Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

HyLiMo: A Textual DSL and Hybrid Editor for Efficient Modular Diagramming

Niklas Krieger

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Sandro Speth, M.Sc.
Commenced:	November 3, 2022
Completed:	May 10, 2023

Abstract

Context. Diagrams are an important artifact in software engineering. They are used to visualize complex systems and for communication with different stakeholders, including domain experts and developers.
Problem. Currently, two major diagramming approaches exist: either the diagram is created using a textual concrete syntax, or using a graphical editor. However, both have their limitations, in particular when precise manual layouting is required. In the area of modeling, recent research introduces the concept of hybrid/blended modeling, which combines graphical and textual notations for an improved user experience. Our goal is to apply this concept to diagramming, too.
Objective. Our concept allows manipulating the layout both graphically and textually, with both views being live-synced. Also, for improved clarity of complex layouts, we support programming language constructs in the textual definition. First, we want to collect and evaluate the requirements of a hybrid textual-graphical diagramming approach, with a focus on the interactive graphical view. Second, we want to prove that a concept of modular hybrid diagramming is feasible.
Method. In order to collect requirements, we perform interviews with experts. Then, we perform a survey with the same experts to evaluate the collected requirements. To prove our concept is feasible, we implement a modular framework for hybrid diagramming. To evaluate it, we implement a module for UML class diagrams.
Result. Overall, we succeed in implementing our modular hybrid diagramming framework and thus show that our approach is feasible. By conducting two case studies, we show that our implementation results in a usable diagramming experience for UML class diagrams.
Conclusion. We developed a tool that allows for hybrid textual-graphical creation of UML class diagrams using a web-based editor. However, future work is required to implement missing features, especially in the area of graphical editing.

Kurzfassung

Kontext. Diagramme sind ein wichtiges Artefakt im Bereich des Softwareengieering. Sie werden verwendet um komplexe Systeme zu visualisieren, und für die Kommunikation mit verschiedenen Interessevertretern, insbesondere Domänenexperten und Entwicklern.
Problem. Aktuell existieren zwei primäre Ansätze, um Diagramme zu erstellen: entweder wird das Diagramm basierend auf einer textuellen Syntax generiert, oder mithilfe eines grafischen Editors erstellt. Jedoch besitzen beide Ansätze Limitierungen, insbesondere sofern ein manuelles als auch präzises Diagrammlayout erforderlich ist. Aktuelle Forschung in Software-Modellierung führt das Konzept der hybriden/vermischten (blended) Modellierung ein. Hierbei werden textuelle und grafische Notationen kombiniert um eine bessere Benutzererfahrung zu erreichen. Unser Ziel ist es, dieses Konzept auf das Erstellen von Diagrammen zu übertragen.
Ziel. Unser Konzept ermöglicht es das Diagrammlayout sowohl grafisch, als auch textuell zu manipulieren, wobei grafische und textuelle Ansichen live-synchronisiert sind. Um bei komplexen Layouts eine bessere Übersichtlichkeit zu erreichen, unterstützt die textuelle definition Programmiersprachenkonzepte. Zunächst sollen Anforderungen von hybridem grafisch-textuellen Erstellen von Diagrammen identifiziert und ausgewertet werden. Hierbei fokussieren wir uns auf den interaktiven grafischen Editor. Folgend soll gezeigt werden, dass unser Konzept technisch umsetzbar ist.
Methode. Um Anforderungen zu sammeln führen wir zunächst Expertengespräche durch. Darauf folgend führen wir eine Umfrage mit denselben Experten durch um identifizierte Anforderungen zu evaluieren. Um zu zeigen, dass user Konzept technisch umsetzbar ist, implementieren wir eine Softwarebibliothek für modulares hybrides Erstellen von Diagrammen. Dieses wird durch ein Modul für UML Klassendiagramme evaluiert.
Resultat. Die Softwarebibliothek für modulares hybrides Erstellen von Diagrammen wurde erfolgreich implementiert. Daraus schließen wir, dass unser Konzept technisch umsetzbar ist. Des weiteren ergeben zwei Fallstudien, dass unsere Implementierung zu einer akzeptablen Benutzererfahrung beim Erstellen von UML Klassendiagrammen führt.
Schlussfolgerung. Wir entwickelten ein Tool für hybrides grafisch-textuelles Erstellen von UML-Klassendiagrammen mithilfe eines webbasierten Editors. Zukünftige Arbeit ist jedoch erforderlich, um nicht umgesetzte Funktionen hinzuzufügen, insbesondere im Bereich des grafischen Editierens.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Acronyms

API Application Programming Interface. x, 86

AST Abstract Syntax Tree. 9

CI Continuous integration. 24

CLI Command-line interface. 24

CSS Cascading Style Sheets. 46

CST Concrete Syntax Tree. 9

DD Diagram Definition. 6

DSL Domain-Specific Language. ix, x, xiii, 2

EMF Eclipse Modeling Framework. 16

GLSP Graphical Language Server Platform. 17

GPL General-purpose Programming Language. 10

HTML HyperText Markup Language. 13

IDE Integrated Development Environment. 13

JS JavaScript. xiii, 10

LSP Language Server Protocol. 13

MDSD Model Driven Software Development. 11

OMG Object Management Group. 6

PDF Portable Document Format. 47

SCSS Sassy Cascading Style Sheets. 62, 84

SVG Scalable Vector Graphics. 14

TS TypeScript. 11

UI User Interface. x, 71

UML Unified Modeling Language. ix, x, 1

UML DI UML Diagram Interchange. 6

URL Uniform Resource Locator. 83

WASM WebAssembly. 52

WPF Windows Presentation Foundation. 48

1 Introduction

Diagramming is the process of creating visual representations of systems, processes, or concepts. Thus, diagrams are common in engineering disciplines, including software engineering. Diagrams are used, for instance, by domain experts to visualize domain concepts, architects when breaking a system into its components, and developers to document and explain code. To fulfill all these use cases, a variety of different diagram types exist, with different degrees of formality. Similarly, two main approaches for creating said diagrams exist: (1) graphical, and (2) textual.

First, a variety of tools exist where users graphically create diagrams, including *Visio* and *diagrams.net*. There, users create diagrams by typically placing diagram elements on a (possibly infinite) canvas. Layouting is done by the diagrammer, i.e., the engineer who creates the diagram, which allows for custom layouting. The second main group is formed by textual tools like *PlantUML* and *Mermaid*. There, diagrams are created by defining the elements of the diagram using a textual concrete syntax. The graphical diagram is then created programmatically, diagram elements are positioned using a layout algorithm. Other textual approaches, e.g., *TikZ-UML*, instead allow the user to provide layouting information. Both textual and graphical approaches have unique benefits. Textual approaches allow for the efficient definition of diagram elements, e.g., classes in a *Unified Modeling Language (UML)* class diagram. Also, developers often prefer to write code instead of using a graphical editor, which can be seen in the increasing popularity of tools like Mermaid. Further, textual concrete syntaxes allow for integrating general-purpose programming features, e.g., custom functions and variables to reuse common constructs, thus, improving efficiency. However, as these tools use an automatic layout algorithm, these tools are insufficient for use cases where manual control over the layout is required to communicate information efficiently. Such use cases include software documentation and scientific publications. While other textual tools like TikZ-UML support precise manual positions, defining such textually can be burdensome and time-consuming. Here, graphical tools provide the benefit of intuitive layouting by simply dragging elements. Yet, manipulating the content of diagram elements can be more difficult compared to textual notations, in particular when interacting with rotated or otherwise transformed elements. Also, achieving precise positioning can be challenging. While snapping can help with aligning elements, when dealing with large diagrams, choosing the correct alignment is complex.

Hybrid textual-graphical diagramming can be a solution to those challenges. Recent research in modeling introduces the concept of hybrid/blended modeling. Blended modeling uses multiple notations, including textual and graphical ones, to manipulate an underlying model, thus, improving user experience and efficiency [CTVW19]. However, compared to diagramming tools, modeling tools are typically less flexible, as the model needs to conform to the metamodel. Therefore, not all use cases of diagramming tools can be fulfilled by modeling tools. Also, modeling tools typically provide fewer styling and theming options compared to diagramming tools. In the area of diagramming, *UMLet* [ATB03] uses a hybrid graphical textual approach. Here, the graphical view allows editing the position and sizes of diagram elements, e.g., UML classes, and changing the routing of connections between those elements. The content of said elements is instead defined

using a textual concrete syntax, which is specific to the element. Yet, as positions cannot be defined textually, precise layouting remains challenging, especially in the case of editing during the life cycle. Also, the programmatic definition of elements is not supported. Thus, existing tools prove to be insufficient to adequately address all the challenges that we have identified.

To solve these challenges, we envision a modular framework that brings the benefits of hybrid/blended modeling to diagramming. First, as it uses a hybrid graphical-textual approach, a hybrid editor, consisting of both a graphical and textual view should be provided. Both views allow editing the diagram, with changes being live-synced. Diagrams are primarily defined using a textual Domain-Specific Language (DSL), which serves as a single source of truth, thus allowing to share and persisted diagrams easily. Also, programming language features should be provided here, to simplify the definition of repeating elements, and improve customizability. To be usable in situations where a high level of control in the appearance of the diagram is required, the framework should support both manual and precise layouting of its contents. Furthermore, the style of diagram elements, including built-ins, should be customizable by the user. As different types of diagrams should be supported in the long term, we envision the framework to be extensible by modules providing support for additional diagram types. Based on this concept, we derive our main research questions:

RQ 1: "Is implementing a framework for modular live-synced hybrid diagramming technically feasible?"

Regarding **RQ1**, the main challenge lies in supporting graphical editing features while allowing programming constructs to define diagrams.

RQ 2: "Do programming language features, in particular variables and control flow structures help with manual layouting?"

RQ 3: "Do custom functions and control flow structures improve customizability and extendability?"

These two questions both focus on the programming language features aspect of our concept, overall, we want to find out how these features support users in creating diagrams.

RQ 4: "Does the hybrid approach improve diagramming efficiency?"

Our last research question focuses on the productivity aspect of hybrid diagramming. Here, we want to find out how the hybrid diagramming concept compares to existing graphical and text-based approaches, in particular when maintaining a diagram over a prolonged period of time.

As a result, the primary contributions of this work are (1) a concept of for modular hybrid diagramming (2) a reference implementation of such a framework (3) UML class diagram support for said framework

As described, our concept focuses on use cases where a high level of styling and layouting control is required. Thus, as the primary user group, we identify researchers creating diagrams for scientific publications. Secondary user groups include developers and architects, e.g., when creating

documentation. Consequently, to evaluate the created framework, we conducted two case studies, where PhD and undergraduate students used our framework to create several class diagrams. In both case studies, each participant was able to create the diagram(s) using the HyLiMo framework, using both styling and programming language features in the process.

The remainder of the thesis is structured as follows:

Chapter 2 – Foundations First, we introduce concepts and technologies relevant to understand the remainder of this thesis.

Chapter 3 – Related Work This section gives an overview of related work.

Chapter 4 – Requirements Engineering Describes the requirements of our approach and the process, how we collected these requirements.

Chapter 5 – Concept Describes the concept of our modular hybrid diagramming framework.

Chapter 6 – Architecture & Implementation This chapter details the architecture of the implemented framework and some aspects of its implementation.

Chapter 7 – Evaluation We describe how we evaluated our framework using two case studies.

Chapter 8 – Conclusion Concludes the thesis and gives an overview of the final results and remaining future work.

2 Foundations

This chapter introduces concepts and technologies important for understanding this thesis. First, Section 2.1 gives an overview of diagrams, with a focus on UML class diagrams. Following, programming language and domain-specific language concepts are described Section 2.2 and Section 2.3 respectively. Last, we outline technologies relevant to this thesis in Section 2.4.

2.1 Diagrams

Diagrams are commonly used in software engineering. They allow visualizing complex systems and make them easier to understand for domain experts. Depending on the specific use case, many different types of diagrams can be used. As a result, they are used during the whole software engineering process. Starting at requirements engineering, use case diagrams allow collecting and organizing requirements from stakeholders. Component diagrams visualize how a system can be broken down into components, class diagrams represent the design and implementation of a system. Overall, diagrams can be categorized into two subcategories: structural and behavioral diagrams. While structural diagrams are used to describe the structure, e.g., its components, of the modeled system, behavioral diagrams show what happens inside the system.

2.1.1 Diagramming vs Modeling

Conceptually, diagrams are graphical representations of underlying models. Modeling is the task of creating models, and then defining one or more views, the diagrams, based on those underlying models [Lim]. In contrast, when using diagramming, one directly defines diagrams, without first defining a non-visual model. Both approaches are used in software engineering practice and provide their own benefits and drawbacks. Diagramming has a low entry barrier and makes it easy to brainstorm ideas. Also, diagramming tools typically support a flexible notation, not constraining users to the formal syntax of a modeling language. Modeling provides the benefit of allowing to reuse model elements across different views. Also, it allows using the models for other purposes than visualization, for example, code generation. Especially, conformance to the metamodel can, but also must be ensured.

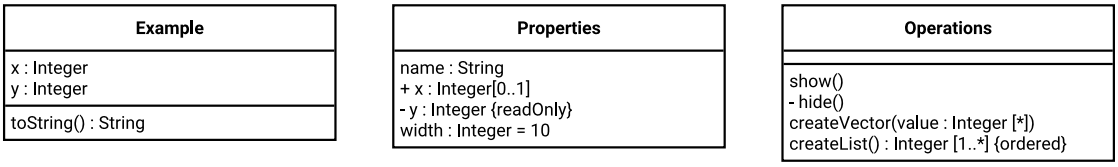

Figure 2.1: Example UML classes which define properties and operations.

2.1.2 UML Class Diagram

UML class diagrams are a specific type of structural diagrams. By being part of the Object Management Group (OMG) UML standard[1], they are one of the most popular types of diagrams. Typically, UML class diagrams are used to model classes and their relationships in an object-oriented system. Before going into the details of class diagrams, we first need to define the difference between models and diagrams. First and foremost, the OMG UML standard defines a metamodel that can be used for modeling. UML models consist of elements, e.g., classes and packages, and is an instance of the UML metamodel [CBR+17, p. 683]. A UML (class) diagram is a graphical representation of said model. Note that the UML metamodel itself does not define any distinction in class, package, or component models. As a consequence, a model can contain both classes and components simultaneously, and therefore, graphical diagrams can too. According to the standard, class diagrams primarily contain structured classifiers [CBR+17, p. 685], including classes, components, associations, and collaborations [CBR+17, p. 183]. Additionally, interfaces and packages are commonly used. As diagrams are visual artifacts, a graphical representation of said elements is required. In the UML standard, this aspect is provided by the *UML Diagram Interchange (UML DI)* specification, which in turn is based on *Diagram Definition (DD)*. Following, we introduce the most important aspects of elements commonly used in UML class diagrams, and their graphical syntax. Note that as UML is a highly complex standard, here, we focus on the aspects relevant to this thesis. Especially, depending on the type of diagram, the same element may be visualized differently, following, we focus on the most common representation in class diagrams.

Classes are the most commonly used element. Among others, a class defines a name, a set of properties, and a set of operations. As Figure 2.1 shows, in a diagram, classes are typically displayed using a rectangle with multiple separated compartments. The top-most compartment contains the name, the second properties, and the third operations. Properties primarily consist of a visibility, a name, a type, a default value, and modifiers. All elements except the name are optional and may be omitted. Class `Properties` in Figure 2.1 shows how properties are notated in a diagram. Especially, visibilities are defined with the characters `'+'` (public), `'-'` (private), `'#'` (protected), and `'~'` (package). The multiplicity is a range of format `lower..higher`, where `'*'` for higher represents any amount. `0..*` can also be replaced with `*`. The type is specified after the colon and modifiers in curly brackets. Similarly, operations primarily consist of a visibility, a name, a set of parameters, and a return type. The return type can also contain a multiplicity and properties. Overall, parameters are similar to properties, however, do not define a visibility. Class `Operations` in Figure 2.1 showcases some example operations. A class can have any amount of superclasses. As shown in Figure 2.2, classes are connected to their superclasses using non-filled arrows, as

[1] https://www.omg.org/spec/UML/2.5.1/PDF

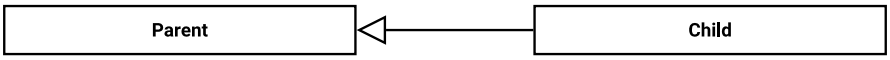

Figure 2.2: Example of a UML class extending another class.

subclasses generalize their parent class. Last, to improve extensibility, UML supports extending classes using stereotypes. Due to not being relevant to this thesis, for their semantics, we refer to the standard [CBR+17, p. 252-272]. Graphically, stereotypes can be added to classes by enclosing the name of the stereotype into guillemets, and placing it above the name of the class, as shown in Figure 2.3.

Figure 2.3: Example of a UML class with the stereotype `ExampleStereotype`.

Like classes, interfaces also define a name, a set of properties, and a set of operations. While the overall graphical notation is similar to classes, «interface» has to be added. Note that while this notation is identical to stereotypes, «interface» is not a stereotype, but a keyword for the metaclass Interface. Similar to classes, interfaces can generalize other interfaces as shown for `InterfaceA` and `InterfaceB` in Figure 2.4. Additionally, classes can realize interfaces, in the diagram, this is shown by the dashed arrow.

An association defines a relationship between classifiers, especially, classes and interfaces [CBR+17, p. 199]. While UML supports n-ary associations, following, we focus on binary associations between two elements. Graphically, associations are represented by connections between the association ends. To further define the association, different textual and graphical elements can be added to the association. First, as associations can define a name, a name can be added near the middle of the connection, as shown in Figure 2.5. A solid triangle defines the reading direction, in this case, “Actor ActsIn Movie”. Also, properties can be associated with the ends of the association. Additionally, properties can be used for association ends. In this case, the name, visibility, multiplicity, and modifiers are added near the appropriate association end. Furthermore, for both ends, one can define navigability, options include navigatable, non-navigatable, and no defined navigability. Graphically, as shown in Figure 2.6, navigatable is represented by an arrow, while non-navigatable is represented by an X symbol. Composition and aggregation are specific types of associations. In the case of composition – also called composite aggregation – being used, the composite is responsible for the lifetime of the composed elements. For aggregation, also called shared aggregation, no precise semantics are defined in the standard [CBR+17, p. 112]. Figure 2.7 shows how composition and aggregation are represented graphically. While for composition, a filled diamond is used, for aggregation, an empty diamond is used.

Figure 2.4: A UML class `ClassA` realizing `InterfaceB` which generalizes `InterfaceA`

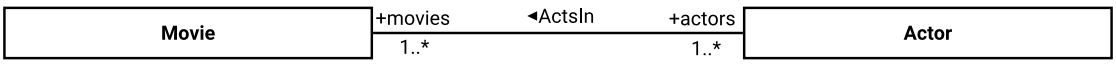

Figure 2.5: An association between two classes.

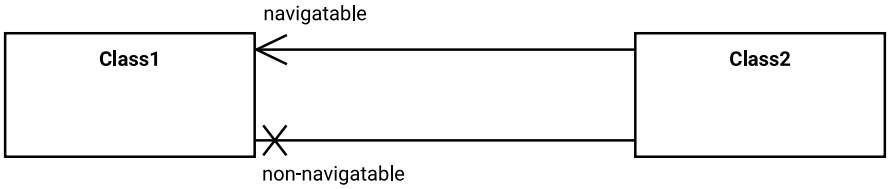

Figure 2.6: UML associations showcasing navigability.

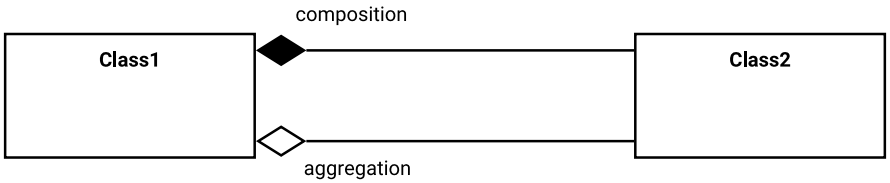

Figure 2.7: UML composition and aggregation associations.

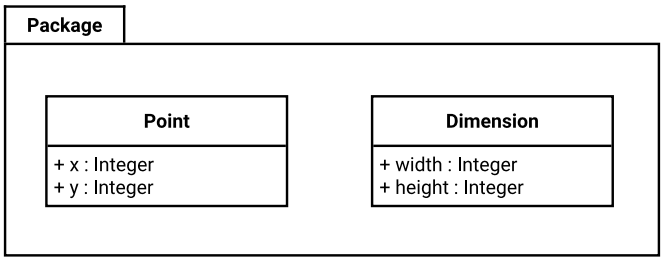

Figure 2.8: UML package containing two classes.

Packages are namespaces for some members [CBR+17, p. 241], in the case of class diagrams, primarily other packages, classes, and interfaces. Similar to those elements, packages define a name. Graphically, as shown in Figure 2.8, a package consists of a rectangle containing the name, and a larger area containing its members.

Every element of the model can own comments [CBR+17, p. 22]. While comments do not add semantics to the model, they can be useful to modelers to understand the model. Figure 2.9 shows how comments are visualized using a rectangle with a bent top-right corner. Comments are connected with a dashed line to the element they describe.

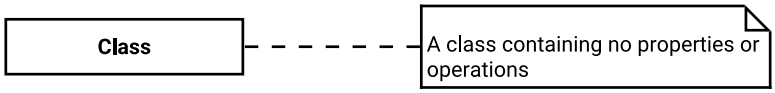

Figure 2.9: UML comment describing a class.

2.2 Programming Language Concepts

As this thesis includes programming language design, we first need to introduce some important concepts. In this section, we give an overview of important concepts relevant to this thesis. For a more complete introduction to programming language design, we recommend the textbook by Mogensen [Mog22] and related literature.

Syntax

The syntax of a programming language defines which textual artifacts are syntactically valid programs in said programming language. It is commonly defined in several layers: The first common layer is lexic. A lexer uses a regular lexical grammar to split the source code into so-called tokens. Typical tokens include identifiers, keywords, string literals, and number literals. Modern programming languages usually define tokens using regular expressions [Mog22, p. 50]. All further steps in the pipeline of a programming language then operate on the sequence of tokens produced by the lexer, instead of directly operating on the source code. The next layer defines the concrete syntax of the language. It uses a context-free grammar which operates on the token defined in the lexical grammar. A parser uses this grammar to build the Concrete Syntax Tree (CST). Usually, this is done as a bidirectional mapping, thus, a CST can be transformed back into the sequence of tokens and thus into the source code. As a result, formatters usually operate on the CST. However, for the parser to be able to use a grammar, it needs to be unambiguous, often resulting in a fairly complex CST. Therefore. further steps in the pipeline, e.g., interpreters, use the abstract syntax instead of the concrete syntax of a programming language. An Abstract Syntax Tree (AST) is obtained by simplifying the CST to only contain the actual meaning of the program.

Scopes

Most programming languages support named entities like functions and variables [Mog22, pp. 105-109]. A scope is an area in the program, where a binding of a value to a name is valid. Typically, multiple types of scopes exist: Functions provide a so-called function scope. It is created when entering the function and is valid inside the function. In contrast, the global scope is valid for the whole execution of a program.

In general, two types of scoping rules exist: dynamic scoping and static/lexical scoping. When using static scoping, the visibility of bindings is determined by the syntax of the program. In contrast, when using dynamic scoping, bindings are determined by the execution history.

