TU WIEN Informatics

Typescript-based bigUML Model Management

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Karol Grilling, BSc.
Matrikelnummer 11922631

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assoc. Prof. Dr. Dominik Bork
Mitwirkung: Dipl.-Ing. Haydar Metin

Wien, 2. Dezember 2025

Karol Grilling	Dominik Bork

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

TU WIEN Informatics

Typescript-based bigUML Model Management

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Karol Grilling, BSc.
Registration Number 11922631

to the Faculty of Informatics

at the TU Wien

Advisor: Assoc. Prof. Dr. Dominik Bork
Assistance: Dipl.-Ing. Haydar Metin

Vienna, December 2, 2025

Karol Grilling	Dominik Bork

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

Erklärung zur Verfassung der Arbeit

Karol Grilling, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 2. Dezember 2025

Karol Grilling

Acknowledgements

First I would like to thank my advisor, Assoc. Prof. Dr. Dominik Bork, for guidance and support during the development of this work. His quick and helpful feedback was key to finishing this thesis.

Furthermore, I would like to express my thankfulness to Dipl.-Ing. Haydar Metin, who was very helpful and patient throughout the development phase of my thesis. His advice always helped move my work forward.

I would like to thank my family, especially my parents, RNDr. Silvia Grillingová and RNDr. Karol Grilling, who have played a very important role in my life and also during my studies. Their help and support were very important to me and I am very grateful for it.

Lastly I need to extend my appreciation to my friends, namely to Ing. Adrián Rojek, whose advice and expertise were always priceless; to Miroslav Mihalčin for his motivation, support and never-ending trust in my abilities; to MA Peter Husár, who was always there for me and pushed me forward; and to Gabriel Ráchela, whose unconventional thoughts and unexpected insights somehow always arrived exactly when I needed them most, and helped me finally finish this work.

Kurzfassung

Modellgetriebene Entwicklung kombiniert üblicherweise textuelle Sprachen und grafische Notationen, um Softwaresysteme zu beschreiben. Derzeit findet eine Verlagerung hin zu webbasierten Tools und Frameworks wie Langium und GLSP statt. Diese ermöglichen eine einfache textuelle und grafische Modellierung im Browser. Trotzdem ist die Modellverwaltung auf der Serverseite oft nicht gut strukturiert, was die Wartbarkeit und Erweiterbarkeit beeinträchtigt. Diese Arbeit entwirft und evaluiert eine generische und wiederverwendbare Architektur für die Modellverwaltung für Tools, die GLSP und Langium verwenden und in der Modellierungsumgebung BIGUML realisiert sind. Sie verbessert die CRUD-Operationen sowie die Eigenschaftenpalette, die Gliederungsansicht und die Validierung auf einheitliche Weise. Die Lösung verwendet eine einzige Definitionsdatei, die Elemente, Containments und viele Metadaten beschreibt. Aus dieser Definition erstellen Generatoren Zuordnungen und Typdeskriptoren, die in abstrakten CRUD-Handlern und -Diensten verwendet werden, anstatt doppelte und nicht konfigurierbare Handler pro Element. Die Architektur ist in den BIGUML-Modellserver und die VS Code-Erweiterung integriert und wird für UML-Klassen- und Paketdiagramme mit nur kleinen Konfigurationsklassen instanziiert. Diese Arbeit wird anhand von Funktionstests der Generatoren und Handler, einem quantitativen Vergleich mit einer früheren BIGUML-Version und Szenarien bewertet. Dieser Ansatz reduziert Boilerplate-Code und Handler-Duplikate, die Code-Teile bleiben dank der einzigen Quelle der Wahrheit synchronisiert und das Hinzufügen neuer Diagrammtypen wird dank des konfigurierbaren Musters für die Modellverwaltung in webbasierten Modellierungstools vereinfacht.

Abstract

Model driven engineering commonly combines textual languages and graphical notations to describe software systems. A shift toward web-based tools and frameworks like Langium and GLSP is happening. These enable simple textual and graphical modeling in the browser. In spite of this, model management on the server side is often not well-structured, what hurts maintainability and extensibility. This thesis designs and evaluates a generic and reusable architecture for model management for tools using GLSP and Langium realized in the BIGUML modeling environment. It improves the CRUD operations as well as property palette, outline view, and validation in unified way. The solution uses a single definition file, which describes elements, containments, and many metadata. From this definition, generators create mappings and type descriptors used in abstract CRUD handlers and services, instead of duplicated and not configurable per element handlers. The architecture is integrated into the BIGUML model server and VS Code extension and instantiated for UML Class and Package Diagrams with only small configuration classes. This work is evaluated using functional tests of the generators and handlers, a quantitative comparison with an earlier BIGUML version, and scenarios. This approach reduces boilerplate and handler duplication, the code parts stay synchronized thanks to the single source of truth file and the addition of new diagram types becomes simple all thanks to the configurable pattern for model management in web-based modeling tools.

Contents

CHAPTER 1

Introduction

This chapter introduces the problem explored by this thesis, highlighting its significance and practical implications for software developers. It outlines the primary research questions that will guide the investigation and briefly explains the methodological strategy adopted to address these questions. Additionally, the chapter provides a concise summary of the subsequent sections, offering the reader a clear roadmap of the thesis structure.

1.1 Problem & Motivation Statement

During recent years Model-driven engineering (MDE) has transitioned from traditional integrated development environment (IDE)-based tools to web-based modeling environments [HB23]. Web-based modeling environments gained popularity because of their platform-independent features and improved collaboration capabilities and lower user adoption barriers and natural scalability of web technologies. The Eclipse Modeling Framework (EMF) [Fou25a] together with Java [Ora25] serves as the base for traditional IDE-based modeling tools which create performance and compatibility problems. The limitations of these tools make them no longer appropriate for the current development environments that use distributed agile approaches.

Web-based modeling solutions have gained rapid adoption through the development of the Language Server Protocol (LSP) [Mic25] and its graphical extension known as the Graphical Language Server Platform (GLSP) [Fou25b]. Through LSP web-based clients execute syntax highlighting and code completion and reference resolution functionalities which were previously managed by IDEs. GLSP enhances the graphical capabilities of these tools to handle complex editing operations and visualization needs which occur during graphical modeling particularly with UML diagrams.

Langium together with GLSP use browser-native lightweight technologies to create flexible maintainable solutions with modularity. The TypeScript framework Langium delivers

strong language engineering capabilities which follow the LSP standards. GLSP enables developers to build scalable modular client-server systems that deliver graphical modeling editor capabilities. The current state of graphical modeling tools still faces important issues regarding the management of complex UML diagrams through scalable and flexible and reusable architectural systems.

The traditional Java-based framework EMF fails to meet the requirements of contemporary web-based modeling applications because it adds excessive weight to system operations. Frameworks like Langium enable developers to create powerful textual domain-specific languages (DSLs) through semantic modeling and cross-referencing capabilities. The current development landscape lacks complete generic solutions which unify Langium's features with graphical modeling clients through GLSP thus creating separate development environments with restricted model management component reusability. Modern modeling environments that implement continuous integration and rapid evolution need tools which provide fast extension capabilities through minimal manual changes. This need aligns with the current research trends, which emphasize extensible, lightweight, and automated modeling platforms. These requirements are also highlighted in recent discussions about the future direction of modeling tools [MBWM24]. The requirement for BIGUML becomes urgent because it needs consistent and extensible UML diagram operation management to maintain high productivity levels and ensure quality assurance.

The web-based modeling tool BIGUML [MB23a] faces major difficulties when performing Create, Read, Update and Delete (CRUD) operations. The current system implements each operation type and model element through specialized handler classes which create dependencies. This approach creates major code duplication, which complicates maintenance work while making the system less scalable. The process of adding new model elements together with operation types forces developers to write extensive manual code which creates obstacles for quick system evolution and flexibility. The management of model complexity and usability in web-based tools needs additional functionalities such as property palettes together with outline views and validation mechanisms for effective operation. User productivity receives substantial improvement through model interaction features and efficient navigation and robust integrity checks, which these components provide [BLO23]. The present manual logic implementation in BIGUML leads to redundant and error-prone and inflexible structures that need extensive manual coding. The current method fails to adopt code generation techniques and declarative configurations which results in substantial maintainability and scalability difficulties.

The industry requires an immediate solution which handles fundamental operations and additional features through generic reusable configuration-based approaches. The solution should decrease developer workload through declarative operation handler definitions which reduce development costs. The proposed unified architecture built with TypeScript, Langium, GLSP, code generation and declarative configurations aims to simplify development processes while minimizing redundancy and improving maintainability to meet modern web-based model engineering needs.

1.2 Aim of the Thesis & Expected Results

The primary aim of this thesis is to develop a generic and reusable architectural pattern for operation handlers in model management, cutting down the necessity for extensive custom implementations. The general aim is the modernization of BIGUML's current Java-based architecture by transitioning to a fully TypeScript-based, browser-compatible solution. The transition will require building a powerful configuration-based system which controls operation handlers automatically so new operation types can be added without much manual coding. The goal is not only to reduce current architectural shortcomings - such as high code redundancy, reduced scalability, and difficulty in maintenance - but also to provide a flexible and extensible architecture that adheres to contemporary web-based modeling best practices.

Moreover, the thesis expands its scope beyond CRUD operations, addressing additional essential components like property palette generators, outline views, and comprehensive validation mechanisms. By incorporating these auxiliary features into the configuration-driven architecture, the solution will improve tool's and developer's productivity, simplify interaction with complex model structures, and improve overall model integrity through automated, declarative rules.

To achieve these objectives, this thesis will develop a two-stage prototype. Initially, a concrete implementation will be developed specifically addressing BIGUML's immediate UML diagram challenges - such as CRUD operations, property palette generation, and validation. In the second stage, the framework will be extended into a reusable and extensible framework compatible with GLSP to enable application across different modeling tools and languages. The specific implementation for BIGUML will directly benefit its users and developers by significantly enhancing scalability, maintainability, and user experience. The generalized architecture will provide a reusable pattern for developers who create or extend web-based graphical editors or domain-specific graphical representations.

Through detailed evaluations, the robustness, efficiency, and scalability improvements of the proposed configuration-driven architecture will be compared against the existing version of the tool, created by Jäger [Jä24]. This should clearly quantify enhancements in developer productivity, maintainability, and system scalability. According to the goals of this work, the following research questions will be answered in the course of this thesis:

- **(RQ1)** How can a generic and reusable architecture be designed for operation handlers in model management to minimize the need for custom implementations?
  - The answer will identify the core elements required to construct a configuration-driven framework that supports the dynamic introduction of new operation types, thus obviating the necessity for redundant handler classes and significantly enhancing flexibility and reusability.

- **(RQ2)** What is the most effective way to generalize CRUD operations for UML diagrams and auxiliary features such as property palettes, outline views, and validation mechanisms while ensuring reusability and extensibility across different diagram types?
  - The solution will present a single unified system for CRUD operations and additional features which enables their expansion across different diagram types through standardized implementation.
- **(RQ3)** How can a TypeScript-based framework enable efficient configuration and execution of new operation handlers and additional features while decreasing development effort for future extensions?
  - The answer will demonstrate the effective use of TypeScript features, including decorators, generics, and static typing, in combination with a declarative, configuration-driven approach to centralize operation management. This strategy will ensure minimal coding effort, streamlined handler integration, and simplified maintenance for developers, ultimately enabling rapid and robust system evolution.

1.3 Methodology

1.3.1 Requirements Analysis

The first step includes gathering the requirements for a generic and reusable operation handler architecture, as well as the model management type definition that is required. For the operation handler framework, requirements will be identified based on the analysis of the current BIGUML tool and other similar tools. Also, the configuration-driven approach will be assessed by identifying the issues encountered while trying to incorporate new operation types in the current system. For the type definition, the assessment will involve determining how CRUD operations, property palette handling, outline generation, validation, and other essential functionalities can be defined generically enough to ensure scalability, reusability, and ease of extension.

1.3.2 Conceptualization

In this step, a concept for the generic operation handler framework will be developed in a way that is configuration-based and there would be no code duplication and the framework should be easily extendable (RQ1). The framework will explicitly include mechanisms to generalize CRUD operations and additional key features such as property palettes, outline views, and validation rules. A reusable, declarative solution will be designed for these operations to enable streamlined and scalable implementations (RQ2). Additionally, the use of TypeScript - with its static typing, generics and decorators - as the foundation will be explored to ensure the configuration and execution of handlers are efficient and maintainable (RQ3). It is emphasized, however, that the configuration-driven design,

beyond just TypeScript features, is what chiefly enables simpler addition of operations and reduces developer effort.

1.3.3 Build Artifacts

During this stage, effective and customized practical improvements designed specifically for BIGUML will be created, dealing with immediate concerns such as implementing and supporting UML diagram operations (CRUD functions update, property palette generation, outline view tiered management, and integrated validation.) This explicit execution is designed to eliminate BIGUML tool's usability and scalability concerns. After this is done, this solution will be implemented in the systems and will enable analysis to be done on the usability along with the abstracting of the implementation. Focus of this won't just stay on the abstracting of the implementation but also on the configuration-based extensibility targeted to extend the reusability for the modeling tools and languages. It will also be examined whether the customized handlers for the property palette will be created and through enabling the dedicated code generators for the property palette and the validation templates. Finally, the implemented solution will be examined for practical use cases within the context of model management and measure robustness for targeted solutions with specific use case analysis for class diagrams.

1.3.4 Evaluation

The evaluation of the code that is written within the work of this thesis is following:

- **Functional Testing:** Detailed unit tests will be developed in order to ensure the correctness and reliability of the framework with focus on the CRUD functionalities, property palette configurations, outline generators, and validation mechanisms without the need of changing the code

- **Descriptive Evaluation - Informed Argument:** The improved solution will be compared to the Jäger's TypeScript-based model server [Jä24], focusing on metrics such as effective lines of code (ELOC), number of files required (e.g., for creating operation handlers), architectural patterns employed, and overall complexity. The comparison of these will help with evaluation of the improvements in reusability, extensibility, and development effort.

- **Descriptive Evaluation – Scenarios:** The framework's effectiveness will be evaluated through its ability to support the development of specific tools and use cases including CRUD operations and property palettes and outline view management and class diagram validation rules. The system will be considered successful if it enables new operation types with minimal reconfiguration needs.

1.4 Structure

This chapter 1 gave a quick introduction to this thesis with describing its problem and motivation statement, aim, expected results and the methodology to achieve this.

In Chapter 2, the necessary background is provided. It introduces central concepts of model engineering and explains the core ideas and architectures of GLSP and Langium, followed by an overview of generator technology in software and language engineering which will later be used.

Chapter 3 discusses the state of the art. It first reviews traditional approaches and then turns to web-based modeling. Finally, it positions this thesis in relation to other works.

Chapter 4 presents the concept for the proposed solution.

Chapter 5 describes the implementation of this concept in detail using the presented concepts.

Chapter 6 contains the evaluation of the developed framework. It reports on the functional testing, descriptive comparison and evaluates scenario use cases. The chapter concludes with an interpretation of the evaluation results with respect to the research questions.

Finally, Chapter 7 concludes the thesis by summarizing the main contributions and findings and outlining directions for future work for web based modeling tools.

CHAPTER 2

Background

This chapter explains the terminology used within model engineering and why it is important in software engineering. Next, the core concepts of GLSP [Fou25b] are introduced and explained, with its use case in BIGUML [Too25]. After that, we look into Langium [Typ25], which is a very useful language development framework, based on Xtext [Ecl25]. Finally, we discuss generators and how they are used in software and language engineering.

2.1 Model Engineering

The field of software development known as model engineering views models as primary artifacts that are used throughout the engineering process. In simple terms, a model is an abstraction of a system that brings out the most important aspects and leaves out the irrelevant details, with the intention of a specific task or purpose. As Brambilla et al. note, “a model [is] a simplified or partial representation of reality, defined in order to accomplish a task or to reach an agreement on a topic” [BCW17]. The definition says, that a model, which is a representation of a real world system cannot express all of its details. Bézivin and Gerbé (2001) similarly define a model as “a simplification of a system built with an intended goal,” such that the model can answer questions in place of the actual system [BG01]. There is also a simple formulation by Ed Seidewitz: "A model is a set of statements about some system under study" [Sei03]. The key idea from all of these definitions is, that a model is an abstraction. It represents the parts of a system, which are relevant for some specific purpose as for example understanding, analysis, communication. There are two possibilities, how models can be used - descriptively, to describe a system that already exists - water pipeline system in a city, showing pipes connecting reservoirs, pumps and households - or prescriptively, to specify how some new system could be built - a blueprint. Significantly, models enable engineers to operate at a higher level of abstraction, which helps with complexity management; as Selic observes,

building complex systems in traditional engineering "involves constructing a variety of specialized models" to understand the problem and potential solutions [Sel03]. This is also the case in software engineering, where modeling greatly improves understanding and explains complex software ideas [Sel03].

2.1.1 Models and Their Key Characteristics

Not all models are created equal. Useful models are known to fulfill certain characteristics. Bran Selic named five qualities, which are essential for engineering models [Sel03]:

- **Abstraction:** The model is a simplified version of the real system because it only shows the important parts and leaves out the ones that aren't needed.
- **Accuracy:** The model faithfully and correctly represents those aspects of the system it purports to model. In other words, the important properties of the real system are captured with accuracy.
- **Cost-Effectiveness:** Benefits should make the work to make and use the model worthwhile; it should be much more cost-, time-, or effort-effective than making or testing the real system directly.
- **Predictiveness:** By studying the model, one can deduce facts about the real system before the actual system is built or changed.
- **Understandability:** The model must be presented in a form that is understandable to a broader audience (e.g. graphical representation).

These criteria, proposed by Selic [Sel03], guarantee that a model is not only a simplified representation, but that it is useful for its engineering purposes. A model which meets these characteristics can serve as a powerful tool - a "well-formed, adequate, and dependable instrument" in the words of Thalheim [Tha22] - in activities such as design, analysis, and decision-making.

2.1.2 Syntax and Semantics

To create and use models in a systematic way, we use a modeling language. The vocabulary and rules, which describe the models are defined by the modeling language. It consists of abstract syntax, one or more concrete syntaxes and a definition of semantics [Kle08]. Each of these has a specific role:

- **Abstract Syntax:** The abstract syntax of a modeling language describes the underlying structure of its models, independent of how they are notated. This is the formal representation of the language. In computer science terms, the abstract syntax is often the metamodel of the language - essentially a model of the modeling concepts themselves. This structural definition is typically realized as an abstract

syntax tree (AST) or an equivalent graph of model elements in a repository [BKP20]. The abstract syntax is hidden from the end-user - modelers usually interact with a model via its concrete syntax, while the abstract syntax is maintained behind the scenes. In summary, the abstract syntax (or metamodel) defines the types of model elements and how they can be composed [Kle08, Sei03].

- **Concrete Syntax:** The concrete syntax of a modeling language is the notation used by humans to create and view models. This can be textual (similar to programming language code or structured text) or graphical (diagrammatic symbols and connections), or even a combination of both. This is how we can understand or edit the model, usually in an editor or some modeling tool. E.g. a UML class model might have a graphical concrete syntax - boxes and arrow in a diagram - while a JSON schema model has a textual concrete syntax - JSON document. One language can have multiple concrete syntaxes, but the most important thing is, that it should be presentable in human-comprehensible way.

- **Semantics:** The semantics of a modeling language define the meaning of models – that is, how to interpret the elements and structures defined by the syntax. While syntax (abstract and concrete) tells us how a model is formed, semantics tells us what the model means. For example in the UML Class Diagram language: its semantics include rules like "if class A has a generalization relationship to class B, then A is a subclass (specialization) of B". In other words, a "specialization" arrow in the diagram implies an inheritance relationship in the system's implementation or domain logic [Obj17]. Semantics can be defined either informally (documentation) or formally (mathematical models, code) [BF14].

In summary, a modeling language provides a formalism for modeling: the abstract syntax is the schema or metamodel that all models in the language conform to; the concrete syntax is what modelers manipulate; and the semantics attaches real-world or computational meaning to the models.

General purpose vs. Domain-Specific Languages

Modeling languages can be broadly categorized into two types: General-Purpose Modeling Languages (GPL) and Domain-Specific Modeling Languages (DSL). This classification relates to the scope and intended use of the language:

- **General-Purpose Modeling Languages (GPL):** Languages that are not tied to any single problem domain, in sense that they are designed to be applicable to a lot of different kinds of systems and industries. An example for this is Unified Modeling Language (UML), which is an universal visual modeling language for software systems [Obj17]. It can be used to model anything from banking systems to spacecraft software, as it provides a wide range of generally useful concepts as classes, states, activities etc.

- **Domain-Specific Modeling Languages (DSL):** On the other hand, a DSL is tailored to a particular domain or problem space and offers concepts and notations, which are closely aligned to its abstractions. As an example, HTML (Hypertext Markup Language) is a domain-specific language for web page structure and content [Wor25] and Simulink is a DSL for control system and signal processing models [Wik25c]. Because domain concepts are built into DSLs from the start, domain experts can model problems more naturally and with less unneccessary complexity than they would need with GPL. DSLs usually produce more understandable and compact models for a specific domain, but at the price of being useless outside of it.

Furthermore, DSLs can be divided into internal and external DSLs.

- **Internal DSL:** Language built on top of an existing GPL - syntax and facilities of a host language are used to create domain-specific constructs. For example a fluent API in Python [vRF25], also called embedded DSL.
- **External DSL:** In contrast, this language is standalone and created from scratch for a specific purpose, using its very own syntax and semantics. Creating of a new external DSL gives the most flexibility to create the notation, that is most appropriate for the domain, whether that be a new textual syntax or a graphical notation. However it requires implementing new parsers, editors and translators for that language.

Martin Fowler notes, that external DSLs can be more adjusted for specific needs for the price of more work, whereas internal DSLs are limited by the host language's ecosystem [Fow10].

2.1.3 Metamodeling and Meta-Layers

While discussing models and modeling languages, the term metamodeling is very important part of the discussion. Metamodeling is the practice of defining the language of models - it deals with the models of models. A metamodel is a model, which defines the structure and rules for the other models. For example, the UML metamodel defines the elements like Class, Attribute, Association and their possible connections - defining what makes a legitimate UML model [Obj17]. A metamodel is effectively the abstract syntax or grammar of a modeling language or as Kleppe defines - "a model used to specify a language" [Kle08]. Seidewitz elaborates that a metamodel defines the structure and constraints of a set of models - i.e., it describes a family of possible models, all of which conform to that metamodel's definitions [Sei03]. Bézivin et al. similarly note that a metamodel is an abstraction of an existing model (in the real world domain) and provides a way to describe a whole class of such models [Bé05]. In other words, if you have a model that represents some system, the metamodel captures what all such models look like in general.

The Object Management Group (OMG) standardized the 4-layer metamodeling architecture [Obj19]. It defines a hierarchy of models and metamodels, usually labeled from M3 to M0, where each level serves as the metamodel for the level below it. From top to bottom, each layer's elements are instances of the layer above and each model conforms to the metamodel at the layer above. The layers can be described in this way:

- **M0-layer:** It contains the actual data or instances that exist in the real world or in the software. Meaning for software, it could be the live object instances or a database of records and for non-software system physical entities. We often say the M1 model represents a set of M0 entities. M0 instance would be a particular Customer record.
- **M1-layer:** It contains models of a real world systems and describes some system or scenario in the real world. M1 class model might describe that "Customer" is a class with certain attributes.
- **M2-layer:** It contains metamodels - models of languages. The UML metamodel, BPMN's metamodel or Ecore (the metamodel used in EMF [Wik25a, Wik25b]) live in this layer. So the M2 metamodel defines the structure of M1 models. So in this Layer, Customer is recognized as a Class in UML, which says a class has attributes and operations.
- **M3-layer:** It contains the meta-metamodel - the language used to define metamodels. It provides the foundational concepts like "Class", "Attribute" or "Association". The Meta-Object-Facility (MOF), which defines how metamodels are structured, lives in this layer. For example, MOF's defines MetaClass, which says a meta-class can define classes like UML's Class.

This four layer architecture is a useful conceptual framework for understanding how modeling languages and models actually relate [Obj19, Obj17]. Each layer is an instance of the layer above. The *conforms-to* relationship is often used interchangeably with *instance-of* in this context. This hierarchy ends at M3, because the meta-metamodel (like MOF) is defined in terms of itself, enabling a closed loop.

In the figure 2.6 we can see a specific example of all four layers, with a specific situation with a video in the M0-layer, then a class named Video in the layer M1, which models the the real world video. In the M2-layer we can see metamodel of the Video class, with Attribute, Class and Instance and the layer M3 contains the meta-metamodel with only Class.

Metamodeling is important not only for defining languages but also for tool support. When new metamodel is defined, we can generate or build tools like editors, code generators or analyzers that work for any model which conforms to that metamodel. This is a key principle of model-driven engineering (MDE): by raising the specification of a problem to the model (M1) level and formally defining its metamodel (M2), we can

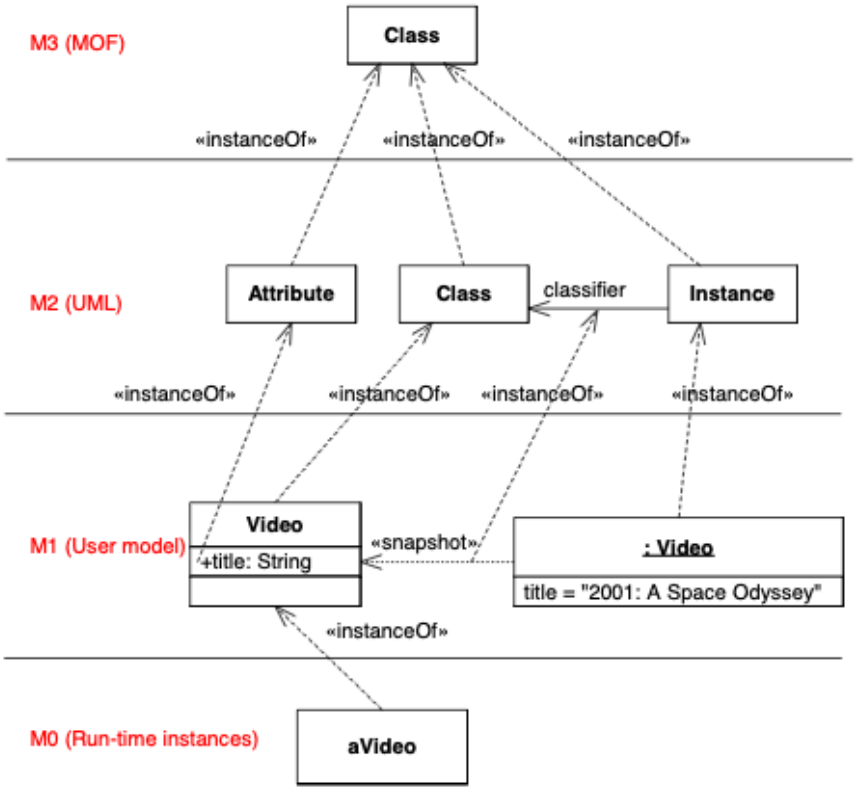

Figure 2.1: An example of the four-layer metamodel hierarchy [Obj11]

leverage automation (at M2 and M3) to manipulate those models. Bézivin described MDE as an approach that "uses formal models, i.e., models which are machine-readable and processable, to produce executable software systems semi-automatically" [Bé05]. In other words, models become first-class citizens in the development process, not just documentation. They can be transformed into other models or code, checked for consistency, and kept in sync with implementations.

