

# **On Developing and Operating GLSP-based Web Modeling Tools: Lessons Learned from BIGUML**

Haydar Metin and Dominik Bork

To appear in:

*Proceedings of the 26th International Conference on Model Driven  
Engineering Languages and Systems, (MODELS 2023)*

© 2023 by IEEE.


Final version available soon:

[www.model-engineering.info](http://www.model-engineering.info)

# On Developing and Operating GLSP-based Web Modeling Tools: Lessons Learned from BIGUML


Haydar Metin

*Business Informatics Group, TU Wien, Vienna, Austria*

haydar.metin@tuwien.ac.at, 

Dominik Bork

*Business Informatics Group, TU Wien, Vienna, Austria*

dominik.bork@tuwien.ac.at, 

**Abstract**—The development of web-based modeling tools still poses significant challenges for developers. The Graphical Language Server Platform (GLSP) reduced some of these challenges by providing the necessary frameworks to efficiently create web modeling tools. However, more knowledge and experience are required regarding developing GLSP-based web modeling tools. This paper discusses the challenges and lessons learned after working with GLSP and realizing several GLSP-based modeling tools. More concretely, experiences, concepts, steps to be followed to develop and operate a GLSP-based web modeling tool, and the advantages and disadvantages of working with GLSP are discussed. As a proof of concept, we will 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. 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.

**Index Terms**—Modeling tool, GLSP, web modeling, lessons learned, LSP, eclipse

## I. INTRODUCTION

Modeling tools assist users in efficiently creating models of high quality by following standards like UML or ER. 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 [1]. Their functionalities are often built on older technology stacks, i.e., they are not compatible with state-of-the-art web technologies [2], [3]. Aside from a tool's functionalities, a well-designed, modern, and responsive graphical user interface is crucial for efficient and enjoyable use [4]. However, current tools are often labeled as not very useful [1], [5]. Tool development is therefore denoted as an essential part of enterprise information systems engineering and software engineering [6]–[8] research.

Historically, IDEs were developed as rich clients with built-in support for all the necessary language handling. Recently, the trend moved to separate the client from the language-specific parts using the Language Server Protocol (LSP). This change allowed small clients focusing on responsive and modern UIs to be hosted on the web being connected to a language server as the backbone which is doing the heavy lifting on the language smarts. As more editors utilize web technologies, we now see similar possibilities arising for modeling tools, i.e.,

*web modeling tools*. However, the development of web-based modeling tools still poses significant challenges for developers.

The Graphical Language Server Platform (GLSP) [9] aims to fill that gap by allowing users to develop modeling tools similar to other LSP-based editors [10], [11]. 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 a 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.

In the paper at hand, we bridge that gap by sharing our experience of realizing several web modeling tools with GLSP. 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. [2], [10], [11]), we believe, with our paper, we can make an original contribution that is of value for researchers and software engineers, who consider developing such a tool or migrating an existing standalone tool (e.g, EMF-based) to a web modeling tool.

In the remainder of this paper, Section II first introduces GLSP before we move to the development and deployment of GLSP-based web modeling tools in Section III. A reference architecture is presented in Section IV and its instantiation in the BIGUML proof-of-concept modeling tool is described in Section V. 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 VI and concluding remarks in Section VII.

## II. 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 [9]. The realized editors can be easily integrated into plain web applications but also into tool platforms such as Eclipse Theia and VS Code, and even in traditional Rich Client Application platforms like Eclipse

RCP. GLSP is an open-source project hosted at the Eclipse Foundation on GitHub<sup>1</sup>. GLSP is under active development by the community, with the next major release v2.0.0 expected for July 2023.

Generally, GLSP adopts the basic protocol structure and way of working as introduced by the Language Server Protocol (LSP) [12], [13]. GLSP extends LSP to account for the specific challenges coming 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 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, etc. In its current version, GLSP-based web modeling tools are built on the following types of essential components (cf. Fig. 1):

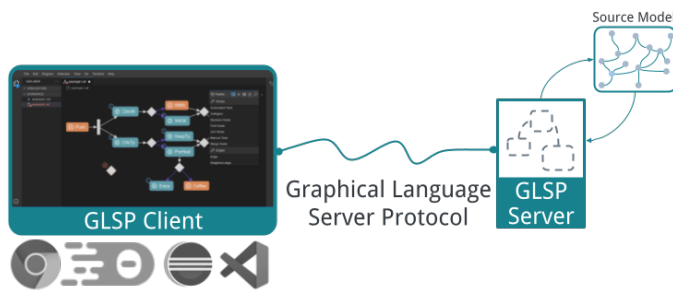


Fig. 1. GLSP Architecture

- **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 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 extends LSP to being able to handle graphical diagram 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 which contains 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 and user interaction and all the language smarts, 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 the LSP, enables high flexibility and interoperability. [14] 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 is to customize the user interaction and the look-and-feel using the client's API and the platform integrations offered by GLSP<sup>2</sup>.