The difference between those scoping rules can be explained using Listing 2.1. In line 1, the code defines the variable `x` with value `0`. Lines 2-4 declare a function `f`, which logs the value of a variable `x` to the console. Lines 5-8 declare a function `g`, which first declares the variable `x` with value `1`, and then invokes `f`. Depending on the scoping rule, the invocation of `g` in line 9 results in different outputs. With static scoping, it outputs `0`. The variable `x` is not defined in the function scope of `f`, and its lexical parent scope is the global scope, thus `x` refers to the variable declared in line 1. However, when using dynamic scoping, the program outputs `1`. Again, the variable `x` is not declared in the function scope of `f`. However, `f` was called inside function `g`. Therefore, the program next tries to find the variable in the function scope of `g`, where it is defined, thus printing `1` to the console.

Listing 2.1 Example code with a JS-like syntax used to explain static and dynamic scoping.

```
1 const x = 0;
2 function f() {
3     console.log(x);
4 }
5 function g() {
6     const x = 1;
7     f();
8 }
9 g();
```

Note that with dynamic scoping, the parent scope is determined using the execution history, here the call stack, instead of the lexical structure of the program. In languages with first-class function support, meaning functions can be used as values and therefore be saved in variables and returned from other functions, static scoping requires closures. A closure consists of the function, and the reference to its environment, typically the scope in which it is created. Thus, even if the function is passed as a value to another context, a reference to its lexical parent scope is retained.

Type Systems

In a programming language, types allow differentiating between values with different properties [Mog22, pp. 167-168]. Their main purpose is to prevent errors by preventing values without specific properties being used where values with specific properties are expected. The type system of a programming language is a set of rules with respect to typing. It can be categorized into two main axes: First, one can distinguish between static and dynamic typing. In statically typed languages, one can determine the type of variables and other constructs on a source level. Validation is typically performed as a step in the compile chain. On the other hand, dynamically typed languages associate types with values, type checks are, if existing, performed at runtime, not at compile time. For example, JavaScript (JS) is dynamically typed, while Java is statically typed. Second, one can distinguish between strong and weak type systems. Strongly typed languages strictly verify that properties associated with a type are not violated. For example, if a number is provided where a string is expected, a strongly typed language will raise an error. Note that dynamically typed languages can use strong typing, too. On the other hand, weakly typed languages do not perform such validations, and often automatically try to convert types where possible. In general, strong vs weak typing represents a more gradual scale without precise definition.

2.3 Domain-Specific Language

As defined by Fowler and Parsons, DSLs are computer languages focusing on a particular domain [Fow10]. Compared to General-purpose Programming Languages (GPLs) like Java or C++, they trade generality for improved expressiveness in their particular domain [MHS05]. Thus, they improve ease of use in that particular domain, resulting in improved productivity and reduced maintenance expenses. They achieve this improved expressiveness by providing notations and

constructs towards their specific domain. In an ideal case, there already exist domain-specific notations which can be directly reused. However, such constructs introduce, if extensively used, additional complexity. Therefore, good tool support is crucial to improve user satisfaction. One may note that in general, there is no clear separation between DSL and GPL. Instead, there is a gradual scale of generality, ranging from very specific, to more general-purpose languages with domain extensions.

One of their main benefits is to allow domain experts with no or limited knowledge in programming to define domain knowledge in a machine-readable way. On the other hand, they also make the domain more accessible to software developers. Today, a variety of different DSLs are in use, and their benefits are empirically proven. They are common in Model Driven Software Development (MDSD), where they are used to define models in a clear and precise way. Thus, it is no surprise that they are also common in diagram modeling and are therefore well-suited for our use case.

Different types of DSLs exist. While some can be executed in one or another way, including interpretation and compilation, others are purely descriptional. Generally, DSLs can be divided into two categories: internal DSLs and external DSLs.

External Domain-Specific Language

External DSLs are computer languages independent of other programming languages. Thus, they usually use a compiler-like infrastructure [WB23, p. 357], similar to GPLs. They define their own concrete and abstract syntax, semantics, and are used with their own tools and backend. One of their main benefits is flexibility: it is up to the language designer how to design the language. This allows expressing data in a domain-specific way, which would not be possible in GPLs. However, this flexibility comes with two major downsides: tooling support and development effort. First, designing and developing a computer language is a major undertaking. A language engineer has to define its grammar, and implement or generate a parser for such grammar. Based on the resulting concrete syntax, one needs to define the abstract syntax. If execution is required, one needs to additionally develop an interpreter, transpiler, or compiler. Yet, to be efficient, tooling is required, e.g., syntax highlighting in an editor and auto-completion.

Language workbenches can be used to overcome these downsides. Typically, they both provide an environment in which the DSL is developed in, and tools to be used with the developed language [ESV+13]. Thus, language workbenches not only reduce development effort but improve the user experience of the resulting DSL. Today, popular open-source language workbenches include *Xtext*[2], *MontiCore* [GKR+08][3] (both Java), *textX* [DVMV17][4] (Python), *langium*[5] (TypeScript (TS)/ TS).

Language Composition is another feature commonly provided. It allows defining DSLs by extending or aggregating other DSLs. A common use-case is to add GPL-like functionality, e.g., expressions, to a DSL. Instead of developing these features from the ground up, one can simply extend a base expression language. Out of the mentioned language workbenches, this feature is most present

[2] https://www.eclipse.org/Xtext/
[3] https://monticore.github.io/monticore/
[4] https://textx.github.io/textX/
[5] https://langium.org

in MontiCore. In [HKR21, p. 110], Rumpe et al. describe four composition features: Language aggregation allows to use different languages together, and language embedding allows to embed one language into another. Language inheritance and language extension both allow reusing a language while modifying some of its elements, with extension being the more conservative approach, disallowing some potentially dangerous modifications. Both Xtext and MontiCore provide grammars that can be extended. In general, composition in textual external DSL is challenging as the grammar needs to remain unambiguous.

Projectional Domain-Specific Language

In contrast to parser-based editors, which store the source code as a single source of truth, projectional editors store the syntax tree persistently. When editing, the projectional editor shows a projection of the syntax tree. For DSLs, this is usually a textual representation. Additional representations include graphical and tabular representations, with some editors allowing embedding different representations, e.g., including a table in an otherwise textual editor. By not relying on a parser, languages can be composited even if the resulting syntax would be ambigous [VP12]. Instead, disambiguation is done by the user when entering potentially ambiguous code. However, this composability benefit comes at a cost, which mainly affects entering code, selecting and modifying code, and infrastructure integration [BVJ+16]. For example, when entering code, historically, editors enforce the tree structure of the syntax tree on the user. For example, when entering an expression like `a + b`, the editor would require the user to first enter the plus symbol before the two variables to add can be entered. Additionally, selection is limited, as it is done on the syntax tree level, not the textual level. Also, infrastructure integration, e.g., code generation, can be challenging. Today, JetBrains MPS[6] is the most popular implementation. Berger et al. show that even for inexperienced users, basic editing efficiency can be achieved quickly in this implementation [BVJ+16]. However advanced editing requires more experience.

Internal Domain-Specific Language

In contrast to external DSLs, internal DSLs are implemented as a library of a host GPL [WB23, pp. 357-35]. This approach is of particular use if the DSL requires GPL features, e.g., logic, as features of the host language can often be used. In this case, the host provides not only the syntax, but also a runtime. Besides, this approach results in further advantages: First, tooling for the resulting DSL is usually provided by the tooling for the host language. Additionally, composability can be achieved by simply adding multiple libraries, disambiguation is then ensured by the host language. However, there are also drawbacks. Most importantly, the syntax of the DSL is limited to what the host supports, as each valid DSL program must automatically be a valid program in the host language. Furthermore, the host GPL limits where the DSL can be used, as in order to evaluate a DSL program, the host program needs to be executed. This can be problematic for compiled languages. Internal DSLs can be split into two categories: ones that use deep embedding, and ones that use shallow embedding [GW14]. When using deep embedding, terms of the DSL, construct,

[6] https://www.jetbrains.com/mps/

when executed, just an AST. Further traversal, e.g., execution, then operates on this constructed AST. On the other hand, when using shallow embedding, terms of the DSL directly implement the semantics, and no intermediate AST is used.

Generally speaking, when used with an appropriate library, any GPL can be used as DSL [MHS05]. Historically, functional languages like Lisp have been popular for implementing internal DSLs. Due to its homoiconicity, when implementing the DSL, one can operate on the code like on any other data structure. Additionally, by using S-expressions, Lisp allows for metaprogramming in a way consistent with the rest of the language. Today, internal DSLs are popular with most programming languages [WB23, p. 389]. To select an appropriate host GPL, its requirements need to be analyzed. Often, a DSL is implemented to allow programmers to define data structures in a specific GPL in a more natural way. A common example of this includes HyperText Markup Language (HTML) builders. In this case, the hosting language is already given. In other cases, existing code needs to interact with the DSL code, or required execution environments limit the choice the language designer has. Last, one needs to consider the syntax one wants to achieve. While in some host GPLs, one is limited to a function call-based syntax, other languages allow for more flexibility. Important aspects include operator overloading, and infix functions in order to use functions like operators. Examples of languages that provide such features include Scala and Kotlin.

2.4 Technologies

This section introduces some of the technologies used for this thesis, starting with the *Language Server Protocol (LSP)* in Section 2.4.1, followed by *Eclipse Sprotty* in Section 2.4.2

2.4.1 Language Server Protocol

Traditionally, language support for Integrated Development Environments (IDEs) and editors had to be implemented separately for each platform to support. However, as most modern IDEs and editors are quite similar regarding programming language support, such functionality can be provided separately from the environment, allowing for broader tooling support with reduced development effort. The most popular protocol/architecture is the *Language Server Protocol (LSP)*[7], initially developed by Microsoft for *VS Code*. Here, all language-specific functionality, e.g., completion, diagnostics, and formatting, is implemented using a language server. The language server is connected via the Language Server Protocol (LSP), a *JSON-RPC*-based protocol, to the IDE-specific language client, which itself interacts with the IDE/editor. Due to the popularity of the protocol, language clients for several platforms, ranging from web-based editors like *Monaco Editor*, to full-fledged IDEs like *Eclipse* and *Visual Studio*[8]. Thus, a single language server can provide language-specific tooling support to a large variety of different platforms.

[7] https://microsoft.github.io/language-server-protocol/

[8] https://microsoft.github.io/language-server-protocol/implementors/tools/

2.4.2 Eclipse Sprotty

Eclipse Sprotty[9] is a diagramming framework based on web technologies, primarily developed by TypeFox. Diagrams are rendered using Scalable Vector Graphics (SVG), resulting in high-quality vector graphics and great extensibility. Features provided by Sprotty include animations, an interactive canvas with zoom & translation support, and layouting of diagram elements. A detailed description of Sprotty's architecture can be found in its wiki[10]. For the scope of this thesis, its client-server support is of particular interest. When using this feature, the model is stored on a diagram server, which communicates via actions with the editor client. To display the diagram, the model is transformed into a view on the client. Layouting of diagrams can be performed on the server, client, or a combination of both, depending on the use case. As actions can be passed via JSON-RPC, integration into a language server is possible, allowing for the creation of diagrams based on textual definitions on the language server.

[9] https://github.com/eclipse-sprotty/sprotty

[10] https://github.com/eclipse-sprotty/sprotty/wiki/Architectural-Overview

3 Related Work

This section describes related work relevant to this thesis. We focus on two areas: First, in Section 3.1, we detail related academic work in the area of hybrid modeling, and diagramming using the LSP. We also describe our research methodology. Second, we give an overview of tools used for both diagramming and modeling in Section 3.2.

3.1 Academic Related Work

Methodology

In order to identify related work, we used Google Scholar[1] as a search engine. For related work, we decided to focus on the hybrid editing aspect of this thesis. Therefore, we used the following search queries:

- hybrid graphical textual diagram
- hybrid graphical textual modeling
- blended modeling

Additional search queries we considered, which however did not give usable results, include "hybrid modeling" and "hybrid diagramming". For each search query, we selected the first 10 results. Based on the title and abstract, we excluded work not relevant to the scope of this thesis. To further collect related work, we performed both forward and backward snowballing.

Results

In [CTVW19], Ciccozzi et al. describe the concept of blended modeling. When using blended modeling, users interact with the same model (abstract syntax) through multiple notations (concrete syntaxes). To improve modeling efficiency, blended modeling allows for temporary inconsistencies between both different notations and a notation and the underlying metamodel. In contrast to multi-view modeling, which focuses on appropriate views for different stakeholders, blended modeling aims at improving editing user experience by using different notations [DLP+22]. Based on a user story, they identify the following improvements in usability: First, multiple notations allow understanding the same model differently, improving understandability. Next, different (groups of) users prefer different concrete syntaxes, e.g., some users prefer graphical ones, while others prefer textual ones. Therefore, providing multiple concrete syntaxes both improves acceptability

[1] https://scholar.google.com

and learnability. Similarly, changeability is improved, as different aspects of the model can be manipulated more easily in an appropriate notation. Last, tabular notations can improve analysability. In [DLP+22], David et al. perform a systematic study on blended modeling in commercial and open-source model-driven software. Out of the 26 analyzed tools, all support graphical modeling, while 19 support textual, 13 tabular, and seven tree-based modeling. Also, 62% allow for synchronized navigation, often resulting in graphical and textual navigation being synced.

Addazi et al. apply the concept of blended / hybrid modeling to modeling for UML and profiles [ACLP17]. They propose a framework allowing for both graphical and textual modeling. Their implementation is based on the *Eclipse Modeling Framework (EMF)*, with the textual part being provided by *Xtext* and the graphical part by *Eclipse Papyrus*. Textual grammars are generated based on the selected UML profile and optionally manually adopted for a better user experience. Changes between views are synced on-the-fly. Also, views do not need to be complete. In particular, their textual editor can work on a more complex underlying model, and changes to the textual concrete syntax are propagated to the model by updating, not replacing it, keeping information not present in the textual concrete syntax. The authors evaluate their concept using the *MARTE UML profile*. A set of modelers use the graphical, tabular, and textual notation for two create and modify modeling tasks. Depending on the task, either the graphical or textual notation is most efficient, they show that a hybrid approach results in faster modeling.

In [CK19], Cooper et al. give an overview of hybrid graphical-textual languages using *Eclipse Sirius* and Xtext. As the main requirements of such languages, they identify

(1) syntax-aware editing, including error markers, syntax-highlighting, auto-completion, and refactoring

(2) references between textual and graphical views,

(3) uniform error reporting in both graphical and textual parts, and

(4) exposing the model to model-to-model management platforms.

The whitepaper “Xtext / Sirius - Integration” [TO] by TypeFox and Obeo is among the reviewed implementations. There, the authors describe the benefits of textual and graphical modeling, and describe two possibilities for hybrid editors with said technology stack: First, editing the same model both graphically and textually. Second, they embed an Xtext-based editor in the properties view of Sirius, resulting in an advanced text editing experience inside Sirius.

An alternative to these *Eclipse*-based approaches is presented by Glaser et al. [GB21]. *BIGER* provides hybrid textual and graphical modeling for Entity Relationship diagrams as a *VS Code* extension. It consists of three main components: a textual view, where an Xtext-based DSL is used to define the Entity Relationship model, a graphical editor, and an SQL code generator. Architecturally, the extension is realized as client-server architecture. The server is implemented as a language server using LSP, where both the textual and graphical models are stored. The graphical view is automatically generated based on the textual model and uses *Eclipse Sprotty* for its implementation. Supported graphical manipulations include renaming labels, and creating elements [Gla22]. To sync the graphical to the textual view, graphical edits are converted to textual snippets which are inserted/replaced in the textual model. A similar approach was presented by Walsh et al. [WDJ22]. There, the authors present an architecture for designing language servers for modeling languages using both graphical and textual notations. Their hybrid language consists of a hybrid language server and a hybrid language client. The textual language server communicates with the textual

client via LSP, while the graphical language server communicates with the graphical client via the *Graphical Language Server Platform (GLSP)*. In contrast to [Gla22], and similar to [ACLP17], the approach by Walsh et al. uses a shared underlying model, which is bidirectionally mapped to both the graphical and textual view. To evaluate their concept, the authors also present a prototype implementation for UML-RT, called the "RTPoet language".

3.2 Modeling & Diagramming Tools

Here, we categorize and describe tools commonly used for modeling and diagramming, with a focus on tools using a hybrid textual graphical approach. Tools were identified based on the review by Ozkaya [Ozk19], and the websites "Modeling Languages"[2] and "Software architecture tools"[3].

Diagramming Tools

Diagramming tools allow modelers to create diagrams directly, without specifying underlying models first. Diagramming tools can further be split into textual and graphical tools. Graphical diagramming editors provide the user with a canvas to draw diagrams. Popular tools include, among many others, *draw.io*[4]*/diagrams.net*[5], *Visio*[6], and *Lucidchart*[7]. Typically, these tools offer a wide range of elements and connections to use on the canvas. Provided elements range from general shapes and boxes to modeling language elements. For example, one can use both UML classes, and general shapes in the same diagrams.net diagram. As a result, those graphical diagramming tools typically do not enforce conformance to modeling language standards. Due to users creating diagram elements graphically, users are also responsible for layouting the diagram elements. Depending on the tool, different features are provided to help the modeler layouting the diagram, including snapping and aligning multiple elements. Some, for example *yEd live*[8], also support auto-layouting.

In contrast, textual tools allow the user to define diagrams by using a textual concrete syntax. The generated graphical diagram is purely read-only and used for presentation only. *Mermaid*[9] and *PlantUML*[10] are two of the most popular available tools. Here, the modeler defines the content of the diagram using a textual DSL. Listing 3.1 shows an example class diagram, as it can be used with PlantUML. In contrast to graphical editors, layouting is performed using a layout algorithm. User influence on the layout depends on the tool. For example, when using PlantUML, one can define the layout direction. Additionally, the (relative) length of edges can be specified, and the layout can be influenced further using invisible edges which affect the layout, but are not drawn. However, note that these only allow to influence layouting, the modeler cannot define an exact layout. Similarly,

[2] https://modeling-languages.com/uml-tools/, https://modeling-languages.com/text-uml-tools-complete-list/
[3] https://softwarearchitecture.tools/
[4] https://github.com/jgraph/drawio
[5] https://app.diagrams.net/
[6] https://www.microsoft.com/en-us/microsoft-365/visio/flowchart-software
[7] https://www.lucidchart.com/
[8] https://www.yworks.com/yed-live/
[9] https://mermaid.js.org/
[10] https://plantuml.com/

Listing 3.1 Example PlantUML class diagram.

```
@startuml
class Class1 {
    -id: int
    -name: string
    +method(): void
}
class Class2 {
    -value: double
}
Class1 --> "1..*" Class2 : contains
@enduml
```

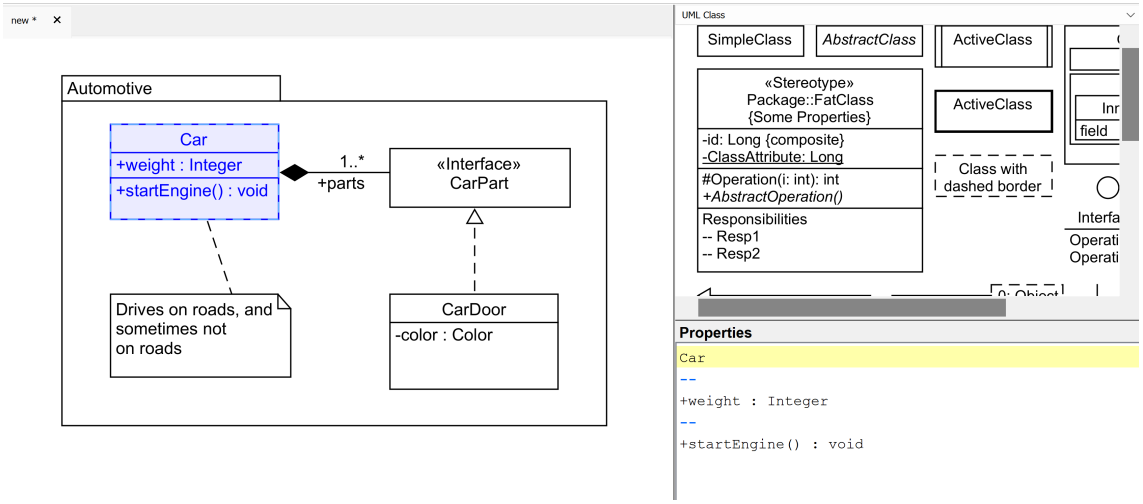

Figure 3.1: UMLet editor with an example diagram.

styling support varies across tools. Tools like *TikZ-UML*[11] provide lower-level access. Here, similar to graphical tools, layouting is performed primarily by the modeler, by specifying coordinates for diagram elements manually. Furthermore, some tools allow the programmatic definition of diagram elements. For instance, *Diagrams*[12] uses an internal Python DSL to define system architecture diagrams. Similarly, PlantUML provides preprocessing capabilities[13] which allows not only to use of expressions but also custom functions.

UMLet[14] [ATB03] is an example of a hybrid diagramming tool. Here, users both use graphical and textual editors to create diagrams. The editor consists of three main areas, as shown in Figure 3.1:

[11] https://perso.ensta-paris.fr/~kielbasi/tikzuml/index.php
[12] https://diagrams.mingrammer.com/
[13] https://plantuml.com/preprocessing
[14] https://www.umlet.com/

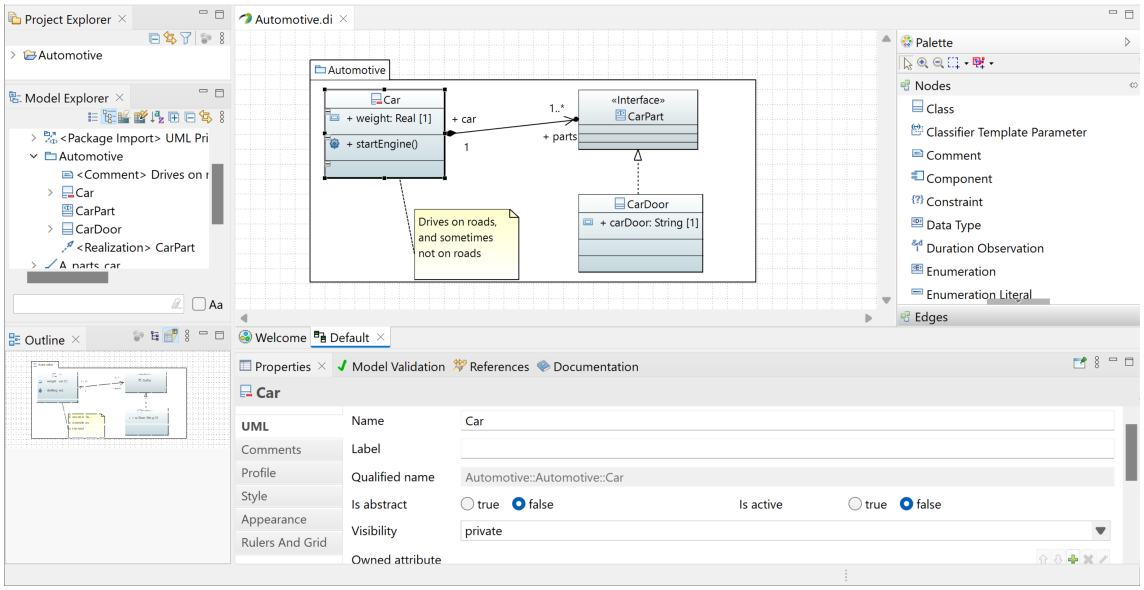

Figure 3.2: Eclipse Papyrus with an example UML class diagram, showing the Papyrus Editor, Model Explorer, and Property Editor for a class.

(1) The main canvas where elements can be graphically manipulated. This includes moving elements, resizing elements, and changing the routing of associations.