2.1.4 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) functions as a widely adopted framework which enables model-driven engineering principles to be applied in practical applications. Ecore, which is the main component of EMF, is a modeling language that implements the OMG's MOF at the M3 level. Developers can also define metamodels (M2) in Ecore for languages such as UML or custom domain-specific notations, and from these EMF can automatically generate Java code, which is then used for creation, persistance and manipulation of models (M1). EMF also provides runtime support for persistence, validation, change notification, and can be integrated with other Eclipse technologies,

which makes it the base for many modeling environments, editors, and code generators in the Eclipse ecosystem [SBPM08, Wik25b]. In the figure 2.2 we can see an example of a core part of Ecore, which focuses on the metamodel specification.

EMF therefore implements the four-layer metamodeling architecture by turning abstract concepts into an executable infrastructure, and for more than a decade it has been the standard for building desktop-based modeling tools. Frameworks such as GMF or Sirius depend on EMF to provide their functionality as rich diagramming or model manipulation. However, because EMF is deeply connected to Java and the Eclipse IDE, it has traditionally been linked with heavy, desktop environments. This limitation is increasingly relevant as the field makes a shift toward more lightweight, web-based solutions such as LSP and GLSP [BLO23], which aim to bring the benefits of modeling to modern and faster browser-based platforms while separating from Eclipse's Java-based runtime [con25].

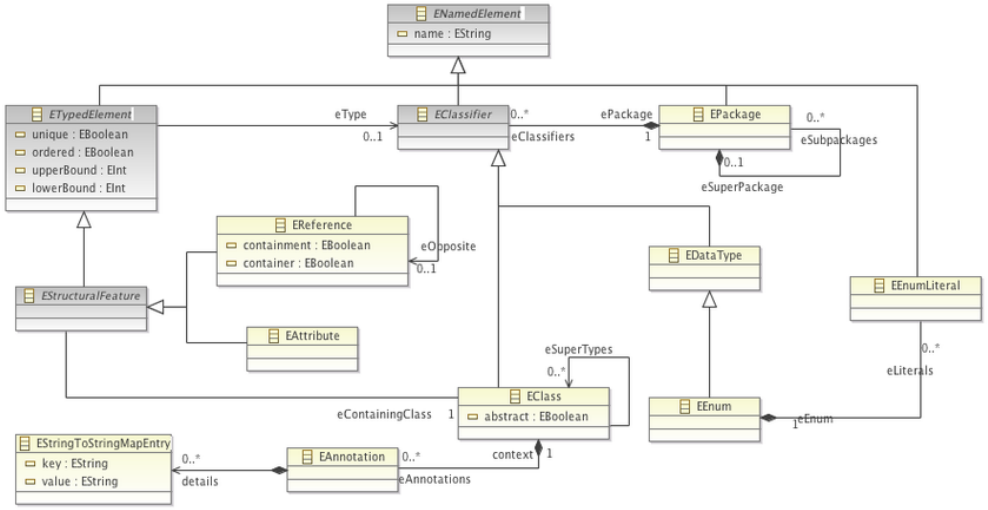

Figure 2.2: Class diagram of a core part of Ecore focusing on metamodel specification and oriented to model decomposition [MKG13].

2.1.5 Importance of Modeling in Software Engineering

Model engineering principles have become increasingly important in modern software engineering - while it started as a paradigm, which should help developers build better software faster, with software completely eating the world, the relevance of it keeps increasing [CGJ+22]. Modeling offers a way to cope with the complexity of today's software systems by giving high-level abstractions that developers and stakeholders can reason about. By using models, teams can visualize architecture, explore design alternatives, detect inconsistencies early, and communicate effectively about system structure and behavior. As Selic notes, in classical engineering disciplines no one would build a complex product (bridge, airplane, etc.) without first constructing models or

blueprints [Sel03]; similarly, in software, modeling brings clarity and attention to detail that source code alone (being very detailed and low-level) cannot easily provide. Models can serve as an engineering blueprint, but also, in the context of MDE, as an executable specification. They allow developers to perform analyses (e.g., performance simulation, model checking) before committing to code, and they enable partial automation like code generation or model-to-model transformations, which can greatly reduce manual coding effort. Moreover, models are often easier to modify than code, supporting agility and adaptability.

The significance of modeling is evident in the emergence of standards and frameworks dedicated to model engineering. The OMG's UML and MOF standards, for example, have provided a foundation for a wide range of modeling tools and methodologies [Obj19, Obj17]. There are also many domain-specific modeling languages and tools in use (from MATLAB/Simulink in engineering to domain-specific schemas in enterprise architecture [Mat24, Wik25c]), each providing notations closer to the problem domain than general code. When properly integrated into the development process, models improve communication (by providing a common visual/textual language for discussing systems), improve quality (through early analysis and design validation), and enable automation (through generation of other artifacts). Empirical studies of model-based software engineering report improvements in software quality, communication, and automation of development artifacts [GPHL+19].

Finally, it is worth noting that as software development tools evolve, there is a trend toward more lightweight, accessible modeling environments. In recent years, development has shifted from heavy IDEs to web-based tools and cloud services. To support modeling in such environments, the community has developed protocols like the Language Server Protocol (LSP) and its graphical extension GLSP, which decouple editing front-ends from language processing back-ends. Recent research has taken this idea further by demonstrating how model management itself can be realized entirely in TypeScript. Jaeger et al. (2025) introduced a TypeScript-only model management framework that enables next-generation web-based modeling tools without relying on Java-based EMF servers. Their work provided the first comprehensive architecture and evaluation of a browser-native model management layer integrated with Langium and GLSP [JLF+25]. While their approach established the feasibility of full client-side model management, this thesis extends that foundation by generalizing the concept into a modular, maintainable framework for TypeScript-based modeling environments. It focuses particularly on improving abstraction layers, reusability, and developer experience within the BIGUML architecture.

However, until recently, sophisticated model management (especially for graphical models) still often required heavy server-side components (e.g., a Java server running EMF for handling the models). This thesis is positioned at the intersection of model engineering and modern web technology: it explores how models and metamodels can be managed in a lightweight, browser-based tool, using a TypeScript-based modeling stack entirely on the front end. The goal, in line with the principles discussed above, is to make

modeling tools more accessible and integration-friendly by moving model management to a client-side web context. By using a TypeScript language framework (e.g., Langium) to implement the modeling language and its metamodel, and by providing a model server API in the browser, we leverage model engineering concepts to enable rich modeling capabilities without a traditional Java/EMF server.

2.2 GLSP

The Graphical Language Server Platform (GLSP) is a client–server framework for building web-based diagram editors, inspired by the success of the Language Server Protocol (LSP) in decoupling language-specific logic from tooling UIs [Mic25]. Like LSP, GLSP follows a split architecture with a thin client and a "smart" server. However, GLSP extends the LSP concept to address the unique challenges of graphical modeling, such as managing two-dimensional diagram layouts and enforcing domain-specific rules. There are no such challenges in text editing [Fou25b]. This separation allows the difficult diagram logic as loading models or applying rules to stay on the server while the platform-independent diagram clients stay lightweight. GLSP-based editors can thus be embedded in various frontends (VS Code, Eclipse Theia, web applications, etc.) while reusing the same backend logic [Fou25b].

Figure 2.3: GLSP: client-server architecture [Fou25b]

Core Concepts and Architecture

The main task of GLSP is to introduce a clear separation between the source model - the underlying data or domain model - and the diagram model, which is also called the graphical model – the visualisation of the data. The responsibility of the GLSP server is to load or access the source model (e.g. a UML model stored in JSON or an EMF model) and to transform it into a corresponding graphical model for the client [Fou25b]. The graphical model is essentially a serialized description of diagram elements (nodes, edges, labels, etc.) that the client can render. The GLSP client is built on a generic diagram framework, typically Eclipse Sprotty [Fou25c]. It receives this model and is in charge of

rendering the diagram in the browser or editor UI. The client is language-agnostic and it simply interprets the graphical model. The client also provides interactive UI tools such as palette, context menus based on what the server decides as acceptable in the diagram. The client can request from the server the allowed editing operations on the model (e.g. which elements can be created or connected at a given context) after the initial diagram is rendered – thus enabling dynamic toolbars and context actions[Fou25b].

When a user adds a node or connects two elements to a diagram or basically does any edit, the GLSP client does not modify the diagram locally. Instead, it sends the operation message to the GLSP server using the GLSP protocol over JSON-RPC, which describes the user's action [Fou25b]. The server then updates the source model accordingly to the changes and regenerates the graphical model to show the change, and sends an update back to the client. When the client receives the updated graphical model, it re-renders the diagram to show the result of the operation. This makes sure that the source model on the server acts as the single source of truth and keeps the client's graphical view in sync.

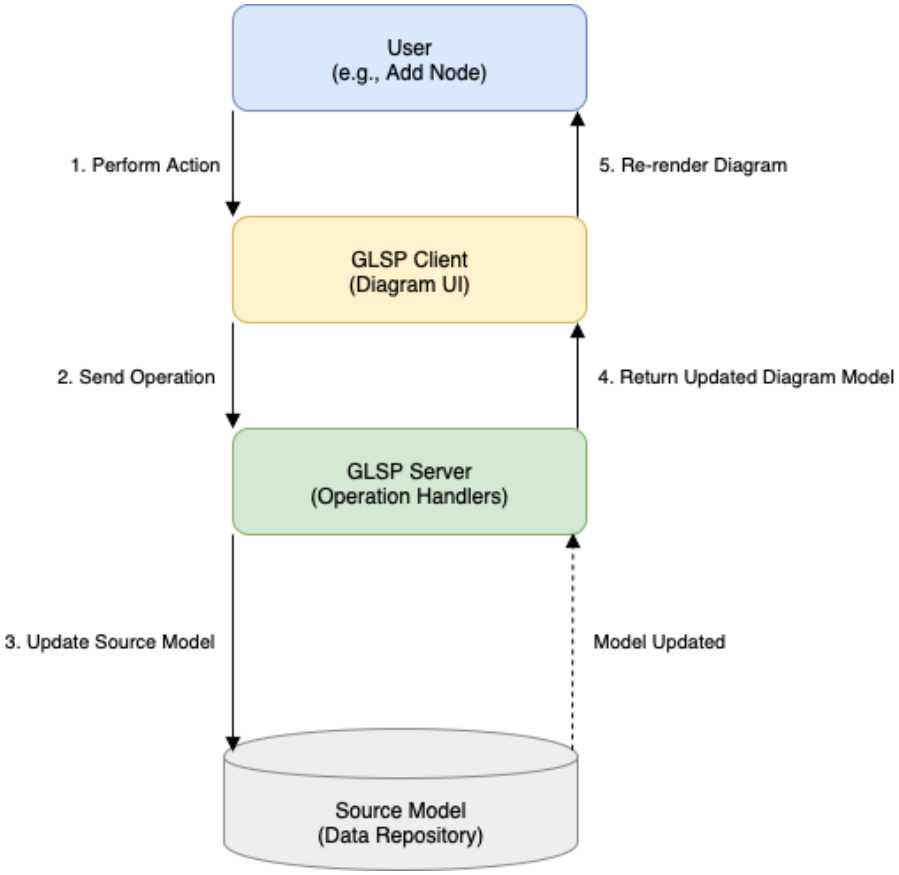

Figure 2.4: GLSP communication cycle

GLSP uses a structured command and operation handling framework internally to process

user edits in a modular and extensible way. The communication between client and server during this process is illustrated in Figure 2.6, which shows how user actions trigger operations that flow through the GLSP client and server components before updating the source model and re-rendering the diagram. GLSP models user model modifications through operations which represent specialized actions within the protocol for model alterations [Fou25b]. Each operation (identified by a unique kind, e.g. addNode, deleteEdge) is handled on the server side by an Operation Action Handler and one or more Operation Handlers. The Operation Action Handler functions as a generic dispatcher (a single global OperationActionHandler exists in every GLSP server) which accepts operation requests from clients. In addition to server-side execution, certain actions can also be invoked directly on the client side through the GLSP Dispatcher, which coordinates hybrid interactions (HSpan) between client and server components. The operation execution receives context from a command stack (for undo/redo functionality) through the Operation Action Handler before the handler directs the actual model modification to the registered Operation Handler for that specific operation kind. The server executes "add class" operations in UML diagrams through custom AddClassOperationHandler which understands how to modify UML source models by creating new class elements within the model repository. The Operation Action Handler executes post-processing activities that include diagram graphical model regeneration through the GModel Factory and model update notification transmission to the client. The command stack receives commands from this interface for the purpose of enabling undo/redo functionality. Operation Handlers contain domain-specific model modification functionality (CRUD operations on the source model) while the Operation Action Handler executes operations and maintains diagram consistency after each modification. The command pattern design allows each user action to serve as a command which supports execution and undo and redo operations while keeping business logic separate from GLSP framework execution steps. The resulting command framework allows developers to add new operations while preserving graphical view consistency with the model state.

Modularity and Dependency Injection

A key strength of GLSP is its highly modular architecture and use of dependency injection (DI) to configure all components [Fou25b]. This modular structure is illustrated in Figure 2.6, which shows how the GLSP server composes its core services, such as the action dispatcher, handlers, GModel state, and model submission cycle, within a dependency injection container. Both the client and server side of GLSP are built using an inversion-of-control container (the GLSP client uses an InversifyJS container in a Node/TypeScript setting, while a Java-based server uses Google Guice) [Jan25, Goo25]. All core services – such as model loading, model transformation, operation handling, model layout, etc. – are wired via DI and can be easily swapped or extended by overriding bindings in the DI container. For instance, if a tool needs a custom strategy for layouting diagram elements or a different persistence mechanism for the model, the developer can replace the default service with a custom implementation by rebinding it in the configuration. This DI-based design, combined with GLSP's use of slim abstractions, gives adopters a

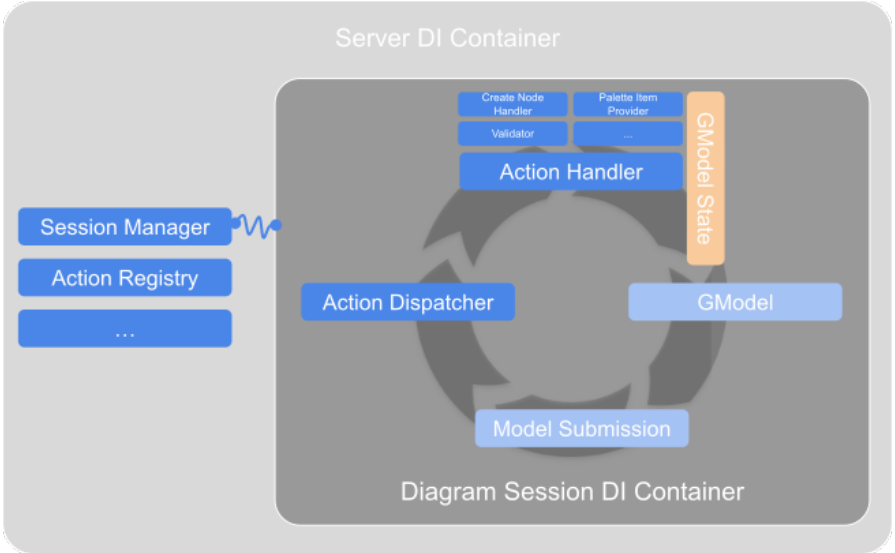

Figure 2.5: GLSP server architecture with Operation Action Handler, action handlers, GModel state, and model submission cycle [Fou25b]

high degree of control and extensibility. Different from other frameworks, GLSP does not use overly generic or restrictive APIs in critical areas, but instead of it, it uncovers the underlying technologies. By revealing for example direct access to the SVG-based rendering via Sprotty, CSS styling, or HTML overlay controls, the developers can adjust the appearance and functionality of the diagram editor according to their requirements [Fou25c]. In other words, GLSP's architecture's strength is, that it separates the concerns and does not sacrifice flexibility: the framework takes care of the communication, base protocol, and integration points, while allowing full customization of visuals and model logic. This modular structure of GLSP enables it to work with various implementation stacks. For example, GLSP servers can be implemented in Java or TypeScript (Node.js) and GLSP provides base classes in both languages. The choice of programming language can be made based on the project's ecosystem or deployment needs. Similarly, the source model can be managed with arbitrary frameworks or formats (EMF models, JSON files, databases, etc.), and GLSP offers integration modules for common cases like EMF/Ecore models or a simple in-memory JSON model [Fou25a]. Because the client–server protocol is well-defined and language-agnostic, it is possible to even switch out one component without having impact the others – for instance, replacing a Java GLSP server with an equivalent Node.js server, or supporting multiple client integrations (VS Code and a web app) using the same server logic. Such deployment flexibility is discussed in detail by Metin and Bork (2025), who describe a reference architecture for GLSP-based tools supporting various client and server configurations [MB25]. The interaction between the client- and server-side components of GLSP is illustrated in Figure 2.6, showing how the protocol connects modules such as **@glsp/client**, **@glsp/server-node**, and **sprotty** across the browser and server environments. This GLSP reference architecture consists

of a generic client and protocol together with pluggable server and model components which enables high extensibility for developing specialized modeling tools.

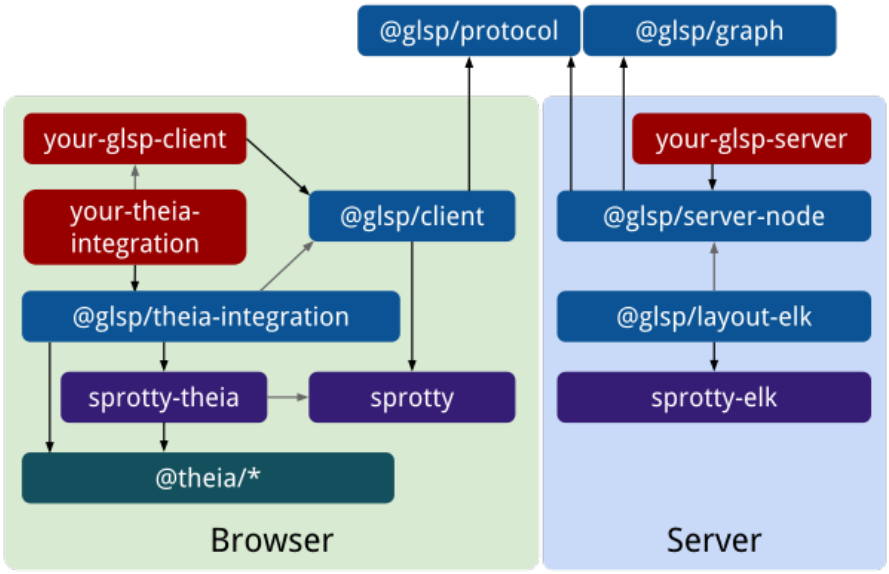

Figure 2.6: GLSP package architecture showing client–server separation and reusable protocol components across platforms [Fou25b]

Use Case: BIGUML and a TypeScript GLSP Backend

The BIGUML tool serves as a great example of a modeling solution, which was built on GLSP's architecture. BIGUML is an open-source web-based UML modeling tool, implemented as a VS Code extension that uses benefits of GLSP for its diagram editing capabilities [Too25]. The BIGUML depends on GLSP to display and modify diagrams through its GLSP client which operates in VS Code using Sprotty diagram framework for visualization and user interaction and its GLSP server which contains UML-specific logic for rule enforcement and model data management. Thanks to GLSP's platform-agnostic design, BIGUML's diagram editor integrates seamlessly into VS Code, but could be reused in other environments as well. The architecture of BIGUML is in fact modular: it separates generic diagram features from UML-specific "diagram features" and higher-level "tool features", following principles of single responsibility and separation of concerns. Notably, BIGUML was initially realized with a heterogeneous stack – a TypeScript-based GLSP client and a Java-based GLSP server, plus a custom Java model management component for the UML model persistence [MB23a]. The setup showed and proved GLSP's flexibility through its ability to combine different implementation languages for client and server and allowed developers to take advantage of existing Java UML frameworks on the server side [Too25].

Building on this foundation, the work in this thesis extends BIGUML by replacing or reimplementing the GLSP server in TypeScript, creating a unified TypeScript backend for

the UML modeling tool. The thesis demonstrates that all BIGUML diagram operations and model management can be handled within a TypeScript backend using GLSP's Node.js server framework with Inversify for Dependency Injection without requiring a separate Java process. This approach takes advantage of the full reference architecture for GLSP-based web modeling tools [MB25] [bor25]. The existing BIGUML GLSP client and protocol remain the same, while the server and source model management are now realized with a TypeScript/Node stack. The motivation for this might be easier integration with web-based technologies and toolchains (as well as avoiding the overhead of bridging between TypeScript and Java). Thanks to GLSP's modularity, the transition is feasible with minimal changes to the client or protocol – the TypeScript GLSP server simply provides new implementations of the required services (e.g. UML model storage, operation handlers for UML actions, etc.), conforming to GLSP's extension points. The result is an extensible, TypeScript-based backend framework for web modeling tools, validated by the BIGUML use case. In summary, this shows how BIGUML uses GLSP to allow independent, but working components (client, server, model store) and how a GLSP-based architecture can be expanded or customized (in this case, moving to a TypeScript server) while keeping the overall design. This shows the value of GLSP's design, which allows to create modern and extensible web-based modeling tools [Fou25b, Too25].

2.3 Langium

2.3.1 Overview and Architecture

Langium is an open-source language engineering framework, which was designed as a modern and web-native successor to Eclipse Xtext. It provides first-class support for the Language Server Protocol (LSP) and is implemented entirely in TypeScript, running on Node.js (and transferable to browsers) [Typ25]. The core idea, as it was described by Spönemann [Sp2], was to keep Xtext's successful concepts as grammar-centric language definition with built-in cross-references. However, we should "lift them onto a new technological foundation" by using TypeScript as the new standard for cloud and web tooling. Langium thus offers a similar architecture of services as Xtext (parser, scope provider, validator, etc.) but on a homogeneous technology stack. It provides a high-level grammar language (akin to EBNF) to specify a domain-specific language's syntax and structure, from which it generates a parser and an in-memory model (AST) representation [Typ25]. In contrast to EMF-based frameworks, Langium uses plain TypeScript types for its AST: each grammar rule corresponds to a generated TypeScript interface or class, and parsing an input file produces a tree of JavaScript objects conforming to these interfaces. Using this design there is no need to maintain a separate metamodel (no need for Java/EMF) and ensure type safety. In the case of grammar change, TypeScript's compile-time checks will catch inconsistencies in any code that manipulates the AST. Overall, Langium comes with out-of-the-box components for parsing, semantic model (AST) creation, reference linking (scoping), validation, and more, all wired together via dependency injection for maximum flexibility.

2.3.2 Grammar to AST, Scoping, and Validation

The starting point in Langium is a .langium grammar that defines the language and directly shapes the AST (abstract syntax tree). The Langium CLI (Yeoman-based generator) can construct a new project and generate AST types and baseline infrastructure from the defined grammar (e.g., via `langium:generate`). Developers then extend or override services to add language-specific logic. Langium's parser reads input text at runtime and produces an AST instance, which is composed out of the generated types. A key feature is cross-references: grammar rules like `Foo -> bar=[Bar]` create references that are initially unresolved and later linked by the framework's Linker using a ScopeProvider. By default, Langium indexes named AST nodes and exposes them to scoping, supporting references within and across files. The scoping logic is modular and can be overridden to implement custom visibility or naming schemes; e.g., Petzold [Pet22] reports restricting targets by type and context. Langium's validation framework reports syntax/parsing errors and lets developers register custom checks in a ValidationRegistry. The validator collects issues during processing and runs custom rules, such as name uniqueness or checks of the required-section. This process then produces a unified list of diagnostics for the IDE. These processes, such as parsing, AST creation, indexing, linking, and validation form the document lifecycle. The whole lifecycle is executed automatically whenever a document is opened or changed.

2.3.3 Document Lifecycle and Incremental Processing

Langium uses a well-defined document lifecycle to maintain model consistency during editing. When a source file is opened or modified, the framework then creates or updates a LangiumDocument. This text is also paired with a parsed AST and starts in the Parsed state. The DocumentBuilder then goes through these states:

- **IndexedContent** - symbols indexed,
- **ComputedScopes** - local scopes prepared,
- **Linked** - cross-references resolved,
- **IndexedReferences** - cross-file references tracked for dependency management,
- **Validated** - all checks performed.

The AST becomes fully linked and diagnostics become known after each build process, thanks to the pipeline.

Importantly the lifecycle supports incremental processing. Langium records cross-file dependencies as ReferenceDescriptions, so when a file changes the affected document is marked Changed and rebuilt. Any document that referenced it, is partially invalidated: its stored links into the changed file are cleared so they will be re-linked on next access. This means the framework recomputes only what is necessary and it does not have

to re-parse or re-validate documents, which were not affected. By using the index of cross-references, it can quickly determine which models need re-linking or re-validation after an edit. This keeps the editor responsive even with large and/or interrelated workspaces. The approach copies the functionality of incremental compilers and Xtext's builder. The figure 2.7 illustrates how a LangiumDocument progresses through the stages Parsed, IndexedContent, ComputedScopes, Linked, IndexedReferences, and Validated, coordinated by the DocumentBuilder and its associated services.

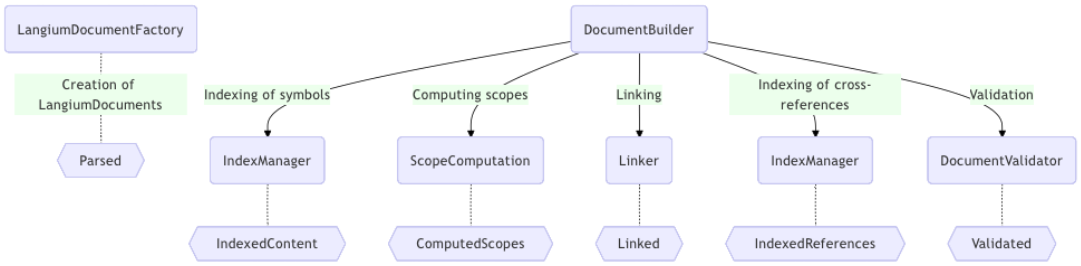

Figure 2.7: Langium document lifecycle [Typ25]

2.3.4 Extensibility and Services

The dependency injection (DI) architecture of Langium makes it highly extensible. The core functionalities exist as service objects inside a DI container which allows developers to replace default services without creating a framework fork. The Langium code generator creates a module class for your language which sets up the DI container through interface bindings for ScopeProvider, Linker, ValidationRegistry and other components. The default bindings in the module point to Langium standard services yet developers can choose to replace them with their own implementations. Developers should create a subclass for their custom validation or scope provider and then register it in the module instead of using the base class. The system will automatically select the custom service for use. The system enables developers to introduce new services through this mechanism. Petzold demonstrated this by creating a DiagramGenerator service, which generated an Sprotty SGraph from the AST in addition to customizing the scope provider and validator [Pet22]. The language server gained diagram request response capability through the registration of her LangiumDiagramGenerator in the module which replaced the default no-op implementation. The Eclipse ELK layout service in Langium includes integration points for diagram layout which users can override. The design of Langium allows users to extend its functionality through service-based additions of new concerns including graphical view generation and code generators which utilize existing AST and indexing features. The DI approach enables advanced users to customize or replace any language infrastructure component (parsing, linking, etc.) for specific domain needs but beginners can use the default settings.