Other tools also enable the construction of graphical editors by utilizing web technologies such as SiriusWeb<sup>3</sup>. SiriusWeb is a low-code platform to create and deploy diagram editors. On the other hand, GLSP focuses on providing the necessary means to develop the whole editor from scratch. The target audience and how the editor is built is different. Both enable the users to create powerful editors while following a separate path. The feasibility of the approach taken by GLSP is also visible in the implemented solutions industrially<sup>4</sup> and in the open-source community<sup>5</sup> having different business domains.

### III. DEVELOPING GLSP-BASED WEB MODELING TOOLS

GLSP provides an extensible client-server framework to develop web modeling tools<sup>6</sup>. This extensible framework offers the developers different implementation options. Currently, the following options for the implementation can be decided by the developers [14]:

- 1) **GLSP-Server.** It is possible to implement the server with Java or TypeScript with NodeJS.
- 2) **Source Model.** The way of saving 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 the 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 different combinations. Fortunately, GLSP offers flexible getting-started templates for quickly setting up the development environment for common combinations. Nevertheless, developers are not constrained to them but can create their solutions without using those 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.

In the following, we will focus on those open questions the developers need to answer and provide a GLSP development and operation process (illustrated in Fig. 2) to structure the

<sup>2</sup>The interested reader is referred to [14] for a comprehensive discussion of the flexibility enabled by GLSP-based web modeling tools.

<sup>3</sup><https://www.eclipse.org/sirius/sirius-web.html>, last visited: 05.07.2023

<sup>4</sup><https://blogs.eclipse.org/post/paul-buck/theia-adopter-story-logicloud-modern-engineering-platform-industrial-automation>, last visited: 05.07.2023

<sup>5</sup><https://github.com/imixs/open-bpmn>, last visited: 05.07.2023

<sup>6</sup><https://www.eclipse.org/glsp/documentation>, last visited: 13.04.2023

<sup>1</sup><https://github.com/eclipse-glsp>, last visited: 06.04.2023

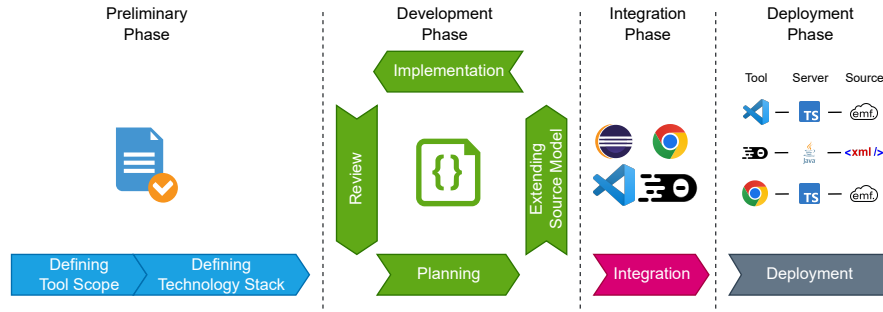


Fig. 2. Development and operation process for GLSP-based web modeling tools.

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 to 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.

#### A. 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.

1) *Defining Tool Scope*: Before deciding on the technology stack and starting to develop, knowing what goal the tool should fulfill is crucial. GLSP is language-agnostic and only provides the foundation to abstract the protocol and the interactions away. 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 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.

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 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 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.

#### B. Development Phase

This phase focuses on realizing 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. The feature is implemented from the source model to the GLSP-Client or the other way around. This approach allows the incremental implementation of new features by different developers without influencing other developers. However, the developers need experience and be aware of all the 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 small to medium-sized modeling tools. In contrast, architecture-oriented development better aligns with medium to larger-sized tools. The latter is significant if, besides GLSP, different other services are used in the architecture. Independently from the chosen approach, the following steps should be followed.

1) *Planning*: After declaring the tool 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 server parts could be affected together. 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.

2) *Extending 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) *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.

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.

### C. 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 require 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.