(2) The properties panel. In contrast to other tools, a diagram element is defined using a textual concrete syntax. Therefore, the properties editor is a text editor allowing to manipulate the content of a single selected diagram element.

(3) An element palette. It allows users to drag and drop diagram elements or copy-paste elements on the main canvas. Furthermore, the provided elements also showcase how to use the textual concrete syntax to configure the element.

Modeling Tools

When using modeling tools, modelers separately define a model, and one or more views based on the model, the diagrams. While modeling tools provide features beyond diagramming, including model validation, model-based simulation, and code generation, for the scope of this thesis, we want to focus on how different tools enable modelers to create diagrams. Tools can be categorized both on how they allow modelers to create the models, and how diagrams can be edited.

Tools that allow modifying both the model and the diagram in a single combined view form the first major group. Well known tools include Eclipse Papyrus[15], *Enterprise Architect*[16], and *StarUML*[17]. Figure 3.2 shows the primary editor of Eclipse Papyrus for an UML class diagram. It supports both modifying the layout of the diagram and adding new and modifying existing elements, resulting

[15] https://www.eclipse.org/papyrus/
[16] https://sparxsystems.com/products/ea/index.html
[17] https://staruml.io/

in a user experience comparable to graphical diagramming tools. Yet, compared to diagramming tools, stricter conformance to the used metamodel is enforced. For example, creating an association between a class and a comment is not supported and therefore prevented by the editor. Furthermore, as the diagram and model are separated, additional views for modifying the model can be provided. In the case of Eclipse Papyrus, this includes a model explorer and a properties view which allows for modifying the properties of diagram elements.

Textual editors are an alternative to graphical and property-based editors. Here, modelers define the model using a textual concrete syntax. As an example, *Structurizr*[18] provides a DSL to define C4 models. Interactive diagrams are automatically generated based on the textual description, with an automatically generated layout. In contrast to diagramming tools using auto-layout, e.g., PlantUML, manually editing the layout is supported, and persisted separately from the model definition, as for graphical modeling tools.

[18] https://structurizr.com/

4 Requirements Engineering

Before being able to create the architecture and design parts of our framework, first, we need to collect and evaluate relevant requirements. Therefore, this chapter gives an overview of our requirements engineering process and the elicited requirements. First, Section 4.1 describes how we structured our requirements engineering phase, and in which areas we collect requirements. Then, Section 4.2 lists requirements we collected ahead of the expert interviews. Section 4.3 details the interview process and lists the requirements collected in that process. Last, Section 4.4 presents the results of our expert survey, where we evaluated and prioritized some of the requirements, and briefly introduce how we evaluate design decisions as part of the survey.

4.1 Overview and Process

In order to perform requirements elicitation, first, a process has to be selected. We outline our process as depicted in Figure 4.1: First, we collect initial requirements. Then, we conduct expert interviews to refine those initial requirements, collect additional requirements, and identify important design decisions. Based on those collected requirements, we conduct a survey to evaluate and prioritize a subset of the collected requirements. However, our concept covers a wide range of technical challenges at different levels of abstraction, starting with designing a diagramming framework and ending with creating an interactive graphical editor. Due to time constraints present both on our side and the experts, for the interview part of the process, we decided to focus on the areas of interest

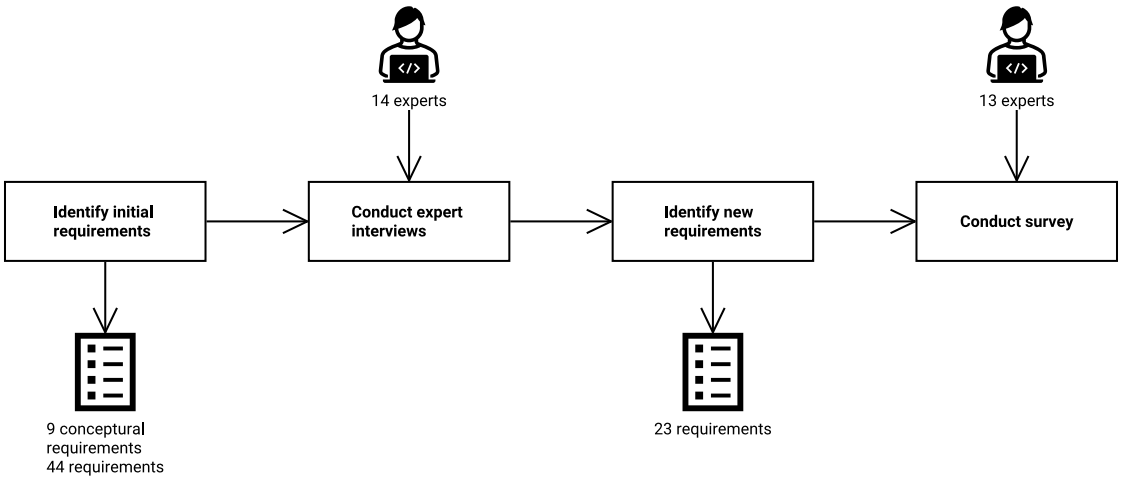

Figure 4.1: Overview of the requirements engineering process.

where we expect their feedback to have the greatest impact. In order to do so, we first need to select the areas in which we want to collect requirements. Therefore, based on our concept, we identify the following areas of interest:

I Environments Environments the framework is used in, e.g., a web browser

II Diagram & Diagram DSL Features of the generated diagrams itself, and the DSL used to define such diagrams. Independant of the graphical and textual views of the editor.

III Programming Language General-purpose programming language features of the Diagram DSL

IV Interactive Graphical View The interactive graphical part of the hybrid editor.

V Textual View The textual part of the hybrid editor.

VI UML Class Diagram DSL Graphical and DSL features required for the first use-case of the framework, UML class diagrams.

Out of these areas, we decide to focus on areas **II**, **IV**, and **VI** for the expert interviews. Multiple reasons exist for this decision: First, those two areas cover the most novel aspects of our approach, especially area **IV**. Therefore, by just mainly relying on requirements collected from prior expertise and existing tools, important aspects will likely be missed. Second, these areas cover features the user interacts directly with when using the hybrid editor. Thus, only interviewing experts can ensure that important user requirements are collected. Last, these areas cover high-level features of the framework. Therefore, domain experts (modelers) without a technical background can provide feedback more easily.

As a result, we decided to first implement the low-level features of the framework, and then conduct the interviews based on that version of the framework. This includes basic features of the diagram DSL, the general-purpose programming language features, a basic text editor, and an initial version of the interactive graphical view. The graphical view should at least support one interaction that modifies the source code. This serves two main purposes: First, it ensures the technical feasibility of our main concept, mainly the hybrid editor. At this point in the process, minor changes to our concept may be required to ensure implementing the framework is possible in the time frame, which may influence the requirements of the higher-level features like the graphical view. Second, having a basic version of both parts of the hybrid editor allows for easier prototyping during the interviews. For example, different syntaxis for diagram DSL constructs can easily be shown with proper syntax highlighting. Also, it demonstrates the concept of making graphical changes resulting in changes to the source code, thus giving the experts an example of how hybrid editing can work in practice.

4.2 Initial Requirements

This section lists the initial requirements collected before starting the implementation of the framework. It is structured based on the identified areas of interest from Section 4.1.

Initial requirements are collected before starting the implementation of the framework and before performing the expert interviews. However, we already collected related work and existing tools. Therefore, those requirements are mainly based on functionality that can be derived from our concept, and functionality present in existing technologies. Additionally, ideas developed during the conceptualization phase of this thesis, together with the supervisor and examiner, are included.

Apart from the already mentioned subdivision into different areas of interest, those requirements can further be subdivided into regular (initial) requirements and conceptual requirements.

4.2.1 Conceptual Requirements

Unlike regular initial requirements, conceptual requirements must be directly derived from our concept of work. Therefore, those can be regarded as a formalization of our concept of work. This formalization is required in order to evaluate our main research question **RQ 1**: Only if all conceptual requirements are met, we can conclude that our concept is technically feasible. Additionally, identifying conceptual requirements is necessary for evaluation and prioritization of other initial requirements, and requirements collected during Section 4.3. As described in Section 4.1, prioritization of requirements is done using a survey with experts. However, as shown in Chapter 3, multiple approaches exist, including purely graphical and purely textual ones. There is a risk that if a majority of the selected experts prefer one of those approaches, only requirements of that approach get prioritized and therefore implemented in the scope of this thesis. Thus, conceptual requirements must be assigned the highest priority, independent of the prioritization process of other requirements. Additionally, requirements collected during expert interviews might be in conflict with our concept, and such must be identified and eliminated. We achieve this by identifying requirements that conflict with at least one conceptual requirement.

The following eight requirements can be derived directly from our concept:

CR 1: Hybrid editor The editor the modeler interacts with provides both a textual and a graphical view. The modeler can manipulate the diagram both in the textual, and graphical view.

CR 2: Diagram DSL In the textual view, the modeler uses a diagram DSL.

CR 3: Programming Language Features The diagram DSL provides general-purpose programming features, including control flow structures like if and while, and user-defined functions.

CR 4: Code as Single Source of Truth A diagram is fully defined by the DSL code. As a result, the code can be shared with other modelers and put into version control.

CR 5: Live-Synced Editing The graphical and textual views of the hybrid editor are live-synced. Modelers should see the effect of graphical edits immediately in the textual view, and the effect of textual edits in the graphical view. As immediate updates are likely, not feasible, syncs should occur as often and fast as possible.

CR 6: Manual Layouting The modeler can layout diagram elements, e.g., classes and associations in a UML class diagram, manually, meaning positions and dimensions are defined by user input, and not by a layout algorithm.

CR 7: Precise Layouting The modeler is able to define positions and dimensions of diagram elements up to (near) arbitrary precision.

CR 8: Styling The modeler can style the diagram and its elements, with detailed control, e.g., to modify the font size or font family.

CR 9: Modular Extensible Framework Modules allow the adoption of the framework to support new diagram types.

4.2.2 Environmental Requirements

For this area, we identify different environments in which the framework can be used. Categorizing those requirements is important for designing an architecture for the framework, as different environments may (1) limit the use of available technologies (2) imposes constraints on how the framework needs to be subdivided into components.

Based on existing tools, we collect the following requirements:

RE I.1: Web Browser Support As a modeler, I want to be able to create diagrams directly from a web browser, without being required to install any tools locally.

RE I.2: IDE Support As a modeler, I want to be able to edit diagrams inside my favorite IDE. Native IDE features should be used wherever possible.

RE I.3: CLI support As a modeler, I want to be able to render diagrams using a Command-line interface (CLI) tool, e.g., as part of a Continuous integration (CI) pipeline. This should be possible in non-graphical environments.

RE I.4: Consistency The resulting diagrams should be consistent across different environments.

RE I.5: Ease of Use The editing user experience of the hybrid editor should be similar across different environments. If different environments enforce the usage of different underlying technologies, those implementation details should be transparent to the modeler.

RE I.6: Integration into Web-Based Tools As a web developer, I want to be able to integrate the hybrid editor into other pages. Importantly, this should be possible (1) independent of the used web framework (2) for purely static pages without a web server (3) for browser extensions.

4.2.3 Diagram & Diagram DSL Requirements

As defined by our concept, a diagram DSL is used to define diagrams. This area covers the requirements of both the diagram DSL itself and the diagrams created by it. As defined by **CR 9**, our goal is to develop a general framework that can be adopted to different diagram types. Therefore, requirements in this area must not be specific to a certain type of diagram. Yet, different families of diagrams might require different features. While we may extend our approach in the future, to limit the scope of this thesis, initially, we want to focus on graph-based diagrams, e.g., UML class, component diagrams.

Here, we obtain the following requirements:

RE II.1: Extensible Diagram Types The framework supports the existance of different diagram types. There is a DSL construct by which the modeler can define which diagram is used. Developers can implement new diagram types.

RE II.2: Diagram Elements Diagram elements are supported. A diagram element consists of a content element (e.g. a UML class), a position on the canvas, and a size.

RE II.2.1: Point Types To position diagram elements on the canvas, modelers can use different point types including

RE II.2.2: Absolute Point A point defined by an absolute x and y coordinate

RE II.2.3: Relative Point A point defined by an offset in the x and y direction, and a point to which it is relative.

RE II.2.4: Line Point A point defined by a line (an association or the outline of a diagram element), and a position on the line.

RE II.3: Associations Associations are supported. An association is a line consisting of one or more segments, with an optional marker (graphical element) at the start and end of the line.

RE II.3.1: Line Association Segment An association segment that goes directly from one point to another point.

RE II.3.2: Bezier Association Segment An association segment which is a bezier curve.

RE II.3.3: Axis-aligned Association Segment An association segment that connects one point to another point only using horizontal and vertical lines.

RE II.3.4: Association Builder DSL Construct The diagram DSL provides a syntax to allow modelers to easily define associations that consist of multiple segments. Associations consisting of segments of different types should be supported too. For common constructs, e.g., a polyline consisting of multiple line segments, higher-level constructs should be provided.

RE II.4: Styling All graphical elements can be styled by the modeler.

RE II.5: Styling DSL A declarative styling DSL is provided to simplify styling.

RE II.6: Low-level Graphics Framework A low-level graphics framework is provided to allow modelers to extend diagrams with graphical features not provided by default. Common graphical elements, including text, polygons, and paths should be supported. Also, the manipulation of elements created by the diagram framework should be supported.

RE II.7: Diagram Export Diagrams can be exported to different file types.

RE II.7.1: SVG Export Diagrams can be exported to svg.

RE II.7.2: PDF Export Diagrams can be exported to pdf.

RE II.7.3: PNG Export Diagrams can be exported to png.

4.2.4 Programming Language Requirements

As defined by our concept, detailed in **CR 2**, the diagram DSL should provide programming features. These capabilities allow modelers to script repetitive elements of a diagram and extend the provided functionality. Especially, in combination with **RE II.6**, a modeler can create graphical elements not included in the framework by default when required. Yet, multiple technical approaches exist which allow for integrating general-purpose programming language into DSLs. Thus, collecting these requirements is necessary in order to choose a suitable approach.

Based on existing DSLs that use this approach, including *PlantUML* and *Structurizr*, we consider the following requirements to be the most important: As a modeler, I want to . . .

RE III.1: Imperative Paradigm In order to allow for user-friendly scripting comparable to commonly used scripting languages, the imperative programming paradigm is used. Importantly, the diagram DSL is not purely declarative.

RE III.2: Control Flow Structures use loops and conditional execution in order to automate repetitive parts of designing a diagram.

RE III.3: Data Types use common data types, including strings, numbers, booleans, objects, and lists.

RE III.4: User-Defined Functions be able to define custom functions to allow for the reuse of functionality.

RE III.5: Extend DSL Functionality be able to define custom structures that are similar in use and functionality to already existing diagram DSL constructs. As an example, if there already exists a construct that allows the user to define classes, it should be possible to define a construct that allows for creating interfaces.

4.2.5 Interactive Graphical View Requirements

As part of our concept of a hybrid editor, the interactive graphical view is one of the two views a modeler interacts with when editing diagrams. It serves several main purposes: First, when using the textual representation, it allows the modeler to gain live feedback on the effect of textual edits. Second, it allows exploration of the diagram. Third, as defined by our concept, it allows the user to graphically manipulate the diagram. For the initial requirements, we want to focus on features where we see the most benefit for a modeler. However, as this is one of the areas we focus on during our expert interviews, we expect to collect more requirements later in the process.

As a modeler, I want to . . .

RE IV.1: Navigatability navigate the diagram interactively, by zooming and panning the diagram canvas. Doing this should be possible without affecting the diagram itself.

RE IV.2: Move Elements modify the position of diagram elements by dragging them on the canvas. This includes moving points which define an association, allowing to edit the association graphically.

RE IV.3: Resize Elements resize diagram elements, e.g., classes in a UML class diagram.

RE IV.4: Rotate Elements rotate diagram elements, e.g., labels on associations.

RE IV.5: Multi-Select select multiple diagram elements at once. If multiple are selected, manipulation operations like move and resize should affect all selected elements.

RE IV.6: Auto-Layout auto layout diagram elements, e.g., by clicking on a button. As an example, for a UML class diagram, this would both affect classes and associations between them. It should be possible to auto-layout the complete diagram or only selected elements.

Those requirements can be split into two categories: First, some features only affect the graphical view, interactions that temporarily modify the graphical view, but are not persisted. From the requirements above, **RE IV.1** and **RE IV.5** fall into this category. Second, other features allow the user to modify the diagram itself, interactions which must be persisted in the diagram DSL. As we consider defining the layout one of the more challenging tasks when using a purely textual concrete syntax, here, we focus on features that allow the modeler to modify the layout of the diagram graphically. However, we expect to collect more requirements of this category during the expert interviews.

4.2.6 Textual View Requirements

RE V.1: Syntax-heighlight Diagram DSL code in the textual view should by syntax-heighlighted.

RE V.2: Auto-Format It should be possible to format the DSL code. The formatter should be configurable, meaning, e.g., indentation and maximum line length.

RE V.3: Error Highlighting Errors in the DSL code should be highlighted, and an error message should be made available to the user.

4.2.7 UML Class Diagram Requirements

UML class diagrams are defined by the OMG UML specification. However, due to being a complex standard, it is not the goal of this thesis to implement the full UML standard with respect to class diagrams. Thus, collecting these requirements enables us to select the most important features to implement. Based on other commonly used tools, including *Mermaid*, PlantUML, and *diagrams.net*, and based on prior experience, we expect the following features to be of most importance to modelers:

As a modeler, I want to. . .

RE VI.1: Classes model classes. Classes should have a name.

RE VI.2: Fields and Methods add fields and functions to classes. It should be possible to subdivide the body of the class into multiple sections, separated by a horizontal divider.

RE VI.3: Stereotypes add stereotypes to classes, e.g., to mark a class as an interface.

RE VI.4: Abstract Classes mark classes as abstract. This should result in the title being written in italics, however, this style behavior should be customizable by the user.

RE VI.5: Associations model associations between classes. Different types of associations should be supported, e.g., extends, composites, and aggregates. Also, I want to model the navigability of the association, by adding arrowheads and crosses to the end of associations.

RE VI.6: Association Labels add labels to associations, e.g., do model role name and multiplicity. Such labels should be positionable at arbitrary positions at the association.

4.3 Expert Interviews

In this section, first, we give an overview of our expert interview process. As part of this, we identify the goals of those interviews and detail how the interviews were conducted. Last, we evaluate the interviews and collect new requirements.

4.3.1 Overview and Process

As stated in the introduction, our main user group for our tool is researchers conducting research in the field of computer science. Thus, we decided to invite 14 current and former researchers working in the area of software engineering to our interviews, of which all 14 responded and agreed on having the interview. We conducted the interviews over the course of two months. As described in Section 4.1, we want to focus on areas **II Diagram & Diagram DSL**, **IV Interactive Graphical View**, and **VI UML Class Diagram DSL**. To guide the interview, we define interview goals, which cover that areas:

Goal 1: Collect Graphical Features To fulfill this goal, we need to collect requirements that describe how a modeler wants to interact with our graphical view. We focus on two areas: First, interactions that modify the diagram itself, and therefore must result in a modification of the diagram code. Second, interactions that only operate on the graphical view, allow the modeler to navigate it but do not modify the diagram itself.

Goal 2: Design Class Diagram DSL As we choose UML class diagrams as the first use-case of our framework, we need to design the DSL which is used to model such diagrams. Additionally, as part of this goal, we identify additional aspects of class diagrams not yet covered by Section 4.2.7 which users want to be able to model.

Goal 3: Collect Diagram Features This covers requirements of the general diagram framework, the diagram DSL, and the rendered diagram itself. However, for the interviewee, it can be difficult to come up with these more abstract requirements. Especially, it can be difficult to identify if a DSL requirement is part of the general diagram DSL, or the UML class diagram-specific extensions. Thus, for the interview, we focus on the example of UML class diagrams and derive general diagramming requirements from there.

To prepare the interviewees, we created a short document which is provided to them several days before the interview. The first part of the document introduces the interviewees to our concept and explains the interview process. First, we first lay-out the background and introduce the two state-of-the-art approaches for creating UML class diagrams, (1) Descriptive DSL with auto-layout (2) Graphical editors. Based on this, we highlight the shortcomings of both those approaches and introduce the concept of hybrid modeling. A video then showcases how a modeler can interact

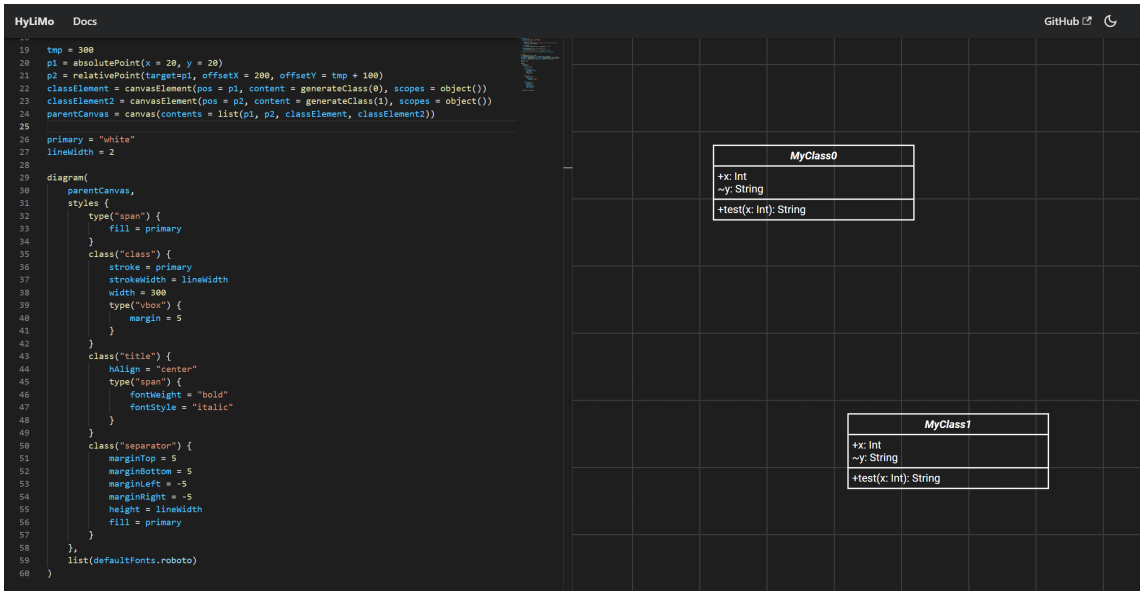

Figure 4.2: View of the hybrid editor with the example diagram provided to the interviewees.

with a hybrid editor, and we list our most important features. Last, we give an overview of the process described earlier. The following part aims at helping us at fulfilling our interview goals, by providing a DSL example, and some questions we try to answer during the interviews. Listing 4.1 shows the DSL example code provided. It shows how one could define two classes, an association between those classes, and some styles. It also defines the layout of the association and the position of each class. Following, a set of questions is presented. For the UML class diagram DSL, those questions focus on how one declares fields and methods as part of a class, and how to declare associations between classes. Regarding the graphical view, the questions focus on which graphical features should be provided. Last, we provide a link to our hosted version of the hybrid editor, and an example diagram that can be pasted into the editor. Figure 4.2 shows how this example code renders two classes with fields and methods as entries. In addition to showing how classes could look, it allows the user to graphically move these two classes and observe the resulting modification of the code on the left side, thus demonstrating one of our main concepts. However, it is important to note that this example, unlike the DSL example in Listing 4.1, does not make use of the (class) diagram DSL, and thus, does not show how a modeler will interact with our finished framework. This is also communicated to the interviewee before the example is shown to prevent confusion. The document ends with a short documentation of the programming language features already present in the diagram DSL. Its purpose is to allow the interviewee to understand the demo code from above, in case they wish to do so, as no other documentation exists when the interviews take place. However, we do not expect interviewees to read that part of the document, and its content is not part of the interview itself.