2.3.5 Integration with GLSP and Sprotty (Overview)

The Langium system enables textual modeling through a contemporary TypeScript/LSP framework. GLSP operates through a client-server protocol for diagram editing and uses Sprotty to handle client-side rendering [Fou25c]. In Langium, the abstract syntax tree (AST) produced by the language parser acts as the single source of truth, while the GLSP/Sprotty client displays and modifies a synchronized diagrammatic view. The system maintains a single model which displays through two different views because textual modifications update the AST through Langium's incremental document model and diagram actions create model-level changes that update the text. This section explains the functions of these frameworks but the detailed analysis of concrete integration patterns and evaluated architectures including BIGUML appears in Chapter 3.

2.4 Generators

2.4.1 Generators in Software and Language Engineering

Generators automate the production of code and artifacts from higher-level specifications, reducing repetitive work and increasing consistency. In DSL and model-driven settings, they transform models or DSL programs into partial or full implementations, improving productivity and lowering error rates [MHS05]. Because generated artifacts follow uniform patterns, large systems gain structural coherence and maintainability [KT08]. In typical toolchains, a DSL's metamodel or grammar serves as the authoritative source from which generators derive libraries, configuration, and executable modules-i.e., model-to-code transformations that keep implementations synchronized with evolving models. Two broad techniques prevail:

- **Template-based generation** fills parameterized text templates with model data, offering explicit control over output structure.
- **Rule-/transformation-based** generation traverses the input model using declarative mappings to emit code programmatically.

Both fall under model-to-code and are supported by established MDE tools (e.g., Acceleo, EGL, Xtend). When generators are integrated into builds or editor tasks, they enable iterative regeneration as models change and that preserves alignment between design and implementation. Generators are tightly coupled to metamodeling: a metamodel specifies structure; a generator derives implementation from it. The canonical EMF example is Java class generation from Ecore, yielding factories, accessors, and infrastructure for model manipulation. Similarly, language workbenches generate typed ASTs, serializers, or interpreter stubs from grammars, bridging high-level specifications and executable code. A common practical use is scaffolding, which means generating a consistent project skeleton - such as files, modules, and build scripts - that gives developers a solid starting point. Many ecosystems also support sub-generators that can apply incremental updates

by adding new features, refreshing generated files etc.. This keeps projects current as the underlying model evolves.

2.4.2 Yeoman for Generator Development in TypeScript Ecosystems

Yeoman is a well-known utility for building and running generators using JavaScript/ TypeScript in web and language engineering tool development. Often described as "the web's scaffolding tool for modern webapps" [Com], Yeoman provides reusable generator modules that scaffold complete projects or components via a CLI. Conceptually, a generator is a Node.js module, usually written in TypeScript or JavaScript. It collects inputs through prompts and then emits files from templates. This pairing of interactive prompts with templating enables flexible and user-friendly automation and the outputs stay predictable and repeatable.

A key strength of Yeoman is modularity and reuse. Generators are published as NPM packages and invoked with `yo`, allowing best practices and standardized project setups to be shared across teams. The runtime also supports composition, so one generator can call others-useful in complex toolchains where different aspects (e.g., syntax, semantics, IDE features) are handled by dedicated modules. This promotes consistency, avoids reinventing common scaffolds, and encourages layering small, focused generators into larger solutions.

Yeoman is particularly well-matched to TypeScript workflows and integrates smoothly with language engineering frameworks such as Langium. The official `generator-langium` bootstraps a complete DSL project (grammar, compiler infrastructure, VS Code extension) with a few CLI steps, so engineers can focus on language design rather than project wiring. Because the generator is itself TypeScript-friendly, generation logic can import typed APIs, leverage static checking, and align with monorepo practices (e.g., Yarn workspaces) without friction.

Beyond one-time scaffolding, Yeoman fits ongoing automation. The sub-generators have the ability to regenerate or extend artifacts when DSL grammar or metamodel evolves, like when AST or configuration is refreshed. Commands (e.g. generation) can be used as build scripts or editor tasks such as `npm run generate`. The execution model of Yeoman includes stages that produce deterministic and idempotent outputs while handling conflicts and supporting dry runs and prompting and configuring and writing and install. The features enable developers to achieve reproducible results between different environments. The implementation of versioning generators as dev dependencies enables traceability because developers can reproduce the project's generated baseline through version pinning of the generator and template.

In sum, The Yeoman framework provides TypeScript developers with an easy-to-use scriptable base for generator development through its interactive prompting features and template-driven output and composition and workflow integration features that render it suitable for model management toolchains. The framework accelerates development while preserving uniform structure and conventions and enables easy distribution of generation

logic as a reusable package which matches the requirements of DSL-oriented tooling and TypeScript-based ecosystems.

CHAPTER 3

State of the art

This chapter describes the state of the art in model-driven engineering and then focuses on two key contributions for this thesis, which prepare grounds for a fully TypeScript and browser-compatible BIGUML architecture. We first review the conventional frameworks EMF [Fou25a] and Xtext [Ecl25] with their limitations. Then we discuss the architecture and goals of GLSP and Langium. We summarize Lencses's approach [Len24] to combine textual and graphical modeling and the look into Jäger's contributions [Jä24] [JLF+25] into a TypeScript native model server and metamodeling. In the end of this chapter we explain how these efforts help to the contribution of BIGUML and give the groundwork for my thesis.

3.1 Traditional Modeling Frameworks: EMF and Xtext

The Eclipse Modeling Framework became a cornerstone for defining metamodels (in Ecore) and generating code for model manipulation. EMF provides a rich Java API and tools for creating domain models and is still widely used for building modeling tools [SBPM08]. On top of EMF, Xtext emerged as a framework for textual domain-specific languages (DSLs). Xtext [Ecl25] allows language engineers to define a grammar for a DSL and automatically generates a parser, an EMF-based metamodel, and an Eclipse IDE editor with features like syntax highlighting and code completion [Ecl25]. These traditional frameworks enable powerful modeling tools, but they come with drawbacks: they tie the solution to the Eclipse/Java ecosystem and thick desktop clients, and integrating textual and graphical views is non-trivial.

Combining textual and graphical modeling in the Java era often meant mixing different technologies. One approach was to embed textual editors inside graphical modeling environments (or vice versa) and synchronize them. Obeo [OT17] and TypeFox [Kö16] demonstrated an Xtext and Sirius integration that allowed embedding an Xtext editor into a Sirius diagram, updating the model only upon save [OT17]. This "synchronize on

save" strategy confirmed the feasibility of hybrid modeling but did not offer real-time blended editing. Another approach combined two different editors: the textual editor (backed by an Xtext language server) and the graphical editor (e.g. Papyrus or Sirius) would each operate on their own representation and exchange updates through a common model store. Addazi et al. (2017) proposed such a solution for UML, combining Xtext for textual notation with Papyrus for diagrams [ACLP17]. While these integrations proved that blended modeling – "interacting seamlessly with a single model through multiple notations" [CTVW19] – is desirable, they were difficult to implement. The requirement to connect Java-based textual servers with graphical frameworks resulted in more complex development and maintenance processes. A notable example was the BIGER tool [GB21], a VS Code extension for ER diagrams that used an Xtext language server (Java) together with a Sprotty-based diagram (TypeScript). BIGER successfully provided a hybrid textual–graphical modeling experience in VS Code, but it required a heterogeneous stack and custom synchronization logic [GB21]. These experiences showed the limitations of traditional frameworks: Java backend components which are heavy, lack of modular web deployment and issues with keeping textual and graphical editors in sync in real time.

3.2 Web-Based Modeling with GLSP and Langium

To overcome the mentioned limitations and use web technologies and open standards, next-generation frameworks have been introduced. Two key players are the Graphical Language Server Platform (GLSP) and Langium. The two tools follow the current trend of cloud-based development tools and browser-based development through the LSP for editor integration. We have discussed these two tools already in the backgrounds Chapter 2. BIGUML, for instance, was initially built with a GLSP client, a GLSP Java server, and an EMF model server (also Java) behind it [MB23a]. Langium is tightly integrated with LSP by design, meaning the textual editor experience in VS Code or Eclipse Theia is largely configured by Langium's generated language server [BL23]. This new web-based stack offers clear benefits:

- no dependency to Eclipse or Java
- more uniform developer Experience - all in TypeScript
- easier deployment - everything can run in browser or web container
- more flexible UI integration - model editors can be embedded in web apps or VS Code

There are also some trade-offs to these tools, as for example their immaturity, because Langium's grammar language is still evolving and it does not support all of the advanced features Xtext/EMF would. For all that, they are being improved in a rapid manner and their viability for complex use cases has already been shown.

3.2.1 Integrating Textual and Graphical Modeling with Langium & GLSP

Combining a Langium-based textual editor with a GLSP-based graphical editor to achieve "one model, two views" is at the heart of Adam Lencses's work [Len24]. Lencses revisited the blended modeling problem using the new frameworks: the goal was to allow simultaneous editing of a model via text and diagram, using a unified web technology stack. In his state of the art, Lencses surveyed prior efforts (many discussed above) and observed that older solutions either synchronized infrequently (only on save) or required complex dual infrastructure. With Langium and GLSP, there was an opportunity to design a cleaner, real-time integration.

Lencses's architecture introduces a model service (or model server), which acts as the single source of truth for the model and makes sure that both editors stay in sync. The concept is that the GLSP server and the Langium language server both connect to this model service to apply and retrieve model changes. In practice, Lencses extended an existing GLSP example – the Workflow diagram editor – by adding a Langium-based textual DSL for the same workflow language and implementing a model management component to coordinate them. When a user edits the diagram, for example by adding a node or renaming an element, the change is passed on as a command to the model service, which updates the underlying model (the AST) and then notifies the textual editor, which can then reflect the change. The user interface updates the AST through the Langium parser when they enter textual DSL which causes the model service to trigger a diagram view update. The model service maintains formatting and comments in the textual side by carefully applying modifications (e.g. using incremental parsing or patching) instead of rebuilding the text from scratch after every change so that non-semantic information is preserved [Len24]. Lencses identified this as a crucial requirement because previous methods typically removed comments and custom formatting after a round-trip edit but his solution maintains them by working directly on the shared model in memory.

Evaluation of the approach was done by implementing hybrid editors for two UML diagram types in BIGUML: the UML Class Diagram and Package Diagram [Len24]. The original graphical editors of BIGUML for these diagrams, which are backed by Java, were recreated using the new stack:

- There was a new Langium grammar defined for UML class/package elements
- Also a GLSP diagram was provided
- the model service kept them both synchronized

The resulting prototype allowed, for example, a class to be added via the diagram palette and immediately appear in the textual UML notation, or an attribute name to be changed in text and instantly updated on the diagram. This showed true combined modeling for UML in a web environment. Lencses reports that both editors operated

on the same underlying model and could be used in parallel, with the system handling consistency and simultaneous edits. One challenge was to make sure that the textual editor's incremental parsing could live together with external changes: his solution leveraged Langium's document update mechanisms so that if the model service changes the AST, those changes propagate into the text document's state without corrupting the editor (essentially treating the model service like another editing client). By validating the concept on UML (a complex, real-world domain), Lencses's work proved that Langium and GLSP can be successfully combined for hybrid modeling. It also provided a blueprint for structuring such tools: the use of a dedicated model server component can be seen as a pattern for future web-based modeling tools

3.2.2 TypeScript-Based Model Management Framework

While Lencses dealt with the synchronization of editors, David Jäger's work focused on the underpinning model management infrastructure needed to modernize tools like BIGUML [Jä24]. Jäger recognized that, even with Langium and GLSP, a typical modeling tool might still rely on Java in the backend for defining the metamodel and managing model state. In BIGUML's case, the UML metamodel was defined in EMF Ecore and a Java model server was used to handle model persistence and multi-client access. To eliminate this Java dependency, Jäger developed a TypeScript-native approach for metamodeling and model management.

A core contribution of Jäger's thesis is a TypeScript grammar language for defining modeling languages (metamodels) in a way that integrates with Langium. The proposal is to define new structure of the modelling language as TypeScript classes and interfaces, without the use of EMF Ecore models. These are then enriched with custom decorators or annotations, which can indicate containment relationships, cross-references or any modeling concepts. For example, one can declare a class ClassDiagram with a property that is an array of Class elements. Then label it as a containment reference, which specifies part of the UML structure in pure TypeScript. This grammar is conceptually similar to Langium's own grammar, but it is meant to be more abstract (focusing on the metamodel) and flexible for extension. Jäger's approach then uses a generator to process this TypeScript grammar definition and produce two things [Jä24]:

- a standard JSON-based grammar that Langium can consume to create a parser/AST (essentially translating the metamodel into a concrete textual syntax),
- a skeleton implementation of a Langium-based model management server that can host and manipulate instances of this metamodel.

In other words, given a metamodel in TypeScript, the generator sets up all the plumbing for a working modeling tool: the AST types, the Langium language definitions, and the model server that allows external edits via an API.

Crucially, Jäger introduced a model server API integrated with Langium. This is an extension to Langium that allows model-oriented operations (rather than text edits) to be applied to the AST. The model server API provides commands to create, delete or modify model elements. These operations are executed by applying JSON patches to the Langium AST. By using JSON Patch (a standard for describing changes in JSON data), the approach can send a brief description of a model edit, like for example, "add an attribute named x to class Y", to the Langium-based model server. The server will then update the in-memory AST accordingly and inform all clients, which are connected. Jäger integrated this API directly into the Langium framework. By this, he enabled Langium to act not only as a language server, which responds to text edits, but also as a model server, which responds to model changes from other sources. This extension is key to allowing GLSP and other "model-oriented clients" to manipulate the model without going through textual edits. For instance, a GLSP diagram tool can invoke the model server API when a user drags a node or edits a property, and Langium will update its AST and ensure that the text view can be updated (or at least remain consistent).

To validate his framework, Jäger applied it to two case studies, mirroring Lencses's evaluation: the Workflow diagram example and the BIGUML tool. For the Workflow editor, which originally had a TypeScript GLSP server but no full model server, Jäger generated a Langium-based model management server and connected it. After small modifications, like for example redirecting GLSP commands to use the JSON patch API instead of directly mutating a model, the workflow editor was able to support all basic diagram operations fully using the new model server. The result was a purely TypeScript implementation of a workflow modeling tool, where both text and diagram (if a text view existed) would share the Langium AST as the single source of truth.

The more complex example was BIGUML, targeting UML Class and Package diagrams. Jäger had to create a TypeScript representation of (a subset of) the UML metamodel to use with his generator. He reports defining a partial UML metamodel in the TypeScript grammar language – for example, a Diagram root class containing either a ClassDiagram or PackageDiagram – and annotating it appropriately. From this, the generator produced a Langium grammar and a model server. Jäger then replaced BIGUML's original Java components with a new TypeScript GLSP server (based on the workflow GLSP server code) and the generated Langium model server. After integrating these, BIGUML's VS Code extension could be run with an entirely TypeScript backend. The functionality was verified against standard UML modeling features: creating and deleting classes, adding attributes and operations, renaming elements, etc., all through the GLSP diagram UI, and all handled by the Langium model server under the hood. Jäger even implemented advanced editor features like a property palette (an inspector form to edit properties of a selected model element) and an outline view in the GLSP client, showing that even these could be supported with the new architecture. The property palette actions were wired to the model server API: when a property value is changed in the form, a JSON patch updates the model in the Langium AST, which then reflects back in the diagram.

One outcome of Jäger's work is that it lowers the barrier to porting existing modeling

tools to a web-friendly stack. He developed a tool to import Ecore models and turn them into a TypeScript grammar, supporting developers in moving from EMF to Langium [Jä24]. Moreover, his approach to integrate the model server API into Langium itself contributed to the Langium project which could benefit others who need multi-editor support. The research findings were presented in a joint publication [JLMB25]. According to it, the two state-of-the-art tools (the workflow editor and BIGUML) could be "rebuilt" using a homogeneous TypeScript stack, with full support for hybrid textual–graphical editing. This shows a significant step towards web-based modeling tools.

3.3 Contributions to bigUML and Thesis Context

Lencses and Jäger work together to develop BIGUML into a web-native blended modeling tool which supports browser-based LSP-driven modeling and textual–graphical workflows [BL23].

- **Lencses's contribution** demonstrates real-time two-way synchronization between textual and graphical UML editing through the combination of a Langium DSL with a GLSP editor and a shared model service. The approach enables practical blended modeling of UML while validating the BIGUML metamodel representation in Langium grammar and maintaining editor features such as comments and formatting while ensuring both notations remain consistent.
- **Jäger's contribution** replaces the Java/EMF backend with a TypeScript-native model management framework and a model-server API which applies JSON-patch operations to the Langium AST. This enables BIGUML to run end-to-end in TypeScript, for example as a VS Code extension. There is no need for a separate Java service, while keeping multiple clients (text and diagram) consistent.

As a result, the development of BIGUML transforms from a prototype that uses mixed technologies into a complete TypeScript-based system which is compatible with web browsers. Prior work describes the development path of BIGUML starting from its GLSP-based introduction to the lessons learned during the migration process [MB23a, MB23b]. Lencses and Jäger solve two essential problems which were found in the previous work: blended notations on the web and EMF-free UML model management.

3.4 Summary

The chapter examined the transition from Java/EMF-based modeling stacks to web-native approaches which focus on GLSP and Langium. The chapter explained how GLSP's client–server diagram framework and Langium's TypeScript/LSP-based language engineering allow for browser-based hybrid textual–graphical tooling. Building on this, we summarized two enabling contributions: (i) Lencses's blended modeling setup that synchronizes a Langium DSL and a GLSP editor through a shared model service [Len24];

and (ii) Jäger's TypeScript-native model management, including a model-server API that applies JSON patches to a Langium AST and removes the dependence on a Java/EMF backend [Jä24]. Together, these works reposition BIGUML toward a fully TypeScript, browser-compatible architecture.

Building directly on these foundations, this thesis focuses on the model-management operation handling layer: This thesis generalizes and systematizes create–read–update–delete (CRUD) and related operations in a configuration-driven, TypeScript architecture atop Jäger's model server and Lencses's blended editing setup, using code generation from a single definition file to cover all diagram types. In doing so, the thesis operationalizes the state-of-the-art concepts for BIGUML and contributes a reusable operation-handler pattern applicable to similar GLSP/Langium-based tools.

CHAPTER 4

Concept

In this chapter, I will introduce the concept for the solution of issues, we have talked about thus far. I will focus on the ideas, patterns and architectural choices I designed or adopted and explain the problems and principles.

4.1 Goals & Constraints

The main aim of this thesis is to improve, optimize and work on the alternative option for graphical modeling, which was proposed by Jäger [Jä24] in his thesis. This alternative moves away from the traditional Java-based EMF technology and utilizes web-friendly TypeScript alternative with the help of Langium - next-generation textual language framwork written in TypeScript, which provides support for the Language Server Protocol.

4.1.1 Project Goals

I will once again summarize the goals of this thesis, in order to improve the Jäger's solution on use all of the benefits of using modern and web-friendly TypeScript:

- **Maintainability and reuse:** One of the biggest goal of this thesis is to make the solution as easy to maintain as possible, what means, that the per-diagram boilerplate needs to be reduced and minimized. The repository then does not have much duplicated code, as this is generalized and inherited for specific cases and thanks to it is then easier to read and understand. This enables also quick onboarding for new developers, who need to extend the tool and also safe refactors of the code, as the main logic is generalized and centralized. In the Figure 4.1 we can clearly see, that the number of files, needed for creation of the single elements is really very big.

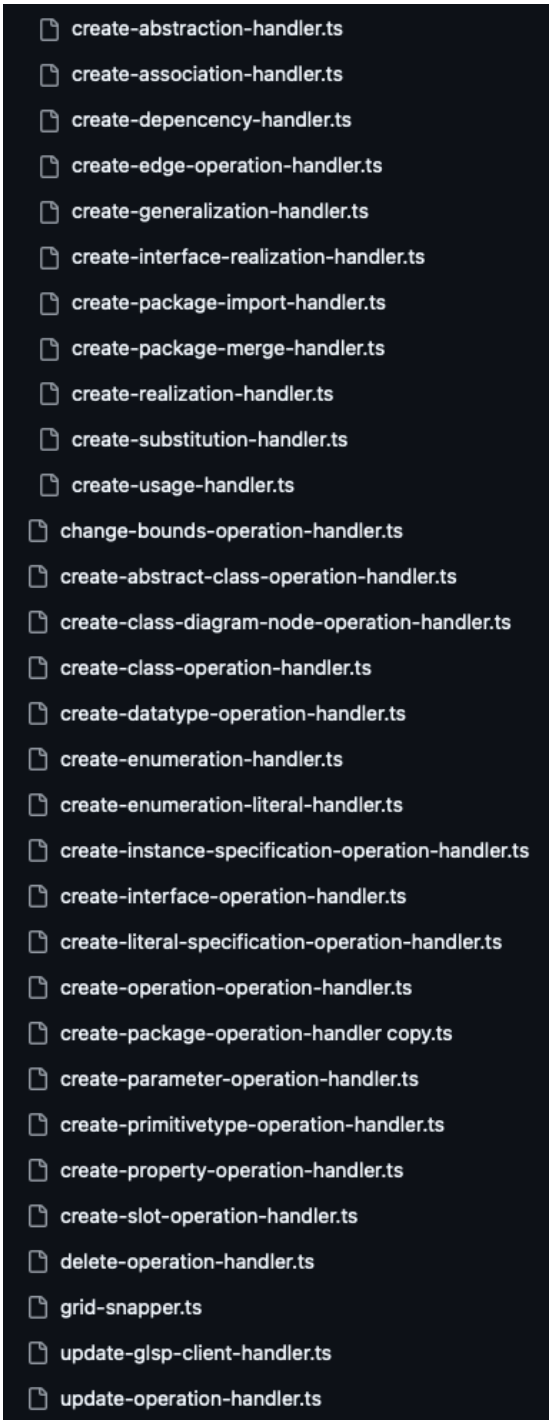

Figure 4.1: Handler explosion in class diagram

- **Single source of truth:** The goal is to create one file (called `def.ts`, where all information a diagram needs are summarized and everything is based on this file, whether it is code generation (needed for the functionality of the tool), finetuning of elements, logic for the diagram or validation. It is important for this file to be the single source as it avoids the duplication and drift across all features of the tool.
- **Type-safety and correctness**: Use the perks of the TypeScript as for example literal unions, strict types (for element types), strongly-typed decorators restricted to compatible fields - misuse is a compile-time error, generics and discriminated unions to enforce exhaustive handling.
- **Scalability:** The goal is to make the option to add new diagram types (e.g. Class Diagram, Sequence Diagram etc.) easy and straight-forward. Adding of a new type of diagram should as easy as creating few very simple files, containing only the type-specific definitions. The whole logic, which would otherwise be needed in these files must be generalised in abstract class, which the newly created must extend.
- **Developer Experience:** Every diagram follows the same order of things - edit facts in the one source of truth and run generators. Each diagram has same folder structure and naming conventions. Generator scaffolds boilerplate from the one source, which reduces duplicated code fragments and setup time. There are clear extension points, as the abstract handlers are documented and the per-diagram code sets only the `elementTypeId` and optional functions.

4.1.2 Context and Constraints

Langium AST stays as a single model source (baseline by Jäger, which is adopted and unchanged), what means, that all the truth lives in the Langium AST, not the diagram. The UI never mutates the diagram directly, but the changes are AST-first via typed JSON-Patch. It is important to state, that it must maintain the referential integrity as stable IDs, the containment between parent and child and ordering.

Also the GLSP operation lifecycle used in this thesis follows the Jäger's baseline, which was adopted and unchanged. Editing is done using a strict cycle, where the client sends a request, which is then validated by the server. The request followingly changes the Langium AST and finally emits GLSP actions to upgrade the diagram. Handlers remain stateless with respect to the UI and validation takes place before any commit, in order to preserve the integrity of the model. The four operations:

- Create
- Update
- ChangeBounds
- Delete

respect the GLSP semantics, including selection context, coordinate handling, and parent resolution.

Another project constraint is the generic over specific architecture, which intentionally minimizes the code needed per diagram. This can be done by introducing abstract base handlers. For specific diagram type handlers only small and declarative overrides are needed as the `elementTypeIds` or factory mappings. Other things as elements and attributes are meant to be configured from the `def.ts` - this keeps the behavior of the tool consistent for all the diagrams. There is also an option for rare edge cases, where documented hook methods can be used without harm to the generic core.

File `def.ts` serves as single source of truth. The types and decorators declared here drive validation, property palette, the outline, code generators and also AST generation. When anything is changed in `def.ts`, it is safely sent across packages through typed registries and templates used in generators. The strict typing prevents any configuration drift or duplication. This centralization minimizes the maintenance effort and keeps the text-based and diagram-based editors aligned.

Validation and UX consistency is also a project constraint, which needs to be mentioned. Rules, which were declared using decorators in the `def.ts` must match the UI constraints as required or optional fields, enum options etc. To guarantee this, the palette and outline schemas are derived directly from `def.ts` and avoid stringly-typed mappings. Errors are shown using the GLSP diagnostics consistently, no matter where the edits originate.

The TypeScript stack remains conceptually interoperable with legacy BIGUML semantics. This serves as an environment constraint, to keep it compatible with the prior Java/EMF approach. It preserves the naming conventions, containment rules and ID semantics.

4.2 Single Source of Truth: `def.ts`

4.2.1 Principle

The goal was to create a single source of truth, which serves as a central and customizable meta-definition - for this purpose file `def.ts` was used. In `def.ts` coexist nodes and edges (domain element kinds), relationships, constraints, so that there is one file, which controls the behaviour across the tool.

In the prior version by Jäger [Jä24] `def.ts` already existed. It was used as the entry point for the generation of the Langium grammar (`uml.langium`) and AST reflection/types (`ast.ts`). These two keep the DSL and AST in sync. The textual model or the Langium AST is the authoritative source and can also be called AST-first worldview. The diagram behavior follows and satisfies to what is declared in `def.ts`. It means it gets all of the containment, references, default values for the attributes or created constraints.

What we need to extend in this thesis is to centralize the configuration in this file and therefore provide consistency by construction. By doing this, all of the newly created consumers, meaning all of the generators, read the `def.ts` directly:

- Default value generator
- Path generator
- Property palette generator
- Outline view generator
- Validator generator

rather than duplicating the configuration for each of these generators separately. For that mainly decorators are used, as they serve the function we need in order to achieve this. However also other constructs of TypeScript can be used, e.g. to enumerate all elements one type of Diagram can have, union type is used.