### D. 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 [14]. 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 [14], [15]:

- **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 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 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 are recommended.

## IV. REFERENCE ARCHITECTURE

GLSP 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 the developers the same power as the framework developers. Yet, if some patterns are ignored, it 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 Sprouty-based modeling tools as well as the extensive collaboration and knowledge exchange with EclipseSource<sup>7</sup> utilizes the following patterns:

- **Separation of Concerns (SoC).** GLSP consists of multiple components (cf. Fig. 1), like the client and server. The complexity rises further if additional services like a model server and ECore are introduced. For this reason, the architecture should be split into distinct sections to address particular concerns. Concerns can be general as *"the model server manages the source models"*, or as specific as *"model mappers create the graphical models"*. By following SoC, modularity can be reached. This approach makes the code simpler, maintainable, and easier to reuse and allows the module to be independently developed.
- **Single Source of Truth (SSoT).** The model should be only modifiable and readable from a single place. Otherwise, invalid modifications could make the source model erroneous, or the services could have outdated data. Therefore using a model server can improve the scalability and decouple the source models from GLSP, thereby allowing multiple GLSP clients to connect to the same server and perform model updates.
- **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.

#### A. Concepts

*Dependency injection* allows developers to modularize and loosely couple their code and thus improve the separation of concerns. It is a core concept used in every aspect of GLSP. Building upon the flexibility provided by dependency injection, we further differentiate between implementation features to further improve loosely coupled modules for: *core features*, *tool features*, and *diagram features*.

*Core features* work with the server directly (e.g., GLSP, model server). They provide the necessary server functionality and provide entry points to the other features. Nevertheless, core features have no language-specific information. The goal is to provide the necessary glue code between the server and the other features, load the other feature modules, and manage the server application. As already mentioned, the GLSP framework is still under active development. Those factors have been taken into account while designing the architecture. The core

<sup>7</sup>EclipseSource (<https://eclipse-source.com/>) is specialized in the development of GLSP-based modeling tools for industrial customers and one of the driving forces behind the further development of GLSP.

module enables the developers to react to introduced changes by the underlying framework in a centralized place.

*Tool features* provide new functionality by reusing core and diagram features. Those tool features provide distinct functionality not currently supported by the server (i.e., tool) framework and thus depict new functionality like a custom diagram outline, copy-paste, or auto-complete functionality.

*Diagram features* provide language-specific functionality and have access to the source model. They provide the necessary CRUD operations to the other features to interact with the source model. Other features are not allowed to modify or read the source model directly. Otherwise, the source model could be modified wrongly.

With this feature separation, we will now provide the concepts allowing the developers to have a scalable, extensible, and maintainable architecture.

1) *Contributions & Manifests*: The architecture needs to define clear structures between the modules based on the feature categorization to achieve clear separation. Every feature module isolates a specific functionality and needs to interact with other modules by means of *Contributions* and *Manifests*.

- **Contributions.** Core features and tool features provide interfaces called Contributions. Those Contributions are used in places in a feature to delegate the execution logic to other features. For example, delegating the GLSP-create operations to the diagram features or allowing new operations from tool features. The core feature uses Contributions to allow other features to extend or override default framework functionality. In contrast, tool features reuse those core contributions to either extend a specific core functionality or to realize something new. As tool features are language-agnostic, they can also provide contributions that the diagram features can implement to allow access to the source models from the tool features. Contributions introduce a system where different features are loosely coupled and allow communication only over well-defined interfaces. Technically, Contributions expose operations to allow other modules to register their implementation to specific injection points utilized by the module by following the dependency injection pattern.
- **Manifests.** Every tool and diagram feature has a Manifest. A Manifest defines all the contributions the feature module wants to make. Consequently, features can only provide functionalities to the application by using the previously defined Contributions. Thus, the Manifest is the glue code that fulfills the Contribution requirements by connecting it to an implementation. It is done by using the exposed operations in the Contribution to register the implementation to the dependency injection container.

Contributions and Manifests work together to loosely couple feature modules. Contributions define the requirements, and Manifests use those Contributions to fulfill those requirements. Consequently, modules that provide Contributions can be sure that their needs will be fulfilled if used. This approach allows the application to load separated, maintainable, extensible, modular functionality.

2) *Source Model Representation Separation*: The source models of diagrams can be similar, but different interactions can be possible depending on the currently active diagram representation. A node element could allow modifying it in a specific source model representation. In contrast, a different representation could not allow the same interaction. Consequently, diagram features must respect the currently active representation of the source model in the modeling tool.