For the process, we choose one-on-one interviews, with a planned duration of around one hour. The interviews usually lasted between 45 and 90 minutes, except for one interview taking approximately 130 minutes. The interviews are structured in the following way: First, we introduce our concept by going over its main features and demonstrating the example code. The level of detail of this introduction is based on how much the interviewee prepared for the interview, ranging from next to no preparation to having read the provided document and having tried out the provided

Listing 4.1 DSL example code provided to the interviewees as part of the introduction document, and used during the interviews

```
classdiagram {
    House = class("House") {
        "roomCount" : Int
        "wallColor" : "Color"
        "createRoom" : fun("floor" : Int) => "Room"
    } at pos(0, 100)

    Room = class("Room") {
        +("size") : Double
    }

    House --> Room {
        start = 0.5
        over = list(pos(200, 300), pos(400, 500))
        end = 0.5
    }

    styles {
        class("class") {
            minWidth = 100
            maxWidth = 600
        }
    }
}
```

example. Then, the interviewee gets to provide feedback, ask questions, or show their ideas. Some interviewees already prepared their ideas or questions before the interview, while others came up with questions and ideas on the spot based on the given introduction. In some interviews, this was skipped completely, as the interviewee neither asked questions nor introduced their ideas. The following phase of the interview is based on provided questions, starting with questions regarding class diagram DSL, followed by questions regarding the interactive graphical view. However, like the preceding phase, the interview does not strictly follow those questions, they rather serve as starting points for the discussion of features. Last, before ending the interview, the interviewee is again given the chance to provide their ideas.

4.3.2 Results

As defined by our interview goals, we both collect requirements and design decisions. Here, we focus on the requirements collected during the interviews. Design decisions will mainly be detailed in their appropriate section of Chapter 5. Similar to Section 4.2, collected requirements will be grouped by area identified in Section 4.1.

Our main requirements engineering goal was to identify new graphical features and additional requirements for existing features. Here, we are able to identify eleven new requirements. Similar to already identified requirements, those requirements can be split into interactions that only affect the graphical view and interactions that modify the diagram itself. Regarding the first category, we identify three new requirements:

RE IV.7: Navigate to Source As a modeler, I can navigate from a diagram element to the DSL code defining the element, e.g., by selecting the element or clicking on a context menu entry.

RE IV.8: Scale to Viewport As a modeler, I can zoom and pan the viewport to fit the current diagram by clicking on a button.

RE IV.9: Search and Filter As a modeler, I want to only show specific elements in the graphical view, e.g., with a textual search-based filter, or a category-based filter. Such filtering should not be persisted in the diagram.

Out of those, **RE IV.7** was mentioned particularly often. It provides a solution to the issue that finding the correct section in the DSL code to modify can be challenging, an issue that already appeared for the rather small demonstration diagrams used during the interviews. With respect to the initially identified requirements, multi-select **(RE IV.5)** was mentioned by several interviewees.

As expected, several new features which fall into the second category were mentioned:

RE IV.10: Snapping As a modeler, when graphically moving elements, the element snaps to positions using alignment lines as seen in Microsoft PowerPoint or bpmn.io.

RE IV.11: Align Elements As a modeler, I can align multiple elements, e.g., left, right, or center, by clicking on a button.

RE IV.12: Edit Text As a modeler, I can edit text elements in the graphical view.

RE IV.13: Edit Styles As a modeler, I can edit the style of diagram elements in the graphical view.

RE IV.14: Create Elements As a modeler, I can create new diagram elements graphically, e.g., by selecting elements to create in a toolbox. As an example, for a UML class diagram, elements to create include classes, and associations between them.

RE IV.15: Modify Elements As a modeler, I can modify the content of existing diagram elements graphically. For instance, for a class in a UML class diagram, one can edit, add and delete fields and methods.

RE IV.15.1: Edit Associations As a modeler, I can edit associations graphically, including adding and removing segments (see **RE II.3**), and changing the type of existing segments.

RE IV.16: Copy & Paste Elements As a modeler, I can add new diagram elements by copying & pasting in the graphical view.

Out of these, **RE IV.10**, **RE IV.11**, and **RE IV.15.1** are most similar to the already identified requirements of this category, as these features improve how the user is able to layout the diagram graphically. In addition, the auto-layout button **(RE IV.6)** was mentioned several times. While interviewees agreed that manual and precise layouting are often required, they also mentioned that this process could be simplified by first generating an initial layout based on a layout algorithm, which the modeler then can modify. As an alternative, we discussed whether to introduce a DSL

construct, which automatically layouts some diagram elements. Yet, this approach was rejected by the interviewees, mainly due to two reasons: First, this approach provides no major benefits, while not allowing the modeler to afterward manually adjust the position of elements layouted using this construct. Second, it conflicts with **RE I.4**, as different versions of the layout algorithm might result in different layouting results, thus not guaranteeing long-term layout consistency. The remainder of the features mentioned focuses on creating new diagram elements and modifying existing ones. With all these requirements met, modelers could interact with our hybrid editor similar to purely graphical tools they are used to working with. Interviewees noted that this would be helpful for domain experts with limited knowledge regarding programming languages and DSLs. Also, it was mentioned, that adding new diagram elements and creating associations graphically would help onboard modelers not familiar with our framework. With these features, instead of being required to find examples or look in the documentation, to learn the syntax of our DSL, new users could just use the graphical features, and then learn the DSL based on the code inserted.

When asked for other features which could help new users, one interviewee suggested implementing auto-completion for the textual editor.

RE V.4: Auto-Completion The textual editor provides auto-completion. The auto-completion should show only valid tokens to auto-completion, and provide documentation where possible.

When talking about environments in which our editor will be used, as expected, web **(RE I.1)** and IDE **(RE I.2)** support was mentioned by several experts. Still, we also identified one new environment: a desktop application:

RE I.7: Desktop Application As a modeler, I want to be able to edit diagrams in a desktop application.

RE I.8: Offline Support As a modeler, when using locally installed diagramming environments, e.g., a desktop application **(RE I.7)** or an IDE extension **(RE I.2)**, I can edit the diagram without requiring an internet connection.

The same interviewee also mentioned offline support as an additional requirement. This requirement also affects area **II Diagram & Diagram DSL**, as we must make sure that diagrams can be defined without any external dependencies.

Additionally, in this area, we identify five new requirements:

RE II.8: Theming As a modeler, I can define different themes for a diagram, including a dark and light theme. It should be possible to then render the diagram with a specific theme without modifying the code defining the diagram.

RE II.9: Element-Specific Styling DSL As a modeler, I want to be able to use a DSL construct, which allows styling a specific diagram element, e.g., a single class in a UML class diagram.

RE II.10: Easy Styling As a modeler, I can define common styling properties, e.g., the primary color or line width, without being required to use the low-level styling framework.

RE II.11: Layout and Style Separation of Concerns As a modeler, I want to structure a diagram element in the DSL code in a way that allows separating its main three concerns: content, layout, and style of the element.

RE II.12: Import As a modeler, I can import diagrams from other sources, e.g., Mermaid or diagrams.net.

While **RE II.8** to **RE II.11** can be implemented using some additions to the diagram DSL, **RE II.12** stands out as import is diagram type-specific, and therefore would need to be implemented in a way that does not conflict with **CR 9**.

Last, four new requirements could be identified regarding UML class diagrams:

RE VI.7: UML Packages As a modeler, I can create UML packages.

RE VI.8: UML Comments As a modeler, I can create UML comments.

RE VI.9: Ownership on Associations As a modeler, when modeling associations, I can model ownership by putting a dot on the owning side of the association.

RE VI.10: Warnings if Standard is Violated As a modeler, I want to see warnings in the textual and/or graphical view when the UML standard is violated.

4.4 Evaluation and Prioritization

After collecting requirements and design options in expert interviews, we surveyed the same experts. The survey consists of two parts: The first part focuses on evaluating requirements collected initially and during expert interviews and is described in Section 4.4.1. The second part of the survey focuses on design decisions. Section 4.4.2 describes the overall structure, while results will be detailed in the appropriate section of Chapter 5.

4.4.1 Requirements

Overall, we focus on environment and interactive graphical view requirements in the requirements part of the survey. In total, we evaluated 33 requirements, for each requirement, both a title and description were provided. Except for minor wording changes, the provided title and description are similar to how requirements are described in Sections 4.2 and 4.3. Our main goal was to evaluate and compare the importance of different requirements. Thus, as scale, we choose an interval scale with equal distance between scale points. Endpoints were labeled with “Unimportant” and “Critically important”. As described by Porst, such scales should generally have between five and seven scale points [Por13, p. 88]. Based on the expert interviews, we expected several requirements, especially regarding the interactive graphical view, to receive high ratings. Therefore, to be able to differentiate between the higher-rated items, we decided to use a seven-point scale. Figure 4.3 gives an example of how the scale is presented to the surveyed expert.

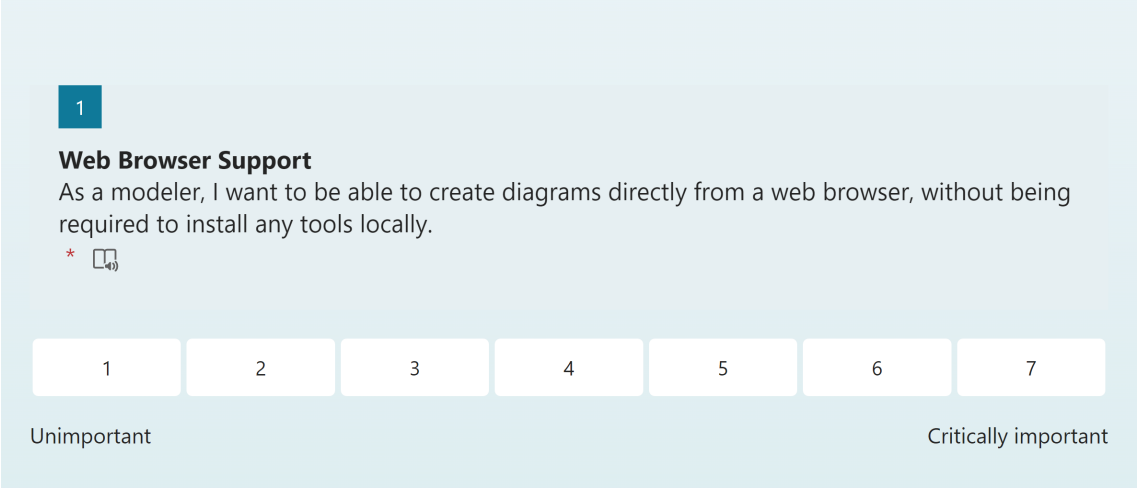

Figure 4.3: Example how requirements are presented to the surveyed experts in the first part of the survey.

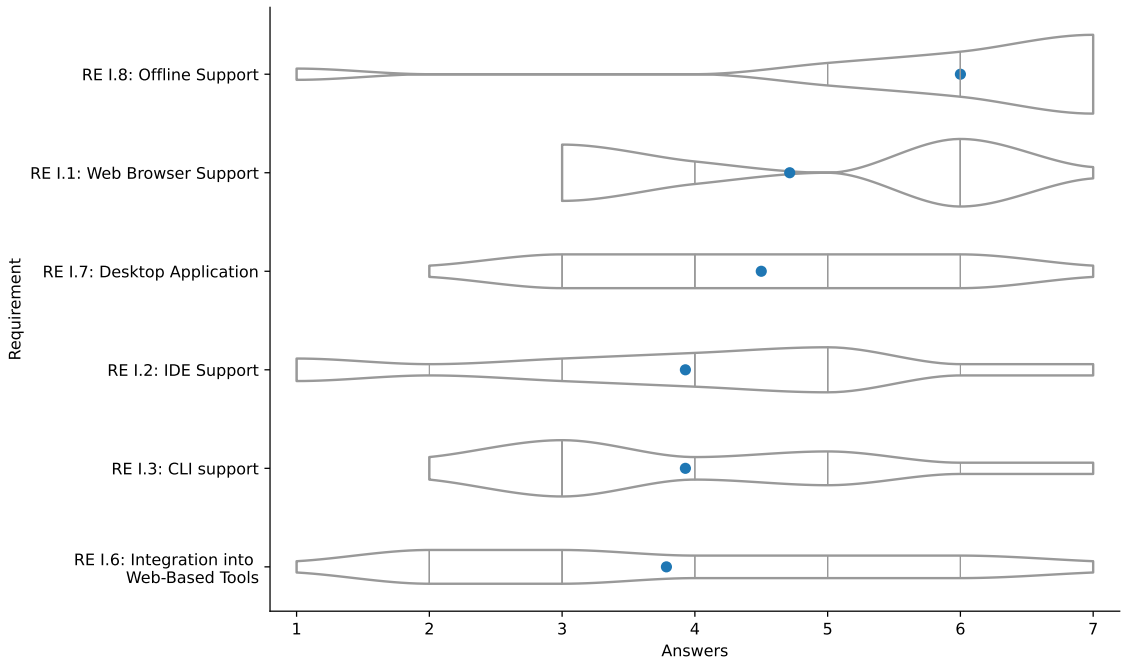

Figure 4.4: Survey results for environmental requirements

Environmental Requirements

For area **I Environment**, we included all six collected requirements in the survey. Our main goal for this area is to find out in which environment we implement our hybrid editor for the scope of this thesis. Figure 4.4 shows the collected results. First, with an average rating of 6.00, offline support **(RE I.8)** was rated the highest. Of the graphical environments in which we could use for our hybrid editor, the web browser **(RE I.1)** was rated the highest, followed by desktop application **(RE I.7)** and IDE **(RE I.2)**. CLI support **(RE I.3)** was rated lower than all graphical environments with an average rating of 3.93. Integration into web web-based tools **(RE I.6)** was rated the lowest of all environmental requirements. Based on these results, we decide to initially prioritize the web-based version of our hybrid editor.

Interactive Graphical View Requirements

Area **IV Interactive Graphical View** was the main focus of our survey. Here, we not only wanted to find out which features to implement first, but also which requirements users expect in general. Figure 4.8 shows the results. Navigability **(RE IV.1)**, meaning zooming and panning the canvas was rated the highest with an average rating of 6.50, followed by moving elements **(RE IV.2)** and edit associations **(RE IV.15.1)**. Overall, features for editing positions and sizes tend to be rated higher than features for editing the content of elements, e.g., edit text **(RE IV.12)** or edit styles **(RE IV.13)**. These results can be explained as editing content can also be performed easily in the textual view, while defining positions textually tends to be tedious. However, both snapping **(RE IV.10)**, aligning elements **(RE IV.11)**, and auto-layout **(RE IV.6)** achieve a – compared to other position-related features – low average ranking of 5.36, 5.36, and 4.64 respectively, resulting in places 11, 12, and 14. Last, search and filter **(RE IV.9)** and scale to viewport **(RE IV.8)** are ranked lowest. For prioritization during this thesis, we largely follow the results of our survey. However, features requiring creating new elements, e.g., editing associations or copying & pasting elements will not be implemented during this thesis and remain as future work due to the expected high complexity of such features.

Other Requirements

For the remaining areas, we primarily wanted to find out how to prioritize features. Therefore, to limit the length of the survey, we omit requirements we expect to implement in any case, either as they provide basic functionality, or due to comparatively low effort to implement, from the survey.

Regarding area **V Textual View**, we only included a single requirement: auto-completion **(RE V.4)**, the result can be found in Figure 4.5. Here, the experts gave an average rating of 4.93.

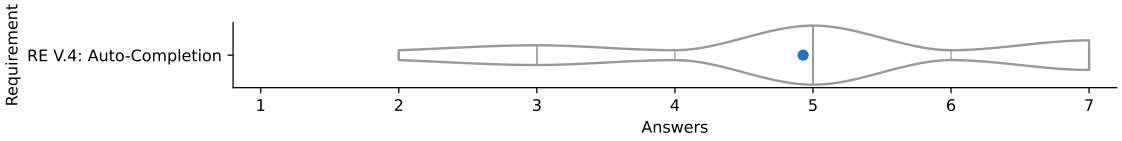

Figure 4.5: Survey results for textual view requirements

Five requirements were included for area **II Diagram**. Out of these, both easy styling **(RE II.10)** and element-specific styling DSL **(RE II.9)** were rated highest with an average rating of 5.64. Closely following, layout and style separation of concerns **(RE II.11)** received an average rating of 5.14. Import **(RE II.12)** and theming **(RE II.8)** were rated the lowest. Based on these results, we decided to primarily focus on the first three requirements.

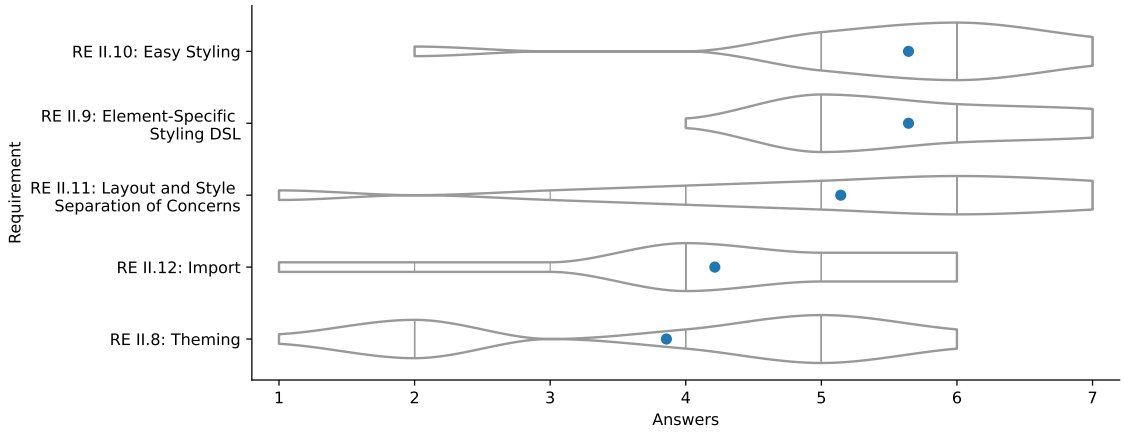

Figure 4.6: Survey results for diagram requirements

Last, four requirements were included for area **VI UML Class Diagram DSL**, asking experts to prioritize specific elements supported by the class diagram module. UML comments **(RE VI.8)** were rated first with an average of 4.71. Warnings if the standard is violated **(RE VI.10)**, ownership on associations **(RE VI.9)**, and UML packages **(RE VI.7)** all received ratings close to 4 (±0.14). Therefore, we decided to prioritize UML comments, while implementing the remaining features based on the required effort.

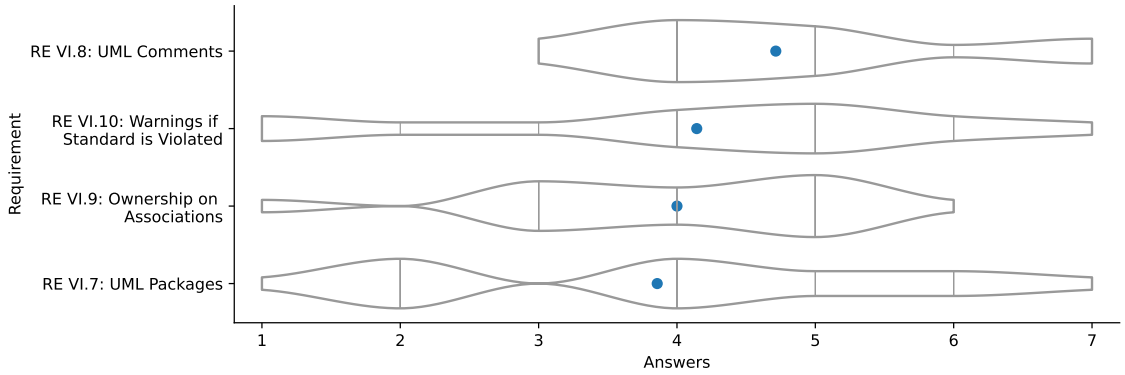

Figure 4.7: Survey results for UML class diagram requirements

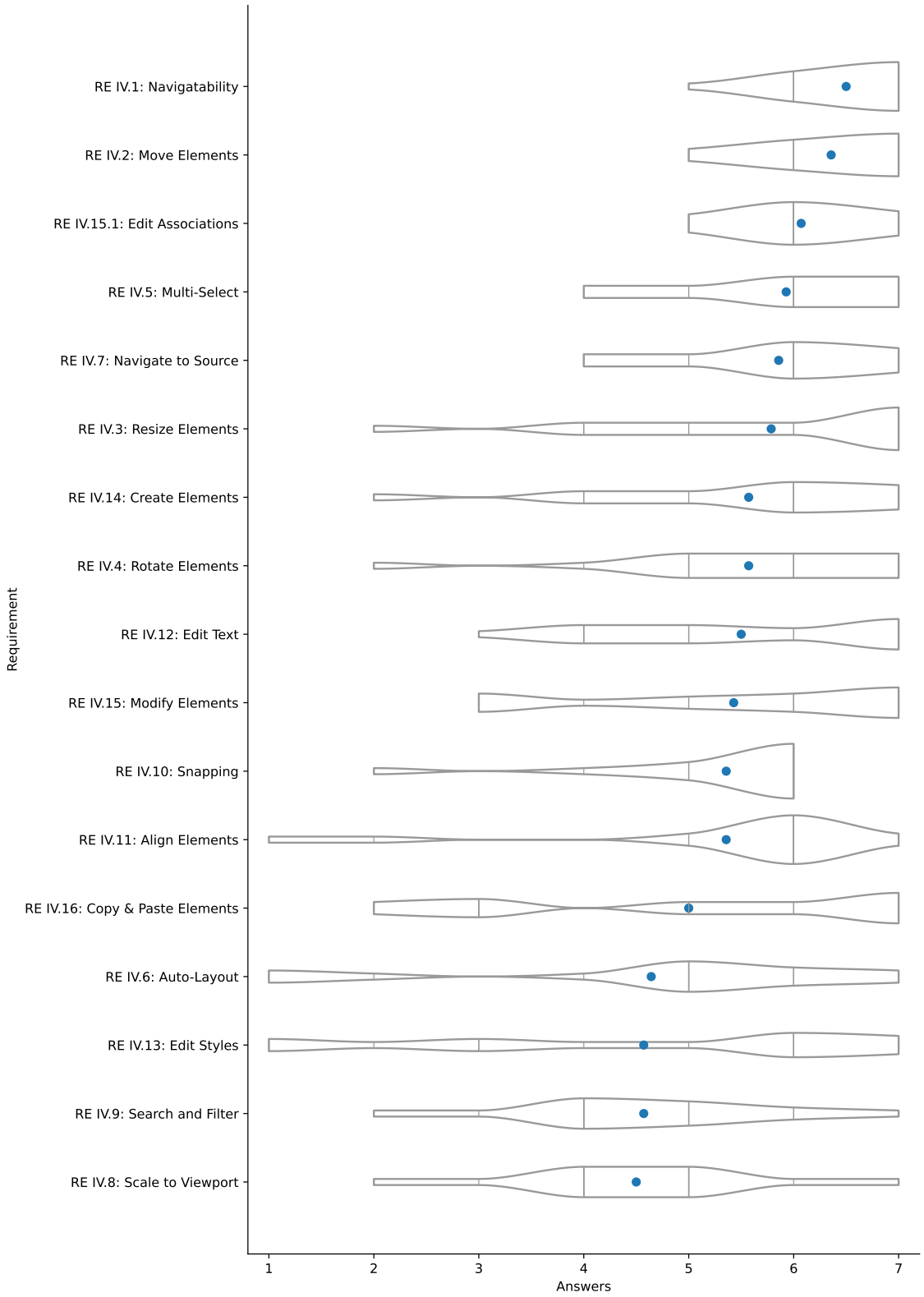

Figure 4.8: Survey results for interactive graphical view requirements

4.4.2 Design Decisions

Before and during the expert interviews, we collected multiple design options regarding different aspects of the framework. The second part of the survey focuses on finding conclusions for these design decisions. The provided scale depends on the number of available options. First, if two options exist, and those options are strictly exclusive, we chose a drop-down where users can select their preferred option. For all other questions, we chose a scale similar to the rating scale chosen to evaluate requirements. Again, we use a 7-point interval scale with a point distance of one. However, instead of using an interval from 1 to 7, instead, we use an interval from -3 to +3. Due to technical reasons, endpoint names were not attached at each scale, however, the introduction text to the section defines -3 as “very negative” and +3 as “very positive”. Our main reason to use a different interval, compared to when evaluating requirements, is to allow surveyed experts to explicitly annotate options as negative.