So to summarize, the `def.ts` acts as a single source of truth, it was created already in the first version and is extended in this thesis:

- **Baseline (Jäger):** Generates Langium grammar and AST reflection
- **Extension:** Uses decorators and other TypeScript constructs in order to drive the generation of files, which strongly inherit details from it

4.2.2 What `def.ts` Captures

In this subsection we will name all of the elements the file `def.ts` must include, to fulfill all of the needs of improvements done in this thesis:

- **Domain taxonomy and roles:** The model splits types into three subtypes:
  - **Nodes (graphical entities) extending from `ElementWithSizeAndPosition`:**
    - `Class`,
    - `Interface`,
    - `DataType`, etc.
  - **Relations (edges) extending from `Relation`:**
    - `Association`,
    - `Composition`,
    - `Generalization`, etc.
  - **Meta information - `MetaInfo` with:**
    - `Size`,
    - `Position`,

    , which capture the layout information independently of domain semantics.

This separation keeps semantics clean, enables AST-first updates, and simplifies reusable handler logic.

- **Diagram root and kinds**
  - The main and entry node needs to be marked with a specific decorator - @root, so that langium generator knows, which it should be.
  - Each diagram type has its own class with arrays of relations and entities.
- **Enumerated vocabularies as literal unions:** Enumeration types with a textual options are given for types, which need specific options

This part of the file originates from the Jäger's version. The following sections contain the extensions and additions introduced in this thesis.:

- **Per-diagram element scopes:** In order to be able to create generated files for each diagram type separately, all valid elements (nodes and edges types) must be listed and defined for each diagram type. This is needed by property palette and outline generators, where they must create a separate file for each diagram type.
- **Containment vs cross-reference semantics:**
  - **Contained children:** e.g package -> entities, class -> properties, class -> operations. These are used to generate creation paths (@path)
  - **Reference to another model element:** Marks a property as such, resolved by name or ID, rather than containment. Common in relationships or type references (@crossReference)
  - **Dynamic property:** Dropdown of all elements in this type (@dynamicProperty(X))
- **Validation Constraints:** Attributes of elements (e.g. name of an element etc.) can be constrained by some rules. These are then checked on the client side and should also align with the server-side validation. There are two types of constraints:
  - **Built-in decorators:** These are validation constraints coming from an external library, in our case it will be TypeScript library class-validator. It contains the most basic ones, which are built-in with this library like @MinLength, @ArrayMaxSize or @ValidateIf. These are then expanded by the second type:
  - **Custom decorators:** These will use the same sematics, but can be created by the developer, when built-in constraints are not enough and some more specific validation is needed. The form also follows the class-validator library, but the logic needs to be set by the developer. Constraining length of a string with upper and lower bound is an example for custom validator, as the library class-validator does not offer this built-in.

- **Defaults and initialization policy:** Part of the generated content by generators are the default values for newly created elements. There can either be explicitly provided default values or the type's implicit default values (e.g. false for booleans, 0 for numbers, "PUBLIC" for Visibility etc.). There is also a difference between optional and non-optional attributes, which needs to be taken into account, when creating a new element. This behavior needs to be controlled in `def.ts`, as it must be clear, where default values are explicitly defined and where they should be implicitly taken.
- **Graph/Visual classification:** Some elements that are either not visual/visible in the diagram or sub-node elements as for example `EnumerationLiteral`, `Property`, `Operation...` must also be marked. This avoids and prevents accidental layout logic for purely semantic items. This is also needed for the newly created generators.
- **AST specialized hints:** In some special cases the AST type needs to be given in order to share the base AST identity. As an example, AST type "Association" on relations Aggregation and Composition shows how an edge family has the same AST identity for the tooling compatibility and still stays semantically distinct.

4.2.3 Why Centralize in `def.ts`?

In this subsection I will explain the rationale and explain the reasons behind centralization of all information in one single file `def.ts`.

These are the reasons for the centralization in `def.ts`:

- **Prevent duplication and inconsistencies:** All consumers and generators read from the same source. This means, that the definition of all elements does not need to be divided across the whole code base, but is centralized in one file. It also removes the possibility for errors and inconsistencies, when there would be different definitions in two different places. This leads to the fact, that changing of the `def.ts` updates the whole tool coherently.
- **Type-safety and safe refactors:** When some changes are done in the `def.ts`, TypeScript immediately indicates when a consumer is out of sync. `def.ts` acts as a typed contract everyone else must obey, as an example:
  - Literal unions act as closed vocabularies. For example if some type changes one of its values (e.g. `type Visibility = "PUBLIC" | "PRIVATE" | "PROTECTED" | "PACKAGE"` and some other actor (e.g. handler) uses an old value, the TypeScript compiler will report an error.
  - Typed decorators have constrained usage. All of the decorators have a very specific task with typed signatures and intended usage. If it is applied to a wrong type (e.g. decorator intended for strings is applied to number), or wrong argument(s) is/are passed, TypeScript flags it as an error.

  - The element sets per diagram (e.g. `ClassDiagramElements` contain all of the elements a diagram type can use. These are types and not just documents. This implies safe scoping.
  - Where code must be fixed, refactors come to surface. That means for exmaple if a property is renamed in `def.ts` (e.g from properties to members) or if there is change to required fields or default values - any generators or generated files (as handlers or palette mappings), which still reference these old values, will not compile.

    To sum up, when consumers drift from `def.ts`, we get early and compile time breakage. This prevents small runtime inconsistencies and make large changes safer and more mechanical

- **Predictable extensibility:** Adding of new elements to a diagram, or creating a new property or constraint is a single edit. Everything that is needed is just to declare it in `def.ts` and all registries and generators will pick it up. This workflow is repeatable and documented.
- **Separation of concerns:**
  - Layout vs. semantics: `MetaInfo` (`Size`, `Position`) holds layout data separately from domain classes, so resizing or moving does not dirty semantic models.
  - Non-visual elements stay semantic: @noBounds marks types like Property, Operation, Parameter as non-graphical, preventing accidental layout logic or handlers from treating them as nodes.

4.2.4 Outcomes

Now I will name all of the outcomes, that are created based on the `def.ts` - what is derived from it. Some points will be explained later in this thesis into detail, so now only short mentions:

- **Langium grammar & AST types:** As already mentioned the `uml.langium` grammar and the `ast.ts` reflection/types with runtime guard, subtyping and references are generated. This was already present from the Jäger's thesis as a baseline. As earlier described, the purpose is to keep the textual DSL and AST in lockstep with the conceptual model.
- **Creation path mapping:** Here valid parent and child creation routes are computed and encode them into creation path mapping.
- **Default value mapping:** A default initializer map for the generic create handler is generated. This supplies default values based on the preferences defined in `def.ts`.

- **Property Palette schema:** The field presence; widget types as text, enum, dynamic reference etc.; required or optional fields and constraints are derived from `def.ts`. This allows generation of palette configs.
- **Outline view schema** Same as for property palette, just with outline hierarchy and visibility.
- **Validation files:** Generate files needed for validation, which honor the decorators over attributes in `def.ts`

4.2.5 Summary

To sum up, the conceptual takeaway from this section is, that `def.ts` serves as the typed, single source of truth: it declares domain elements, containment, references, defaults, and constraints once, and from that one definition the system derives the DSL grammar & AST (baseline from Jäger), as well as creation paths, default initializers, property palette and outline schemas, and validation wiring (this project).

4.3 Generic Operation Handlers & Model State Abstraction

The goal of this section is to give the motivation and the explanation to the two core architectural concepts in this thesis, which aim to improve the maintainability of the GLSP-based tool at scale:

- **Generic Operation Handler pattern** and
- **Model state abstraction**

shared across diagrams.

To introduce the problem context, the current implementation in the tool version from Jäger is briefly described. Each element in a diagram has its own create operation handler. That means, that for all elements as `Class`, `Package`, `Enumeration` etc. there is one file in the project, which mainly looks the same for all of them and differs just in few values or small details. This is done in this way not only for nodes, but also for all edges, what grows the complexity and number of files. Important to say is, that the logic in these files is also very similar. Another issue with current situation is, that this setting is needed for all diagram types. This leads to huge code duplication, as for example Class diagram and Package diagram both use Class element and Package element. In the current state of the tool, there are four files (two in class diagram and two in package - one for class element create and one for package element create), where two of them are completely the same and the other two differ just in small details.

```
1  /** Before: per-element operation handlers */
2  binding.add(CreateClassOperationHandler);
3  binding.add(CreateClassPropertyOperationHandler);
4  binding.add(CreateClassMethodOperationHandler);
5  binding.add(CreateEnumerationOperationHandler);
6  binding.add(CreateEnumerationLiteralOperationHandler);
7  binding.add(CreateInterfaceOperationHandler);
8  binding.add(CreateAbstractClassOperationHandler);
9  binding.add(CreateParameterOperationHandler);
10 binding.add(CreatePackageOperationHandler);
11 binding.add(CreateDataTypeOperationHandler);
12 binding.add(CreatePrimitiveTypeOperationHandler);
13 binding.add(CreateInstanceSpecificationOperationHandler);
14 binding.add(CreateSlotOperationHandler);
15 binding.add(CreateLiteralSpecificationOperationHandler);
16
17 binding.add(CreateAbstractionOperationHandler);
18 binding.add(CreateAssociationOperationHandler);
19 binding.add(CreateDependencyOperationHandler);
20 binding.add(CreateGeneralizationOperationHandler);
21 binding.add(CreateInterfaceRealizationOperationHandler);
22 binding.add(CreatePackageImportOperationHandler);
23 binding.add(CreatePackageMergeOperationHandler);
24 binding.add(CreateRealizationOperationHandler);
25 binding.add(CreateSubstitutionOperationHandler);
26 binding.add(CreateUsageOperationHandler);
27
28 /** After: generic per-operation-kind handlers */
29 binding.add(ClassCreateNodeOperationHandler);
30 binding.add(ClassCreateEdgeOperationHandler);
```

Listing 4.1: Binding list before & after adopting generic handlers

In Listing 4.1 we can see the state of the operation handler bindings in the current version for the specifically class diagram. As we can see, each element has its own create handler, what leads to much duplicated code and many redundant files. In Listing 4.1 we can also see, how it could look like using the proposed solution, where we have one create handler per operation kind (node and edge).

We use configuration over specialization and composition over duplication, what enforces the consistent semantics with a minimal and stable set of abstract base classes and interfaces. Specifically what this means is, that we introduce a new behavior with a generic algorithm (configuration), which receives data and policies. By putting together small reusable strategies (composition), instead of creating many subclasses as shown in Listing 4.1 (specialization) or just copy-pasting logic (duplication). Abstract operation handlers and base model-state interface act as a contract, which makes the semantics same across all diagram types.

The clear outcome of this is, that per diagram, the long list of create handlers collapses from a big amount of element specific files to exactly two generic ones (for creation of nodes and creation of edges) and as a bonus, their logic is offloaded to the one abstract class, so that they can stay very simple in code. Handlers depend on a single base model-state interface.

Why it matters? Separating diagram semantics from infrastructure that keeps rules for naming, defaults and bounds in one place, while keeping the transport and command wiring generic. This makes the task to add a new element or new diagram type very simple and just small declaration to configuration, instead of creating a big amount of new classes and copying content. The additions are the auditable and testable - differences in the configuration clearly show what and why it was changed and behavior remains reproducible across diagrams by design.

4.3.1 Generic Operation-Handler Pattern and Abstraction

As described in the introduction to this section, there are many issues that need to be solved with the operation handlers. The problems are:

- **Handler explosion:** With creation in the current version each element required its own `CreateXXXOperationHandler` and then also often separate handlers for properties, literals, etc.
- **Fragile wiring:** Dependency injection modules, responsible for the binding of the operation handlers for each diagram contained very long and error-prone binding lists. Forgetting of just one binding leads to runtime errors.
- **Slow iteration:** When a new element is added, it requires creation of new files, new bindings and new tests. In all of these files, the behavior is nearly identical and differs only in small details so the code is very duplicated
- **Update and Delete Handlers:** Update and Delete Operation Handlers were practically the same for each diagram type and needed to be generalized into one common abstract class.

The design decision is to introduce an abstract and generic handler per operation kind (CreateNode, CreateEdge, Delete, Update). What was changed is, that the Template Method Pattern is being used, where the base class implements the algorithm:

```
1 parse op;
2 resolve container;
3 assemble semantic patch;
4 add meta info;
5 return command;
```

Listing 4.2: Pseudo algorithm of the base class for generic handler

As shown in the Listing 4.2, each handler firstly parses the operation, then resolves the container in the whole diagram, followingly assembles the semantic patch and adds the meta information to it, like the position and size etc. and finally returns the command. There are also hook methods, which provide points for small per diagram adjustments and variations.

As already mentioned, the approach is to configure rather than specialize. This means that diagrams contribute a minimal amount of configuration (e.g. elementTypeIds, creation-path resolution, default-value providers) instead of custom class implementations. This will consolidate and simplify the bindings. Now only one class per operation kind per diagram must be binded (e.g. `ClassCreateNodeOperationHandler` or `ClassCreateEdgeOperationHandler`) instead of a large number for each element per diagram. The binded components are also more deterministic, as in this version we always know, what to expect to bind in the module for the specific diagram:

- `(DiagramName)CreateNodeOperationHandler`
- `(DiagramName)CreateEdgeOperationHandler`

and this completes the set of creation handlers for each diagram type.

Several invariants are enforced centrally by the base handler, in order to keep the behavior predictable and uniform:

- **Naming policy:** It guarantees collision free and predictable default names. The default values come from the generated file, based on the earlier mentioned canonical schema in `def.ts`. Thanks to this, handlers remain deterministic.
- **Bounds policy and grid alignment:** Based on the definition from `def.ts` excludes the size and position patches to maintain a clean separation of UI concerns. Grid alignment is ensured by consistently applying snapping.
- **Atomicity:** The semantic and meta changes are commited as a single command. This prevents partial updates and improves the reliability and undo behavior.

What remains diagram specific are still the `elementTypeIds`, which the GLSP-side element type IDs and decide, for what element is the operation going to be executed. These `elementTypeIds` are then provided in the subclass specific to diagram like this:

```
1 readonly elementTypeIds = [
2         ModelTypes.ABSTRACT_CLASS,
3         ModelTypes.CLASS,
4         ModelTypes.DATA_TYPE,
5         ModelTypes.ENUMERATION,
6         ModelTypes.ENUMERATION_LITERAL,
7         ModelTypes.INSTANCE_SPECIFICATION,
```

```
8           ModelTypes.INTERFACE,
9           ModelTypes.LITERAL_SPECIFICATION,
10          ModelTypes.OPERATION,
11          ModelTypes.PACKAGE,
12          ModelTypes.PARAMETER,
13          ModelTypes.PRIMITIVE_TYPE,
14          ModelTypes.PROPERTY,
15          ModelTypes.SLOT
16      ];
```

Listing 4.3: `elementTypeIds` provided in the create node operation handler subclass for the class diagram

The Listing 4.3 clearly shows, that this is a very simple list, which can be generated based on the definition in `def.ts` for each diagram. When Listings 4.3 and 4.1 are compared, it is clear, that this is much simpler, as instead of binding classes for each element, we only provide an array with names of the elements in the subclass of the abstract generic create node handler.

Why it works? Generalizing and centralizing the algorithm, while pushing the variability just into small hooks removes the duplication and makes the behavior predictable. These are the principles this concept follows:

- **DRY (Don't Repeat Yourself):** Originating with Hunt & Thomas [HT99], DRY means each piece of knowledge lives in one place. The creation/update/delete algorithms exist once in the abstract handlers. That means, that the naming rules, defaults from the definition, grid snapping and patch composition are not copied into big number of classes. This removes the need for copy-pasting, makes the maintenance easier and a single fix or improvements applied everywhere.
- **Open/Closed Principle (OCP):** Formulated by Bertrand Meyer's but mainly popularized in SOLID, all of the software entities should be "open for extension, closed for modification" [Mar02]. The behavior is extended via configuration and strategies (`elementTypeIds`, creation-path policies etc.) instead of editing the main algorithm in the abstract handlers. This way the algorithm is not changed so much, what prevents errors, preserves stable contract and the evolution and improvement is incremental - new types and features are added without touching the core.
- **Consistency:** Default values, naming, bounds and patch structure all originate from the earlier described `def.ts`. They are received centrally by the base algorithm. This makes the behavior predictable and eliminates the small possible errors between element types.
- **Testability:** A shared algorithm enables contract tests, which can cover the invariant path once. The logic, which is specific for some diagram types, like some specific hooks can be tested using simple small unit tests.

- **CRUD-wide uniformity:** A similar template is used in Create, Update and Delete. It resolves the element paths using the index, transforms values if it is needed, chooses the correct patch operation and commits atomically. This prevents the differences between diagrams, the documentation is shorter and onboarding is simpler.

The following **Alternative Solutions** were considered but rejected:

- The status quo solution, with per element handlers is not ideal for use in this tool. As reasoned before this carries inconsistencies with it and causes combinatorial growth of files, while new file is needed for each element, with much of duplicated code. This issue gets even bigger, when we take more diagram types into consideration.
- Another possibility was to switch all operation handlers to a one single handler. This would create a "god class", which contains logic for create, update and delete operations for all elements (and also possibly all diagram types). The issue with this solution would be that it inhibits testability and is prone to merge conflicts. The readability and therefore the maintenance of this class would also be very hard, as it would consist of many lines of code and could lead to confusion.
- Code generation of the handlers in the status quo was also considered an option. This would on one hand reduce the typing, but it supports the code duplication as code in the single classes would still be very similar with just small changes needed. The generated code would still need maintenance and diffs would get noisy.

Although this solution appears to be a good fit for this scenario, it also brings some **Trade-offs** with it. To name some, also with their **mitigations**:

- Moving to generic handlers causes loss of specialized behavior for some specific elements.
  - Mitigation: Hook/strategy points, which can keep this specific needs local with per-type overrides, without need to change the base class.
- With more features adding to the tool, it is possible that the generic base class gets bloated or too big and may get confusing and harder to read.
  - Mitigation: The generic class should stay as minimal as possible and add all variations into injected strategies.
- More and more injected classes into the base class can get the control flow too messy, what can lead to longer debugging - the issues can become harder to trace.
  - Mitigation: The logs should stay clear to help with future debugging.

This generic handler design builds on established software engineering patterns. This means that the approach is based in proven practices, what makes it more understandable, comparable, and extendable solution:

- **Command Pattern** The operations in this solution are packaged as atomic commands which can be replayed, in case redo or undo is needed. This allows also batching and auditability. The intent and execution stay separated as the command is separated from the model state applying a JSON patch.
- **Strategy** Information or functions as naming, defaults, creation path resolution and other similar concerns act as policies which are injected. These can be changed per diagram, what keeps the generic base diagram simple and stable with the core diagram and also allows targeted variation.
- **Inversion of Control and Dependency Injection** The configuration for the specific diagrams is done at the module level, where strategies and handlers are wired without hard dependencies. This leads to lower coupling, clear composition and simpler substitution in unit tests.

After these changes, we expect these quality attributes:

- **Maintainability**: There will be fewer moving parts, which are needed to update and also standardized hooks.
- **Consistency and Correctness**: Uniform semantic patches and meta handling is used.
- **Scalability**: Adding of new elements or diagrams will become linear in configuration, not in new files/classes.
- **Developer Experience**: All the changes will lead to easier maintenance and what varies is explicit -> clearer mental model.

In summary, it can be concluded, that a single, template-based handler per operation kind, configured by per-diagram type lists and policies, replaces huge number of per-element handlers and makes creation, update, and deletion logic uniform, testable, and easy to extend.

4.3.2 Model State Abstraction

The goal of this subsection is to explain a small issue, which was discovered while creation of the generic handlers. The handlers depend strongly on the model states of the diagram, which are provided by dependency injection in the sub classes for each diagram. Ad-hoc union types:

```
ClassDiagramModelState | PackageDiagramModelState | ...
```

caused type narrowing, branches and would not be easy to maintain, as for each new diagram type, new ModelState would be needed to be added in a specific spot(s) to update the solution.

The design decision was, to create a thin and stable surface - the abstract model state - called `BaseDiagramModelState`, which exposes the services, all of the handlers (and also other classes, which were switched to abstract) depened on:

- Semantic context
- AST and ID index
- Transaction/command gateway
- Selection and UI context
- MetaInfo access

This enables the generic operation handlers, which can run unchanged across all diagrams, selected only by element type policies. The state shape drift is reduced and consistent semantics are kept for patching, (de)serialization and naming across tools. The testability is much better as the handlers can be unit tested against the abstraction and the specific diagram stuff can be covered by small adapter tests. And the most important is, that the extensibility is much better and simpler, as adding a new diagram means creating a new specific model state, which needs to extend the base diagram, without the need to change or rewrite handlers.

This works for these reasons:

- **Facade**: This abstraction of the model state acts as a facade, which through a single entry point exposes needed services as index, serializer, model service etc. The handlers do not have to know specifics and detail to each diagram type and therefore reduces its complexity.
- **Dependency Inversion**: Handlers do not rely on specific and concrete diagram states, which can then later be injected at runtime, but on the abstract interface. Thanks to this the whole system is more flexible and changes or extensions to the code are possible with no need to change the handler code.
- **Decoupling**: Generic handlers do not know which diagram they are operation on as they see only the abstraction. This allows the same handler logic to be reused across different diagram types.
- **Testability**: The abstracted base model state allows to test handlers using simple mock states. As a result tests do not require whole diagram state, what speeds up feedback and simplifies maintenance.

4.4 Validation & Derived UI/Navigation

4.4.1 Decorator-Driven Validation Strategy

As the users are giving input into the tool, logically a validation of this input is much needed. That was the reason, why it was decided to add checking of what they put into the tool and constraining some attributes.

Why the tool needed validation? The lack of validation and wrong inputs from user could lead to invalid AST states, which were routinely persisted:

- Names do not follow needed and used conventions, for example class names written in lowercase instead of being capitalized.
- Entities can also miss required semantic data, as for example an interface without a name
- Arrays could extend their intended limits
- and many more...

These errors could typically appear late, what could leave to some parts of the tool crashing late, rather then giving an error message about incorrect input.

The following **approach** was therefore chosen, to build validation into the tool and to keep it in sync with others added features:

1. **Declarative constraints in the `def.ts`**: To continue using our single source of truth `def.ts`, the decision was met to put the validation constraints into this file, as it already contains all of the elements with all of the attributes, so it is logical to include them here. In order to avoid unnecessary reimplementation, it was decided to use already defined TypeScript library (class-validator). This library allows to set constraints to different data types using decorators and then validate the entities based on these. This solution also helps the readability, as the rules (decorators) sit directly on the domain properties - so literally next to the thing it constraints.

```
1        export class Class extends Entity {
2
3              @Matches(/^[A-Z]/, {
4                message: "First letter of class name must be
   uppercase.",
5              })
6              @MinLength(5, { message: "Class name must be at
   least 5 characters long" })
7              @LengthBetween(5, 10, { message: "Class name must
   be 5-10 characters" })
8              name: string;
```

```
9                isAbstract: boolean = false;
10               @ValidateIf((o) => o.isActive === true)
11               @ArrayMaxSize(3, {
12                 message: "Active classes must declare at most 3
    properties.",
13               })
14               @path
15               properties?: Array<Property>;
16               @path operations?: Array<Operation>;
17               isActive?: boolean;
18               visibility?: Visibility;
19           }
20
```

Listing 4.4: Example of an entity, which has validation decorators on it

Listing 4.4 shows the readability of the constraints, but also the single specific validation decorators:

- @Matches - Using regex checks the string,
- @MinLength
- @LengthBetween - this is a custom validator, to demonstrate, that it is possible and can be created for future needs.
- @ArrayMaxSize
- @ValidateIf - First this decorator must be true, in order to check the decorator under it, so in listing 4.4, the max array size will be checked only if isActive is true,
- and many more provided by the TypeScript library class-validator.

There are default validators, custom validators and conditional constraints.

2. **Code generation**: Based on this configuration in `def.ts`, the validator generator create needed artifacts in order to proceed with validation. It parses the `def.ts` and collects only the entities and only those properties, which actually carry some constraints. It also collects only those used decorator imports from class-validator and custom validators.
   - It then emits minimal DTOs - validation elements, only with those properties, which were constrained in some way.
   - It also generates a validator function, which uses guards from genereted AST reflection `ast.ts` (isClass, isEnumeration) to select the correct DTO and then calls class-validators validateSync function.
3. The generated validator function is then called inside of ModelService's PATCH operation. Here, the validation is executed, if it passes, the function proceeds, otherwise it stops and the invalid states are never persisted.

4. Error messages are displayed to the user via tool's UI, with error messages provided in the `def.ts`.

This solution satisfies the following quality attributes and that is why this works:

- **Single source of truth**: The rule is defined only once at one place and all other parts of the approach consume this truth.
- **Maintainability**: Adding new or changing a constraint is very simple, as only one line of code is needed, with human-readable decorators. Everything else regenerates automatically.
- **Correctness and safety**: Invalid states are blocked at the PATCH operation pre-commit, so only valid states go through.
- **Consistency and UX**: The constraints and messages are the same throughout the entire tool and the declarative messages make it clear to the user, what needs to be corrected.
- **Developer experience**: No need for code duplication and constraint are discoverable by reading `def.ts` - easily readable and code-friendly.

Important to mention about the **generated code** is, that it is minimal and concise. The file, which creates the minimal DTOs, declares only fields, which are being constrained and also imports only needed decorators, which are used in the DTOs. Another generated file is a simple switch-like dispatcher function, which uses already existing guards from (also generated) `ast.ts` and is easy to read and debug.

These are **alternative solutions**, which were considered, but rejected:

- **Imperative checks in handler:** These would be closer to the edit location and could also access the context of the UI. However, the logic would have been duplicated across multiple files, which would be hard to keep consistent. This was rejected, because we decided to use a generic model-driven solution.
- **External validation registry:** This would improve the readability of already long `def.ts`, it would explicitly separate the concerns and everything could be organized by domain area. However, this could be taken as extra steps and duplicated code, what could lead to differences with `def.ts`. This idea was rejected, as the decorators used in `def.ts` are pretty readable and keep the file structured.

To sum this up, new validation pipeline executed using decorators was introduced. It starts in the `def.ts`, which already has all types which can be validated. From this file minimal DTOs are generated, which are then used in a centralized pre-commit check. This keeps the model always valid and the constraints are declared only once next to the domain properties.

4.4.2 Property Palette and Outline Generation

The property palette is the part of the tool, which allows the user to edit selected Element. In Figure 4.2 the property palette of a Class element with the name Professor is displayed. Below it is the outline view, which shows all elements in the diagram, also with their properties in a tree view.