The knowledge about the current representation needs to be available all the time to overcome this problem. The representation should be a unique identification to be able to differentiate. Moreover, this approach allows the core module to load only the current representation's supported feature modules with this information.

3) *Model Server*: It is possible to allow GLSP to manage the source models; however, allowing that would make the GLSP-Server carry multiple concerns. For small modeling tools, this drawback can be neglected. Still, with increasing complexity and the requirement to connect the source models with other services, this decision can negatively impact the whole architecture. Therefore, using a model server improves flexibility. In this case, the source models are managed outside the GLSP-Server in a model server. Outsourcing helps to have a single source of truth, and modifying the source models would be the sole responsibility of the model server to prevent invalid states. Additionally, connecting other services aside GLSP to the model server would be possible to allow a different view of the source model aside from a graphical (e.g., form-based or text-based) one to support blended modeling [16], [17] with GLSP.

4) *Flow Similarity*: In most cases, apart from necessary validation, the only difference in creating two distinct elements in the source model by a modeling tool is the modification done to the source models. The elements are semantically different, but until modifying the source model, the flow is similar. Simply, the user needs to trigger the create operation, which needs to be handled by the GLSP-Server; finally, either the source models are modified by the GLSP-Server or a model server. A generic functionality can be provided for similar flows, where only the modifications to the source models differentiate. Deciding how to implement generic approaches in the modeling tool is up to developers. Regardless, introducing such approaches helps the developers to maintain more readable, maintainable, and extensible code and allows easier fixes, as the interactions follow a similar pattern.

## B. Architecture

In Fig. 3, we present the reference architecture composed of the previously mentioned concepts. The architecture applies to the GLSP-Server and the model server. We have a single-core module. The core module aims to manage the whole application and the underlying framework. It makes the framework and core functionality accessible by defining Contributions for the other modules. Moreover, it also loads the other feature modules by using their Manifest definitions

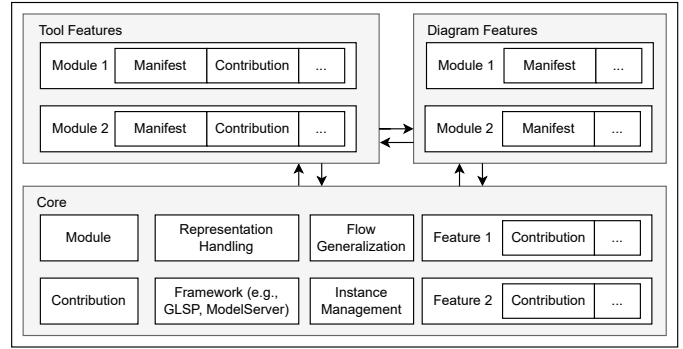


Fig. 3. Reference Architecture

for the application. The core module also needs to handle the different representations to load the correct diagram module.

The tool features have, for every feature, their own module. The modules are isolated from each other, and no direct communication is allowed. All communication between features needs to be done through well-defined interfaces and Contributions. The core module needs to provide the necessary Contributions so that the feature module can provide new custom functionality. If the framework functionality is not accessible because a Contribution from the core module is missing, then the core module needs to provide a workaround (i.e., by defining a new Contribution). Outside of the core module, no module is allowed to use the framework directly. Moreover, tool modules can make use of language-specific functionalities by defining new Contributions which need to be implemented by the diagram modules.

Diagram features provide the necessary language-specific implementation. They have the knowledge required to work with the underlying source model. They cannot define new Contributions because circular dependencies could arise. Diagram modules use their Manifest and the Contributions from the other features to provide for those features language-specific functionality. Those can be, for example, mapping the source model into a structure required or updating the source model as requested from a feature.

This approach decouples the modules and enforces the use of well-defined interfaces if a module wants to modify functionality from another module, which is done by using the Contribution in the Manifest. Further, this approach allows easier testing and maintaining of the code, as the communication only happens through the interfaces, and the module can be internally changed without influencing the outside.

## V. PROOF-OF-CONCEPT: THE BIGUML TOOL

By following the development and operation process described in Section III (cf. Fig. 2) and the reference architecture in Section IV (cf. Fig. 3) a GLSP-based UML modeling tool called BIGUML was developed<sup>9</sup>. Software engineers, architects, business users, and more utilize UML regularly. The UML specification consists of multiple diagrams which increases the challenges of developing good tool support. A further challenge arises from the possibility of visually ren-

dering the same source model element differently in different UML diagram representations. The constraints regarding e.g., the allowed relationships between nodes also vary depending on the diagram representation.