During the interviews, when discussing design decisions, some interviewees mentioned that we could simply implement multiple approaches if said approaches are non-conflicting. However, other experts stated that we should instead choose only one, even if it’s not their preferred approach. Therefore, in the survey, we asked the meta-design-decision “If multiple implementations for a feature exist, should I support multiple or stick to the most popular one?”. Implementing multiple options would have the benefit of more users being able to use their preferred version, at the cost of feature bloat, pollution of the global scope in the DSL, and increased cognitive load when looking at code written by other users. Out of the 14 interviewed experts, 12 chose the option of only supporting one implementation. As mentioned before, all other design decisions will be detailed in Chapter 5.

5 Concept

This section elaborates our concept we developed based on our vision and collected requirements. As mentioned before, our overall concept is to develop a hybrid editor and diagramming framework. Figure 5.1 provides an overview of the main components of our concept, and how these components relate to each other. First, Section 5.1 introduces diagrams in the context of our framework. Then, Section 5.2 details the textual DSL used to define diagrams. Our general framework is diagram type-independent, diagram types are supported using a modular approach. Section 5.3 introduces the UML class diagram module, with a focus on how it integrates into the general diagram DSL. Last, the hybrid editor is the focus of Section 5.4. Here, we focus on how interactive graphical editing features can be used to manipulate the definition of a diagram.

5.1 Diagram

Due to being a diagramming framework, diagrams are central entities in our concept. They are defined by users using our diagram DSL, displayed and manipulated in the hybrid graphical editor and rendered to different output formats. Figure 5.2 gives an overview of our data pipeline from the DSL code the user inputs, to the finished rendered diagrams. In this section, we focus on the three types of diagrams: First, in Section 5.1.1, we explain the diagrams the user defines, with a focus on styling and supported diagram elements. Following, we detail the layouting process and the resulting layouted diagrams in Section 5.1.2. We also briefly go over how layouted diagrams are simplified to better support different rendering targets.

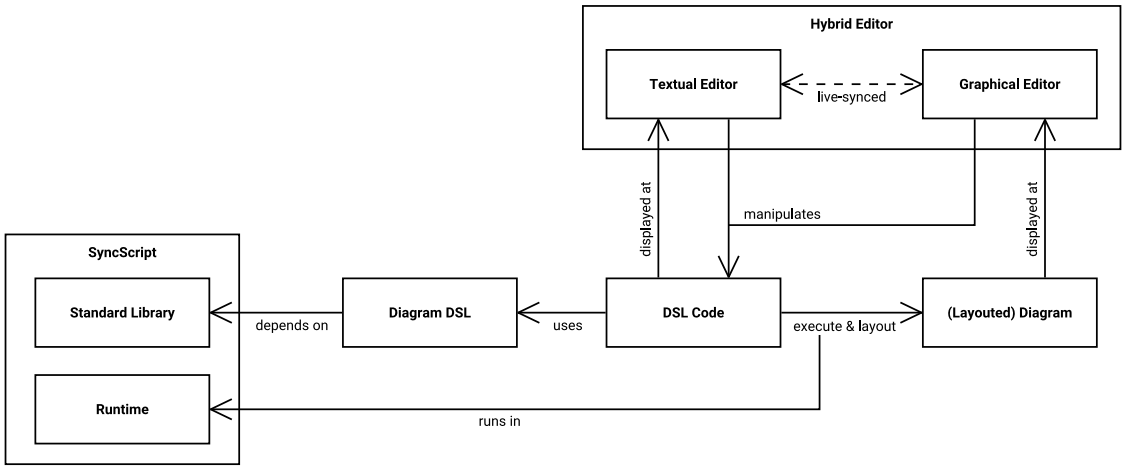

Figure 5.1: Overview of the main components of our concept.

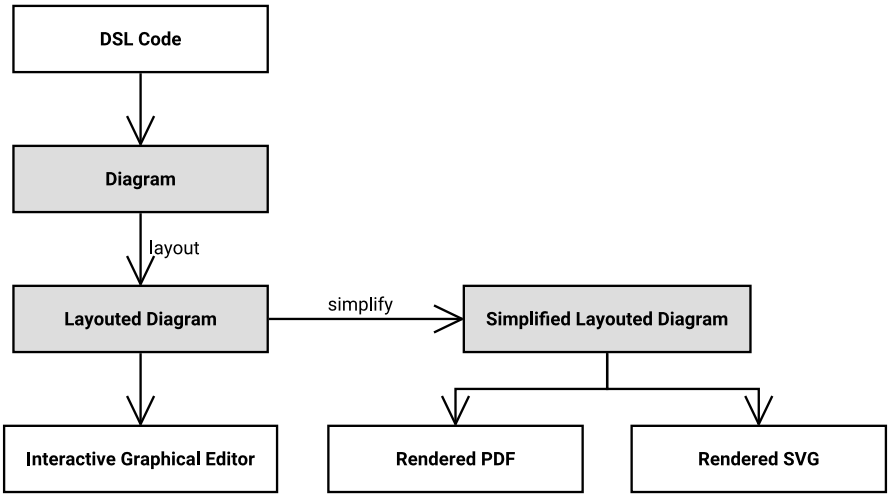

Figure 5.2: Data flow from DSL code to rendered diagrams. Highlighted parts are detailed in this section.

	type	
category	layouted elements	specialized elements
primitives	rect, ellipse, path, text	span
panels	vbox, hbox, stack	
canvas	canvas	canvas element, canvas connection, marker, absolute/relative/line point

Table 5.1: Overview of all supported elements by category and type.

5.1.1 Diagrams

Diagrams are the central entities a user defines when using our framework. As seen in Figure 5.2, they are defined using the diagram DSL. To support different diagram types, diagram entities should be diagram type independent, and rather provide reusable lower-level constructs. The definition of higher-level constructs, e.g., classes in a UML class diagram is done using the diagram DSL, as described in Section 5.2.3. Additionally, as required by **RE II.6**, users should be able to directly define low-level graphical elements and manipulate elements defined by the framework. As a result, we design diagrams as hierarchical graphical models, comparable to HTML documents. Conceptually, a diagram is a triple of a tree of diagram elements, a set of styles, and a set of font families which can be used by the styles. Following, we first describe the supported diagram elements, and then detail the styling framework.

Supported elements can be partitioned in two ways: First, elements fall into three main categories: primitives, panels, and canvas. Categories – and their elements – are further detailed in the following paragraphs. Second, two types of elements exist: layouted elements and specialized elements. Table 5.1 gives an overview of all elements both by type and group. Specialized elements can only be used as children of a specific parent element. They do not support our general layout algorithm,

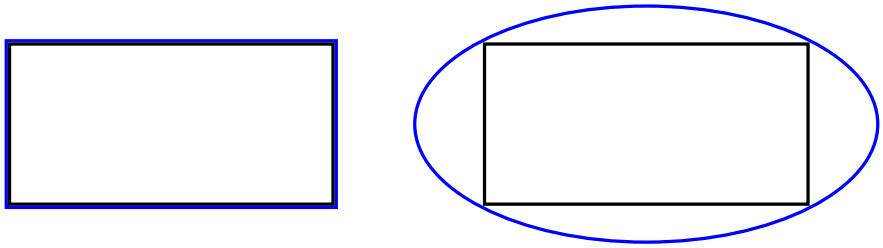

Figure 5.3: Rectangle and ellipse (blue, outer) containing a rectangle of a fixed size (black, inner)

and rely on the layouting by their parent. Especially, they cannot be used as root element. As an example, span elements can only be used inside a text element, and a text element can only contain spans. Layouted elements are the more general element type, supporting layout style attributes. They can be used everywhere where layouted elements are expected, especially as the root element, and children of panels. Following, we further detail the supported elements by category.

Primitives

Primitives are the most basic provided elements. In the context of diagramming, (almost) all visible elements are formed by compositing primitives. Currently, we support four different primitive elements: rectangle (rect), ellipse, path, and text elements. Rectangles, ellipses, and paths form the group of shapes. With full support for SVG paths, paths are the most flexible element of this group. In a diagramming context, they can be used to create arbitrary shapes or to create different arrowheads. Due to providing an arbitrary shape, paths do not support containing other elements. Rectangles and ellipses on the other hand support containing a content element. As both can be created using paths, they are typically used as container elements. As Figure 5.3 shows, their size is calculated so that they completely contain their content, with the content not overlapping its stroke. Last, text elements allow to include text in diagrams. In order to support inline styles, e.g., to only mark some words as bold, a text element consists of a list of spans, where each span can define its own styling. Spans themselves do not provide any layout information, as their position and breaks completely depend on the flow of their containing text element.

Panels

Panels are used to layout other elements. They solely layout their child elements, and do not add any additional graphical features. Currently, as shown in Figure 5.4, three types of panels are supported: Vbox and hbox allow stacking elements vertically and horizontally, respectively. The stack element allows the user to stack elements on the z-axis, thus allowing to overlay multiple elements.

Canvas

Initially, we focus on graph-based diagrams, like UML class diagrams. While panels and primitives allow creating elements used in such a diagram graph, we still need elements that allow arranging those in a graph-like manner. In our framework, the canvas provides this functionality. A canvas

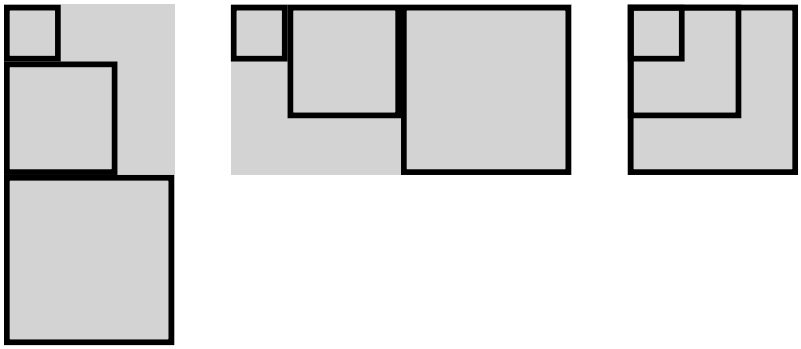

Figure 5.4: Different panels containing three rectangles as contents. The gray area highlights the area the panel covers. From left to right: vbox, hbox, stack

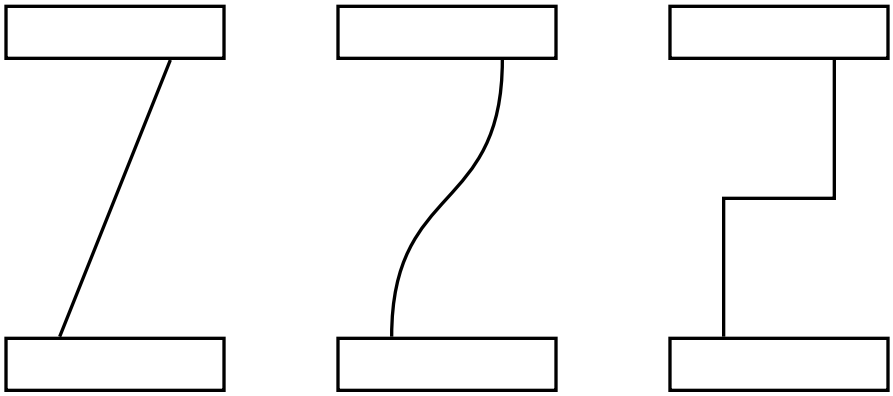

Figure 5.5: Three canvas connections showcasing the three main segment types: line, bezier, and axis-aligned

can contain two types of elements: Canvas elements and canvas connections. Canvas elements act as wrappers which allow them to position their content freely on the canvas. Besides, a canvas element can define size and rotation. Like panels, both the canvas element and the canvas itself do not provide any graphical features. In a diagramming context, they are typically used to represent diagram elements, e.g., as defined by **RE II.2**. Canvas connections are used to create graphically visible connections between points on a canvas, typically between canvas elements. Thus, they can be used to create diagram associations as defined by **RE II.3**. Graphically, they are similar to the primitive path element. However, paths can only be used as content of canvas elements. Also, they do not provide the means to be graphically manipulatable. As a result, they are insufficient to represent connections on the canvas. Conceptually, a connection is a path with a defined start and end point and any amount of intermediate points. Consecutive points are connected by connection segments. Different segment types allow to customize how two points are connected. Based on conducted expert interviews and existing tools, we identify three segment types we want to support initially: line segments, which draw a direct line, bezier segments, which draw a cubic bezier curve, and axis-aligned segments, which only use horizontal and vertical lines, as shown in Figure 5.5.

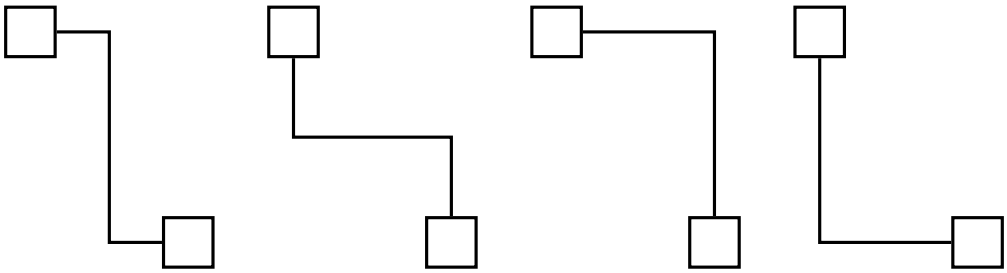

Figure 5.6: Four canvas connections consisting of a single axis-aligned segment.

For bezier segments, we considered using both cubic or quadratic bezier curves. While cubic bezier curves require two control points per segment, quadratic bezier curves only require one. When joining bezier curves, typically, geometric (G^1) continuity is desired, resulting in a smooth curve. To achieve it at a joint of two bezier segments, one must ensure that the first control point of the second segment lies on a ray formed by the joint point and the last control point of the first segment [BD90]. For our use case, this would result in a confusing user experience when using quadratic bezier curves. When moving a control point, in order to maintain continuity, one either needs to modify the start and end points or modify the control points of the adjacent segments. However, when using the second approach, as each segment only has a single control point, the control points of all segments need to be modified, which is not practical, as changes would not be contained locally. Therefore, the first approach is used, e.g., by *diagrams.net*. Thus, the user cannot move control points without moving the start point, end point, or both, which can be confusing to users. Cubic bezier curves do not have this downside. When moving a single control point, one can maintain continuity by moving the closest control point of the adjacent segment. As this adjacent segment itself has two control points, the change does not need to be propagated and therefore is locally contained. Similarly, when moving a start/end point of the segment, maintaining the relative position of the adjacent two control points is sufficient to maintain continuity. Thus, we decide to only implement cubic bezier curves.

As described before, axis-aligned segments only use vertical and horizontal lines. Thus, in contrast to cubic bezier curves, it is possible to create graphically identical connections by using multiple line segments. However, we decided to implement it as a standalone segment type to allow for simpler definitions and improved graphical editing features. To achieve this, we define a single segment as a step-like path. As Figure 5.6 shows, four options exist: First, a horizontal part can be followed by a vertical part, followed by an additional horizontal part. Alternatively, horizontal and vertical parts can be swapped, resulting in two vertical and one horizontal part. Options three and four are similar, with either the start or end part having a length of 0. Initially, we decided to only support the first case. A segment was defined by a start and end point, and a number between 0 and 1, defining where the vertical part is located. As a result, options three and four are automatically supported by setting this position to 0 or 1 respectively. However, option two would require at least two axis-aligned segments, or one axis-aligned segment and one line segment. Yet, this proved to be insufficient in practice, where option three is often required. As a solution, we extended the value of the relative position parameter to a range of -1 to 1. Keeping the semantics for positive numbers, negative numbers can now be used to create a segment using option three, where now the parameter defines the relative position of the horizontal part.

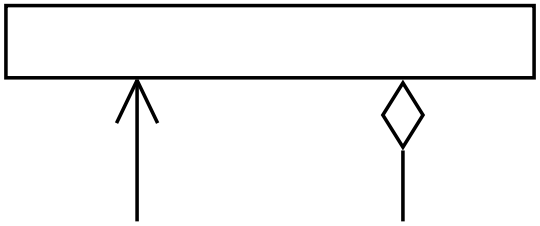

Figure 5.7: Two types of markers at the start of a canvas connection. While the first marker does not affect the start of the connection, the second shifts it to the end of the marker.

Similar to SVG paths, connections support markers at both the start and end of a connection. Typically, they are used to draw arrowheads or similar graphical elements, yet, in order to improve flexibility, technically they can wrap any element. Similar to canvas elements, they define the position, rotation, and optionally size of their content. However, position and rotation are automatically determined by the tangent at the respective end. Some markers require the line to not start at its original defined point, but instead at the end of the marker, as shown in Figure 5.7. To allow this, a marker defines a position between 0 and 1, with 0 representing the start of the marker, and 1 its end. Similar to diagrams.net, the start of the actual line is then shifted to this position.

Both canvas elements and canvas connections require points in the canvas for layouting. During requirements engineering, we identify three types of points to support: absolute points **(RE II.2.2)**, relative points **(RE II.2.3)**, and line points **(RE II.2.4)**. The distinction of different point types is required to both ease defining points and manipulating positions graphically in the interactive editor. Absolute points are the simplest form of point, they directly define a x and y-coordinate. Relative points require a target, and x and y-direction offsets. Both other points and canvas elements can be used as target, if a canvas element is used, its position is used as the target point. Line points are the most complex supported point types. As the name suggests, they are defined as a point on a line. Both canvas connections and canvas elements can be used as line providers. If a canvas element is used, the line is defined as its outline. The outline itself depends on the content of the canvas element. While most elements simply use the bounding box as their outline, rectangles, and ellipses provide a more sophisticated outline calculation. As those two are commonly used as diagram container elements, associations between diagram elements can start directly at the visual outline, and not at the bounding box. Line points need, in addition to the target line, also define the relative position on the line. During expert interviews, we identify three possible ways to do this:

position on segment The position is defined by the segment index and a relative position on the segment. For example, to obtain the center of the second segment, one would use the tuple $(1, 0.5)$.

global position Similar to the first option, but combined into a single number. The whole range $[0, 1]$ is defined into subranges for each segment. For example, if a line consists of two segments, numbers in the range $[0, 0.5]$ represent positions on the first segment, while numbers in the range $]0.5, 1]$ represent positions on the second segment.

n **points on segment** With this implementation, one first divides a segment into n points. Then, one defines the index of the point to obtain. As a result, the position is defined as a triple (idx, n, i) of segment index idx, segment point count n, and point index i.

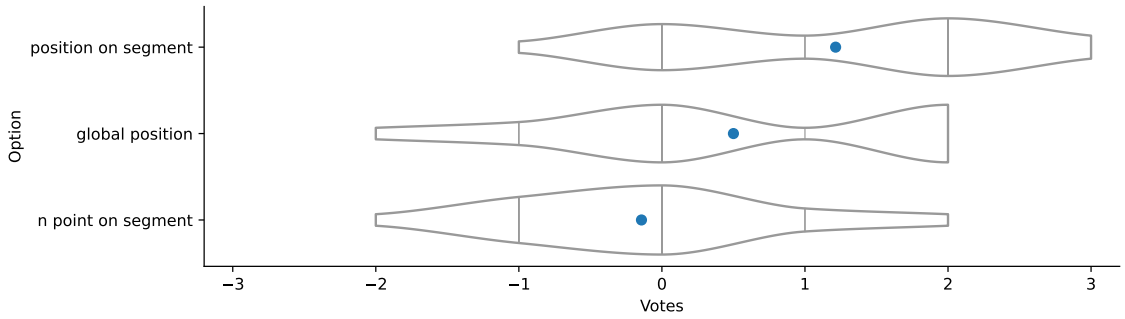

Figure 5.8: Survey results regarding which implementation to choose for line points.

All three approaches provide their unique benefits and drawbacks. Both segment-based approaches have the disadvantage, that if the segment count changes, the behavior is unstable, and updating the segment index is typically required. However, keeping the same relative position on a segment is easier to achieve, as only the segment index needs to be implemented. When mainly using the graphical editor, the second option provides the benefit that only one number needs to be defined in the code. Yet, when manually defining a position in code, calculating the correct number can be difficult. Approach three provides arguably the best user experience when manually defining points, as commonly used points, e.g., center, start and end, can be intuitively defined. However, it has the disadvantage that when using the graphical editor, only discrete steps are supported, instead of a continuous manipulation provided by the first two options. To decide which type(s) of line point to implement, we added included a question in the survey asking the surveyed experts to rate all three options. Figure 5.8 shows the results obtained. With an average rating of 1.21, the experts rate the first option, position on a segment, the highest, followed by global position with an average rating of 0.50. Last, n points per segment receives a negative average rating of -0.14. As a result, we decide to choose the first implementation option for the default line point. However, we additionally support the second option by making the segment index optional. If not provided, we fall back to the second approach and calculate the segment index and relative position on the segment.

Styling

Styling allows the user to further specify the look of the diagram elements, as required by **RE II.4**. Generally, each type of element defines a set of style attributes that are supported. Table 5.2 gives an overview of which attributes are supported for which elements. Style attributes can further be subdivided into several groups: Layout attributes affect the positioning and size of an element, they are further detailed in Section 5.1.2. Stroke and fill attributes affect the stroke and fill of a shape respectively, to provide some familiarity to users, the name and semantics of available attributes are highly influenced by SVG presentation attributes. As element-specific styling is required **(RE II.9)**, setting style attributes globally is insufficient. Therefore, we allow attributes to be specified both locally on an element and globally using styles. As local attributes always take precedence over global attributes they can not be used to provide defaults and are therefore rarely used by our framework. However, they are available to users as an easy way to define styles for attributes when using the low-level graphics framework directly. A style is a tuple of a set of style attributes it defines, and a list of selectors which specify which elements the style affects. In contrast

style attributes		supported elements
group	attributes	
layout	width, minWidth, maxWidth, height, minHeight, maxHeight, margin, marginTop, marginBottom, marginLeft, marginRight, vAlign, hAlign	rect, ellipse, hbox, vbox, stack, canvas element
stroke	stroke, strokeOpacity, strokeLinecap, strokeLinejoin, strokeMiterlimit, strokeDash, strokeDashSpace	rect, ellipse, path, canvas connection
fill	fill, fillOpacity	rect, ellipse, span
	rotation	canvas element
	lineStart	marker

Table 5.2: Overview which style attributes can be applied to which elements.