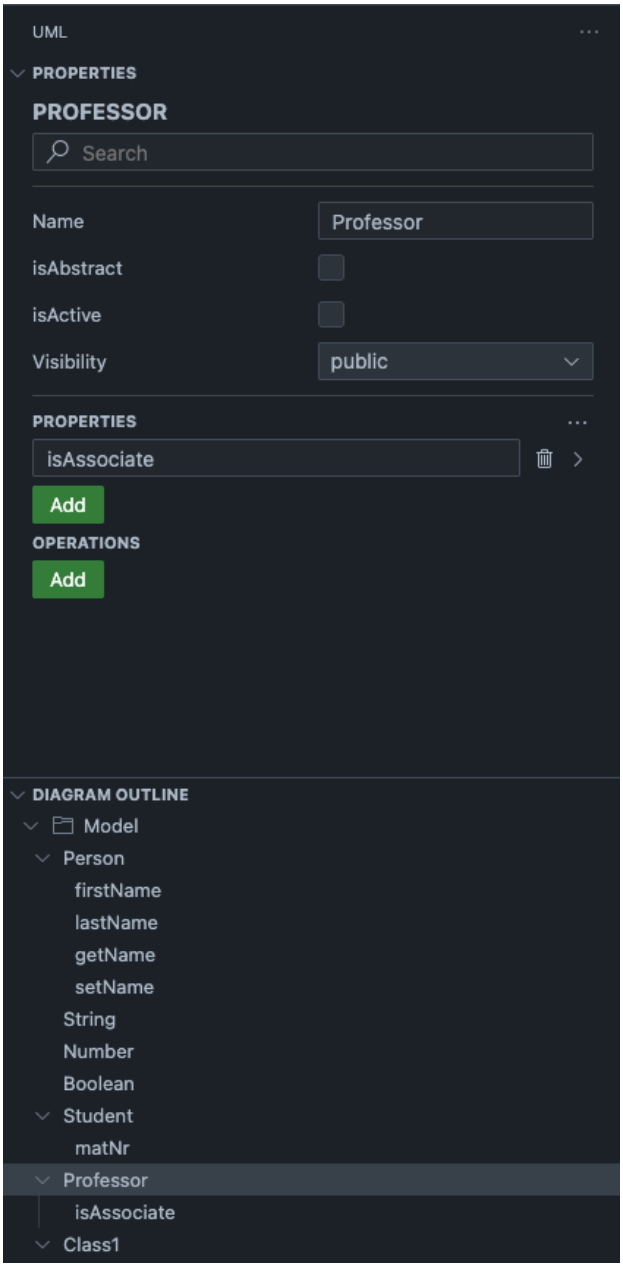

Figure 4.2: Screenshot of the Property Palette and Outline view in bigUML tool

This functionality was already built in the version from Jäger. The issue with it, was its strong duplicity, weak possibility for code maintenance and time expensive extension. The **problems** associated with editing (Property Palette) and navigation (Outline) are as follows:

- The code needed for property palette and outline navigation was located in many manually created handlers, which were spread across diagrams for all possible elements. These were also duplicated, as some diagram types used the same element, but code for it was located twice in different spots. Also the difference between single property handlers (e.g. `ClassPropertyPaletteHandler` and `InterfacePropertyPaletteHandler`) was minimal and the code was very similar.
- An issue also was, that the code in the single element property handlers was very dependent on the types and their properties. When something was changed - property added or renamed etc. - in the definition, also the property palette handler needed to be adjusted. This could lead to mistakes and inconsistencies, because the developer had to always think about this coupling.
- The very same issue was also with the outline, where if something was changed in the definition - as a new element or different naming for child - the outline action handler must have also been updated.
- Review and testing was very demanding as the diffs were mechanical and noisy, what hindered the real intent behind a change.

In the Listing 4.5 we can clearly see, that the elements can very simply be derived from the definition file, as each variable type is mapped to some type of tool in this handler, as for example string and number to text; or boolean to bool. The only not explicitly clear type is the `.reference`, but this is also strongly derived type.

```
1         export namespace ClassPropertyPaletteHandler {
2           export function getPropertyPalette(
3             semanticElement: Class,
4           ): SetPropertyPaletteAction[] {
5             return [
6               SetPropertyPaletteAction.create(
7                 PropertyPalette.builder()
8                   .elementId(semanticElement.__id)
9                   .label((semanticElement as any).name ??
    semanticElement.$type)
10                  .text(semanticElement.__id, 'name', semanticElement.
    name, 'Name')
11                  .bool(
12                    semanticElement.__id, 'isAbstract', !!
    semanticElement.isAbstract, 'isAbstract',
13                  )
14                  .reference(
15                    semanticElement.__id, 'properties', 'Properties',
16                    (semanticElement.properties ?? [])
17                      .filter((e: any) => !!e && !!e.__id)
18                      .map((e: any) => ({
19                        elementId: e.__id,
20                        label: e.name ?? '(unnamed property)',
21                        name: e.name ?? '',
22                        deleteActions: [DeleteElementOperation.create
    ([e.__id])],
23                      })),
24                    [
25                      {
26                        label: 'Create Property',
27                        action: CreateNodeOperation.create(ModelTypes.
    PROPERTY, {
28                          containerId: semanticElement.__id,
29                        }),
30                      },
31                    ],
32                  )
33                  ...
34                  .build(),
35              ),
36            ];
37          }
38        }
39
40
```

Listing 4.5: Simplified code of a Class Property Palette Handler

The **approach** to solve this problem is to treat both - the property palette and outline - as a projection of the same semantics from the definition file `def.ts`. The palette is just a projection of all attributes and contained parts an element has in the definition, whereas the outline is a tree structured navigational projection of the containment and types.

Both use a uniform request handler with similar structure:

1. validate the action
2. resolve semantic element through model index
3. short circuit on invalid inputs
4. dispatch using AST type guard (`isClass`, `isEnumeration`, etc.)

The individual palette handlers are derived from the definition in `def.ts` from property kinds in a deterministic way:

- text and numbers (string and number) map to `.text`,
- booleans to `.bool`,
- types like enum to `.choice`,
- and * containments to `reference` lists

Outline nodes are also derived from the `def.ts` from the containment structure. The top level specific entities, which are under the "model" root become "parents" and each * containment becomes a child.

The logic in all of these files stays the same. A difference from the previous solution is, that all of the element Property Palette Handlers (used in all diagrams) are created in one place and then the request action handler of all diagram types, import the same handlers, which they need. This means that there does not have to be two e.g. `ClassPropertyPaletteHandler` files, when class diagram needs it and also the package diagram needs it. Only one file needs to be generated as these would be completely same files by content.

All these files:

1. Property palette handlers for all elements,
2. request palette action handler for all defined diagram types and
3. request outline action handler for all defined diagram types

are generated by a generator, which takes information solely from the `def.ts` and based on these generates the files. There is also an option to skip some property from the property palette - this is done by a decorator in the `def.ts`.

The following quality attributes are met using this solution:

- **Maintainability:** Because everything is co-located and generated from the single file `def.ts`, which acts as a single source of truth, a change in semantics updates both Property Palette and Outline, without need for duplication.
- **Correctness:** For the same reason, because both views are derived from one structure, which is also an underlying model for the whole the diagram, the property palette cannot expose fields the model does not have and same for the outline - it cannot invent nesting, which does not exist.
- **Developer experience:** No need for "glue" code and clear structure of handler skeleton, which is also predictable means, that developers can focus on the semantic changes or additions and not wiring. The need to remember all spots, where something needs to be fixed, when some new element/property is added or changed exists no more, as it is needed only to be described once and both views follow.
- **Performance and Readability:** The code generation scales with the number of properties and contained items in O(n) and uses the already available model index. No extra passes or reflexion are needed. Files have a clear structure, there are no duplicated files and everything is nice and tidy in the generated folder.

Possible **alternatives** for this solution were considered, but rejected because the chosen solution is the best fit:

- Forms, which would be written manually for each element and custom outline trees, so basically the current solution. The reason for rejection was already explained, even though this would bring maximal flexibility but very high code duplication and constant risk of possible drift from the model. Mechanical changes would also mean expensive reviews, because of the amount of code.
- A single monolithic runtime handler was also taken into consideration. It would be driven by a deep reflection. On one hand, this would lead to fewer files, but unclear and chaotic code flow with many lines of code. It would be also harder to test in isolation and in case of special cases could break very easily.

While the selected design brings clear benefits, it also comes with certain **trade-offs**. The following list outlines these drawbacks and decisions taken to reduce the negative impact - mitigations:

- The per element palette UX is not so descriptive by default, because the introduced mapping favors consistency over small details.
  - Mitigation: Possibility to skip specific fields from the generated palette and possible supply of targeted custom control, where default is insufficient.
- Icons and labels in some cases might not be correct, as the naming might be slightly different in some domains.
  - Mitigation: Add label or icon details to new decorator, which transports this information to the generated files.
- Generated files do not decrease the file count as they are still needed per element and per diagram and therefore the file count is still relatively high.
  - Mitigation: Generated files are separated under a dedicated generated folder and this lets them be treated as build outputs, which can be ran through quickly by reviewers.

To sum up, by deriving both the property palette (editing) and the outline (navigation) from one file with consistent semantics, the code duplication is removed, UI stays consistent and updated with the model and additions/edits are turned into small and declarative changes.

4.5 Generator-Driven Scaffolding & Monorepo Architecture

4.5.1 Yeoman Generators

The main **problem**, which this subsection needs to solve, is that adding a new domain element meant touching multiple places in the code base, what could very quickly lead to drifting out of sync between single components. This was like a repetitive glue, which could lead to many mistakes.

Important information about diagrams was stored inside of specific handlers, which were on one side too repetitive and duplicated, but on the other side could become out of sync very quickly. The switch to the generic handler for all elements also meant, that this change was very needed, as element specific details could not live in element specific handler anymore. Hidden rules like where can I create X under Y or what defaults does X start with need a new place to live in.

The switch statements and copy-paste templates were also easy to get wrong. The onboarding process for new developers, who come into the project was long and hard, as there were many conventions and processes to be learned, before being productive. Every small change in the model definition lead to risk, as long lists of hand-written code needed to be checked, if they do not need also an update.

In the version from Jäger there is already a baseline Yeoman scaffolding and project wiring. It generated the core structure and the parser for our definition file was also already created. This baseline was needed to be extended, as new needs were created and therefore new generators were needed to be added and parser adjusted to read the newly updated `def.ts` file.

To solve this problem, the following approach was introduced:

- As already introduced and described in Section 4.2, single source of truth plays an important role in generating. The model declarations and decorators are taken as authoritative and then parsed once, so that all generated files can rely on it deterministically.
- For the handlers we need some deterministically generated properties. These are derived from the `def.ts`. Following code generators are introduced:
  - **Creation-path mapping:** The generator scans all parent properties, which are marked with `@path` decorator, then computes allowed child types for each types via inheritance and create a function which acts as a pure lookup used in the creation handler `getCreationPath(parentType, childType)`
  - **Default-value mapping:** This generator either derives the default values from its explicit values or takes the implicit default value for the specific type. This creates a long map for all properties in all types and creates a function `getProperties(elementTypeId`, which returns `property, defaultValue` entries, used while creation of the elements.
  - Other generators, that were already introduced, as validation, property palette and outline.
- Next a thin CLI orchestrator is needed, with a `generate` command, which writes the creation mapping and default value mapping into a dedicated `yo-generated` folder and then executes a registry of contributions (with other generators).
- Generators produce stable and formatted files (using prettier) with "DO NOT EDIT" headers, which enables clean diffs and safe regeneration.
- Finally, what was the goal of this approach is that the generic create handlers call the newly generated lookups with all needed information. Hard-coded rules are no longer needed, which reduces branching and makes behavior uniform.

The following quality attributes are delivered thanks to this solution:

- **Consistency by construction:** `def.ts` sets the pace with declarations, which drive the model structure, grammar, validation and the default value and path maps. This leads to fewer mismatches, as everything originates from the same source.

- **Maintainability:** Changes and additions need to be updated only in one place and are regenerated everywhere where needed. This means that small changes in semantics propagate safely and developer does not have to "hunt" for all switch cases and potential mismatches across whole code base.

- **Correctness and Testability:** Simple mappings and lookup functions are simple to test. All the information about default values and child sets live inside of one file and are easy to check.

- **Simple workflow and deterministic build:** Developers need to know only one simple workflow - update the `def.ts` and run the generator, instead of many files, where changes are needed for one modification. Stable file location and formatting enable great readability and minimize chaos in code base.

Using this approach, LOC touched per new element drops drastically from the current version, as the only manual change needed to be done by the developer is an update of `def.ts` file. The rest is generated. In comparison, as mentioned before, also the handlers, property palette, outline, grammar etc. would need an update and new LOC by the developer. The create handler complexity has shrunk and it is possible to make it completely generic and abstract, thanks to the generated values. Number of bugs caused by model information not being in sync is also minimal using this system and the generation time for all of the cases (default values, creation paths, validation, property palette, outline) is negligible in comparison to manual edits a developer needed to make.

Possible **alternatives** to this solution are as follows:

- The solution described by Jäger, with lots of manual boilerplate. It may be very simple at the beginning, but diverges quickly; and reviews and changes turn into cognitively heavy process with many steps to remember.

- Ad-hoc templating without a normalized model. It feels fast at first, but each template invents its own assumptions. Rules for inheritance and decorators may get duplicated and drift from the baseline. All changes in schema mean going through many files. Small inconsistencies can slip through and reviews become hard and more like guessing.

- Runtime reflection instead of codegen. This could reduce repository noise and may seem flexible, but the complexity is shifted to runtime. Cold starts and per request overhead would grow strongly. Artifacts are not diffable in PRs, which means that CI/reviews miss changes. Versioning and testing become harder and environment-dependent behavior will also play some role in negative meaning.

There are not many **trade-offs** to this solution, however to name few:

- Generator must be ran after each change of the `def.ts` file. If this is not executed, it may lead to errors during runtime and inconsistencies. This can be mitigated with automatic regeneration every time a build runs and caveat in README file.
- Central defaulting may not fit every edge case, where opinonated defaults may make better sense. This can be easily mitigated by opt-out decorators or some hooks in handlers, which can then be overriden in the subclasses for specific diagrams.

To sum up, domain intent is defined only once in the definition file, what allows to generate the maps actually needed by the handlers. Handlers become thin and simple, behavior is uniform and changes are applied from the declaration instead of manual stitching. Code generation reduces code duplication, improves code readability and makes it more readable.

4.5.2 Package-per-Concern Monorepo

To keep the generators organized and clean, it is important to think about the structure, how the generation is done. It is important to note, that this generative process can in future enhancements of the tool get expanded and more generators can be added, in order to simplify the processes and structure. Monolithic and loosely structured repositories could couple concerns and make the reuse and testing more difficult.

Adding a new projection or generator would require touching of central code and dependencies must have been entangled. The boundaries were unclear.

The **approach** to solve this issue is following:

- **Split the logic by concern into small workspace packages**, to clearly separate different code parts:
  - `generator-langium-model-management`: Original part of the generated created by Jäger, extended in this current project. It includes a simple CLI, which executes the whole generation including parsing and validatin of the `def.ts`; generates the core artifacts as the grammar, serializer, validation files, default values and creation path mappings; and then runs the registry of these contributions.
  - `model-management-common`: Acts as a kernel contract package - it defines shared types and the plug-in port, which is implemented by all of the generators. It also contains small utilities. By centralizing these abstractions, it enforces Dependency inversion as the core depends on interfaces, not implementations.
  - Additional concern specific generators - specifically we have `property-palette` and `outline`, which live as separate packages and implement the common contribution contract from the `model-management-common`.

- **Registry pattern for extensibility:** As mentioned, all contributions implement the same `codeGeneration` function, which returns an array of path and content for all generated files. The CLI only needs to register needed contributions and execute them using `registry.register(contribution)` and `registry.execute(params)`.
- **Versioning and boundaries**: Stable and minimal public surfaces, and lots of internal code remains private. The code dependencies have clear directions, where CLI depends on common contracts, contributions depend on common and not on each other. The generated outputs are grouped under `yo-generated` in consumers to avoid mixing the auto generated code with editable code.

This solution works because of these reasons:

- **Separation of concerns**: Each package has one task to do. The single generators are clearly split across folders and the mental model is simpler. The changes to single packages are also localized and clear to track.
- **Reusability**: If new domain is created, the CLI still generates correct mappings, which can later be used by the handlers. No edits are needed in the CLI.
- **Evolvability**: Extra projection, for example for Legend means, that you only need a small package, which implements the `ModelManagementContribution`. This must be registered in the registry, so only single line added in the CLI.
- **Testability**: Small packages compile more quickly and are easier tested. Contracts can also be mocked in isolation and failure modes are clearer.

What can be observed is that the PR size and the review time with this extension shrinks heavily, because changes are done only in one package per concern at a time. As mentioned, new diagrams and domains can be added by wiring the registry and no forked CLI logic is needed. When new generator is needed, only new package with one register in the registry is needed.

Another **alternative solution** would be one big package, where the initial setup would be minimal, however it could quickly get very chaotic, reviews would get harder and builds would be slower.

A small and typed contract and a CLI driven by registry enables us to grow a number of generators in one monorepo. All of the concerns stay isolated, can be very easily reused and adding of a new generator does not mean changing of the core logic, only registering new contribution.

CHAPTER 5

Implementation

In this Chapter, the implementation of the described concepts will be presented.

5.1 Definition file `def.ts`

The file, which acts as a single source of truth should make the code more readable, the maintenance of the code simpler and keep all elements that needs to be adjusted in one place.

The implementation of this file builds on the file that was created in the project version from Jäger, but adds some customization to it. This file is then used by all of the generators defined in this project.

5.1.1 Decorators

The biggest change and addition to the `def.ts` file are the decorators, which are then used for different parts of the tool. In the Table 5.1 all of the currently available decorators are listed, also with a short description and their usage.

In Listing 5.1 we can see the definition of two entities - `Property` and `Operation`, which have few attributes removed in order to keep it simple. In this listing we can clearly see the use case of the decorators, but also the main and essential function of the `def.ts`, which was already provided in the previous version. We can see that elements are defined as classes, with attributes of different types, as needed for the UML diagrams.

When it comes to the decorators, few of them are used in this definition. For both of the elements the decorator `@noBounds` is used. That is because both of them do not have a graphical representation, what means that the create handler must not create a position and size value for them.

Decorator	Description & Usage
Class (Entity) Decorators	
`@root`	Marks the class as the entry point (root AST node) of the model; it is used on the top-level container `Diagram`.
`@withDefaults`	Enables default value propagation at creation time. Without it, only required or explicitly defaulted fields initialize; with it, optional fields also get implicit defaults, based on its type (e.g., `false` for boolean, `0` for number or `"PUBLIC"` for Visibility).
`@noBounds`	This is for entities with no size and position in the diagram (e.g., `EnumerationLiteral`, `Property`) and it skips the layout-related configuration.
Property Decorators	
`@path`	Marks a property as a containment path (child container) in the model hierarchy; used by the creation-path generator to decide which elements can be created under a parent.
`@crossReference`	Marks a property as a non-containment reference (by name/ID) to another model element; common for relationships or type references.
`@dynamicProperty('X')`	Used with `@crossReference`; instructs tooling (e.g., property palette) to provide a dynamic dropdown of all available elements of type `X` for convenient linking.
`@skipPropertyPP`	Excludes the annotated property from the property palette even if it exists on the model.
Validation Decorators (example of few decorators from TypeScript library class-validator)	
`@MinLength`	Makes sure a string has at least the specified number of characters.
`@ValidateIf`	Runs the validation only when the given condition is true (conditional validation).
`@ArrayMaxSize`	Limits the maximum number of elements allowed in an array property.
`@Equals`	Makes sure the field value equals the specified constant.
`@LengthBetween` *(custom)*	Validates that a string's length lies within an inclusive lower/upper bound; this decorator shows the possibility of extension by own custom decorators.
AST Mapping Decorator	
`@astType('X')`	Overrides the default AST node type name for the class with `'X'`; useful when the auto-derived name is insufficient or conflicts with diagram/edge handling.

Table 5.1: Overview of decorators in `def.ts` and their description and what are they used for

A difference in class decorators between these two is the @withDefaults decorator on Operation. In this specific case, this means, that in Property only the defined or required values are initialized on creation of the element. That means, that the multiplicity and aggreagation will not get initialized on Property. On the other hand, because Operation has the @withDefaults, all of the attributes will get initialized and if none explicit value is provided, then a default value for its type will be taken.

There are also the dynamicProperty and crossReference decorators at the propertyType in Property. This instructs the property palette to create a dropdown of type DataType and marks this field as a non-containment reference.

In the Operation on the parameters, there is the @path decorator. This tells the creation handler (via generated file), that the property is a containment path and a child container should be created for the parameters in the model hierarchy.

```
1         @noBounds
2         export class Property {
3           name: string;
4           isDerivedUnion?: boolean = false;
5           visibility?: Visibility = "PUBLIC";
6           multiplicity?: string;
7           @dynamicProperty("DataType") @crossReference propertyType?:
      DataTypeReference;
8           aggregation?: AggregationType;
9         }
10
11        @noBounds
12        @withDefaults
13        export class Operation {
14          name: string;
15          isQuery?: boolean;
16          visibility?: Visibility;
17          @path parameters?: Array<Parameter>;
18        }
19
```

Listing 5.1: Simplified definitions of Property and Operation

5.1.2 Diagram Types

Small addition to the def.ts are also the element unions, which serve as an overview for the generators. They make it clear, which elements are needed in the generation process for the specific diagram type. In Listing 5.2 is the definition of type ClassDiagramElements, which explicitly names all of the Elements a Diagram Type can use. These are then used during the Property Palette and Outline generation, because the specific Request Action Handlers need to know, which elements to include.

```
type ClassDiagramElements = | Enumeration | EnumerationLiteral |
Class | AbstractClass | Interface | Package | Property | Operation
 | Parameter | DataType | PrimitiveType | InstanceSpecification |
Slot | LiteralSpecification | Relation | Abstraction | Dependency
| Association | Aggregation | Composition | InterfaceRealization |
 Generalization | PackageImport | PackageMerge | Realization |
Substitution | Usage;

```

Listing 5.2: Definition of `ClassDiagramElements`

5.2 Generic Operation Handlers & Model State Abstraction

The creation of generic operation handlers was needed to remove code duplication and to simplify definition of the default values. In previous version from Jäger, each element needed its own file for a creation, where much of the code was duplicated and also definitions for default values were spread across multiple files. As we can see in Listing 5.3 from the `CreateClassOperationHandler`, the values for a class element are hard-coded in the `value` prop, even though they seem very generic and have no special semantic value in them.

```
createClass(operation: CreateNodeOperation): string {
    const containerPath = this.getContainerPath(operation);
    const newName = findAvailableNodeName(this.modelState.
semanticRoot, 'NewClass');
    const patch = JSON.stringify({
        op: 'add',
        path: containerPath ?? '/diagram/entities/-',
        value: {
            $type: 'Class',
            __id: createRandomUUID(),
            name: newName,
            isAbstract: false,
            properties: [],
            operations: [],
            visibility: 'PUBLIC'
        }
    });
    return patch;
}

```

Listing 5.3: `createClass` function in `CreateClassOperationHandler` in Jägers version

This needed to be fixed and was done so using generated content from a definition file. More to that will come later, but in this section we explain, how the creation operation handlers (but also others) were generalized.

5.2.1 Abstract CRUD Handlers (Template Method)

As discussed in the concept Chapter 4, all the create operation handlers for all elements were compacted into two abstract generic operation handlers:

- `AbstractCreateNodeOperationHandler`
- `AbstractCreateEdgeOperationHandler`

As these handlers are used for all diagram types, they must be abstract, so that each diagram type can then have a subclass, with the minimal implementation for the specific type.

The same principle was then applied to all the other operation handlers for editing, deletion, changeBounds etc. For these a new `common` folder was created, where all of the abstract handlers now live.

But back to the abstract create node operation handler, which works like a **template algorithm**. The steps are allways the same, but the content of the operations or parameters change for different element types:

1. First the handler receives a `CreateNodeOperation` and verifies the `elementTypeId` is supported, by the subclass's `elementTypeIds` (as can be seen in Listing 4.3). The input is normalized.
2. Then it is resolved, there to insert the element, by asking the index for container and using the generated `getCreationPath(...)`
3. Next the node's value is created, with id, type, newly generated name and new defaults are applied from the generated `getProperties(...)`
4. One semantic `add` patch is emitted for the node
5. In case the type is visual (meaning it was not decorated with the `@noBounds` in the definition file) also size and position meta data are added and it is snapped to the grid.
6. Finally the semantic and meta data are combined into one single atomic command

```
    const allProps = getProperties(operation.elementTypeId);
        for (const { property, defaultValue } of allProps) {
            if (property !== 'name' && nodeValue[property] ===
    undefined) {
                nodeValue[property] = defaultValue;
            }
        }
        return JSON.stringify({
            op: 'add',
            path: containerPath,
            value: nodeValue
        });

```

Listing 5.4: Code snippet, to compare, how the default values are created new approach.

In the listing 5.4 it is clearly visible, how now the input of the default values is not hard coded into the separate classes, but it is derived using the function `getProperties`. The values originate from the generated code, which is generated based on the definition file. This solves the maintenance of the code, as everything is now controlled from one file.

The same is valid also for other policies, which are injected via hooks into this generic handler and can also be added in the future. It means that not only the default initializers, but also the creation path resolution comes from a generated mapping based on the definition file. Also the naming and grid snapping is injected. This eliminates the per element classes as they are no more needed, because we can centralize the information in one central file and derive everything from it.

Other abstract handlers The same principle was also applied to the other handlers, which were duplicated for each diagram type and needed almost identical code for their function. Here belong handlers like:

- `AbstractChangeBoundsOperationHandler`,
- `AbstractDeleteOperationHandler`,
- `AbstractUpdateElementPropertyActionHandler`,
- `AbstractUpdateOperationHandler`

The approach here was same as in the create node and edge handlers, just applied to the specified needs. The algorithm needed for the specific operation was generalized and put into the abstract class. This allows the specific subclasses for these handlers to be minimal, what allows simple maintenance and extensibility.

Hooks or in other words small overridable methods in the abstract handlers, which let subclasses adjust some information without changing the control flow. With these, the skeleton of the handler stays still the same and stable, however the subclasses can now plug some additional small logic where needed.

```
1    @injectable()
2    export abstract class AbstractUpdateOperationHandler extends
     OperationHandler {
3        override operationType = UpdateOperation.KIND;
4
5        abstract override readonly modelState: BaseDiagramModelState;
6
7        override createCommand(op: UpdateOperation): Command |
     undefined {
8            ...
9        }
10
11       protected createUpdatePatch(operation: UpdateOperation): { op:
      'add' | 'replace'; path: string; value: unknown } | undefined {
12           ...
13           const value = this.transformValue(operation, element);
14           const opKind = this.chooseOp(element, operation.property,
      value);
15           ...
16       }
17
18       /** transform incoming value if needed (default: return as-is
     ). */
19       protected transformValue(operation: UpdateOperation, _element:
      any): unknown {
20           return operation.value;
21       }
22
23       /** choose between 'add' and 'replace' (default: always '
     replace'). */
24       protected chooseOp(element: any, property: string, _value:
     unknown): 'add' | 'replace' {
25           return 'replace';
26       }
27   }
28
```

Listing 5.5: Simplified `AbstractUpdateOperationHandler`, which shows its overridable methods - hooks

In Listing 5.5 we can see the `AbstractUpdateOperationHandler` with all its methods. This is an example of a handler, which needed these hooks, to modify some the flow in some specific subclasses. For subclasses, which do not need to adjust this information, default value is returned (when the hooks are not overriden). There are two hooks:

- `transformValue` - its purpose is to normalize the UI payload into data, which is ready for the model,
- `chooseOp` - which prevents the replacing of non existent key errors, in case it could happen.

```
1    protected override transformValue(operation: UpdateOperation,
    _element: any): unknown {
2        const raw = operation.value;
3        if (typeof raw === 'string' && raw.endsWith('_refValue')) {
4            const refId = raw.slice(0, -'_refValue'.length);
5            const target = this.modelState.index.findIdElement(refId);

6            if (!target) return raw; // fallback: leave as-is
7            return {
8                ref: {
9                    __id: target.__id,
10                   __documentUri: target.$document?.uri
11               }
12           };
13       }
14       return smartCast(raw);
15   }
16
```

Listing 5.6: Overridden method `transformValue` in `ClassDiagramUpdateOperationHandler`

These are overridden for the Class Diagram subclass, but not for the Package Diagram. In the Listing 5.6, we can see, how the method is overridden for the class diagram. It is needed, because the class diagrams are reference heavy. The UI often returns lightweight tokens or strings for these, which must either be transformed to correct form or "smart-casted" (`"true"` → `true`, `"42"` → `42`). Package diagrams mostly edit only simple fields and hierarchical containment; and the cross element reference are rare - that is why the transformation is not needed.