Given the noted specific challenges, and the community experience in how difficult it is to provide rich modeling support for UML [18]–[20], we believe the UML case is an excellent candidate to thoroughly test the strengths and weaknesses of GLSP 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 our realization of BIGUML in detail.

### A. Realization

We followed the feature-driven development approach to implement the features. In every iteration, one feature was implemented through the architecture from the model server to the clients bottom-up. 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. Moreover, Eclipse Theia and VS Code integrations were developed to test different tool platforms for BIGUML.

As UML consists of multiple diagram types and representations, it was essential to have a scalable, maintainable, and extensible architecture. Consequently, we used the introduced concepts and instantiated the reference architecture. An exemplary overview of how the BIGUML architecture is applied to the reference architecture can be seen in Fig. 4. In the following, we will share more architectural and implementation aspects of BIGUML.

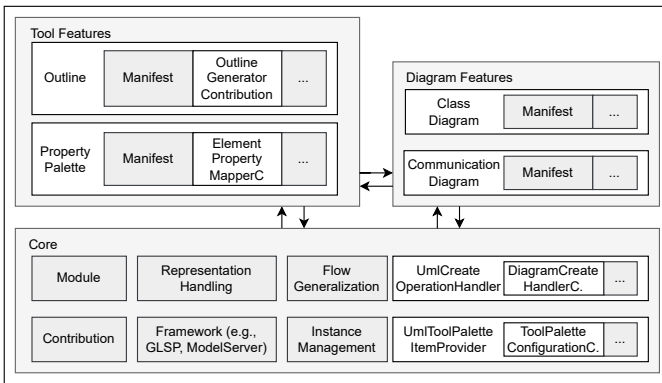


Fig. 4. BIGUML architecture

1) *Contributions & Manifests:* The core module provides 11 distinct Contributions used by diagram modules and three general Contributions usable by all modules realized as Java interfaces. The diagram-specific Contributions from the core allow the diagram modules to handle or override the GLSP-specific functionality, like CRUD operations or overriding the tool palette. Three tool feature modules exist: *Outline*, *Editor Panel*, and *Property Palette*. Only the Outline and Property Palette features provide Contributions that diagram modules can use. Those Contributions allow the diagram modules to update the behavior of those features or provide the necessary

language-specific logic. An excerpt of the contributions can be seen in Table I. Due to this clear separation between the modules, the feature modules can be developed in isolation or in parallel. This approach allowed completely updating features without modifying other modules as the interfaces stayed the same.

2) *Source Model Representation Separation:* Together with the source models, the representation of the diagram is saved to the file system. The model server provides the representation to all read requests to differentiate between the different diagram types. With that information, the GLSP-Server loads the correct language-specific diagram module to interpret the source model. Afterward, the interpreted source model and the active representation are provided to the GLSP-Client to render it. The GLSP-Server can then redirect any successive requests based on the representation to the correct diagram module to handle it for the specific diagram.

With the separation between the source model and the representation, the BIGUML modeling tool can provide different diagram types with similar source models that do not influence each other. This approach allows having different diagram types with different rules and visual representations.

3) *Model Server:* We are using the EMF.cloud model server<sup>8</sup> that allows using the Eclipse Modeling Framework (EMF) in the cloud. With this approach, managing the source model in a central place and independently from GLSP is possible. By doing this, only the model server can modify the model files, and the outside has only read access. For the source models, we are using the EMF-based implementation of the UML metamodel for the Eclipse platform, and for the graphical elements, we are using a customized version of the ECore notation models provided by the used model server. The notation model has information like the bounds of the UML element in the diagram. Moreover, GLSP can access the UML metamodel and the notation for semantic information regarding the graphical element. This knowledge allows GLSP to allow or restrict specific user actions or to execute different actions depending on the case.

