36] Ossher H, van der Hoek A, Storey MD, et al (2010) Flexible modeling tools (FlexiTools2010). In: Taylor RN, Gall HC, Medvidovic N (eds) 32nd ACM/IEEE Int. Conf. on Software Engineering - Volume 2, pp 441–442, https://doi.org/10.1145/1810295.1810419

[37] Ozkaya M (2019) Are the UML modelling tools powerful enough for practitioners? A literature review. IET Softw 13(5):338–354. https://doi.org/10.1049/iet-sen.2018.5409

[38] Paige RF, Cabot J (2024) What makes a good modeling research contribution? Software and Systems Modeling pp 1–5. https://doi.org/10.1007/s10270-024-01177-x

[39] Philip Langer (2024) Diagram editors with GLSP: Why flexibility is key. https://www.youtube.com/watch?v=mSTXgUZCBVE, accessed: 14.04.2024

[40] Popov G, Lu J, Vishnyakov V (2024) Toward extensible low-code development platforms. In: Shaikh A, Alghamdi A, Tan Q, et al (eds) Advances in Emerging Information and Communication Technology. Springer Nature Switzerland, pp 487–497, https://doi.org/10.1007/978-3-031-53237-5_29

[41] Pourali P, Atlee JM (2018) An empirical investigation to understand the difficulties and challenges of software modellers when using modelling tools. In: Wasowski A, Paige RF, Haugen Ø (eds) 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. ACM, pp 224–234, https://doi.org/10.1145/3239372.3239400

[42] Rocco JD, Ruscio DD, Salle AD, et al (2023) jjodel - A reflective cloud-based modeling framework. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023 Companion, Västerås, Sweden, October 1-6, 2023. IEEE, pp 55–59, https://doi.org/10.1109/MODELS-C59198.2023.00019, URL https://doi.org/10.1109/MODELS-C59198.2023.00019

[43] Rodríguez-Echeverría R, Izquierdo JLC, Wimmer M, et al (2018) An LSP infrastructure to build EMF language servers for web-deployable model editors. In: Hebig R, Berger T (eds) Proceedings of MODELS 2018 Workshops, CEUR Workshop Proceedings, vol 2245. CEUR-WS.org, pp 326–335, URL https://ceur-ws.org/Vol-2245/mdetools_paper_3.pdf

[44] Rodríguez-Echeverría R, Izquierdo JLC, Wimmer M, et al (2018) Towards a language server protocol infrastructure for graphical modeling. In: 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. ACM, pp 370–380, https://doi.org/10.1145/3239372.3239383

[45] Sarioglu A, Metin H, Bork D (2023) How inclusive is conceptual modeling? A systematic review of literature and tools for disability-aware conceptual modeling. In: Almeida JPA, Borbinha J, Guizzardi G, et al (eds) Conceptual Modeling - 42nd International Conference, ER 2023, Lisbon, Portugal, November 6-9, 2023, Proceedings, Lecture Notes in Computer Science, vol 14320. Springer, pp 65–83, https://doi.org/10.1007/978-3-031-47262-6_4

[46] Selic B (2002) The real-time UML standard: Definition and application. In: 2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), 4-8 March 2002, Paris, France. IEEE Computer Society, pp 770–772, https://doi.org/10.1109/DATE.2002.998385

[47] Steinberg D, Budinsky F, Merks E, et al (2008) EMF: eclipse modeling framework. Pearson Education

[48] Stone D, Jarrett C, Woodroffe M, et al (2005) User interface design and evaluation. Elsevier

[49] Syriani E, Vangheluwe H, Mannadiar R, et al (2013) Atompm: A web-based modeling environment. In: Liu Y, Zschaler S, Baudry B, et al (eds) 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), Miami, USA, September 29 - October 4, 2013, CEUR Workshop Proceedings, vol 1115. CEUR-WS.org, pp 21–25, URL https://ceur-ws.org/Vol-1115/demo4.pdf

[50] Vincenzo DD, Rocco JD, Ruscio DD, et al (2021) Enhancing syntax expressiveness in domain-specific modelling. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS 2021 Companion, Fukuoka, Japan, October 10-15, 2021. IEEE, pp 586–594, https://doi.org/10.1109/MODELS-C53483.2021.00089

[51] Walker M, Fischer M, Neubauer M, et al (2024) Towards a domain specific language for the development of distributed real-time systems. In: Bauernhansl T, Verl A, Liewald M, et al (eds) Production at the Leading Edge of Technology. Springer Nature Switzerland, pp 268–279, https://doi.org/10.1007/978-3-031-47394-4_27

[52] Walsh L, Dingel J, Jahed K (2022) A general architecture for client-agnostic hybrid model editors as a service. In: Kühn T, Sousa V (eds) Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS 2022. ACM, pp 749–754, https://doi.org/10.1145/3550356.3563131

[53] Wasowski A, Berger T (2023) Domain-Specific Languages - Effective Modeling, Automation, and Reuse. Springer, https://doi.org/10.1007/978-3-031-23669-3, URL https://doi.org/10.1007/978-3-031-23669-3

[54] Wohlin C, Runeson P, Höst M, et al (2024) Experimentation in Software Engineering, Second Edition. Springer, https://doi.org/10.1007/978-3-662-69306-3

[55] Yohannis AR, Kolovos DS, García-Domínguez A (2024) Exploring complex models with picto web. Sci Comput Program 232:103037. https://doi.org/10.1016/J.SCICO.2023.103037