Subclasses of abstract generic handlers The abstract and generic create node handler now lets the concrete implementations to be very minimal as it can be seen in the Listing 5.7. The only information that needs to be defined on top of the abstract handler are the `elementTypeIds` and the specific `modelState`. This simplifies the code structure strongly, as when new diagram type is added, only these two information need to be provided for the subclass.

```
    @injectable()
export class ClassCreateNodeOperationHandler extends
    AbstractCreateNodeOperationHandler {
    readonly elementTypeIds = [
        ModelTypes.ABSTRACT_CLASS,
        ...
    ];

    @inject(ClassDiagramModelState)
    declare modelState: ClassDiagramModelState;
}

```

Listing 5.7: Subclass of the create node operation handler for class diagram (part of the code which was already shown in Listing 4.3 is omitted)

5.2.2 Base Diagram Model State Interface

To make the generic GLSP operation handlers abstract and work across all diagram types, the `BaseDiagramModelState` needed to be introduced. It acts as a thin and facade over the GLSP's `DefaultModelState`, which is aware of the semantics. Its purpose is to simplify the access to:

- the Langium semantic model - `semanticRoot` and `semanticUri`,
- language services, which are needed by the handlers,
- and the `BigUmlModelIndex`, which resolves the graphical IDs back to AST nodes.

We can see these in the Listing 5.8 and also that the class exposes just the minimal set of mutation primitives (`sendModelPatch`, `undo`, `redo`) which are backed by a `ModelService`. This means the handlers can apply changes via patches or some history operations and they do not have to know the specific diagram they are operating on. The semantic concerns are separated in one place and it keeps the handlers free of diagram-specific wiring.

All of the generic handlers depend only on this contract and the specific model states as class or package, just bind the services and re-index after mutations. The logic does not have to change in the handlers.

```
    abstract class BaseDiagramModelState extends DefaultModelState {
      abstract index: BigUmlModelIndex;
      abstract semanticUri: string;
      abstract semanticRoot: Diagram;
      abstract modelService: ModelService;
      abstract semanticSerializer: DiagramSerializer<Diagram>;
      abstract nameProvider: QualifiedNameProvider;
```

```
8
9       abstract sendModelPatch(patch: string): Promise<void>;
10      abstract undo(): Promise<void>;
11      abstract redo(): Promise<void>;
12    }
13
```

Listing 5.8: Definition of `BaseDiagramModelState`

The usage of these is clearly visible in Listings 5.5 (line 5) and 5.7 (line 9). The `AbstractUpdateOperationHandler` uses the `BaseDiagramModelState` and uses its minimal interface and then in the specific subclass for the `ClassCreateNodeOperationHandler` the specific model state for the class diagram is injected.

5.3 Decorator-Driven Validation

The main goal of this implementation is to auto generate code needed for the validation based on the definition file `def.ts`.

5.3.1 Generated DTOs & Validator Dispatcher

A new simple generator was introduced next to all the generators, which using small ts-morph script scans the definition file. It looks for properties, which are decorated by the known validation decorators. Based on this, it builds a `ValidationInfo` object. The generation works as fo:

1. The algorithm starts with import harvesting - it goes through all the imports in the `def.ts` file and finds all those, which are either from the `class-validator` or from the custom validators (`validation/custom-validators/*`), so the custom rules are also automatically included. This is visible in Listing 5.9 in line 7. For this scraping of the `def.ts` a typescript library called `ts-morph` is used, which allows a easy manipulation with TypeScript code. For that a "Project" needed to be created (line 2) and source of the file we want to use added (line 3). After that, we can programatically control the file and manipulate with it. We use the function `getImportDeclarations()` to get all the imports and to filter just the correct ones we need from the `def.ts`.

```
1     function collectDecoratorImports(defPath: string) {
2       const proj = new Project({ tsConfigFilePath: path.join(
      process.cwd(), "tsconfig.json") });
3       const src = proj.addSourceFileAtPath(defPath);
4       const res: ValidationInfo["decoratorImports"] = [];
5       src.getImportDeclarations().forEach((imp) => {
6         const mod = imp.getModuleSpecifierValue();
```

```
7         if (mod === "class-validator" || mod.includes("/
    validation/custom-validators")) {
8           const names = imp.getNamedImports().map((n) => n.
    getName());
9           if (names.length) res.push({ from: mod, names });
10        }
11      });
12
13      return res;
14    }
15
```

Listing 5.9: `collectDecoratorImports` function from the validation generator

2. For each AST class in the `def.ts`, which has at least one decorated property (with validation decorators), a new DTO class called `${AstName}ValidationElement` is generated. Each DTO is a clone of the original class, and it mirrors the original property type and the optionality symbol (`?`). These DTO classes are needed later in the process. In the Listing 5.10 shows how these DTOs are created - `ts-morph` goes through all the classes, in these it goes through all the props, which are compared to the `class-validator` imports, to see if they have some validation decorators on them. Then these props are added to an array and if this array is not empty, the class can be added to the array of `entities` and will be created as the special minimal validation DTO.

```
1     function buildValidationInfo(defPath: string): ValidationInfo
      {
2       const decoratorImports = collectDecoratorImports(defPath);
3       const decoratorNames = new Set<string>(
4         decoratorImports.flatMap((i) => i.names)
5       );
6         ...
7       const entities: EntityInfo[] = [];
8       src.getClasses().forEach((cls) => {
9         const props: PropertyInfo[] = [];
10        cls.getProperties().forEach((prop) => {
11          const decos = prop
12            .getDecorators()
13            .filter((d) => decoratorNames.has(d.getName()));
14          if (decos.length) {
15            const typeNode = prop.getTypeNode();
16            props.push({
17              name: prop.getName(),
18              decoratorTexts: decos.map((d) => d.getText()),
19              typeText: typeNode?.getText() ?? prop.getType().
    getText(),
20              isOptional: prop.hasQuestionToken(),
21            });
```

```
22            }
23          });
24          ...
25          if (props.length) {
26            entities.push({
27              name: cls.getName()!,
28              dtoClassName: `${cls.getName()}ValidationElement`,
29              props,
30            });
31          }
32        });
33        return { entities, decoratorImports };
34      }
35
```

Listing 5.10: Entity discovery in validation generator

In the Listing 5.11 the generated DTO for element `Enmueration` is displayed. In the `def.ts` it has one property decorated with the custom validation decorator `@LengthBetween` - that is why this is the only property, that was needed in this `EnumerationValidationElement`, which is needed for the validation.

```
1     export class EnumerationValidationElement {
2       constructor(src: Enumeration) {
3         Object.assign(this, src);
4       }
5       @LengthBetween(3, 10, { message: 'Enumeration.name must be
    3-10 characters' })
6       name: string;
7     }
8
```

Listing 5.11: Entity discovery in validation generator

3. The generator needs to collect the AST import identifiers for the validation dispatcher - only for the referenced types (the generated DTOs). These are then imported using a single import from the also (sooner) generated file `generated/ast.js`. It also needs to import all of the generated DTOs.

4. The validator dispatcher then decides the AST type of a node based on the imports from `ast.js` (`isClass`, `isEnumeration`...) and validates it with the newly created minimal DTO of the same type and the class-validator function `validateSync`. Errors are then collected into the `error` variable to be later thrown as an error message. Listing 5.12 shows, how all the "is x" from `ast.js` are imported for all referenced entities, how all the created DTO validation elements are imported, how the `AstNode` is validate using the `validateSync` from the `class-validator` and how the errors are collected.

```
1   import { validateSync } from 'class-validator';
2   import type { AstNode } from 'langium';
3   import { isEnumeration, isClass, isInterface, isDataType }
    from '../../generated/ast.js';
4   import { EnumerationValidationElement, ClassValidationElement
    , InterfaceValidationElement, DataTypeValidationElement }
    from './validation-elements.js';
5
6   export function validateNode(node: AstNode): void {
7     let errors = [];
8     if (isEnumeration(node)) { errors = validateSync(new
    EnumerationValidationElement(node)); }
9     if (isClass(node)) { errors = validateSync(new
    ClassValidationElement(node));
10    }
11    if (isInterface(node)) { errors = validateSync(new
    InterfaceValidationElement(node)); }
12    if (isDataType(node)) { errors = validateSync(new
    DataTypeValidationElement(node)); }
13    if (errors.length) {
14      const msg = errors.flatMap((e) => Object.values(e.
    constraints ?? {})).join(', ');
15      throw new Error('Validation error: ' + msg);
16    }
17  }
18
```

Listing 5.12: Generated validator dispatcher

So to sum up, the validation generator creates two files:

- `validation-elements.ts`, which includes the minimal DTO validation elements, which are then used in the other generated file,
- `validator.ts`, which has the function `validateNode`.

This function (`validateNode`) is used in the `UmlModelService.patch` pre-commit hook, right before delegating to `super.patch`.

`UmlModelService.patch` accepts JSON Patch operations, then opens the UML document and for each operation it resolves the target via JSON pointer. For `add` and `replace` it applies the change on a clone and delegates it to `super.patch` to persist the changes. This workflow of the patch functionality already existed in the Jäger's version of the tool - now we just added the validation into it.

The validation runs inside the overriden `patch` method, right before the final `super.patch(...)` call. The function derives the `parentPointer` from the JSON

pointer and resolves the whole parent via `getNodeByPointer(root, parentPointer)`. The parent is then shallow-cloned, because the `validateNode(clone)` is only applied to clone, so that failures never change the real model and so they do not persist. On validation error the hook logs the failing operation and parent pointer, throws and the entire patch gets aborted. The user then gets a clear message about the data, that was not correct and did not come through the validation, without change of the model.

This cheap, local and pre-commit validation check prevents the data to be corrupted with unsifficient/invalid data and also provides precise and immediate feedback.

5.3.2 Custom Validators

Custom validators are project specific rules, which are still expressed as decorators on the newly generated DTOs. The standard decorators, which come from `class-validator` have use case and cover the common cases as string length, regex, arrays etc. But domain constraints like name policies per stereotype or conditional cardinatilites may need more advanced logic. To show, that this is possible using the TypeScript library `class-validator` and that it is supported in our pipeline, `LengthBetween` decorator was created as a small proof of concept. It behaves like a built-in rule from `class-validator`, even though it is custom and the generator automatically collects its import and uses it as if it was built-in decorator.

As the Listing 5.13 shows, the implementation is just a thin wrapper around the `registerDecorator`, which accepts min and max and opitonal `ValidationOptions`. It then checks the string length in validate and provides a customizable message.

```
1    ...
2    @ValidatorConstraint({ name: 'lengthBetween', async: false })
3    export class LengthBetweenConstraint implements
     ValidatorConstraintInterface {
4        validate(value: any, args: ValidationArguments): boolean {
5            const [min, max] = args.constraints as [number, number];
6            return typeof value === 'string' && value.length >= min
     && value.length <= max;
7        }
8
9        defaultMessage(args: ValidationArguments): string {
10           const [min, max] = args.constraints as [number, number];
11           return `${args.property} must be between ${min} and ${max}
     characters long`;
12       }
13   }
14
15   export function LengthBetween(min: number, max: number,
     validationOptions?: ValidationOptions) {
16       return function (object: Object, propertyName: string) {
```

```
17              registerDecorator({target: object.constructor,
    propertyName, constraints: [min, max], options: validationOptions,
     validator: LengthBetweenConstraint
18              });
19          };
20      }
21
22
```

Listing 5.13: Custom validator class - `LengthBetween`

The usage in `def.ts` is relatively straightforward. as we can see in the Listing 5.14, where we just set the min and max and the message, which should be returned in case of error.

```
1     @LengthBetween(3, 10, { message: 'Enumeration.name must be 3-10
    characters' }) name: string;
2
```

Listing 5.14: Usage of custom validator `LengthBetween`

At runtime `validateNode` evaluate this rule in the same way as it does with built-in decorators inside the `patch` pre-commit hook. In this subsection we wanted to show, that it is possible, to add more advanced rules in case it is needed, and also provide a simple example how such class should look like.

5.4 Generated Property Palette & Outline

To make the whole tool more maintainable and to utilize the one single source of truth `def.ts`, the property palette and outline files need to be auto generated. It is important to say, that property palette and outline was already functional in the version from Jäger. What we do, is make the whole process more maintainable, so that it is not dependent on the developer to update the property palette or outline when some new element or diagram type is added or updated.

The starting point is an explanation of what is generated:

- Per element palette builders:
  `yo-generated/property-palette/elements/<Entity>PropertyPaletteHandler.ts`
- Per diagram palette dispatcher:
  `yo-generated/property-palette/request-<diagram>-property-palette-action-handler.ts`
- Per diagram outline dispatcher:
  `yo-generated/outline/request-<diagram>-outline-action-handler.ts`

This structure guarantees stable names and predictable imports for other generated files. We generate three kinds of files into the `yo-generated` folder and we treat them as build artifacts. All diagram types reuse the same per element palette handlers, what as we already know removes the duplication, as dispatchers for all diagram types can import the same handler e.g. `ClassPropertyPaletteHandler`.

5.4.1 Property palette

The property palette generator needs to generate both - element handlers and the central dispatcher for all diagrams.

The element handler generation starts with the filtering of things, we do not want the palette UIs for. We need to remove types from the `def.ts`, which are not needed as property palette element handlers. We filter `Diagram` and entities which end with `Diagram`, `Relation`, `Entity`, `ElementWithSizeAndPosition` and types which are inheriting `MetaInfo`. This has as an effect that only specific domain nodes, which users can logically edit get the handlers.

Then we come to the creation of single element handlers. Each property becomes exactly one builder line via function `emitBuilderLine()`, which can be seen in the Listing 5.15. The function starts with additional filtering, based on decorators. It skips for the decorator `skipPropertyPP`, which is a decorator, which allows the user to deliberately skip a field, which they do not want to have in the property palette. The second skip is for the decorator `crossReference` and this is also skipped, because these properties are shown using other means.

Then the builder line continues with detemrinistic mapping rules, which create the actual handler based on the properties, the entity has:

- In case the property has the decorator `dynamicProperty(DataType)`, it gets mapped to `.choice` and the choices are defined by the parameter of the decorator. So in this type, the options would be `DataTypeChoices`.
- In case the property has some multiplicity (`=== "*"`, it gets mapped to `.reference`,
- and `string` or `number` are mapped to `.text`

```
1    function emitBuilderLine(prop: any, decl: LangiumDeclaration):
     string | undefined {
2      if (prop.decorators?.includes("skipPropertyPP")) return;
3      if (prop.decorators?.includes("crossReference")) return;
4
5      const dyn = prop.decorators?.find((d: string) => d.startsWith("
     dynamicProperty:"));
6      if (dyn) {
7        const typeName = dyn.split(":")[1];
8        return `.choice(
```

```
9             semanticElement.__id, '${prop.name}', ...
10          )`;
11        }
12        if (prop.multiplicity === "*") {
13          return `.reference(
14            semanticElement.__id, '${prop.name}', '${human(prop.name)}',
15            ...
16        }
17        if (first?.typeName === "boolean") return `.bool(
    semanticElement.__id, '${prop.name}',...`;
18        if (first?.typeName === "string" || first?.typeName === "number
    ")
19          return `.text(semanticElement.__id, '${prop.name}', ${first.
    typeName === "number" ? `String(semanticElement.${prop.name})` : `
    semanticElement.${prop.name}`}, '${human(prop.name)}')`;
20        return;
21      }
22
23
```

Listing 5.15: Builder emission (simplified)

Each element handler then follows the same template as it can be seen in Listing 5.16 and adds the whole builder line created in Listing 5.15 in the `$body` part.

```
1     export namespace ${name}PropertyPaletteHandler {
2       export function getPropertyPalette(
3         semanticElement: ${name}${sigExtras}
4       ): SetPropertyPaletteAction[] {
5         return [
6           SetPropertyPaletteAction.create(
7             PropertyPalette.builder()
8               .elementId(semanticElement.__id)
9               .label((semanticElement as any).name ?? semanticElement.
    $type)
10      ${body}
11              .build()
12          )
13        ];
14      }
15    }
16
```

Listing 5.16: Element template for property palette handler

This finishes the first generated part - the per element palette builders: `yo-generated/property-palette/elements/<Entity>PropertyPaletteHandler.ts`. In the next step, the per diagram palette dispatcher:

yo-generated/property-palette/request-<diagram>-property-palette-action-handler.ts gets generated.

```
1     const diagramAliases = declarations.filter(
2         (d) => d.type === "type" && d.name?.endsWith("DiagramElements")
3       );
4
```

Listing 5.17: Filtering of all diagram types

As we can see in the Listing 5.17, the types, which end with `DiagramElements` are used in this generation process and that is why it is important to define them for all diagram types in the `def.ts` (we can see example in Listing 5.2). Thanks to this, the generator knows, for which diagrams the central dispatchers should be generated and gets the name from it as in the for example `ClassDiagramElements`, the name is then `Class`. From this type it also knows, which elements belong to the specific diagram type, as they are defined in it as a union.

It then generates the needed template for all diagram types. All per diagram specifics collapse to two things:

- Which guards are imported and used
- Which dynamic choice lists are built from the index

All other information is shared between the diagram types.

In the Listing 5.18 we can see these two specifics in a shortened version of a generated property palette dispatcher. At the top, we can see the collection of options `dataTypeChoices` with the help of the model state and function `getAllDataTypes`. In the second part, there are the AST guards (`isProperty` and `isSlot`) for all elements a diagram type can have. These control the dispatch of the correct handler and also forward the computed `*Choices`, which were created in the first part of this generated file.

```
1     const dataTypeChoices = (this.modelState.index.getAllDataTypes
      ?.() ?? [])
2       .filter(i => i?.__id && i?.name)
3       .map(i => ({ label: i.name, value: i.__id + '_refValue',
      secondaryText: i.$type }));
4
5     if (false) {
6     } else if (isProperty(semanticElement)) {
7       return PropertyPropertyPaletteHandler.getPropertyPalette(
      semanticElement, dataTypeChoices);
8     } else if (isSlot(semanticElement)) {
```

```
9       return SlotPropertyPaletteHandler.getPropertyPalette(
    semanticElement, definingFeatureChoices);
10    }
11    // ...
12    return [SetPropertyPaletteAction.create()];
13
```

Listing 5.18: Short snippet from the property palette dispatcher (simplified)

5.4.2 Outline

Outline generation works with a very similar principle as the property palette dispatcher generator. In this case we do no need single element handlers, therefore only one file per diagram is created: outline dispatcher: `yo-generated/outline/request-<diagram>-outline-action-handler.ts`.

With very same technique the generator gets all of the diagram types so that it can create a dispatcher for each type.

It uses filter to add only entities, which transitively extend the `Entity` type and based on the union type `<Type>DiagramElements` (as shown in Listing 5.2), it knows, which element types to include. All of these and also their children are then added, e.g. as we can see in the Listing 5.19, `isClass` has `properties` and `operations` set as its children. It finds the children simply by the multiplicity of its properties - if it is an array, then it is a children.

```
1     if (isClass(entity)) {
2       node.iconClass = 'class';
3       node.children.push(
4         ...(entity.properties ?? []).map(child => ({
5           label: child.name, semanticUri: child.__id, children: [],
    iconClass: 'property'
6         }))
7       );
8       node.children.push(
9         ...(entity.operations ?? []).map(child => ({
10          label: child.name, semanticUri: child.__id, children: [],
    iconClass: 'operation'
11        }))
12      );
13    }
14
```

Listing 5.19: Short snippet from the outline dispatcher (simplified)

The generation of the property palette and outline simplifies the whole workflow, as it continues the narrative of a single source of truth. It gets all the information from `def.ts` and when some values are added or updated in there, everything stays in sync.

5.5 Default Values & Creation Paths Mapping

Other generators needed for high maintainability are the default values generator and creation paths mapping generator. These both also take the information from the `def.ts` file to simplify the creation of new nodes and edges in the diagrams. They also use the already mentioned decorators in the single source of truth file `def.ts`. These feed both mappings.

The goal of these generators was to remove the fragile and not very maintainable logic from the handlers into central, static, fast and typed lookup tables, which are always updated and in sync, when changes are done in the `def.ts` file. These generators output two files in the folder `yo-generated`:

- Default values mapping: `yo-generated/getDefaultValue.ts`,
- Creation path mapping: `yo-generated/getCreationPath.ts`,

which are both formatted using the shared `format()` function (using prettier) for stable diffs.

5.5.1 Default value mapping

The purpose of this generator is to produce per type property defaults, which are used when elements are being instantiated. As input it uses data from the `def.ts` file converted into `LangiumDeclaration[]` type, which was created already in the version from Jäger and we have expanded it to serve our needs. It includes all the information about types and its properties with decorators, optionality and literal defaults.

As output, the generator creates the already mentioned file with the following functions and constants:

- `getProperties(elementTypeId) → Array<property, propertyType, defaultValue>` - function used in the handlers, which provides an array of properties with their default values when new node or edge is created
- `isNoBounds(typeId) → boolean` - function, which forwards the information about visual representation of an element
- `getRelationTypeFromElementId(elementTypeId, upperCase)` - function, which returns relation type in correct form
- `defaultMapping` - constant mapping, with default values for each type and its properties
- `noBoundsClasses` - constant set of class names, which do not have a visual representation

- `astTypeMapping` - constant record of types, which have different AST name, than their name in the `def.ts`

The generator does these steps:

1. It starts with empty `defaultMapping`, `noBoundClasses`, and `astTypeMap`, which were just explained,
2. then it collects all the class level decorators as `@noBounds` and `@astType("...")` (normalized as `astType:<value>`
3. For each class then the array `entries` is computed and filled, while:
   - duplicate property names are skipped using the `seen` set,
   - in case the class does not have the `@withDefaults` decorator (which was already explained in Table 5.1) and the property is optional and has no explicit default value, it is also skipped. In this case the implicit default value will be taken.
   - record `property, propertyType` is created,
   - if the explicit default value was provided, it is added to the property
4. The results are collected into the payload of form `defaultMapping, noBoundsClasses, astTypeMap` , which are then injected into the template

Not explicitly defined defaults are calculated at runtime, using the function `getProperties` as displayed in Listing 5.20, which is in the generated file and it uses the auto-generated `defaultMapping`.

```
1   export function getProperties(elementTypeId: string):
    DefaultMappingEntry[] {
2     ...
3     const entries = defaultMapping[parentType] || [];
4     return entries.reduce((acc, e) => {
5       if (e.defaultValue !== undefined) {
6         if (e.defaultValue === '[]') acc.push({ ...e, defaultValue:
      [] });
7         else acc.push(e);
8         return acc;
9       }
10      switch (e.propertyType) {
11        case 'string': return acc;
12        case 'boolean': acc.push({ ...e, defaultValue: false });
    return acc;
13        case 'number':  acc.push({ ...e, defaultValue: 0 });
    return acc;
```

```
14          case 'Visibility':  acc.push({ ...e, defaultValue: 'PUBLIC'
    });       return acc;
15          case 'Concurrency': acc.push({ ...e, defaultValue: '
    SEQUENTIAL' });  return acc;
16          default: acc.push({ ...e, defaultValue: [] }); return acc;
17        }
18      }, [] as DefaultMappingEntry[]);
19    }
20
```

Listing 5.20: Short snippet from generated `getProperties` function (simplified)

Listing 5.20 shows the function, which distributes the properties for a class at creation. It firstly searches in the lookup, if the property has an explicit default value. In case it does not, it decides based on the type. In case of `string` it is interesting, that no implict default value is returned to avoid noisy empty strings. We can also see the switch from explicit `"[]"` into real `[]` to initialize arrays. And also specific types as `Visibility` and `Concurrency` have special and logical UML defaults `PUBLIC` and `SEQUENTIAL`. These can be later on extended for other special implicit defaults which may be needed.

```
1     Class: [
2         { property: 'name', propertyType: 'string'},
3         { property: 'isAbstract', propertyType: 'boolean',
    defaultValue: false},
4         { property: 'visibility', propertyType: 'Visibility'},
5         ...
6       ],
7
```

Listing 5.21: Simplified snippet from `defaultMapping` of class `Class`

In the Listing 5.21, we can see how the generated `defaultMapping` elements look like and what they include - `property`, which is the property name, `propertyType` and optionally the explicit `defaultValue`. In this case the `getProperties` function, would take the explicit default value for `isAbstract`, but for the properties `name` and `visibility`, the implicit values would be taken, as no explicit default value was provided for these in the `def.ts`.

5.5.2 Creation path mapping

The main purpose of the generated artifact by this generator, should be a function, which for a given parent type and child type decides, which containment property should be used during creation. It again uses the single source of truth `def.ts` and the most important role here plays the decorator `@path`, which makes the property considered as a creation target. As in the previous subsection, this generator also works with the `LangiumDeclaration` array.

The outcome of this generator is a small runtime resolver function from a generated lookup table. This prevents from using some hard coded switch case block, simplifies the maintenance and allows the usage of generic handlers.