4) *Flow Similarity:* While developing BIGUML we noticed multiple similar flows. For example, all CRUD operations can be generalized. To accomplish this, we needed to generalize two parts. First, the GLSP framework handlers had to be overridden to respect the different diagram modules and representations. Afterward, we provided abstract base classes for every CRUD handler in the diagram modules. Those base classes hid the complexity required from the architecture and allowed the language-specific handler to focus only on the necessary semantic modifications. As a result, adding new, for example, create handlers for different elements in the UML model, just required reusing the base class and providing the semantic modification as the base class would handle the required other parts. Aside from the CRUD operations, the same approach of using base classes has been applied to the mapping from the semantic model to the graphical model and

<sup>8</sup><https://github.com/eclipse-emfcloud/emfcloud-modelserver>



TABLE I  
CONTRIBUTIONS AVAILABLE IN BIGUML AND WHERE IT IS USED

Contribution	Provided by	Used by	Description
ActionHandlerContribution	Core	Tool	Used to override GLSP actions handlers
DiagramCreateHandlerContribution	Core	Diagram	Allows diagram modules to provide language-specific create handlers
DiagramLabelEditMapperContribution	Core	Diagram	Allows diagram modules to edit the label of an element
DiagramLabelEditValidatorContribution	Core	Diagram	Used to validate label edit operations
GModelMapperContribution	Core	Diagram	Allows diagram modules to map the semantic element to a GModel
OverrideOperationHandlerContribution	Core	Diagram / Tool	Used to override GLSP operation handlers
OperationHandlerContribution	Core	Tool	Allows tool modules to provide new operation handlers
OutlineGeneratorContribution	Outline Tool	Diagram	Allows diagram modules to customize the generated outline

other parts. Those base classes allowed us to only focus on the necessities without worrying about how the architecture or framework handles them in the background.

By using representation separation, contributions, and base classes, we were able to generalize the necessary GLSP actions, operations, and default implementations to allow language-specific handlers without the architecture overhead. Table II lists an excerpt of the common implementations in GLSP with a similar flow that can be generalized and the necessary complexity hidden using the approach.

TABLE II  
GLSP IMPLEMENTATIONS THAT CAN BE GENERALIZED

GLSP Implementation	Description
CreateOperationHandler	Handles create operations
DeleteOperationHandler	Handles delete operations
ApplyLabelEditOperationHandler	Handles label edit operations
ToolPaletteItemProvider	Provides tool palette items
DiagramConfiguration	Provides diagram configuration

### B. Application

BIGUML is inspired by Eclipse Papyrus [20]. However, it aims to use the newest technologies to provide good usability and experience. Further, it aims to be accessible anytime offline and online. The initial release v0.2.1, already fully supports the UML Class diagram, while further UML diagram types will be added in the coming weeks and months. Internally, BIGUML already supports other UML diagrams besides the class diagram; however, those diagrams are currently not accessible as they are not finalized. The current version already supports a Property Palette and CRUD operations for all elements of Class diagrams.

We released BIGUML openly as an extension to the VS Code marketplace<sup>9</sup> and online by Eclipse Theia. This combination allows the users to decide how they want to use it. However, by bringing UML support to the EMF.cloud model server, other developers can implement their solutions based on the work already done. The reference architecture can also

be taken as a blueprint that developers can easily extend to support other language-specific features for other modeling languages by only changing or extending the diagram modules.

## VI. DISCUSSION

In the following, we discuss our experience of 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.

### A. 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

<sup>9</sup><https://marketplace.visualstudio.com/items?itemName=BIGModelingTools.umlDiagram>