Algorithm 5.1 Evaluating whether a style matches an element or not, given the style's selectors and an element.

```
function MatchesElement(selectors, element)
    i = length(selectors) − 1
    if ¬MatchesSelector(selectors[i], element) then
        return false
    end if
    loop
        if MatchesSelector(selectors[i], element) then
            if i = 0 then
                return true
            else
                i = i − 1
            end if
        end if
        if HasParent(element) then
            element = GetParent(element)
        else
            return false
        end if
    end loop
end function
```

to Cascading Style Sheets (CSS), where selector specificity is used to calculate style priority, we use the order in which styles are defined on the diagram. Later styles will overwrite previous styles. Currently, we provide three types of selectors:

Type Takes a string parameter and matches only elements of the specific type.

Class Takes a string parameter and matches only elements where the class attribute contains the provided string.

Any Matches all elements.

Each style provides a list of selectors, Algorithm 5.1 shows how we evaluate whether a style matches an element or not. Generally, an element matches a style if elements on the path from the diagram root to the chosen element match all selectors in order. Note that except for the element, which must match the last selector, not all elements on the path need to match selectors, it is sufficient if all selectors are matched. Thus, conceptually, our implementation is identical to the CSS descendant combinator.

While styling using selectors provides flexibility, it also requires users to have an understanding of the structure of diagram elements, especially used classes. However, as defined by **RE II.8**, users also want to use different themes, e.g., a dark and light theme, for diagrams. To support themes, we support style variables, similar to CSS variables. Style variables consist of two aspects: First, when assigning values to attributes using a style, one can also assign a variable, consisting of the name of the variable, to the attribute. Second, a style allows defining a set of variables, each being a tuple of name and value. Semantically, when evaluating styles for an element, first, all variables defined on the element are evaluated. Then, if for any attribute a variable is used, the value of the variable is looked up on the element. Style variables allow the definition of multiple style attributes using a single variable. Consequently, color variables like primary and secondary colors allow for defining a color theme.

5.1.2 Layouted Diagrams & Layouting

With their hierarchical structure, relative positions, and styling framework, diagrams are designed to be flexible and easily creatable by users. However, they are not optimized to be easily renderable to our target platforms, including SVG and our interactive graphical editor. Therefore, as shown in Figure 5.2, we first transform diagrams into layouted diagrams in the layout process.

Layouted Diagrams

Layouted diagrams differ from diagrams in three major ways: First, most elements of a layouted diagram define its position and size. These attributes are calculated in the layout process. Additionally, in order to produce consistent outputs, we limit the use of relative positions and use absolute positions where applicable. While relative positions are easier to define for users, they can lead to accumulative errors when transforming to the target platform. Second, layouted diagrams do not provide styles separately, only local attributes on elements are supported. Therefore, during layouting, styles need to be evaluated. Attributes that are only required for layouting are omitted completely. Last, elements required only for layouting are removed, as due to absolute positions, they are no longer required. This includes all panels, but also text elements, which only position their child spans. However, note that information required for the interactive graphical editor cannot be removed, as the editor also operates on layouted diagrams. Consequently, canvas points cannot be evaluated and removed, as this would result in the graphical editor not being able to display and allow modification of intermediate points. Yet, these limitations do not apply when rendering diagrams to non-interactive output formats, e.g., SVG and Portable Document Format (PDF). Thus, for these targets, as shown in Figure 5.2, we are able to further simplify layouted diagrams into simplified layouted diagrams. Here, we evaluate canvas points, replace canvas connections with paths, and convert markers to regular canvas elements. This allows us both to obtain a consistent rendering result and give other developers the ability to easily add new rendering targets.

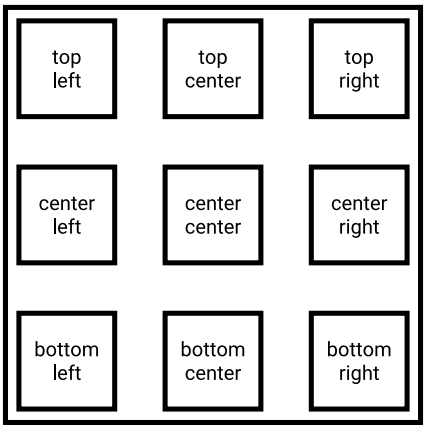

Figure 5.9: Rectangle containing nine rectangles demonstrating all possible alignment options. To improve readability, the child rectangles also define a small margin.

Layout Style Attributes

Before discussing the layout algorithm, we first need to define the style attributes which affect the layout. As shown in Table 5.2, we define the following layout attributes: width, minWidth, maxWidth, height, minHeight, maxHeight, margin, marginTop, marginBottom, marginLeft, marginRight, vAlign, hAlign. First, width and height constrain the size of the element. The user can define a minimum, maximum and exact value for both width and height. These attributes are prioritized in the following order: exact, maximum, and minimum. Margins define the extra margin space around the element, it can be specified either for each side of the element or for all sides at once. If both are defined, side-specific values are prioritized, and the default margin has a thickness of 0. Vertical and horizontal alignment define how the element with added margin is positioned relative to its parent. By default, elements are aligned at the top-left corner. Figure 5.9 shows all possible combinations of alignments.

Layout Algorithm

As shown in Figure 5.2, the goal of the layout algorithm is to transform a diagram into a layouted diagram. Inspired by *Flutter*[1] and *Windows Presentation Foundation (WPF)*[2], we decide to implement a two-step layout algorithm which first measures the size of each element of the diagram tree, and then positions them in a second step. Figure 5.10 shows how these two steps work based on an example diagram consisting of formed by three nested rectangles. The initial phase is referred to as "measure", with the objective of computing the `measuredSize` of each element, given the element's size constraints, meaning minimum and maximum dimensions. Generally speaking, measure itself consists of two steps for each layouted element: The goal of the first step is to evaluate layout style attributes and modify the provided constraints based on it. First, it decreases both maximum and minimum dimensions by their respective margins. For both dimensions, if an

[1] https://docs.flutter.dev/development/ui/layout/constraints
[2] https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/layout?view=netframeworkdesktop-4.8

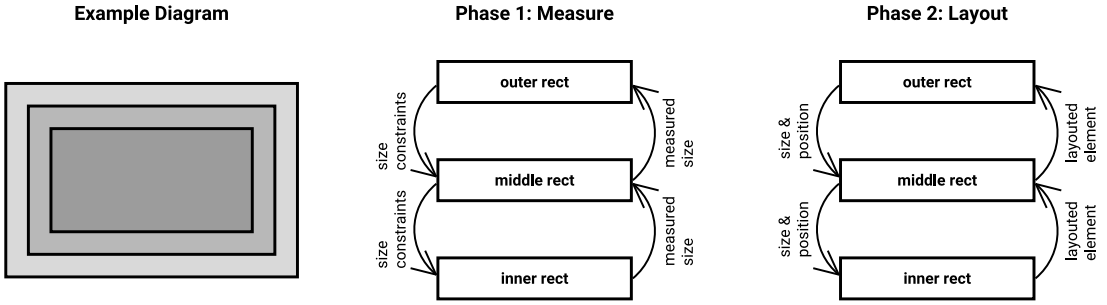

Figure 5.10: Layout Process for an example diagram consisting of three nested rectangles.

alignment is set, the minimum for the dimension is reduced to 0. Next, minimum and maximum width and height of the element are used to further constrain the constraint dimensions. Last, if an exact width or height is given, both minimum and maximum of the constraints are set to the exact value. As a result, local attributes take precedence over constraints provided by the parent. Especially, child elements can overflow their parent. As this step only relies on the provided constraints and style attributes, it can be implemented for all layouted elements. As the second step, the modified constraints are provided to the element-specific measure function, which calculates the `requestedSize`. Its implementation depends on the type of element. Generally, most container elements invoke measure on their child elements and calculate their size based on their children's preferred size. The returned `requestedSize` is saved to be later used during layouting. To compute the `measuredSize`, the margin is added to the `requestedSize`, and the resulting value is constrained to the originally provided constraints. Semantically, for the parent element, the element uses a size valid respective to the provided constraints. Overflow and underflow are handled locally at the child element. Last, the `measuredSize` is returned to the parent.

The second phase, layout, converts the diagram into the layouted diagram. When applied to an element, it takes a position and size as input, together, these form the rectangle in which the element should be drawn. Similar to measure, layout consists of two steps: The first step adjusts the position based on element layouting style attributes, margin, vertical and horizontal align, and the `requestedSize` from measure. First, the margin is added to the `requestedSize`. Then, a new position is calculated so that the modified size from the previous step and the new position form a rectangle, which aligns with the original layout rectangle provided to layout at the position defined by the vertical and horizontal alignment of the element. Then, the element type-specific layout method is invoked with the adjusted position, and the `requestedSize`, which returns the layouted diagram element. Note that this step uses the position of the element without the margin. Again, for wrapper types, this method invokes layout for its child elements, thus, the whole diagram is recursively transformed, resulting in the layouted diagram.

5.2 Diagram DSL

In order to use the diagrams defined in Section 5.1.1, the user needs a way to define those diagrams. As defined by **CR 2** and **CR 4**, our concept uses the diagram DSL to fully define diagrams. Figure 5.11 gives an overview of the three layers, that make up its structure: On the base layer,

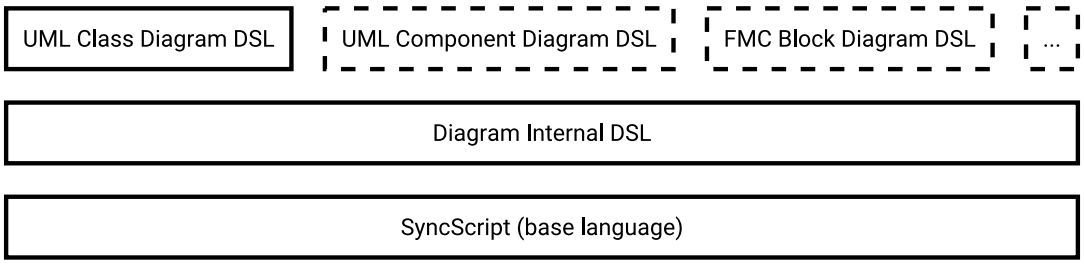

Figure 5.11: Overview of the Structure of the diagram DSL. Modules in dashed lines are not implemented as part of this thesis.

we use a custom GPL, SyncScript. It provides general-purpose programming language features as required by **CR 3**. In Section 5.2.1, we explain why we choose this approach, followed by Section 5.2.2, where we explain the language itself. Building upon that is the diagram internal DSL, which is, as its name suggests, implemented as internal DSL in SyncScript. It provides general diagram features and allows creating diagrams as defined in Section 5.1. Section 5.2.3 further explains its design and features.

5.2.1 Choosing the Approach

As defined by **CR 9**, the framework should be extensible to support additional diagram types in the future. As a result, the diagram DSL also needs to be extensible. We envision this concept as a plugin-based approach: to support an additional diagram type, one can add a plugin for that specific diagram type. Using language composition, the plugin then extends the DSL to support the new diagram type. Based on Section 2.3, we identify three different approaches:

(1) Projectional external DSL

(2) Parser-based external DSL

(3) Internal DSL

In order to select the best approach for our framework, we need to evaluate them against the requirements collected in Chapter 4. Mainly, three areas affect the DSL: First, the chosen approach must be compatible with all supported environments. As diagramming inside a web browser should be supported **(RE I.1)**, the DSL must be executable in a web environment. The same applies to all tooling required to use the language. Second, GPL features need to be incorporated. Last, it must be possible to edit the DSL code based on graphical edits, as defined by **CR 5**. Depending on the approach, this might require additional tracing during execution, in order to choose the correct modification.

In our context, projectional DSLs provide the benefit of allowing easy programmatic AST manipulations. Mainly, AST nodes can be replaced by just creating a new AST node. As the textual editor is purely projectional, we do not need to generate a textual representation of the new node. Additionally, depending on the chosen framework, a base expression language might be provided. Projectional DSLs require by definition a projectional editor and are not compatible with classical text editors. Thus, they are usually developed and used with a language workbench. Based on the

work of Erdweg et al. [ESV+13] and https://github.com/yairchu/awesome-structure-editors, we select six language workbenches for further evaluation[3]: *Jetbrains MPS*[4], *Más*[5], *Whole Platform*[6], *JOY*.js[7], *Fructure*[8], and *Freon*[9]. Out of these, Más, JOY.js, and Fructure were last updated more than three years ago and therefore are not considered active projects. As the Whole Platform is based on *Eclipse*, it cannot be used on the web and is therefore incompatible with **RE I.1**. For the same reason, we can rule out MPS. While there are projects to bring MPS to the browser, most notably *Modelix*[10], these still require a headless server-version of MPS[11], which we try to avoid for our concept. Thus, Freon is the only remaining workbench. It is implemented in TS and used out of a web browser. While it still requires a model server in its current version, we deem a purely browser-based version feasible in the scope of this project, as the server mainly stores models, and is also written in TS, thus functionality could be easily migrated to the browser. However, when testing the projectional editor, we concluded that it is, at least in its current state, not sufficient for our purpose. Mainly, compared to the textual editor of MPS, it lacks convenience features important to give users a familiar user experience to classical text editors. Mainly, when entering code, one needs to follow the tree structure of the AST rather than the linear structure of the code. Additionally, language composition features are missing. Similar to implementing a projectional editor from scratch, we deem implementing these features in the scope of this thesis not possible.

For the remainder of this section, DSL always refers to parser-based DSLs. When comparing internal and external DSLs in the context of our work, both approaches have their benefits and drawbacks. First, external DSLs allow for more syntactic flexibility concerning different diagram type plugins. When using an internal DSL, the syntax for the diagram type-specific part of the DSL remains limited by the syntax of the GPL it is embedded in. Also, tooling support can be tailored more easily towards specific parts of the DSL, as these parts can be determined during parsing. However, tooling support is also required to adapt to the changes in grammar resulting from adding different plugins. As each plugin extends the grammar, all tooling based on the grammar needs to dynamically adapt to these changes. This not only affects the parser and interpreter/compiler but also syntax highlighting, formatting, auto-completion, and similar tools. However, in some contexts, some of these tools are required to be static. For example, a *VS Code* extension cannot dynamically contribute additional syntax highlighting grammars. Yet, with this architecture, each combination of plugins results in a new grammar. Additionally, programmatic source code modification can get more difficult with increasing grammar complexity. Internal DSLs provide the additional benefit that users can easily natively extend the DSL, fulfilling requirement **RE III.5**. Ultimately, for these reasons, we decided to implement the diagram DSL as internal DSL.

As our next step in the process, we had to decide which GPL to embed the diagram DSL in. Two main options exist: using an existing GPL, or implementing a custom one. At first glance, using an existing GPL seems to be the obvious choice, as it already provides not only an execution

[3] [ESV+13] also mentions a language workbench called *Onion* which supports projectional DSLs. However, we were not able to find sufficient additional information to evaluate. Therefore, it was excluded from the evaluation.

[4] https://www.jetbrains.com/mps/

[5] http://mas-wb.appspot.com/

[6] https://whole.sourceforge.io/

[7] https://ncase.me/joy/

[8] https://fructure-editor.tumblr.com/

[9] https://www.freon4dsl.dev/

[10] https://modelix.org/

[11] https://modelix.org/docs/architecture/

environment but also additional tooling, cutting down development effort significantly. Still, not all programming languages are suitable for our approach. First, they need to be usable in a purely web environment. While for most use cases, it is sufficient to transpile to JS or compile to *WebAssembly (WASM)*, in our case, the compiler and all required tooling must be runnable in the web environment. Also, due to implementing live-sync **(CR 5)**, the compiler and startup of the execution environment need to be sufficiently fast. Next, the grammar of the language should be simple enough to allow for programmatic reliable modifications of the AST, yet, flexible enough to implement an internal DSL. Kotlin, Scala, and Groovy are languages that initially came to our minds. For implementing DSLs, they provide multiple benefits in syntactic flexibility, including:

Operator Overloading Operators, e.g., +, can be overloaded for custom data types. For example, addition and multiplication operators can be overloaded for vector classes in a math library.

Infix Notation Infix notation allows using other identifiers like operators. In Scala and Kotlin, functions can be declared as infix functions. In this case, `vec1 plus vec2` becomes identical to `vec1.plus(vec2)`. Groovy provides a similar feature called command chains. Here, functions can take arguments without requiring parenthesis. For example, `goto a then b` is identical to `goto(a).then(b)`.

Trailing Lambdas In Kotlin, a lambda can be defined simply by creating and enclosing one or more expressions in curly brackets. If a function takes a lambda as its last argument, one can provide it using a special syntax where the lambda is placed after the function call. In case the function takes no further arguments, one can omit the round brackets usually required for a function call. For example, `forEach { println(it) }` is identical to `forEach({ println(it) })`. Scala provides a similar feature, however, it can only be used if the function only takes one argument. In Groovy, this feature also exists as a consequence of its command chain feature. When implementing a DSL, this feature allows defining new code structures similar to built-in syntactic structures like `while` or `if`.

Yet, none of these languages can be used, as none provides a compiler that can be used in a web browser environment. Another possible GPL to use is Python, which can be run in the browser using Brython[12]. Python supports operator overloading, and custom DSL structures can be implemented using `with` expressions. However, Python has a rather complex grammar, which could make modifications complex. Additionally, limiting unwanted features, both for security and diagram consistency reasons **(RE I.4)**, would likely be difficult. While countless other languages which transpile to js[13] exist which can be directly used in a browser environment, we were not able to identify one which provides comparable syntactic flexibility to the languages mentioned before, and has tracing and a simple enough grammar required for our hybrid approach. Thus, we decided to instead implement our own GPL, SyncScript.

5.2.2 SyncScript

SyncScript is a GPL in which the diagram DSL is implemented as internal DSL. Its main design goals are

[12] https://brython.info/index.html

[13] https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS

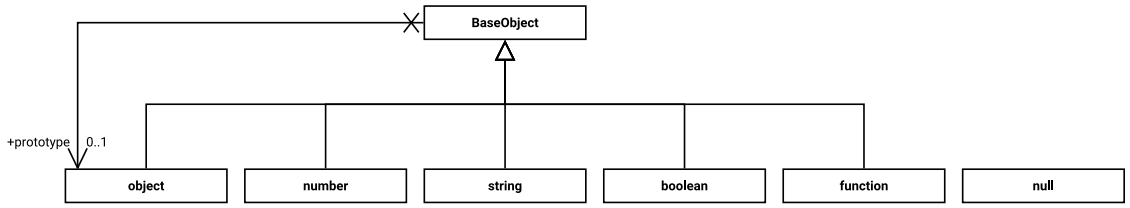

Figure 5.12: Simplified conceptual type model of SyncScript.

Flexible Syntax The syntax should be flexible to facilitate implementing internal DSLs. Mainly, features present in languages commonly used for this use case and described earlier should be implemented.

Simplicity Focusing on the most important aspects allows us to cut down the design and development effort of the language and related tools significantly, thus making realization as part of this thesis feasible. Concerning the grammar, it also makes programmatic transformations on the AST easier to implement and test to be working in all cases.

Familiarity SyncScript should feel familiar to users with experience in modern imperative languages, e.g., TS or Java, ensuring a good user experience. Additionally, requirements identified in area **III Programming Language** should be fulfilled.

Modular Approach It should be possible to extend the runtime with additional modules which provide additional functions, constants, etc. These extensions may implement functionality not originally present in SyncScript, e.g., short-circuit-evaluation.

Security Users should be able to execute untrusted SyncScript code in a sandboxed environment.

The remainder of this subsection details the lexical and syntactical grammar, data model, and conceptual aspects of the runtime.

Data Model

Figure 5.12 show the conceptual type model of syncscript. In total, we provide 7 basic types: object, number, string, boolean, function, and null. As SyncScript will be used in a web environment, in order to simplify implementation, the data model is highly inspired by the data model of JS. Thus, we support number, string, and boolean primitives, with similar semantics: Booleans can be either `true` or `false`. Strings store a sequence of (UTF-16) characters, and numbers represent 64-bit floating point numbers. However, for the remaining types, we diverge from the JS type system. First, SyncScript only supports `null`, compared to JS, which also provides `undefined`. Function support, similar to Python, Kotlin, and several other programming languages both index-based and named arguments. Inspired by Lua tables, objects conceptually are associative arrays. Fields can be stored both under string and number keys, where numbers are limited to integers. In contrast to JS, where numerical keys are implicitly converted to strings, keys `1` and `"1"` do not refer to the same field, but two separate fields. Numerical keys mitigate the need for a built-in low-level list/array type. Also, they allow converting a set of arguments provided to a function into a single arguments object, which will later be used during the design of the syntax of the language.

Algorithm 5.2 Retrieving field from BaseObject taking the prototype chain into account

```
function GetField(target, key)
    if HasLocalField(target, key) then
        return GetLocalField(target, key)
    else if HasProto(target) then
        proto = GetProto(target)
        return GetField(proto, key)
    else
        return null
    end if
end function
```

In order to further support object-oriented programming, we support prototype-based programming. In contrast to class-based programming, it provides – in our context – multiple benefits: First, prototypes are a better fit for our simplistic and minimalistic language design, as no conceptual distinction between classes and instances of classes exists in the data model, grammar, and runtime. Second, prototypes provide additional flexibility, as all objects can be used as prototypes of other objects. Third, as JS shows, classes can be implemented on top of prototypes, therefore, if the need for classes arises, they could still be implemented. However, it is not possible to implement prototypes based on classes, as creating instances of instances of classes is not possible. In our data model, prototypes are available for all BaseObjects, which includes all data types except null. For the user, the prototype is stored as a field under the key `"proto"`, which must be, if present, an object. Thus, the prototype can be retrieved and set like any other field. Yet, replacing the prototype is only supported by objects. As objects themselves can have a prototype, prototypes form a chain, the so-called prototype chain. Field lookup is the main purpose of prototypes in SyncScript. When retrieving a field by a key, if it is not found on the target itself, the field is looked up on the prototype of the target.

Algorithm 5.2 shows how this algorithm works in detail. It requires that the prototype chain does not form a cycle, as this would result in the algorithm running indefinitely. Thus, acyclicity must be ensured when replacing the prototype of an object.

Syntax & Semantics

With the data model now designed, next, we need to design the grammar and associated semantics. While the full syntactical grammar is displayed as a railroad diagram at Figure 5.13, following, we focus on its most important aspects. Table 5.3 gives an overview of all expressions SyncScript supports. Note that it shows the abstract syntax, while Figure 5.13 shows the concrete syntax of SyncScript, which must be unambiguous to a compiler and thus has a different structure.