The output of this generator is simpler and consists only of a single function and a constant mapping:

- `getCreationPath(parentType, childType) → string | undefined` - function used in the handlers, which provides the correct containment property based on the parent and child `mapping` - generated Record<ParentName, Array< property; allowedChildTypes » lookup table

The generation steps are following:

1. An empty `mapping: Record<string, Array< property; allowedChildTypes »` is initialized
2. Then for each declaration - also for each type from the `def.ts` file with properties (shown in Listing 5.22):
   - It scans the properties and if the decorator `@path` is present, it makes sure, that the parent bucket exists,
   - it creates the `allowedChildTypes` array, which are all the declarations (classes), which extend the `typeName` of the property (`allowedType`
   - When no subtypes are found, then the fallback creates an array only with the `allowedType`
   - `property: prop.name, allowedChildTypes` is then pushed into the mapping
3. These mappings are then injected into formatted template and the file is generated in `yo-generated/getCreationPath.ts` with all needed helpers and the resolver function.

```
1  export function buildCreationPathMapping(
2    langiumDeclarations: Array<LangiumDeclaration>
3  ): Record<string, Array<{ property: string; allowedChildTypes?:
   string[] }>> {
4    const mapping: Record<string, Array<{ property: string;
   allowedChildTypes?: string[] }>> = {};
5
6    langiumDeclarations.forEach((parentDecl) => {
7      if (!parentDecl.properties) return;
8      parentDecl.properties.forEach((prop) => {
9        if (prop.decorators && prop.decorators.includes("path")) {
```

```
10              if (!mapping[parentDecl.name]) mapping[parentDecl.name] =
    [];
11
12              const allowedType = prop.types[0].typeName;
13              let allowedChildTypes = langiumDeclarations
14                .filter((childDecl) => childDecl.extends?.includes(
    allowedType))
15                .map((childDecl) => childDecl.name);
16
17              if (allowedChildTypes.length === 0) allowedChildTypes = [
    allowedType];
18
19              mapping[parentDecl.name].push({ property: prop.name,
    allowedChildTypes });
20            }
21          });
22        });
23        return mapping;
24      }
25
```

Listing 5.22: Simplified snippet from `buildCreationPathMapping` function in the generator

In the Listing 5.23 the structure of the generated `getCreationPath.ts` file is displayed. As mentioned, it consists of the static lookup, where for each class type (which has the `@path` decorator on some of its properties), we have these properties with allowed child types; and also the function, which is used in the creation handlers and returns the correct containment schema. In the creation, the parent is the container, we are creating a new element in (e.g. a `Class`, when we create a new `Property`) and the child is currently created type (`Property`). The function `getCreationPath` then searches for entry `Class` in the mapping and looks for the `childKey` in the `allowedChildTypes` array of all entries in the `Class` Record. In this specific example, the 'properties' would be returned for the containment path. It must not always be the same name, e.g. as shown in Listing 5.23 in the mapping for Package, there are more allowedChildTypes for the 'entities' property.

```
1
2       const mapping: Record<string, Array<{ property: string;
    allowedChildTypes?: string[] }>> = {
3         Enumeration: [{ property: 'values', allowedChildTypes: ['
    EnumerationLiteral']} ],
4         Class: [{ property: 'properties', allowedChildTypes: ['Property'
    ]},
5           { property: 'operations', allowedChildTypes: ['Operation']}],
6         Package: [{ property: 'entities', allowedChildTypes: [ '
    Enumeration', 'Class', 'Interface', 'DataType', 'PrimitiveType', '
    InstanceSpecification', 'Package']}]
```

```
7       ,...}
8     export function getCreationPath(parentType: string, childType:
      string): string | undefined { const parentKey = stripPrefix(
      parentType); const childKey = stripPrefix(childType);
9       if (mapping[parentKey]) {
10        for (const entry of mapping[parentKey]) {
11          if ( entry.allowedChildTypes && entry.allowedChildTypes.
      includes(childKey)) {
12            return entry.property;
13          }
14        }
15      }
16      return undefined;
17    }
18
```

Listing 5.23: Simplified snippet from generated `getCreationPath.ts` file

The generation of this file allows the creation handlers to stay declarative without complicated switches, which would have to be updated often. Using this method the paths are always up to date according to the `def.ts` file.

5.6 Generator Orchestration, Registry & Monorepo Packaging

To keep the codebase clean, consistent and very easy to expand by new generators, the generation was moved to a small contribution contract and orchestration done using registry. This makes the repo more readable and tidy, centralizes file writes and formatting and the output of generation is deterministic and reviewable. Generators in packages, split by concerns in a monorepo ensures clear boundaries and dependency inversion, so that new functionalities can be added without changes to the main logic.

5.6.1 Contribution Contract

The goal is to separate all of the generators (where possible and it makes sense) behind a small and stable port. In order for that, we created and defined a port `ModelManagementContribution`, which has `codeGeneration(...) → path, content []` exposed for the generators. This brings few benefits with it:

- It centralizes all of the types, e.g. the already mentioned `LangiumDeclaration` and shares these with the generators.
- The surface stays restricted, as only pure data with the paths and content of the to be created files is returned. No write and I/O operations are done inside of the contributions.

- Dependency injection is enforced, as the `cli.ts` and registry depends on the `ModelManagementContribution` as a port and not on the specific generators.
- This also enables great testability as the port can be mocked and produced file list can be asserted with no side effects.

As mentioned the `ModelManagementContribution` is treated as a **port** and all of the implemented generators are the **adapters**. Each generator also acts as an **Strategy**, because it is modeled with the same `codeGeneration` signature.

```
1        export interface ModelManagementContribution {
2          codeGeneration: (options: {
3            langiumDeclarations: LangiumDeclaration[];
4            glspRoot: string;
5          }) => { path: string; content: string }[];
6        }
7
```

Listing 5.24: `ModelManagementContribution` interface

Listing 5.24 shows, that the function `codeGeneration` takes the declarations - that is the parsed data from `def.ts` - and the root, which is needed in the generators. Then simple output is returned as an array of paths and content for the generated files. We can see, that each generator, which can conform to this contract can be very simply added and the tool can be simply extended.

Now in the Listing 5.25 is a specific implementation of the `ModelManagementContribution` for the property palette. It is visible, how thin the adapter is, as it just delegates the write function and that is it.

```
1        export const PropertyPaletteContribution:
   ModelManagementContribution = {
2          codeGeneration: ({ langiumDeclarations, glspRoot }) => {
3            return writePropertyPaletteHandlers(glspRoot,
   langiumDeclarations);
4          },
5        };
6
```

Listing 5.25: `PropertyPaletteContribution` implementation

5.6.2 Registry and Orchestration

The contributions from the generators, need to be collected in a registry, so then they can be executed sequentially. In the Listing 5.26 is the implementation of the `GeneratorRegistry`, which consists of three parts:

- `contributions` array, which stores the contributions of a type `ModelManagementContribution`
- `register` function, which registers the contribution to this registry - adds it to the array. It is the accumulation of the generators.
- `execute` function, which after the contributions are collected, executes all of them sequentially. This is the composition.

In the function `execute` in the Listing 5.26 it is clearly visible, that it also formats the content of the file using `format`, which is an util function that utilizes prettier. The function goes through all of the contributions, runs it and for the whole array (of paths and contents) it creates the formatted file.

```
1         export class GeneratorRegistry {
2           private contributions: ModelManagementContribution[] = [];
3           register(contribution: ModelManagementContribution): void {
4             this.contributions.push(contribution);
5           }
6           async execute(glspRoot: string, langiumDeclarations:
    LangiumDeclaration[]
7         ): Promise<void> {
8             for (const contribution of this.contributions) {
9               const results = contribution.codeGeneration({
    langiumDeclarations, glspRoot });
10              for (const { path: filePath, content } of results) {
11                const formatted = await format(content);
12                fs.mkdirSync(path.dirname(filePath), { recursive:
    true });
13                fs.writeFileSync(filePath, formatted, "utf8");
14              }
15            }
16          }
17        }
18
```

Listing 5.26: `GeneratorRegistry` implementation

The usage of the `GeneratorRegistry` is very simple as it can be seen in Listing 5.27. The registry needs to be initialized, the contributions are registered (in correct order, if the order of files creation is important) and executed and the parameters need to be set only once, as each ocntribution follows the same contract. This is a straightforward process, which can be very quickly extended as new generator only needs one new `register` line in `cli.ts` to be executed. Furhtermore, CLI does not have to know anything about internal rules of the generator or the contract, so the concerns are separated.

```
1         const registry = new GeneratorRegistry();
2         registry.register(PropertyPaletteContribution);
```

```
3         registry.register(OutlineContribution);
4
5         await registry.execute(glspRoot, langiumDeclarations);
6
```

Listing 5.27: Short snippet from `cli.ts`, which shows the wiring of the Registry in here

5.6.3 Monorepo Packaging

As mentioned, the generators are located in small packages separated per concern inside of a TypeScript workspace scoped as `@borkdominik/*`. The small contract interface `@borkdominik/model-management-common` contains the types, which are shared and the already mentioned `ModelManagementContribution` port. Each feature package - currently `@borkdominik/property-palette` and `@borkdominik/outline` - depend only on the `ModelManagementContribution` contract. This enforces **dependency inversion**, as the main core depends only on the abstraction and not the explicit implementations. Each package can be compiled and extended independently from the others and the CLI only registers contributions and the executes the registry. All of the generated files are written to the `yo-generated/` folder, what guarantees, that the auto generated code is not mixed with the editable code.

When new generator or a functionality is added, it is a very simple and uniform task. New package `@borkdominik/<feature>` must be created and implement the `ModelManagementContribution` as a pure `codeGeneration` which returns an array of paths and content and add one line to the CLI, which registers this new contribution. The registry is built from interchangeable strategies using the same port with `codeGeneration`, no new wiring except for the registration is needed and reviews need to adjust only one package at time. Scalability of the system remains also very simple - by registration, not modification.

CHAPTER 6

Evaluation

This chapter presents the evaluation of the results along three perspectives: functional testing, descriptive comparison to a baseline and a case study using scenarios. With this, it is shown

- that the correctness of the framework is valid and that the operations do what they should,
- that the solution improves the maintainability and reusability,
- and that it can be instantiated for multiple UML diagram types with minimal reconfiguration.

In this chapter, the results of this thesis are evaluated. The first step includes functional testing, which should verify the implementation done during this thesis. Then the first part of the descriptive evaluation is done - the informed argument, where the results are compared to the baseline from Jäger [Jä24], based on metrics and architecture. In the second part a scenarios are described using the BIGUML Class and Package diagrams. To conclude this chapter, the results will be interpreted and summarized.

6.1 Functional Testing

This section verifies the implementation of the TypeScript model management framework and not the scientific claims themselves. The main goal of the functional testing is to check, verify and prove, that the concept and code parts described in the Chapters 4 and 5 are working correctly. All of the test written are unit tests done using TypeScript framework `vitest` [1], which run against small model states without requiring a running GLSP server or a VS Code instance.

[1] https://vitest.dev/

These tests cover all parts of the system, which were described in the chapters above:

- Generic create node handler,
- Generic delete element handler,
- Generic update handler,
- Validation generator and the runtime `validateNode` dispatcher,
- Creation path and Default value generators,
- outline and property palette generators.

This results in many tests across several test files and all tests pass.

6.1.1 Tests for Generic CRUD Operation Handlers

These tests verify the generic logic used to create, update and delete, which is reused for all diagrams. The following code parts are tested:

- Create node (edge) handler:
  - Element type handling,
  - Trigger actions for the GLSP client,
  - Container path resolution,
  - Patch construction for node creation,
  - Nodes without bounds.
- Delete element handler:
  - Correct GLSP operation type,
  - Handling of missing or empty element lists,
  - Handling of empty delete patches,
  - Wrapping patches in a `BigUmlCommand`,
  - Deleting meta information,
  - Building delete patches,
  - Merging and sorting delete operations - this guarantees that relations referencing a node are deleted before the node itself and the metadata is removed last - this prevents dangling references.
- Update element handler:

  - Operation type,
  - No patch when element path is missing,
  - Property update,
  - Command creation,
  - helper function, which casts values to correct types.

6.1.2 Tests for Validation Generator and Runtime Validation

These tests verify the behavior of the generated DTO classes together with the generated `validateNode` dispatcher. The generation of these files is also verified. The following code parts are tested:

- Runtime validation:
  - Constraints checking,
  - Custom constraints checking,
  - Property limits,
  - Conditional validation,
  - Error messages,
  - Combined error messages,
  - Unknown node types,
- Validation Generator:
  - DTO generation,
  - decorations,
  - decorator imports,
  - `validation-elements.ts` and `validator.ts` creation

6.1.3 Tests for Creation Path and Default Value generators

These tests verify the correctness of the generated files for the creation path and default values using mocked `def.ts` declarations. The following code parts are tested:

- Creation Path Generator:
  - Resolving derived types,
  - Fallback to direct type,
  - Ignoring non-path properties,
  - Multiple path properties per parent,

  - File creation

- Default Value Generator:
  - Setting property, propertyType and default value,
  - Ignoring entries without `@withDefaults`,
  - Entries with `@withDefaults`,
  - `@noBounds` classes,
  - `astTypeMap` generation,
  - File creation

6.1.4 Tests for Outline and Property Palette Generators

These tests verify, that the files generated by the outline and property palette generators are correct and include everything they need. The following code parts are tested:

- Outline generator:
  - Declarations,
  - Imports,
  - Guards,
  - Iteration over entities and children,
  - File creation
- Property Palette generator:
  - Calls with `.text`, `.bool`, `.reference` and `.choice`,
  - Labels,
  - Dynamic References,
  - File creation

6.1.5 Summary of Functional Testing

These tests guarantee, that the generic CRUD handlers produce correct JSON Patch operations using the creation path and default values configuration and integrate with `BigUmlCommand` without side effects. Also that the validation is functional and enforces simple, custom and also conditional constraints and are integrated safely into the patch pre commit hooks. And that the code generations generates expected outputs.

After all these components were successfully tested, the implementation of the framework is considered to be strong enough to serve as a basis for the descriptive evaluation and scenario case studies in the following sections.

6.2 Descriptive Evaluation - Informed Argument

The purpose of this section is to compare the result of this thesis with the baseline produced by Jäger [Jä24]. By this we want to show, if it actually improves maintainability and extensibility. A quantitative comparison is done using LOC and number of files for operations, property palette and outline and also a qualitative look at architectural patterns. This section argues, that centralizing semantics in `def.ts` and generating projections reduces duplicated and handwritten code, avoids drift between views and adding of new diagram types to the tool is significantly easier than in the baseline.

6.2.1 Baseline: Jäger's TypeScript-Based Model Server

As a baseline for this evaluation, Jäger's TypeScript based BIGUML model server is used. This has already been introduced in Chapter 3 and only the parts relevant for the evaluation are summarized here. It is built on Lences's blended editing stack [Len24] and Jäger replaces the Java and EMF backend using the TypeScript native metamodel and model management framework, which applies JSON patches to Langium AST [Jä24]. In his thesis, as a result, he gets a fully TypeScript stack, with textual and graphical UML running on the same model server, which uses Langium.

In this baseline, different UML diagram types (as Class and Package diagrams) are implemented using per element GLSP operation handlers for creation, update and deletion, which is combined with manually written property palette and outline handlers. Each diagram type has its own number of handlers and bindings, what leads to completely duplicated code for shared elements and very similar files for others. The overly complicated binding was shown in Listing 4.1 and the redundant complexity was also shown in Figure 4.1. Property palettes and outline views are also defined with many handwritten and redundant code files per element and per diagram.

6.2.2 Evaluation Criteria and Measurement Setup

In this section the results are compared against the described baseline. As presented in the Chapter 1, ELOC and number of files per aspect are compared first. Subsequently, the architectural patterns used, as well as the resulting complexity and effort, are examined. This will be qualitative, based on these metrics and code organization.

Measurement procedure To count the ELOC and number of files, the visual studio extension VS Code Counter [2] is used. ELOC is defined as "effective lines of code", what means, that we do not count lines with comments or empty lines. The measured parts will be the CRUD operation files, property palette files and outline files.

[2] https://marketplace.visualstudio.com/items?itemName=uctakeoff.vscode-counter

6.2.3 Quantitative results

In the Table 6.1, we can see the comparison between the baseline (from Jäger) and the current version - result of this thesis. In this table we compare

- the number of lines of code a developer needs to add to add new diagram type (UML Class diagram) to the tool (ELOC)
- the number of files a developer needs to add to add new diagram type (UML Class diagram) to the tool (Files)

From the results, we can clearly see, that for all of the CRUD operations, the developer in the baseline needed to add 28 files, whereas in the current version only 4 are needed, with fractional number of ELOC. This is consistent with the conceptual reduction of handler explosion shown in Listing 5.7. Also a clear difference is visible in the Property Palette and Outline, where the developer in the current version does not have to write no code, because of the `def.ts` file, where the diagram types are defined.

The Table 6.2 shows very similar results for the package diagram. To add this to the tool, the developer also needs only very simple CRUD handlers with few lines of code (because of the abstract and generic handlers), compared to the baseline. The property palette and outline still do not need any lines written by the developer. In the baseline there was no outline handler for the package diagram.

In the last Table 6.3, same values (ELOC and files) were counted for the abstract handlers, just to get a picture about how many lines of code these files have and to compare them with the baseline's numbers. So to mention some, the abstract generic create operation handler for nodes has 105 lines (Table 6.3, combined with the 25 lines (Table 6.1 for the subclass in class diagram makes in together 130 lines of code with two files. On the other hand, in the baseline solution, for the same features 1077 lines of code and 15 files were needed (Table 6.1). It is important to notice, that these abstract handlers must be written only once - when new diagram types are added, only the subclasses with very few lines of code need to be added.

6.2.4 Architectural Comparison

The quantitative results above reflect a deeper architectural shift from per element per diagram handlers towards configuration using definition file, generic operations and generated projections. This subsection interprets these results in terms of maintainability, duplication and extensibility.

Duplicated handlers or generic CRUD In the baseline's architecture each UML required its own create handler. Each diagram type required also own create handlers for each element and also own update and delete handler. This resulted in handler explosion as seen in Listing 4.1 and Figure 4.1. These show long lists of bindings, which seem very

Name	Current ELOC	Baseline ELOC	Current Files	Baseline Files
CRUD Operations - Class Diagram				
Create*OperationHandler (nodes)	25	1077	1	15
Create*OperationHandler (edges)	23	568	1	11
UpdateOperationHandler	27	54	1	1
DeleteOperationHandler	8	79	1	1
Property Palette - Class Diagram				
Property Palette Handler	0	77	0	1
Property Palette Elements	0	582	0	12
Outline - Class Diagram				
Outline handler	0	46	0	1

Table 6.1: Comparison between current implementation and baseline - number of ELOC and files the developer needs to add, when creating these features (in Class Diagram)

Name	Current ELOC	Baseline ELOC	Current Files	Baseline Files
CRUD Operations - Package Diagram				
Create*OperationHandler (nodes)	10	200	1	2
Create*OperationHandler (edges)	17	82	1	1
UpdateOperationHandler	8	44	1	1
DeleteOperationHandler	8	79	1	1
Property Palette - Package Diagram				
Property Palette Handler	0	28	0	1
Property Palette Elements	0	126	0	2
Outline - Package Diagram				
Outline handler	0	/	0	/

Table 6.2: Comparison between current implementation and baseline - number of ELOC and files the developer needs to add, when creating these features (in Package Diagram)

Name	Current ELOC	Current Files
Abstract Generic CRUD Handlers		
AbstractCreateOperationHandler (node)	105	1
AbstractCreateOperationHandler (edge)	58	1
AbstractUpdateOperationHandler	46	1
AbstractDeleteOperationHandler	100	1

Table 6.3: Number of ELOC and files needed for the generic abstract CRUD handlers

exhaustive to be maintained. High duplication in code means repeated logic, which needs to be copied each time a new diagram type is added. Adding or changing parts of code requires a lot of maintenance, as many files need to be adjusted. The scalability is also poor and it increases the cognitive load for developers.

The solution provided during this thesis makes this much simpler and clearer, with tidier structure. Introduction of generic and abstract handlers solves the per element issue on one side and also the per diagram issue on the other. Instead of N handlers per each element in each diagram type, we now have only one global abstract handler for each of the CRUD operations (create has two - for nodes and for edges) and very tiny subclasses, which define only accepted elements and small hooks in specific cases. The code, which must be written by the developer becomes trivial and huge binding lists disappear.

The difference is clearly visible in Listing 5.7, where we can see the concrete subclass of the create node operation handler for the class diagram. This shows how small have these classes become in contrast to the many files with many lines needed before, as we can see in the Table 6.1. It is important to mention, that this was possible not just using the generic and abstract classes, but also the default value and creation path generation mechanism, which moved a lot of specifics from the single element files, to a reusable mapping, without the need for regular update to these files (thanks to the regeneration).

Tables 6.1 and 6.2 in the CRUD operations section demonstrate this collapse from dozens of handwritten handlers to generic handler implementation and very few small subclasses. This explains the large reduction in ELOC and file count.

Single source of truth `def.ts` The definition file `def.ts` serves as a configuration for all of the diagram types and was introduced in our solution. It contains definition for the element types, properties and metadata for validation, property palette and outline. This file serves as the only source of truth for all of the generators and therefore for the whole framework.

In the baseline, the operation handlers, property palette and outline files were handwritten separately. The logic was separated between all of these files and the versions could easily go out of sync.

In our solution, all of the information about the diagram types is located solely in the `def.ts`. That means, that the logic can never drift between files as the only source for the types is located here. This file serves as an input to all of the generators:

- Property palette,
- Outline,
- Validation,
- Default values,
- Creation path

and all of the generated files live under `yo-generated/...` to keep the file structure clear. When a property is changed in `def.ts`, all of the above mentioned generators take it into account and regenerate the files according to this update.

`def.ts` makes sure, that all views, as property palette, outline, validation and operations, derive from the same declarative element definition and description. Any modification to this file continues to all generated code and removes the "out of sync possibility", which could previously occur between single parts of code.

Let us assume that the BIGUML tool supports only UML Class diagram. To also add support to the package diagram, the developer in the baseline would have to:

- Add create operation handlers for each element (nodes and edges) needed for package diagram,
- Copy and adjust update and delete handlers,
- Add all files for property palette (which are identical with the class diagram ones),
- Add all files for outline

In our solution, the developer would only have to add small subclasses for the handlers (which contain only the supported elements), which are very simple as we can for example see in the Listing 6.1, which provides just the correct model state and element type ids.

```
1        export class PackageCreateNodeOperationHandler extends
     AbstractCreateEdgeOperationHandler {
2        readonly elementTypeIds = [ModelTypes.CLASS, ModelTypes.
     PACKAGE];
3
4        @inject(PackageDiagramModelState)
5        declare modelState: PackageDiagramModelState;
6     }
```

Listing 6.1: Creation handler for the package diagram (imports are omitted)

The second needed adjustment is to update the `def.ts` file as seen in the Listing 6.2, what means that a class for the package diagram needs to be added and definition for all elements, which are supported in this diagram type. All these elements need to be also defined in this file.

```
1        export class PackageDiagram {
2          diagramType: "PACKAGE";
3          @path entities?: Array<Entity>;
4          @path relations?: Array<Relation>;
5        }
6
7        type PackageDiagramElements = Class | Package | Abstraction |
     Dependency | PackageImport | PackageMerge | Usage;
8    }
```

Listing 6.2: Additions to the `def.ts`, to add support for Package diagram

After this the generators need to be run and all of the needed code is generated.

It is important to note, that in both solution new folders and some boilerplate code needs to be created.

6.2.5 Summary of informed argument

Overall the metrics and the architectural analysis indicate that the frameworks using configuration and generators reduces a lot of duplicated and handwritten code, centralized the information in the definition file and simplifies the addition of new diagrams compared to Jäger's baseline. The generated files remain numerous, they are mechanically generated and grouped and the area needed to be created by developers is significantly smaller and more regular. This also makes the maintainability of the code much more effective and it ensures the consistency between concrete code parts.

6.3 Descriptive Evaluation - Scenarios

The purpose of this section is to evaluate the results of this thesis using real scenarios. We instantiate the framework in BIGUML for two diagram types: Class diagram and Package diagram. Both will run in the VS Code BIGUML extension with the TypeScript GLSP server and model management framework, produced as a result of this thesis. Both diagram types are currently configured declaratively in `def.ts` and all needed files are generated using the created generators. The goal is to show, that realistic modeling workflows can be done using our tool and that adding new diagram types works as expected.

In the definition file, all needed metadata are already prepared, with root `Diagram` element, `ClassDiagram` with entities and relations containment and other node and relations elements. The union type `type ClassDiagramElements` is also present.

These elements also contain all of the decorators, which control the code generation for all of the already mentioned generators. We have already seen the example for the definition of some elements in Listing 5.1. There we can clearly see the usage of decorators and the structure of element classes itself.

6.3.1 Workflow scenario

Creation The user with BIGUML extension starts with an empty Class diagram. They click "Class" in the GLSP palette, then into the canvas and new class node appears with default name and visibility (which are read from default-value-mapping, generated by the generator). During this in backend, the generic `CreateNodeOperationHandler` for Class diagram receives operation. It looks up the creation path and default values (which have been generated from the `def.ts` file). It then emits a JSON Patch applied to the Langium AST and triggers a model update. In the Figure 6.1 we can see, how the tool looks like after creating the first class.

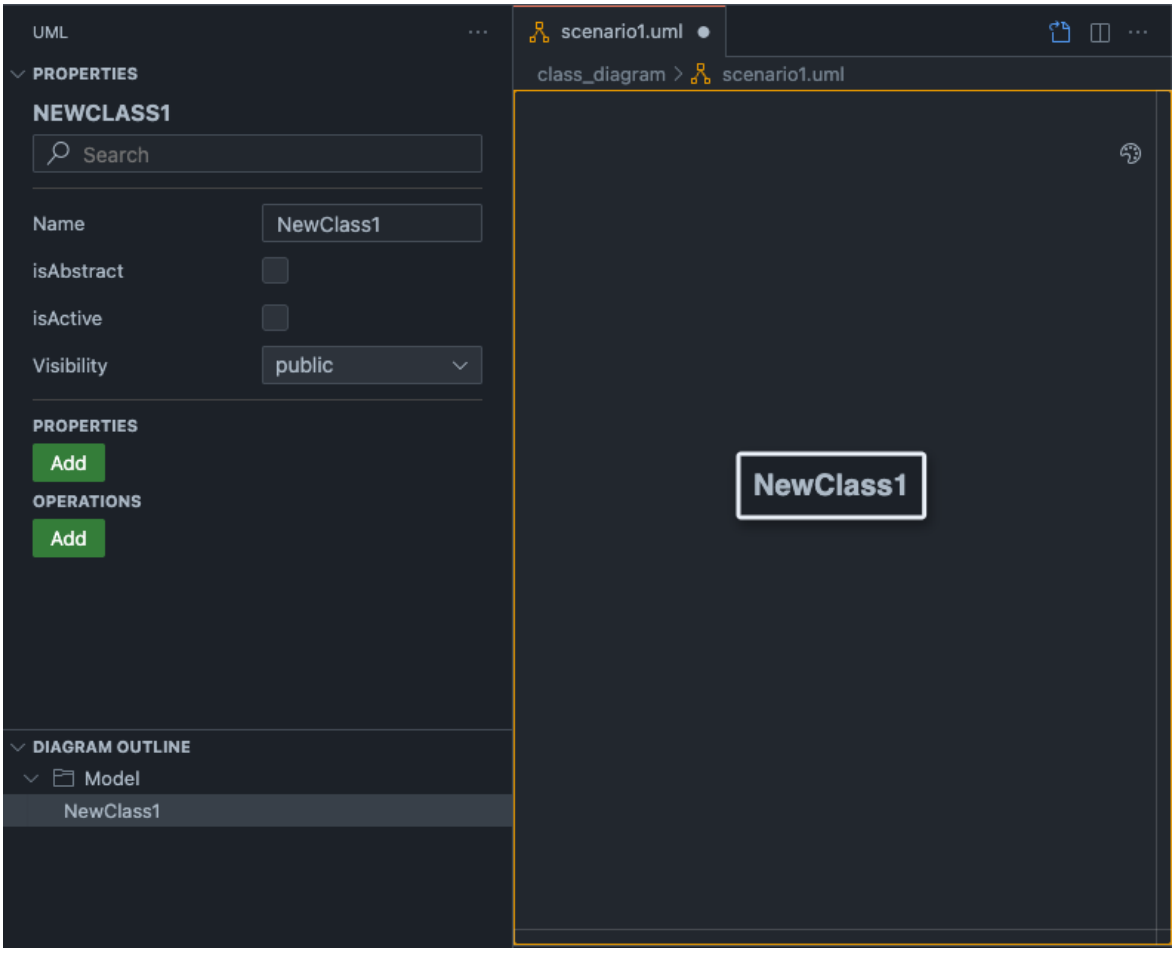

Figure 6.1: Initial Class diagram with a single class created via palette.