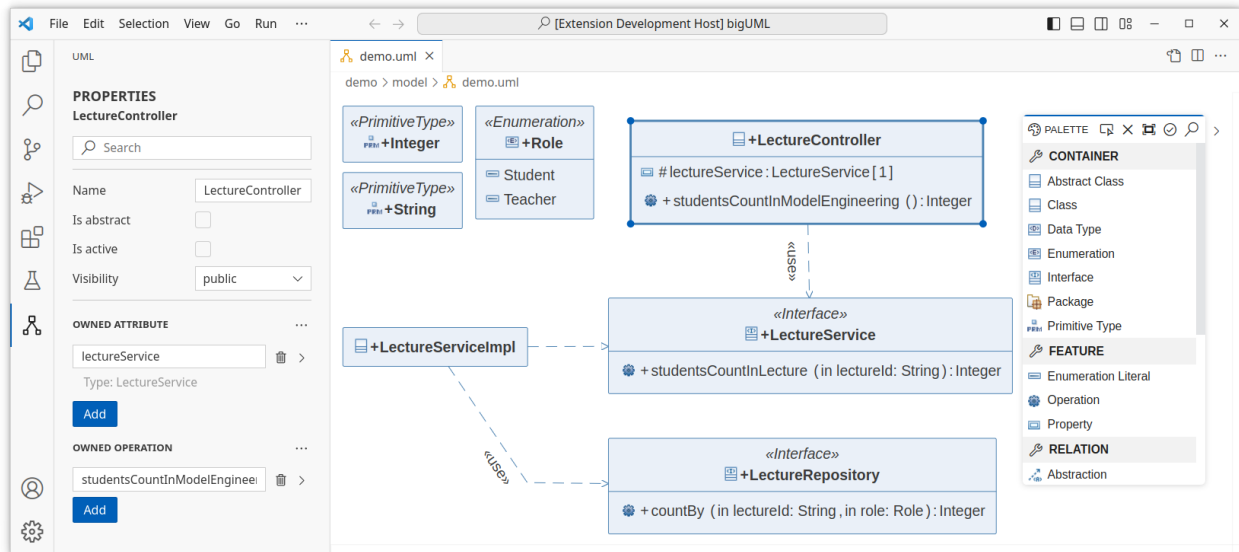


Fig. 5. BIGUML VS Code Extension (available via: [shorturl.at/sAX23](https://shorturl.at/sAX23))

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 frontend 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 frontend 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 outweighs the initial effort required.

### B. Lessons Learned

The BIGUML modeling tool has already gone through multiple iterations and architecture changes to accomplish the requirements better. Initially, the most significant problems were the Separation of Concerns, the Single Responsibility Principle, and Source Model Representation Separation. They were not respected and the implementation took multiple responsibilities. This caused different unexpected behavior while using the modeling tool. Consequently, it was necessary to re-design the whole architecture to have a well-defined architecture. Clear architecture 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 on advanced model engineering 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 have already several years of industrial software engineering experience—and monitoring their progress showed use the flaws with respect to clarity. This is why we abstracted and introduced the architectural concepts of Separations of Concerns, the Single Responsibility Principle, and Source Model Representation Separation 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 very 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 dramatically. 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.

### C. 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.

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 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 [2], [3] (a gallery of examples is provided online<sup>10</sup>) 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 GLSP and the modern, feature-rich, cross-platform web modeling tools that one can develop with it clearly outperforms the challenges discussed at the outset. Our experience is that as soon as modelers start working with a GLSP-based editor they do not want to return to full-fledged stand-alone modeling editors.

The responsiveness of these new breeds of modeling tools is outstanding and will hopefully help elevate modeling tools to the level users are used to working with in other web applications.

### D. Recommendations

GLSP fundamentally changes the development of modeling tools by bringing them into the cloud. 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 put attention to the *Preliminary* phase of our development and operation process (see Fig. 2). Aside from the browser constraint, GLSP works for small and large modeling tools. Yet, depending on the size, 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 small modeling tools, which will never use additional services, using it can cause more drawbacks.

## VII. CONCLUSION

The development of web-based modeling tools still poses significant challenges for developers. In this paper, we reported our experience in developing web modeling tools with the Graphical Language Server Platform (GLSP). Moreover, 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. BIGUML is released as a VS Code extension<sup>11</sup> and currently supports the Class diagram of UML. We show, that GLSP is a powerful framework and provides a foundation that 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 making 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.

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 one another and the technology stack of GLSP also allows a much easier integration of generic solutions that were provided by others. From a research perspective, we aim to investigate migration strategies, automated if possible, to enable the vast amount of community-driven EMF-based modeling tools to evolve into GLSP-based tools.

<sup>10</sup><https://www.eclipse.org/gls/gallery/>

<sup>11</sup><https://marketplace.visualstudio.com/items?itemName=BIGModelingTools.umlDiagram>