Generally, the grammar represents a tradeoff between our overall goals of flexible syntax, simplicity, and familiarity. Two major design decisions influence its design: First, we design SyncScript as a dynamically typed language. This improves flexibility and reduces the development effort of the language and interpreter, however at the cost of less correctness guarantees and worse tool support, including but not limited to auto-completion. Second, the grammar of SyncScript uses (next to) no keywords. As keywords are usually reserved tokens that cannot be used by identifiers, this improves

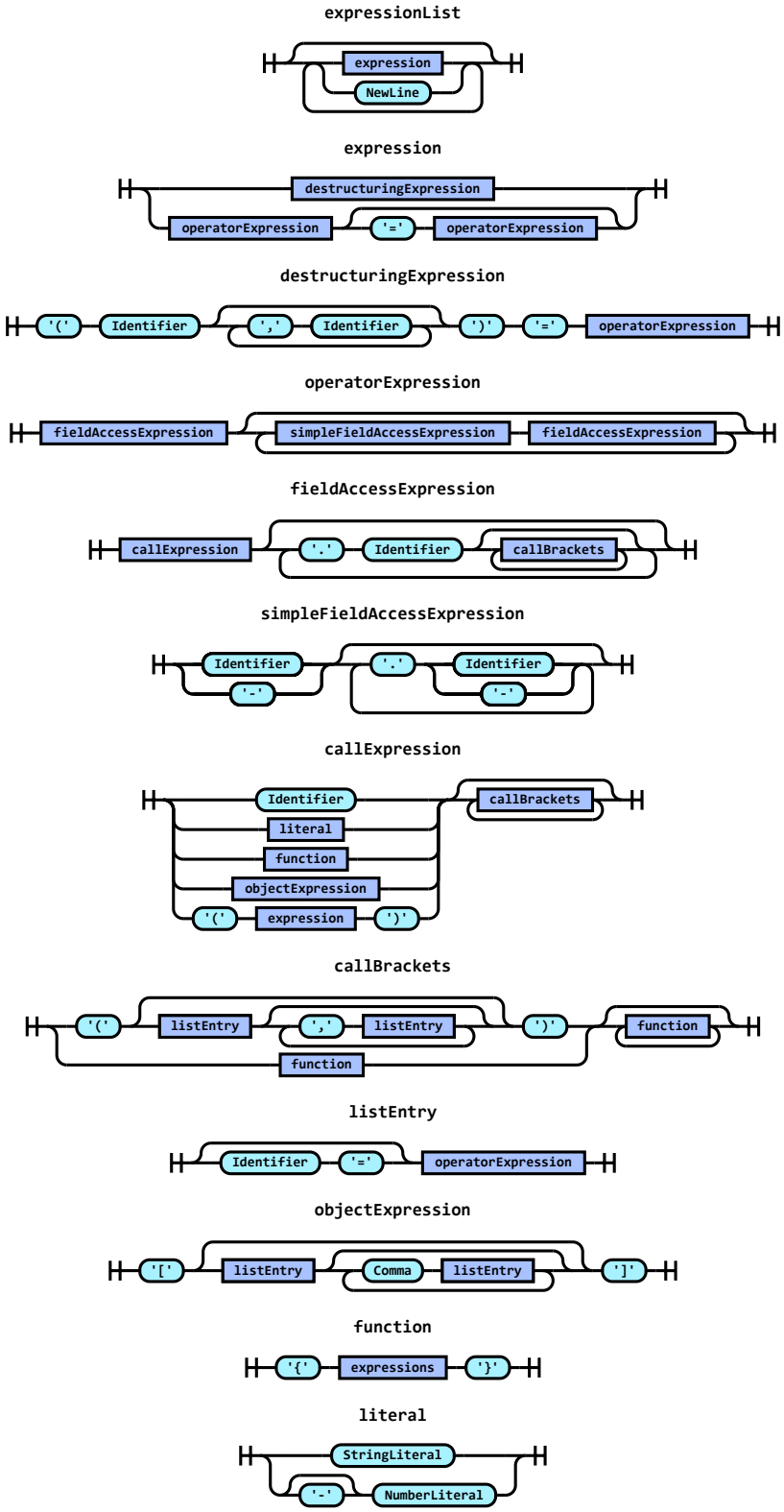

Figure 5.13: Syntax of SyncScript displayed as railroad diagram.

expression	description	example
identifier	get a variable in the current scope	`x`
field access	retrieves a field	`point.x`
operator	invokes an operator with left and right side expressions	`a + b`
call	invokes a function with possible arguments	`point(1, 2)`
function	creates a new function	`{ print("TODO") }`
assignment	assigs a value to a variable or field	`x = 1`
destructuring	assignes multiple variables by destructuring an object	`(x, y) = list(1, 2)`
string literal	creates a new string	`"a string"`
number literal	creates a new number	`1.0`
object	creates a new object	`[x = 1, y = 2]`

Table 5.3: Overview of expressions SyncScript supports.

syntactic flexibility of which internal DSLs profit. For several basic syntactic constructs, we are able to achieve all goals. This includes string and number literal, bracket, and assignment expressions which are all similar to their equivalents in languages like JS, Python or Java. However, most other constructs require some tradeoffs.

First, without keywords, control-flow constructs like `while`, `for`, and `if` cannot be included directly included in the syntax in a common way. As these features are required **(RE III.2)**, we instead need to derive a general construct, thus automatically fulfilling requirement **RE III.5**. As mentioned in Section 5.2.1, trailing lambdas, can be used as this required general construct. Syntactically, we take the same approach Kotlin and Scala: a lambda is defined by enclosing any amount of expressions in curly brackets. The lambda can then be added directly after the function call. Listing 5.1 shows how this can be used to implement an `if` statement expression.

Listing 5.1 Example use of trailing lambda to implement `if` using a function call.

```
1 if(condition) {
2     println("condition is true")
3 }
```

Yet, for our use case, the design Kotlin chose has one major drawback: using this syntax, only one trailing lambda is supported. If multiple lambdas need to be provided, all except the last one must be provided inside the round call brackets as regular parameters. As we also need to implement `while`, which requires two lambdas (both the condition and body) using this approach, we extend the syntax to support multiple trailing lambdas for the same function call. Listing 5.2 shows how this is used for `while`-loops.

Listing 5.2 Example use of two trailing lambdas to implement `while` using a function call.

```
1 while { condition } {
2     println("running while loop")
3 }
```

Listing 5.3 Kotlin lambda which takes two arguments.

```
listOf(1, 2).mapIndexed { index, value ->
    println(index)
}
```

In order for this to work, lambdas automatically return the result of the last inner expression, like in Kotlin. While not needed for these examples, in general lambdas need to access arguments provided on function calls. Like in Kotlin, the first index parameter is provided under the identifier `it`.

As Listing 5.3 shows, to access further arguments, Kotlin uses a special syntax. In order to not require such a syntax, SyncScript instead provides all arguments as an object under the identifier `args`. This is possible as objects can store fields under both integer and string keys. Yet, this introduces significant overhead for accessing multiple arguments. To simplify the retrieval of index-based arguments we introduce a destructuring expression. By putting n comma-separated identifiers enclosed in round brackets at the left side of an assignment, one can retrieve numerical fields 0 to $n-1$ from the object provided on the right side of the assignment. Listing 5.4 shows how this can be used to retrieve two index-based arguments in a lambda.

Listing 5.4 Destructuring expression used to retrieve index-based arguments provided to a lambda.

```
someList.map {
    (value, index) = args
}
```

Unlike most programming languages, SyncScript does not provide a separate syntax for functions. Instead, to create a function with a specific name, one assigns a lambda to a variable of said name. Thus, in SyncScript, there is only one way to create functions. Therefore, we call lambdas function expressions.

Like most modern programming languages, SyncScript uses static scoping, functions are implemented as closures. Technically, scopes are plain objects. Parent scopes are implemented by setting the prototype of a scope object to its parent scope. Thus, when looking up fields, if the field is not defined on the current scope, it is also searched for in all parent scopes thanks to the prototype chain. When using variables (identifier expressions) in a function, it retrieves the value by getting the field with said identifier. Also, each scope object stores a reference to itself under the key `this`. As a consequence, one can directly access the parent scope using the expression `this.proto`.

An operator expression consists of a left and right side expression and an operator in between. Without using keywords, they are implemented using infix functions. We consider two different implementations: First, similar to Groovy, the operator is transformed into a function call on the left side expression. Thus, `1 + 2` is identical to `1.+(2)`. However, this has the drawback that one cannot change the semantics of an operator in a specific context without modifying the result of the left-side expression (or its prototype). To allow for more flexibility, instead, the operator is resolved as an identifier expression in the current scope and then invoked with the left and right side expressions. Thus, `1 + 2` is identical to `+(1, 2)`. To further improve syntactic flexibility, SyncScript also allows field access expressions as operators. For example, `1 Math.sum 2` is also valid. However, in order for remaining unambiguous, call expressions cannot be allowed, as seen in Figure 5.13.

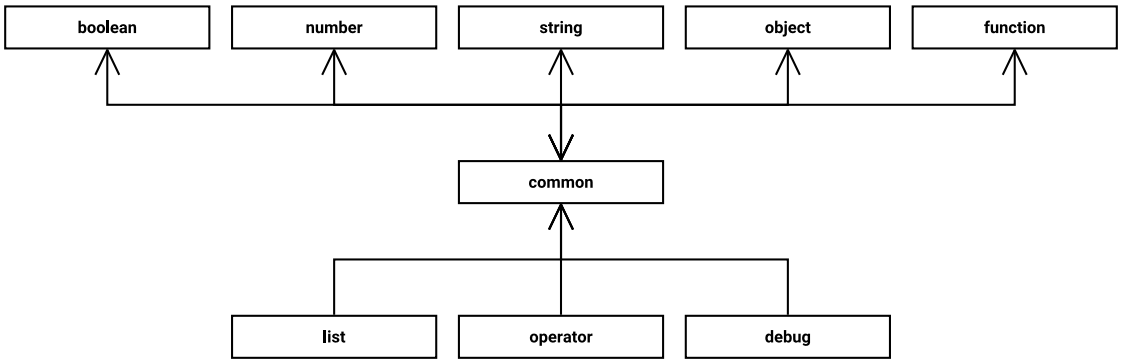

Figure 5.14: Modules of the standard library of SyncScript. Arrows represent dependencies that must be included when using the module.

As a consequence of using regular identifiers as operators, identifiers cannot be limited to alphanumerical characters, but also need to support other characters commonly found in operators, including `'+'`, `'-'` and `'*'`. An approach focusing on syntactic flexibility would allow all characters not used otherwise in syntactical constructs, e.g., `'='` and `'('`. Yet, this would result in `a+b` being parsed as one identifier instead of three as in most other programming languages. To improve familiarity for programmars, at the cost of reduced syntactic flexibility, we split identifiers into two categories: Alphanumerical identifiers consist of alphabetical characters, digits, underscores, and dollar signs. Special character identifiers consist of characters typically found in operators. To allow for operators like ==, `'='` is also allowed in identifiers if combined with at least one other special character. Similarily, to allow operators like ..., dots may be included in such identifiers, however only two or more consecutive dots.

Standard Library

Like any programming language, SyncScript provides its own standard library. In order to provide more flexibility, it is split into several modules. Figure 5.14 shows the modules the standard library consists of. Following, we give an overview of these and their provided functionality. Modules can be separated into two groups: First, core modules, including boolean, number, string, object, function, and common module. These should be included in all programs, as they provide required control flow structures and functionality for built-in types. As all core modules transitively depend on each other, they can only be included as a group. Common mainly provides the already mentioned `if` and `while` constructs, therefore making SyncScript Turing-complete. Both are implemented as higher-order functions, meaning functions that take themself functions as parameters.

The remaining core modules provide type-specific functionality. They mainly add functions to the prototype of these built-in types. As mentioned before, operators are functions that take the left and right-hand sides as arguments. However, when directly implementing the operator in this function, the operator function needs to be manually extended for each added type. Therefore, as Listing 5.5 shows, the default operator implementation simply delegates the call to the left-hand side. As a consequence, the type modules implement operators for the specific type by adding the operator functions to the respective prototype. For each type, the == function is added which serves as an equality operator. For both function and object, this is implemented as reference equality. For the

Listing 5.5 Simplified implementation of the + operator function, which delegates the function call to the left-side expression.

```
+ = {
    (left, right) = args
    left.+(right)
}
```

type	comparison operators	other operators
boolan	`==, >, <, >=, <=`	`||, &&`
number	`==, >, <, >=, <=`	`+, -, *, /, %`
string	`==, >, <, >=, <=`	`+`
object	`==`	
function	`==`	

Table 5.4: Operator support for built-in types provided by core modules.

primitive types, their implementation is identical to the JS === operator, thus comparing their value without performing any type coercions. As a result, `1 == "1"` evaluates to `false` in SyncScript, while it evaluates to `true` in JS. Table 5.4 gives an overview of which operators are implemented for which built-in types. Both boolean, string, and number support comparison operators less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (<=). For objects and functions, == is the only supported operator. Number additionally supports common arithmetic operators, while string supports concatenation via +. Boolean additionally provides and (`&&`) and or (`||`) operators. Similar to JS, these use short-circuit evaluation: `&&` only evaluates the right side if the left side is `true`. `||` only evaluates the right side, if the left side is `false`. Thus, expressions like `(value != null) && value.isPresent()` can be used without causing an error in case `value` is `null`.

In order to allow creating new objects, the object module provides the global `object` function. It takes any amount of arguments, both index-based and named arguments, and creates an object containing all those as fields. Additionally, the module provides the functionality required to interact with fields on objects. `get` and `set` can be used to get and set fields respectively. This can be used to retrieve fields dynamically at runtime and to interact with field names that cannot be used with the built-in dot operator, e.g., due to containing spaces. The `delete` function allows to completely remove a field under a specific name, similar to the JS `delete` operator. To facilitate functional programming, we provide a `forEach` function, which iterates over all fields and provides, similar to JS, both the value and name of af a field. As SyncScript does not support unary operators, boolean negation is instead implemented as the global function `!`, which takes one boolean as input and negates it as a result.

Extension modules form the second group of modules provided by the standard library. They depend on the common module, and thus, transitively on all core modules, yet, can be independently used from each other and may be omitted completely. As seen in Figure 5.14, currently, we support three such modules: The debug module provides a global `println` function. It allows to print any sort of value to the console, allowing for easier debugging. The list module provides the list data

type, as required by **RE III.3**. Lists are implemented as objects with a special list prototype, where list entries are stored in index-based fields. A list always maintains a `length` field, and stores the size of the list. A global `list` function allows creating new lists with the specified entries. To support functional programming, lists both support the `forEach` and `map` functions. In contrast to plain objects, the `forEach` function is overwritten to only iterate over the entries of the list, both `proto` and `length` are omitted. To modify lists, provide two additional functions which automatically maintain the `length` value: `add` allows to add a new value at the end of the list. `remove` removes and returns the last current value of the list. Inspired by Kotlin, lists support two types of operators: `+` concatenates two lists, and returns the new list. `+=` can be used to add a new value to an existing list. Last, the operator module provides commonly used operator functions. In addition to operators shown in Table 5.4, the following functions are provided globally: `!=`, `<<`, `>>`, `+=`, `??`. To support different datatypes, most operators are implemented as shown in Listing 5.5. However, the following operators require to provide a different implementation:

`+` To support string concatenation similar to languages like Java and Kotlin, if either the left or right side is a string, the other side is converted to a string, before invoking the regular implementation.

`-` To allow negating numbers, this method also supports taking only one numerical argument and negating it directly.

`??` Similar to Kotlin and JS. If the left side is `null`, the right side is returned, otherwise the left side.

`==` Allows comparing with `null` by first checking if the left side is null before delegating to the regular implementation.

`!=` Implemented by negating the result of `==`.

Design Decisions from Expert Survey

Based on the conducted expert interviews, we collected three remaining design decisions:

Should SyncScript have a special syntax for comments? Most programming languages support comments syntactically, both line-end comments and multi-line comments. However, adding comments to the syntax also makes the chosen syntax unavailable as DSL construct. For example, if `//` is used for starting a line-end comment, it cannot be used as an operator (as in Python). As an alternative, one could use string literals as comments, similar to how Python handles multi-line comments. Yet, multiple interviewees asked for comments during the interviews, and mentioned, that users likely expect this feature.

To what should assignments evaluate? Two options exist: First, assignments could evaluate to the assigned value. This can be useful when both assigning a variable and using it in a larger expression context. However, it increases the risk of accidentally using the assignment operator, when instead the equality operator `==` should be used. As an alternative, assignments could evaluate to `null`, effectively disallowing the use in larger expressions.

To what should loops (e.g. while) evaluate? As loops are implemented by function calls, loops are expressions that must evaluate to a value. Similar to assignments, we identify two options: First, they could always evaluate to `null`. Second, they could evaluate to the result of their body of the last iteration, or `null` if the loop was never executed. While easily obtaining

the last value might be useful for implementing some algorithms, it can also lead to faults if the loop is never executed and therefore evaluates to `null`. Initially, we also considered the option of evaluating to a list of the results of the body. However, while this solves the inconsistency if the loop is never executed, it results in a large memory overhead which is linear in the number of interactions of the loop, even in case the result is not used. Therefore, during the expert interviews, we decided to omit this option from the survey.

Figure 5.15 show the result obtained. First, experts clearly voted for supporting comments syntactically with 9 of 14 votes. From the eight experts voting for comments, C-like comments were mentioned the most, meaning line-end comments using `//` and multi-line comments using `/* ... */`, as their preferred syntax. Therefore, we implemented comments this way. Second, for both the second and third question, experts voted against always assigning `null` and therefore for the more flexible approach using the assigned value for assignments and the last inner value for loops. Interestingly, while 12 experts voted for the assigned value for assignments, only 9 experts voted for the last inner value for loops. Therefore, we expect the former to be the more common use case.

5.2.3 Diagram Internal DSL

With SyncScript, the GPL to implement our diagram DSL in being defined, next, we detail the diagram DSL itself. Similar to how the standard library of SyncScript is structured, the diagram internal DSL is implemented as two modules. Following, we first the lower-level diagram module. Finally, we conclude this section with the higher-level DSL module based on the diagram module

Diagram Module

Primarily, the diagram module provides functions used to create the basic data types described in Section 5.1.1. These functions are structured similarly to how diagrams are structured: First, the `diagram` function allows defining diagrams as a triple of an element tree, styles, and fonts. Two functions, `font` and `fontFamily` allow creating the fonts required to display text in a diagram. To be independent of a network connection **(RE I.8)** and locally installed fonts, we provide three

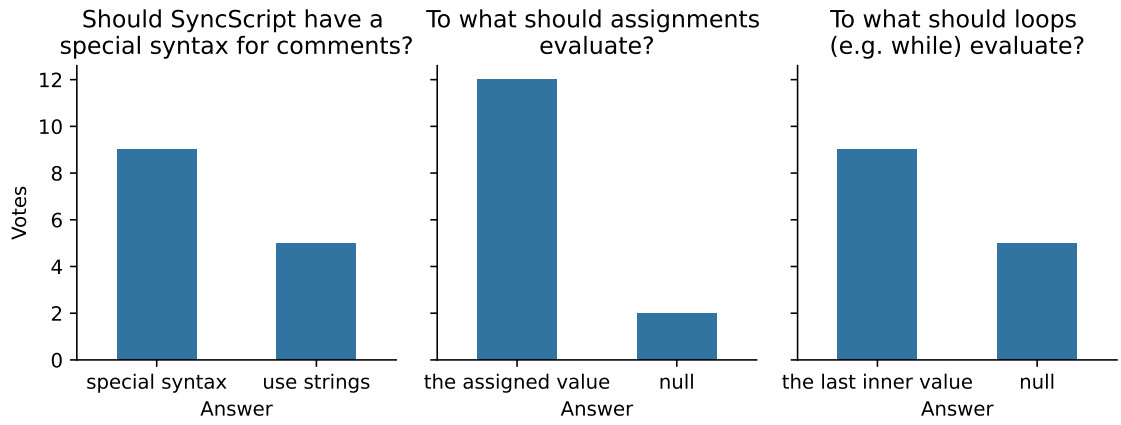

Figure 5.15: SyncScript design questions survey results.

Listing 5.6 Creating a nested diagram element using the trailing lambda syntax.

```
vbox {
    rect()
    rect {
        text {
            span(text = "lol")
        }
    }
}
```

Listing 5.7 Styling with the diagram DSL using nested selectors.

```
styles {
    type("rect") {
        stroke = "black"
        strokeWidth = 5
        cls("title") {
            fontSize = 10
        }
        any {
            maxWidth = 200
        }
    }
}
```

open-source fonts by default: Roboto, Open Sans, and Source Code Pro. To create the element tree, for each element defined in Table 5.1, the diagram module provides a function to create said element. Attributes of the element are provided to the function as named arguments, including the child elements. To simplify creating nested elements, child elements can also be defined using a nested trailing lambda syntax, inspired by *Jetpack Compose*[14], as shown in Listing 5.6. Last, the `styles` function allows creating custom styles. To achieve a consistent user experience, we use an SCSS-inspired trailing lambda syntax to allow nesting selectors and defining attributes inside the block. Listing 5.7 gives an example with a class- and any-selector inside a type selector. Note that attributes can both be defined in the inner selectors, but also inside the top-level type selector. Furthermore, as Listing 5.8 shows, style variables can both be used and defined inside selectors. The `var` function allows defining variables of a name. To assign a value to variables, one can either set fields on the `variables` object provided in each scope, or use the `vars` function, where variables can be set similarly to regular attributes. While `variables` only sets the variables on elements matching the selector, the `vars` function implicitly creates an any-selector, thus also setting the variable on any child object. We decided to provide this function, as style variables are often used for theming, where typically, users want to theme either the whole diagram, or some part of the diagram (e.g. a UML class), however, not only a specific element, e.g., a rectangle.

[14] https://developer.android.com/jetpack/compose

Listing 5.8 Style variables used with the diagram DSL.

```
styles {
    type("rect") {
        fill = var("primary")
        stroke = var("secondary")
    }
    type("rect") {
        variables.primary = "green"
    }
    vars {
        secondary = "black"
    }
}
```

DSL Module

The DSL module provides common functionality required for internal DSLs for different diagram types. First, as specified by **RE II.1**, a DSL construct should be used to define which type of diagram is defined. Syntactically inspired by *PlantUML*, we use a function call with a tailing lambda to create a block where the diagram is defined. The function used, also called the diagram environment function, defines the type of diagram, and creates the resulting diagram by executing the callback provided to it. To not pollute the global scope and prevent conflicts between diagram types, diagram type-specific functionality is only available inside the callback provided to the diagram type function. To simplify defining new diagram types, the DSL module provides the `generateDiagramEnvironment` function. It returns the diagram environment function and takes itself a callback which can be used to enhance the scope used in the diagram environment function, as shown in Listing 5.9.

Listing 5.9 Definition and usage of a diagram environment using `generateDiagramEnvironment`

```
classDiagram = generateDiagramEnvironment {
    it.class = { /* implementation */ }
}

classDiagram {
    class("Test")
}
```

While most functionality available inside the diagram environment depends on the diagram type and is, therefore, provided by the diagram type plugin, some common functionality is provided by this module. First, diagram types often use associations between diagram elements, when using our framework, these are represented by canvas connections. Different associations are typically differentiated by using different markers at the start and or end of the canvas connection, e.g., an arrowhead. Inspired by PlantUML, we decided to use an ASCII-art-like syntax to define such

canvas connections. For example, to create an arrow from `A` to `B`, one could use `A --> B`. Here, `-->` is implemented as an operator. To ease the definition of such functions, the DSL module provides the `createConnectionOperator` function, which takes a start marker factory, end marker factory, and a style class list, and returns the operator infix function. To further improve the flexibility of the created operator, it both supports canvas elements and canvas points as the start and end. By default, the created connection consists of a single line segment.