Property Palette The user wants to edit the newly created class, so he/she uses the generated property palette, which lets the user do it. The user renames the class, toggles the class to abstract and adds two new properties, using the Add button (this shows `.reference` mapping and child node creation). In the Figure 6.2 it can be seen, how the diagram, the property palette and the outline view look like, after these edits. Editing triggers the `UpdateOperationHandler`, what creates again new JSON Patch and creates the AST update. The outline view updates automatically and the structure of the created class with its properties is mirrored. This behavior shows, that both property palette and outline are derived from the same origin in `def.ts` file.

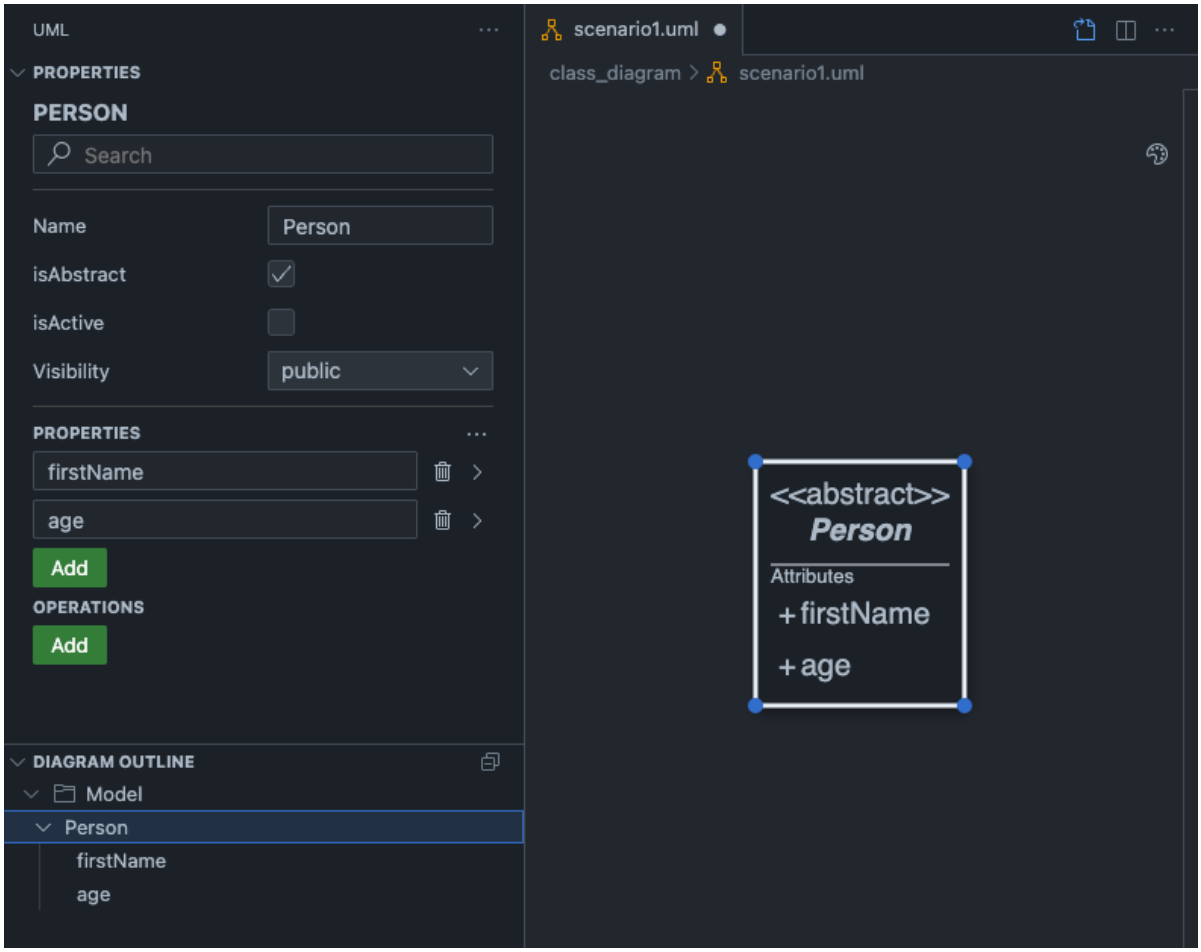

Figure 6.2: Edited Class diagram using the Property Palette with opened Outline view.

Validation From the `def.ts` file we know, that the name of the class must start with a capital letter and it must have at least 5 letters. The user enters invalid name: 'bert' and in the Figure 6.3 we see the error message that is shown to the user. The pre-commit validation (using the generated `validateNode` dispatcher runs and blocks the patch.

The user sees an error message, explaining what is wrong with the input. It is important to emphasize, that the validation is configured in the `def.ts` once and it is reused across diagrams. This demonstrates, that invalid changes never reach the AST, what keeps the model integrity.

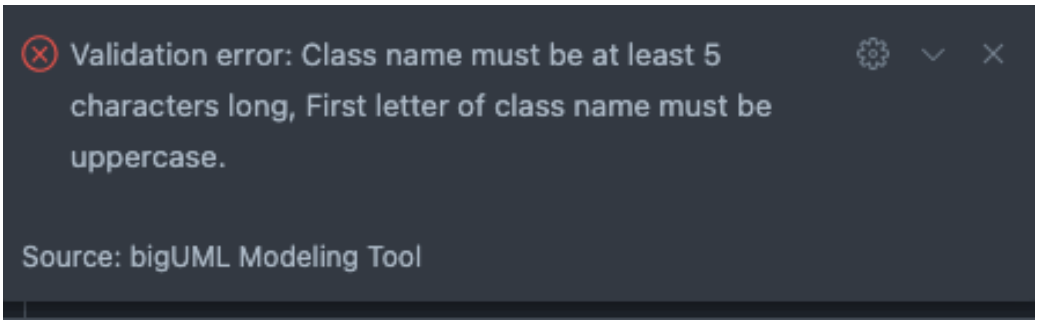

Figure 6.3: Error message shown to the user, when the input is invalid.

Deletion and consistency Finally, the user selects the class and deletes it. By doing this it is shown, that also the generic `DeleteElementHandler` is fully functional, by collecting related nodes, edges and metainfo, sorts it in correct delete order, to prevent inconsistencies (relations first, then node, then metadata) and builds JSON Patch for the delete operation. Diagram, property palette and outline view are all updated, without dangling references.

We repeat these same steps also for the Package Diagram, to show that the new solution supports also different diagram types (instead of class element, we created package diagram).

By doing this scenarios we proved, that:

- generic CRUD handlers are sufficient, to drive all class diagram edits,
- property palette and outline views are fully generated from `def.ts` and stay synchronized,
- validation mechanism successfully blocks invalid edits,
- All diagram specific code is tiny with small subclasses and `def.ts` configuration and rest is reused.

6.3.2 Scenario summary

Both scenarios show, that the framework is not just abstract, but it can actually be used in actual BIGUML editor. For both diagrams we demonstrated full CRUD, generated property palette and outline view and integrated validation. We also proved the possibility

of reuse of the same infrastructure and minimal configuration effort for another diagram type.

In the Table 6.4 we can see all of the functionalities, which are supported in the tool. All of the important operation and capabilities are fully operational and functional in both of the diagrams types.

Requirement	Class Diagram	Package Diagram
Core Diagram Operations		
Create node	✓	✓
Edit node	✓	✓
Move node	✓	✓
Resize node	✓	✓
Create edge	✓	✓
Delete node / edge	✓	✓
Property Palette Capabilities		
Show model element in property palette	✓	✓
Create child node via property palette	✓	✓
Edit nodes via property palette	✓	✓
Delete child nodes via property palette	✓	✓
Model Management & Validation		
Show model structure in outline view	✓	✓
Validation of edits (pre-commit)	✓	✓

Table 6.4: Evaluation of the supported capabilities in the Class and Package diagram scenarios

6.4 Interpretation of the evaluation results

The evaluation followed the plan from the Section 1.3.4, where functional testing verified the core framework of the bigUML tool and if it is correct. Generic CRUD handlers, generators, validation, property palette and outline were all fully tested. A descriptive comparison against Jäger's baseline counted ELOC and number of files and also compared the architectural patterns. Finally the evaluation of scenarios showed, that the framework can be instantiated for two specific UML diagram types (Class and Package diagrams) in the bigUML VS Code extension. Altogether, these perspectives allow the thesis to answer the research questions from the Section 1.2.

Research Question 1 How can a generic and reusable architecture be designed for operation handlers in model management to minimize the need for custom implementa-

tions?

All abstract CRUD handlers (create node, create edge, update, delete) are tested at the level of their shared algorithm with element type handling, container path resolution, patch generation, command wrapping and delete order sorting, with all these tests passing. These tests were run on small model states without a running GLSP server or VS code, what showed that the logic is not binded to the specific clients and can be also used for different diagram types.

For all of the CRUD operation in the Class diagram, the baseline needed 28 files with 1778 ELOC, while our new solution needs only 4 files with 43 ELOC. The generic abstract CRUD handlers are implemented once with no need to be reused or copied for other solutions with 105, 58, 46, and 100 ELOC respectively and the subclasses needed per diagram are tiny with very few lines of code - they consist mostly of `elementTypeIds` and wire model states, instead of providing full algorithms.

The proposed architecture collapses many handlers into a very small set of abstract CRUD base classes, following the template method pattern and the subclasses needed per diagram give only specific configuration rather than repeated algorithms. The base classes centralize all of the logic for name creation, default value application - which is done by code generation from the definition file - grid snapping and creation of the JSON Patches. This was previously duplicated in each of the handlers. This structure follows principles as DRY, Open/Closed and configuration over specialization. Variable parts are moved down to the configuration or small hooks, while the main algorithm stays stable in the core.

To answer the research question, the generic and reusable architecture for the operation handlers can be done by implementing the CRUD algorithm once in abstract base classes, using only small subclasses per diagram, which configure the supported element types and model states and giving the base handlers data using shared configuration - for creation paths, default values, validation - from a single definition file. The reduction of files and ELOC and the successful reuse in different diagram types with comprehensive unit tests shows, that the solution minimizes the need for custom implementation compared to the baseline.

Research Question 2 What is the most effective way to generalize CRUD operations for UML diagrams and auxiliary features (property palettes, outline views, validation) while ensuring reusability and extensibility across different diagram types?

A definition file `def.ts` was proposed, which contains element classes, properties, decorators or union types. It also contains the metedata for default values, bounds, property palette behavior, outline structure and validation constraints. All of the generators proposed in this solution take `def.ts` as the only source of truth, what ensures that all parts of code have the same semantics.

From the quantitative comparison we have an evidence, that for the class diagram, adding property support takes 659 ELOC in 13 files in the baseline (for package diagram it

is 154 ELOC in 3 files). For the outline view adding of one new file is needed for all diagram types. In our solution, the developer does not have to write any lines of code for these two views to work, as they get completely generated from the definition file.

Editing a class using the property palette starts the generic update handler, while the property palette and outline stay in sync, because both are generated based on the same definition. Validation constraints are also once defined and are checked during pre commit validation. The invalid inputs do not go through and a meaningful error message is shown. This is correct for all diagram types and the Table 6.4 shows, that both tested diagram types support the full set of required operations.

To answer the question, the evaluation suggests that the most effective way, how we can generalize the CRUD and auxiliary features for all diagram types is to keep the domain knowledge for elements, containment, properties and constraints in a single centralized definition file. This file then drives the code generation, which is synchronized between all code parts. The generic CRUD handlers then consume these generated mappings and do not have to use hard coded and repeated per diagram logic. The fact that class and package diagrams share the same definition and no additional code is needed for the property palette and outline, while the validation and creation rules are reused, shows that this approach achieves reusability and is extensible with just very small per diagram configuration effort. This mechanism itself is not connected to any specific diagrams and therefore adding further diagram types would follow the same logic - adding knowledge to definition files, rerun generators and adding small handler subclasses. This shows, that the solution generalized more than just the two evaluated diagram types.

Research question 3 How can a TypeScript-based framework enable efficient configuration and execution of new operation handlers and additional features while decreasing development effort for future extensions?

In the solution design, some of the TypeScript features have been used. The core artifacts of the framework as definition classes, unions, abstract handlers and DTOs are implemented using TypeScript. It benefits from the static typing, generics and module structure. The decorators in the `def.ts` give us a declarative way to annotate the model with configuration, which is then consumed by generators and handlers. The generators themselves are TypeScript monorepo modules, which depend only on small shared contract. This guarantees dependency inversion and modularity.

Adding a new generator is reduced to only implementing a new package, which fulfills the `ModelManagementContribution` contract and returns the `{path, content}[]` for all generated files. It then needs to be registered and executed in the registry in the CLI once. All of the generated files are written to `yo-generated` folder. This separates the generated and manually written code and simplifies regeneration and merge conflicts. To add support for new diagram type, the instructions are simple and repeatable.

The big reduction in CRUD ELOC and file counts as seen in the Tables 6.1 and 6.2 shows us, that future extensions won't need much manual coding in comparison to the baseline.

Abstract handlers need to be implemented only once and they are not modified when adding support for new diagram type. Extensions happen only through configuration and small subclasses. This matches the open for extension, closed for modification principle.

To answer the research question the evaluation shows that a TypeScript based framework can enable efficient configuration and execution of new operation handlers and features. Model management concepts are defined as typed TypeScript classes and unions annotated with decorators, which provide configuration. Generic handlers and generators are implemented as reusable TypeScript modules and are used with a small registry interface. The only files which need to be extended are definition file and tiny subclasses. The core algorithm stays stable. Although actual developer time was not measured explicitly, the reductions in ELOC and files, simple extension workflow and successful instantiation of two diagram types indicate, that the developers' effort should be decreased for future extensions in comparison to the baseline.

All of the research questions are answered and addressed positively and the evaluation supports the goal of this thesis initial claim, that a configuration driven TypeScript based framework can improve BIGUML's architecture and maintainability, extensibility and reusability for web based model management.

CHAPTER 7

Conclusion and Future Work

This final chapter concludes the thesis by summarizing the efforts done and presenting the results and findings. Future work for the discussed research topic will be presented after that.

7.1 Conclusion

The goal of this thesis was to modernize the BIGUML's model management and to provide a configurable and reusable architecture for model operations, which also supports features as property palette, outline and validation for GLSP diagram editors. The main three building block are the TypeScript based definition file `def.ts`, which describes diagram elements and their semantics; generators, which generate code parts based on the definition; and a set of generic CRUD operation handlers, which use the generated files to operate on models.

New definition file `def.ts` was implemented, which uses TypeScript classes, union types, and decorators to define diagram elements with their attributes; containment relationships and creation paths; default values; property palette and outline metadata; and validation constraints. It serves as a single source of truth.

A new generator pipeline with registry has been introduced, where all created generators which consume `def.ts` have been added. These produce TypeScript source files in a dedicated `yo-generated` directory. All of the generators are run from a single CLI file, so one execution step regenerates all derived artifacts. The generated and manually added files are separated, what simplifies the workflow.

Implementation of the generic CRUD handlers improved the extensibility and maintainability of the tool, as the create node, create edge, update and delete operations are now generic and duplicated code is not needed anymore. These base classes implement the shared behavior as the lookup of element types and default values using generated

mappings, grid snapping or name generation, Patch creation and command wrapping. Thanks to this, only very simple per diagram subclasses are needed, with just short configuration. The algorithm is not duplicated.

Features as property palette, outline view and validation are also supported and fully generated from the definition file, without the need of writing any code elsewhere, because everything gets generated.

Thanks to the evaluation it has been shown, that set of abstract handlers using the template method in combination with generated mappings provides a generic and reusable architecture and this makes custom implementations simpler. The centralization of all domain knowledge in a single definition file and using it for different code aspects serves as a great strategy for generalization. This approach is conceptually applicable to any additional diagram types. Strong typing, decorators and modular packages from TypeScript make the implementation very efficient as the configuration and extension is simple. This reduces the development effort.

The work was evaluated in three ways - functional tests for generators and handlers, which verified the correctness of added implementation; descriptive comparison with baseline from Jäger [Jä24], which showed big reductions in lines of code and files number and highlighted improvements in architecture; and evaluation with scenario, which showed, that end to end behavior works perfectly. This proved, that the developed result is capable of managing metamodel instances for multiple diagram types; serving as model management backend for GLSP editors; and providing greater maintainability and extensibility with new architecture compared to the baseline.

The thesis shows that a model management framework using configuration can successfully connect diagram editors and metamodels defined in TypeScript. BIGUML benefits from reduced implementation effort per diagram, clearer difference between definition and generated code; and improved potential for future improvements. This work offers a reusable pattern for other projects, which want to manage models using modern web technologies.

7.2 Future Work

The presented solution satisfies the research questions. However, not all aspects were researched fully, as the number of diagrams, complexity of constraints, collaboration features or the empirical user studies. The following section discusses potential improvements and challenges:

The generators could have even a bigger role in the modeling tools. In future works even more generators could be added, which would generate even more code, so that all parts of the code, which are just slightly duplicated and can be logically derived would be generated by the generators, so that the developers do not have to manually write code, that is not needed to be written by them. The infrastructure of the generators could be investigated more into depth, so that the generation is triggered by file changes

incrementally or in some "watch mode". Validation of the definition file itself could be added, which checks the consistency of decorators and unions.

The BIGUML tool itself could get some improvements, which can be investigated in future works for example multi user and collaborative editing, which would allow more users to work on a single project/diagram. This would probably need some remote and persistent storage, which would need API endpoints for loading and saving models and handling versioning. Concurrent edits from multiple clients, conflict resolution and notifications about changes would have to be supported.

Overview of Generative AI Tools Used

I used ChatGPT 4o, 4.5 and 5[1] during development of this thesis (always the most recent version available). For programming, it was used to generate and suggest short code snippets, which were always carefully reviewed and adapted. During the writing stage, it helped with formulating sentences and translating certain parts, but no generated text was used word-for-word and everything was rewritten in my own style.

Only exception is the German translation of the abstract, which was done using DeepL[2].

[1] https://chatgpt.com/
[2] https://www.deepl.com/en/translator

List of Figures

List of Tables

Listings

Bibliography

[ACLP17] Lorenzo Addazi, Federico Ciccozzi, Philip Langer, and Ernesto Posse. Towards seamless hybrid graphical-textual modelling for uml and profiles. In *Modelling Foundations and Applications (ECMFA 2017)*, volume 10376 of *LNCS*, pages 20–33. Springer, 2017.

[BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool, 2nd edition, 2017.

[BF14] Dominik Bork and Hans-Georg Fill. Formal aspects of enterprise modeling methods: A comparison framework. In *Proceedings of the 47th Annual Hawaii International Conference on System Sciences, HICSS 2014*, pages 3400–3409. IEEE Computer Society, 2014. 47th Hawaii International Conference on System Sciences ; Conference date: 06-01-2014 Through 09-01-2014.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280. IEEE Computer Society, 2001.

[BKP20] Dominik Bork, Dimitris Karagiannis, and Benedikt Pittl. A survey of modeling language specification techniques. *Information Systems*, 87:101425, 2020.

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

[BLO23] Dominik Bork, Philipp Langer, and Fabian Ortmayr. A vision for flexible glsp-based web modeling tools. *arXiv preprint arXiv:2307.01352*, 2023.

[bor25] borkdominik. bigUML: A UML web modeling tool based on GLSP. `https://github.com/borkdominik/bigUML`, 2025.

[Bé05] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[CGJ+22] Jordi Cabot, Sébastien Gérard, Jean-Michel Jézéquel, Dimitris Kolovos, Richard Paige, and Alfonso Pierantonio. Modeling should be an independent scientific discipline. *Software and Systems Modeling*, 21:1501–1512, 2022.

[Com] Yeoman Community. Yeoman: The web's scaffolding tool for modern webapps. `http://yeoman.io/`. Accessed: 23.08.2025.

[con25] Wikipedia contributors. Eclipse modeling framework, 2025. Wikipedia page available at: `https://en.wikipedia.org/wiki/Eclipse_Modeling_Framework` Accessed: 21.08.2025.

[CTVW19] Federico Ciccozzi, Matthias Tichy, Hans Vangheluwe, and Danny Weyns. Blended modelling - what, why and how. In *MODELS 2019 Companion (ACM/IEEE 22nd Intl. Conf. on Model Driven Engineering Languages and Systems)*, pages 425–430. IEEE, 2019.

[Ecl25] Eclipse Foundation. Eclipse xtext (online documentation). `https://eclipse.dev/xtext/`, 2025. Accessed: 23.08.2025.

[Fou25a] Eclipse Foundation. Eclipse modeling framework (emf). `https://www.eclipse.org/modeling/emf/`, 2025. Accessed: 19.08.2025.

[Fou25b] Eclipse Foundation. Graphical language server platform - overview. `https://eclipse.dev/glsp/documentation/overview/`, 2025. Accessed: 19.08.2025.

[Fou25c] Eclipse Foundation. Sprotty: A diagramming framework. `https://github.com/eclipse/sprotty`, 2025. Accessed: 19.08.2025.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

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

[Goo25] Google. Google guice: Lightweight dependency injection framework. `https://github.com/google/guice`, 2025. Accessed: 19.08.2025.

[GPHL+19] Katerina Goseva-Popstojanova, Tamer Hassan, Grace Lewis, Edwin Morris, and Lutz Wrage. Benefits and challenges of model-based software engineering. Technical Report NASA/TM–2020–220479, NASA, 2019. NASA Technical Memorandum, National Aeronautics and Space Administration.

[HB23] Florian Hölzl and Simon Barner. Implementing a model-based engineering tool as web application. *CoRR*, abs/2302.14091, 2023.

[HT99] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Boston, MA, 1999. Introduces the DRY principle.

[Jan25] Remo H. Jansen. Inversifyjs: A powerful and lightweight inversion of control container for javascript & node.js apps powered by typescript. `https://inversify.io/`, 2025. Accessed: 19.08.2025.

[JLF+25] David Jäger, Adam Lencses, Martin Fleck, Philip Langer, and Dominik Bork. A model management framework for next-generation web-based modeling tools. *Journal of Object Technology*, 24(2):2:1–14, May 2025. The 21st European Conference on Modelling Foundations and Applications (ECMFA 2025).

[JLMB25] David Jäger, Adam Lencses, Haydar Metin, and Dominik Bork. Towards frontend-only modeling tools: Experiences from rebuilding two modeling tools with a typescript-based model management solution. *Journal of Object Technology*, 24(2):1:1–23, 2025.

[Jä24] David Jäger. Frontend-only browser-based modeling tools. Master's thesis, TU Wien, 2024.

[Kle08] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley Sons, Inc., 2008.

[Kö16] Jan Köhnlein. Embedding xtext editors in graphical modeling tools. `https://typefox.io/blog/xtext-sirius-integration`, 2016. TypeFox Blog.

[Len24] Adam Lencses. Combining textual and graphical modeling with next-generation frameworks. Master's thesis, TU Wien, 2024.

[Mar02] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, Upper Saddle River, NJ, 2002. Popularizes SOLID, including the Open/Closed Principle.

[Mat24] MathWorks. *MATLAB*. MathWorks, Natick, Massachusetts, 2024. Version R2024a. Accessed: 20.08.2025.

[MB23a] Haydar Metin and Dominik Bork. Introducing biguml: A flexible open-source glsp-based web modeling tool for uml. In *Proceedings of the 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 40–44, 2023.

[MB23b] Haydar Metin and Dominik Bork. On developing and operating glsp-based web modeling tools: Lessons learned from biguml. In *Proceedings of the 26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2023)*, pages 129–139, 2023.

[MB25] Haydar Metin and Dominik Bork. A reference architecture for the development of glsp-based web modeling tools. *Software and Systems Modeling*, 2025.

[MBWM24] Judith Michael, Dominik Bork, Manuel Wimmer, and Heinrich C. Mayr. Quo vadis modeling? *Software and Systems Modeling*, 23(1):7–28, 2024.

[MHS05] Miran Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[Mic25] Microsoft. Language server protocol. `https://microsoft.github.io/language-server-protocol/`, 2025. Accessed: 19.08.2025.

[MKG13] Qin Ma, Pierre Kelsen, and Christian Glodt. A generic model decomposition technique and its application to the eclipse modeling framework. *Software Systems Modeling*, 14, 2013.

[Obj11] Object Management Group. OMG Unified Modeling Language (OMG UML): Infrastructure, Version 2.4.1. Technical Report formal/2011-08-05, Object Management Group, 2011. See Figure 7.8 for the four-layer metamodel hierarchy (M0-M3). Accessed: 21.08.2025.

[Obj17] Object Management Group (OMG). Omg unified modeling language (uml) specification, version 2.5. Specification, OMG, 2017. Accessed: 20.08.2025.

[Obj19] Object Management Group (OMG). Meta-object facility (mof) core specification, version 2.5.1. Specification, OMG, 2019. Accessed: 20.08.2025.

[Ora25] Oracle Corporation. The java platform, standard edition. `https://www.oracle.com/java/technologies/java-se.html`, 2025. Accessed: 09.11.2025.

[OT17] Obeo and TypeFox. Xtext/sirius integration white paper. `https://www.obeodesigner.com/resource/white-paper/WhitePaper_XtextSirius_EN.pdf`, 2017. Accessed: 23.08.2025.

[Pet22] Julia Petzold. Langium meets sprotty: Combining text and diagrams in vs code. TypeFox Blog, February 2022. Accessed: 23.08.2025.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition, 2008.

[Sei03] Ed Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, 2003.

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[Sp2] Moritz Spönemann. Langium: A new toolkit that simplifies textual language creation. *Eclipse Newsletter*, May 2022. Accessed: 23.08.2025.

[Tha22] Bernhard Thalheim. Models: the fourth dimension of computer science — towards studies of models and modelling. *Software & Systems Modeling*, 21(1):9–18, 2022.

[Too25] BIG Modeling Tools. biguml - uml diagram modeling tool (visual studio code extension). `https://marketplace.visualstudio.com/items?itemName=BIGModelingTools.umldiagram`, 2025. Accessed: 19.08.2025.

[Typ25] TypeFox GmbH. Langium – language engineering in typescript. `https://langium.org`, 2025. Accessed: 21.08.2025.

[vRF25] Guido van Rossum and Python Software Foundation. *Python Language Reference, Version 3.x*. Python Software Foundation, 2025. Available at `https://docs.python.org/3/reference/`. Accessed: 20.08.2025.

[Wik25a] Wikipedia contributors. Business process model and notation, 2025. Wikipedia page available at `https://en.wikipedia.org/wiki/Business_Process_Model_and_Notation`. Accessed: 20.08.2025.

[Wik25b] Wikipedia contributors. Ecore, 2025. Wikipedia page available at `https://en.wikipedia.org/wiki/Ecore`. Accessed: 20.08.2025.

[Wik25c] Wikipedia contributors. Simulink, 2025. Wikipedia page available at `https://en.wikipedia.org/wiki/Simulink`. Accessed: 20.08.2025.

[Wor25] World Wide Web Consortium (W3C). Hypertext markup language (html) — living standard. Specification, W3C / WHATWG, 2025. Latest version as of 2025. Accessed: 20.08.2025.