## ACKNOWLEDGMENTS

Part of this research was funded through the FFG Innovationscheck 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] J. Gulden and H. A. Reijers, "Toward advanced visualization techniques for conceptual modeling," in *Proceedings of the CAiSE 2015 Forum*, ser. CEUR Workshop Proceedings, J. Grabis and K. Sandkuhl, Eds., vol. 1367. CEUR-WS.org, 2015, pp. 33–40.
- [2] G. D. Carlo, P. Langer, and D. Bork, "Advanced visualization and interaction in GLSP-based web modeling: realizing semantic zoom and off-screen elements," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, 2022*, E. Syriani, H. A. Sahraroui, N. Bencomo, and M. Wimmer, Eds. ACM, 2022, pp. 221–231.
- [3] —, "Rethinking model representation - A taxonomy of advanced information visualization in conceptual modeling," in *Conceptual Modeling - 41st International Conference, ER 2022, Hyderabad, India, 2022, Proceedings*, ser. Lecture Notes in Computer Science, J. Ralyté, S. Chakravarthy, M. K. Mohania, M. A. Jeusfeld, and K. Karlapalem, Eds., vol. 13607. Springer, 2022, pp. 35–51.
- [4] D. Stone, C. Jarrett, M. Woodroffe, and S. Minocha, *User interface design and evaluation*. Elsevier, 2005.
- [5] P. Pourali and J. M. Atlee, "An empirical investigation to understand the difficulties and challenges of software modellers when using modelling tools," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018*, A. Wasowski, R. F. Paige, and Ø. Haugen, Eds. ACM, 2018, pp. 224–234.
- [6] H. Ossher, A. van der Hoek, M. D. Storey, J. Grundy, and R. K. E. Bellamy, "Flexible modeling tools (FlexiTools2010)," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 441–442.
- [7] K. Sandkuhl, H.-G. Fill, S. Hoppenbrouwers, J. Krogstie, F. Matthes, A. Opdahl, G. Schwabe, Ö. Uludag, and R. Winter, "From Expert Discipline to Common Practice: A Vision and Research Agenda for Extending the Reach of Enterprise Modeling," *Bus. Inf. Syst. Eng.*, vol. 60, no. 1, pp. 69–80, 2018.
- [8] U. Frank, S. Strecker, P. Fettke, J. Vom Brocke, J. Becker, and E. Sinz, "The research field modeling business information systems," *Bus. Inf. Syst. Eng.*, vol. 6, no. 1, pp. 39–43, 2014.
- [9] Eclipse Foundation, "Eclipse graphical language server platform," <https://github.com/eclipse-glspl/glspl>, accessed: 13.04.2023.
- [10] R. Rodríguez-Echeverría, J. L. C. Izquierdo, M. Wimmer, and J. Cabot, "Towards a language server protocol infrastructure for graphical modeling," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*, A. Wasowski, R. F. Paige, and Ø. Haugen, Eds. ACM, 2018, pp. 370–380.
- [11] —, "An LSP infrastructure to build EMF language servers for web-deployable model editors," in *Proceedings of MODELS 2018 Workshops*, ser. CEUR Workshop Proceedings, R. Hebig and T. Berger, Eds., vol. 2245. CEUR-WS.org, 2018, pp. 326–335.
- [12] "Microsoft language server protocol specification," <https://microsoft.github.io/language-server-protocol/specifications/specification-current/>, accessed: 13.04.2023.
- [13] "Microsoft language server protocol implementations," <https://microsoft.github.io/language-server-protocol/implementors/servers/>, accessed: 13.04.2023.
- [14] D. Bork, P. Langer, and T. Ortmayr, "A vision for flexible GLSP-based web modeling tools," *CoRR*, 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2307.01352>
- [15] Philip Langer, "Diagram editors with GLSP: Why flexibility is key," <https://www.youtube.com/watch?v=mSTXgUZCBVE>, accessed: 14.04.2023.
- [16] I. David, M. Latifaj, J. Pietron, W. Zhang, F. Ciccozzi, I. Malavolta, A. Raschke, J. Steghöfer, and R. Hebig, "Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study," *Softw. Syst. Model.*, vol. 22, no. 1, pp. 415–447, 2023.
- [17] P. Glaser and D. Bork, "The bigger 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, 2021, pp. 337–340.
- [18] M. Ozkaya, "Are the UML modelling tools powerful enough for practitioners? A literature review," *IET Softw.*, vol. 13, no. 5, pp. 338–354, 2019.
- [19] H. Eichelberger, Y. Eldogan, and K. Schmid, "A comprehensive survey of UML compliance in current modelling tools," in *Software Engineering 2009: Fachtagung des GI-Fachbereichs Softwaretechnik 02.-06.03. 2009 in Kaiserslautern*, ser. LNI, P. Liggesmeyer, G. Engels, J. Münch, J. Dörr, and N. Riegel, Eds., vol. P-143. GI, 2009, pp. 39–50. [Online]. Available: <https://dl.gi.de/20.500.12116/23336>
- [20] A. Lanusse, Y. Tanguy, H. Espinoza, C. Mraïdha, S. Gerard, P. Tessier, R. Schnekenburger, H. Dubois, and F. Terrier, "Papyrus UML: an open source toolset for MDA," in *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*. Citeseer, 2009, pp. 1–4.