Next, multiple experts requested support for an element-specific styling DSL (**RE II.9**). The chosen design should be both consistent with how elements are defined, similar to how global styles work, and respect separation of concerns (**RE II.11**). To satisfy these constraints, we introduce the section concept, as shown in Listing 5.10. Here, `sectionX` are operators which modify the left-hand side

Listing 5.10 The section concept allows defining different aspects of a canvas element, e.g., styles and layout, while maintaining separation of concerns.

```
aCanvasElement section1 {
    // content
} section2 {
    // content
} section3 {
    // content
}
```

element using the function provided as the right-hand side argument. To remain flexible and allow the chaining of sections, `sectionX` functions should always return the left-hand side argument. The availability of sections depends both on the element to modify and the used diagram type. By default, we provide three types of sections with the DSL module: The `styles` section works similarly to the global `styles` function introduced earlier. However, styles are only applied to the element on which the section is present. Also, attributes can be added directly in the `styles` block without an additional selector, these attributes are applied directly to the element. The `styles` section is supported both for canvas elements and canvas connections. Next, the `layout` section is only available for canvas elements. It allows to define the width, height, and pos(ition) of the element by assigning the respective field. Last, the `with` section is available for canvas connections. It is used to define the segments of the connections and add labels to the connection. Listing 5.11 gives an example of how such a path can be defined. To change the segments, a user assigns the `over` field. The path is defined using a fluent builder syntax. `start` and `end` functions are used to refer to the start and end elements of the connection. In case a canvas element is used, the functions

Listing 5.11 Canvas connection with a `with` section defining a two-section path and three labels.

```
A --> B with {
    over = start(0).line(apos(0, 0)).line(end(0.5))
    label("Label 1", 0.5)
    label("Label 2", 0.6, 30)
    label("Label 3", 0.8, -30, 90)
}
```

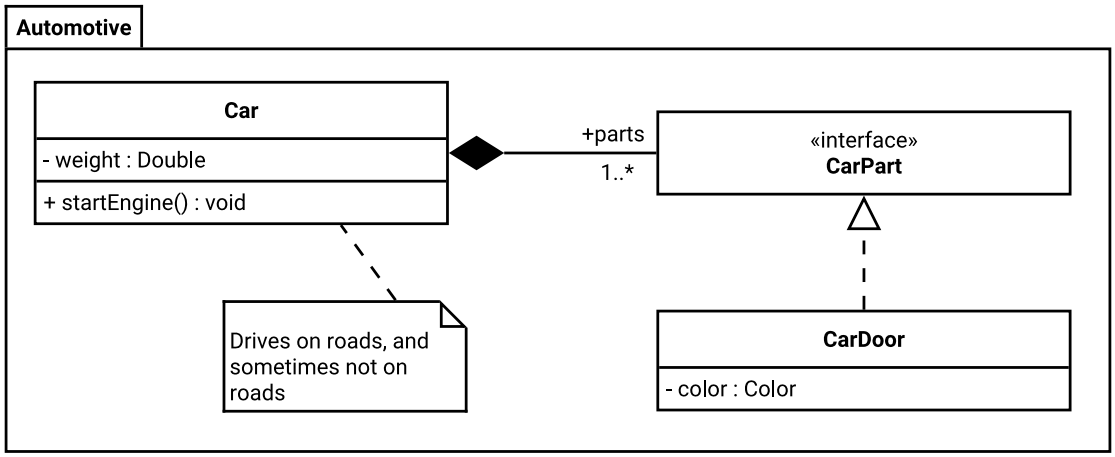

Figure 5.16: An example UML class diagram with a class, interface, comment, and package.

take a number as input which defines the relative position on the outline, in case a canvas point is used, no input is required. To create the different types of segments, `line`, `axisAligned`, and `bezier` functions can be used with the same semantics as described in Section 5.1.1. In order to support a brief notation, each segment also supports taking multiple points as inputs. Furthermore, in contrast to the primitive bezier segment, which takes 4 points (start, end, and both control points), the `bezier` function automatically uses relative coordinates for the control points. Additionally, when defining multiple sections with one call, except for the last segment, only the start control point is provided, and the adjacent end control point is automatically calculated. Both decisions were made to achieve a better user experience. Last, the `with` section also allows defining labels. The `label` function takes one to four elements with the following meaning: displayed text, global position on the line, distance to the line, and rotation (in degrees). If a segment-based position is preferred, one can define the segment index using the named argument `seg`. As labels are implemented as canvas elements and the `label` function returns the created element, both `styles` and `layout` sections can be used for such labels if desired.

5.3 UML Class Diagram

Overall, our framework is diagram type-independent. Additional diagram types are supported by a modular approach. In order to evaluate said approach, we decided to implement a UML class diagram module. Overall, our goal is to support all diagram elements described in Section 2.1.2. As our concept is built around flexibility, the low-level diagram described in Section 5.2.3, to which users have access, would allow creating all required graphical elements. Therefore, in this section, we focus on how higher-level DSL constructs simplify the creation of said graphical elements. Figure 5.16 shows an example diagram with all major supported elements: classes, interfaces, packages, and comments. Furthermore, it shows a realization and an association. Following, we introduce the DSL constructs used to create these elements.

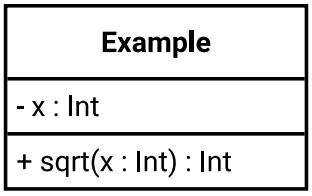

Figure 5.17: Class with a property `-x : Int` and an operation `+sqrt(x : Int) : Int`

As the name suggests, classes are the most common element used in a class diagram. As described in Section 2.1.2, a class primarily consists of a name, properties, operations, and a set of stereotypes. To create a class, we provide the `class` function. It defines a class in two parts: First, it takes the name of the class as a positional argument. Second, it takes a function, which is used to define the content of the class, properties, and operations. Typically, it is provided using the trailing lambda syntax, to create a syntax known to users from other tools, including *Mermaid* and PlantUML. Stereotypes can be defined using the named argument `keywords`. We decided to use the term keywords instead of stereotypes, as several terms used commonly, e.g., interface and enumeration, are in fact keywords for metaclasses [CBR+17, p. 747]. While semantically, as part of a UML model, there is an important difference between keywords and stereotype names, from a graphical perspective, there is none. Also, this increases internal consistency, as keywords can be defined for elements that do not support stereotypes, e.g., packages. Formally, with respect to the UML DI standard, the `class` function thus creates a UMLClassifierShape. To define properties and operations, we consider three approaches:

Strings Here, a `section` function allows adding a new compartment with a list of text entries. Each entry is added in its own line and can contain arbitrary text. Thus, while this approach is the most flexible, it also performs no formatting and validation. Furthermore, users need to manually separate properties and operations.

DSL Functions With this approach, DSL constructs, especially functions, and operators, are used to define properties and operations. Still, all identifiers, especially types, are defined as strings. In comparison to **Strings**, both validation and formatting can be provided easily. We considered different syntaxes for both properties and operations. However, due to being limited to the syntax of SyncScript, it is not possible to replicate the exact UML notation.

Declarative With this approach, properties, and operations are created declaratively. Similar to Java reflection, instead of executing the SyncScript code to generate the entries, its syntactic structure is analyzed. This brings two major benefits: First, the resulting syntax is, as Listing 5.12 shows, significantly closer to the generated entries. Second, no quotation for identifiers is required, as instead of using strings, SyncScript identifiers can be used directly. Yet, due to its declarative nature, in contrast to **Strings** and **DSL Functions**, this approach limits the use of programming language constructs. Especially, no variable can be used, as the code is not evaluated during execution.

Listing 5.12 shows how a simple class with a single property and operation (Figure 5.17) can be defined with all three approaches. Note that for **DSL Functions**, this only represents one possible implementation for the DSL constructs. To decide which approach to choose, we added three design decision questions to the survey: First, we asked the experts to rate the three approaches

Listing 5.12 Definition of Figure 5.17 using all three potential approaches to define properties and operations.

```
// approach 1: Strings
class("Example") {
    section("-x : Int")
    section("+sqrt(x : Int) : Int")
}

// approach 2: DSL Functions
class("Example") {
    -("x" : "Int")
    +("sqrt") : fun("x" : "Int") : "Int"
}

// approach 3: Declarative
class("Example") {
    private {
        x : Int
    }
    public {
        sqrt(x : Int) : Int
    }
}
```

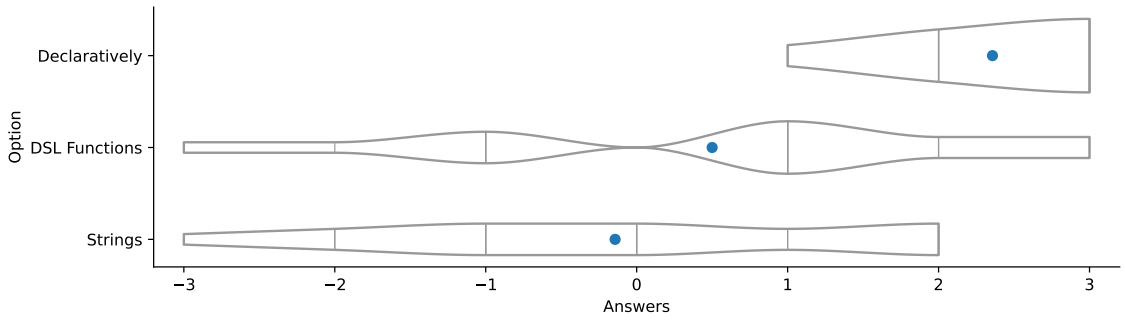

Figure 5.18: Survey results regarding which overall approach to choose for declaring properties and operations.

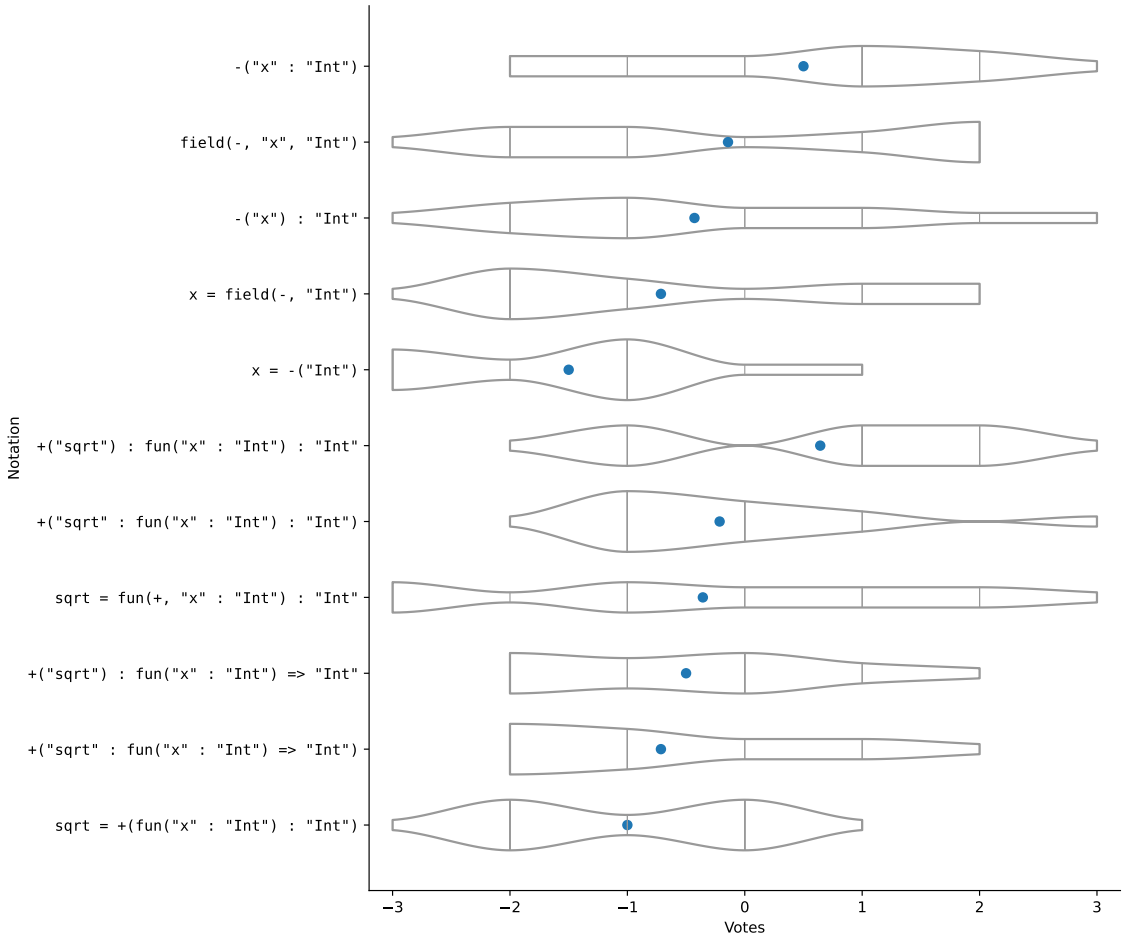

Figure 5.19: Survey results regarding how to declare properties and operations using DSL Functions.

overall. Figure 5.18 shows the collected results, with an average vote of 2.36, the experts clearly prefer **Declarative**, followed by **DSL Functions** with **Strings** last with an average vote of -0.14. Regarding **DSL Functions**, we also asked the experts to rate different options on how to define operations and functions. Figure 5.19 show the results obtained, with the same example property and operation as shown in Listing 5.12. Note that for both properties and operations, only one possible implementation was rated positive on average. Still, with an average rating of 0.50 (property) and 0.64 (operation), both top-rated options were rated worse than the **Declarative** approach. Therefore, we decided to implement the declarative approach. Due to being the low-level implementation, **Strings** will also be supported, but will likely not be used by users primarily. During the expert interviews, some experts preferred to define visibilities using symbols (`'+'`, `'#'`, `~`, `-`), while others preferred the textual version (`public`, `protected`, `package`, `private`). In the survey, 8 out of 14 chose the textual syntax, which we therefore implemented.

With respect to the UML subset we support, visually, interfaces are similar to classes, and only add the `interface` keyword. Therefore, the DSL function `interface` works similarly to `class`. Last, we also support UML comments and packages. Comments are declared using the `comment` function, which takes the text to display as sole input. Similarly, packages are created using the `package` function. It takes the name as the first parameter, and similar to classes, any amount of keywords

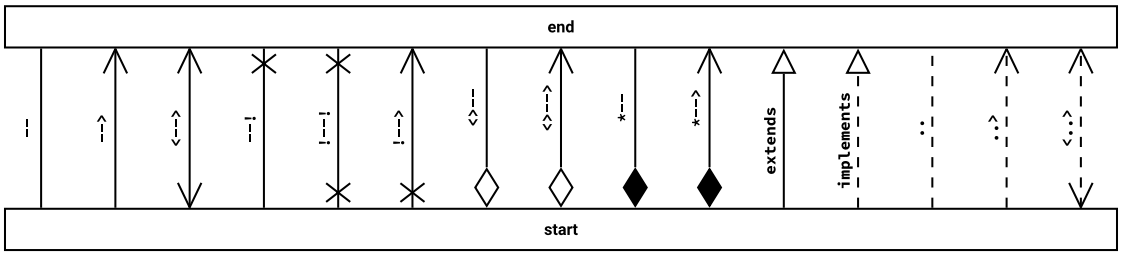

Figure 5.20: Connection operators can be used in a class diagram. Note: mirrored versions are also supported for most operators.

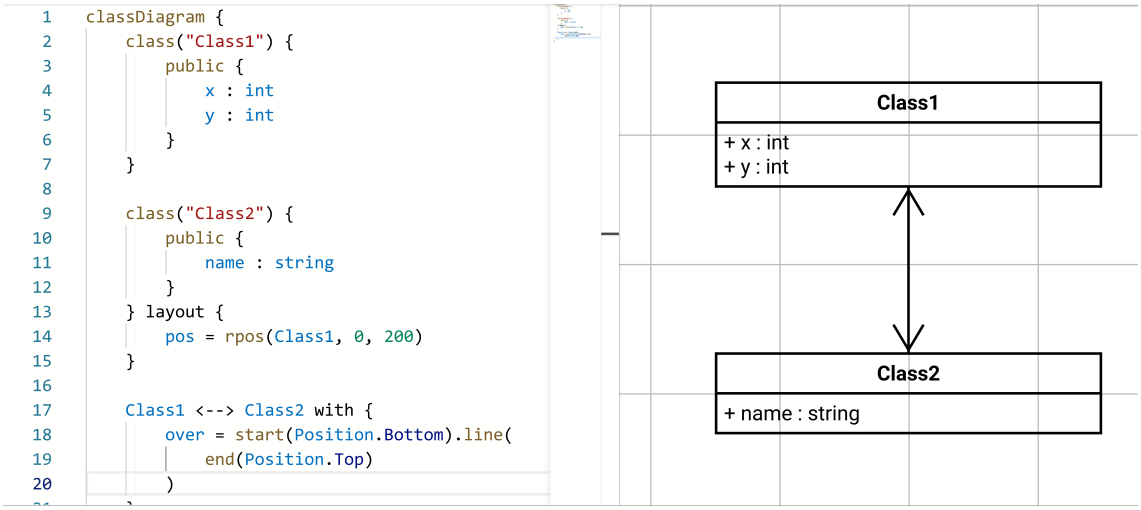

Figure 5.21: The hybrid editor how it is implemented in the web environment.

under the `keyword` parameter. Also, a function can be provided as a second positional parameter, yet, in contrast to classes, it only enables the user to structure the code, it is executed after creating the package and provides no further semantics.

To create connections between diagram elements, operator infix functions are provided, as described in Section 5.2.3. Figure 5.20 gives an overview of the supported operators. Generally, a syntax close to Mermaid and PlantUML was chosen. However, in SyncScript, identifiers are limited to either alphanumerical or special characters. Therefore, `x--` cannot be used as an identifier. For this reason, as shown in Figure 5.20, for non-navigatable association ends, we use `'!'` instead, while for aggregations, we use `'<>'`.

5.4 Hybrid Editor

The hybrid editor is the primary diagramming environment users use to define diagrams. As Figure 5.21 shows, it consists of two separate views: the textual view, and the graphical view. As defined by our hybrid editing concept, both views give the user the ability to modify the diagram. Furthermore, both views are live-synced. Whenever the DSL code in the textual view is updated, the code is executed, and the new resulting diagram is displayed in the graphical view. Also, the

graphical view allows to modify the diagram, these changes are live-persisted in the textual view. Next, we first detail important aspects of the textual editor in Section 5.4.1. Following, Section 5.4.2 introduces the concept of our interactive graphical editor.

5.4.1 Textual View

The textual view provides the environment which allows a user to define the diagram using our diagram DSL. In contrast to the graphical view, a different component is used depending on the environment the hybrid editor is set up. Especially, when used as an IDE integration, the basic textual view is provided by the chosen IDE. However, independent of the environment, to provide a good editing experience, some basic features, e.g., syntax highlighting, need to be provided. To support a wide range of textual editors, where possible, features are implemented using the *Language Server Protocol (LSP)*. Currently, the following features are provided:

Syntax Highlighting To improve understanding of the DSL code, the following constructs are syntax highlighted: String and number literals, functions, variables, methods, and fields.

Diagnostics Ranges of the source code that cause errors are highlighted, and an error message is provided. This includes lexical, syntactical, and runtime errors. Where possible, errors are marked as fine-granular as possible. For example, if a parameter causes an error due to being of the wrong type, only the parameter is marked as an error.

Auto-completion When entering text, auto-completion allows the user to see valid continuations of the currently entered code. Currently, we support completing all identifiers valid in the current scope. Additionally, if a completion item represents a function, documentation for the function can be shown. Furthermore, for some identifiers, additionally, snippets are provided that insert a block of code. For example, when using the `with` operator on canvas connections, a block defining a line from start to end can be inserted.

Formatting The formatter allows for beautifying the code, thus allowing the user to write code without worrying about manually formatting the code. Whitespace, especially indentations, are automatically normalized, and lines exceeding a certain limit – when possible – split into multiple lines.

5.4.2 Graphical View

The hybrid graphical view serves three main purposes: It allows live feedback on textual edits, allows exploring the diagram, and also allows modifying the diagram itself. To support those use cases, the graphical view supports editing and non-editing features:

Non-editing Features

Non-editing features allow users to explore the diagram, without modifying the diagram itself. Currently, we support the following features:

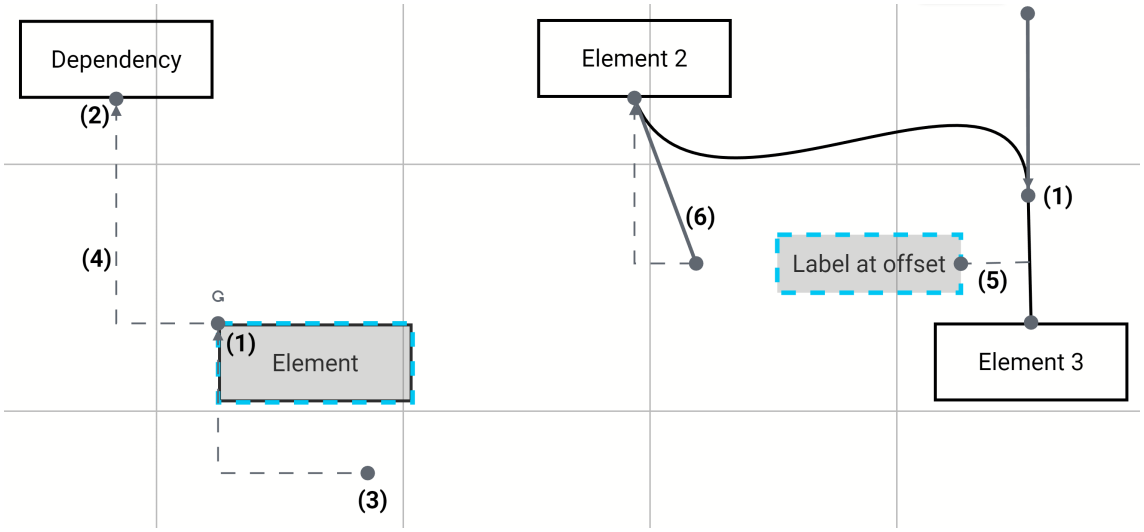

Figure 5.22: Diagram with two selected elements, showcasing added UI elements.

Navigatability As defined by **RE IV.1**, users want to navigate the diagram interactively. This includes both zooming and panning the canvas. While zoom and panning are not persisted in the diagram itself, they will be retained after diagram updates.

Animated Updates To improve the editing experience, the graphical view animates transitions between diagrams. This allows the user to focus on updated elements, and understand how textual edits affect the diagram.

Selection Currently, canvas elements and canvas connections can be selected. Selecting diagram elements serves two purposes: First, only selected elements can be manipulated using the editing features. Therefore, in order to manipulate multiple elements at once, e.g., when moving elements, experts required multi-selection **(RE IV.5)**. From a usability perspective, elements are selected by left click. If the modifier key Ctrl is pressed, currently selected elements aren't deselected when clicking on another element, resulting in multi-selection. Second, for selected elements, the graphical view shows additional User Interface (UI) elements which enable the user to understand the structure of the diagram. Figure 5.22 provides an overview of those UI elements. First, for both canvas elements and canvas connection, points they depend on are added (1). Next, both dependencies (2) and dependant (3) points are visible, with a dotted dependency arrow (4) to showcase the direction of the dependency. Furthermore, for line points, if the line point defines a distance to the line, a dotted line is added towards the root point on the line their part of (5). Last, for bezier segments, control points are connected with a handlebar (6) to their adjacent point on the segment.

Navigate To Source For larger diagrams, finding a diagram element in the textual view can slow down the user. Therefore, as defined by **RE IV.7**, we support navigating to the definition of the element in the textual view. Here, to focus the user's attention on the correct region, the defining code is both scrolled into view and selected. As different ways for triggering the navigation exist, we included a design decision question to the survey. Figure 5.23 shows the results, as "Left click + modifier key" achieved the highest average vote of 1.71, we decided

to implement this feature as Left click + alt, as both Ctrl and Shift were already used for other functionality. Also, to allow for mouse-only navigation, we also decided to trigger navigate to source on double click.

Editing Features

In contrast to non-editing features, editing features allow manipulation of the diagram itself. Thus, such changes are persisted in the textual definition of the diagram. For the scope of this thesis, we support four different types of interactions: (1) Move points and canvas elements **(RE IV.1)** (2) Resizing canvas elements **(RE IV.2)** (3) Rotating canvas elements **(RE IV.3)** (4) Manipulating axis-aligned canvas connection segments. All four supported interactions are continuous transformations of the diagram, meaning that the user gradually manipulates the diagram, e.g., by dragging a canvas element from a start to a target position. As part of our live-synced concept, the textual definition of the diagram is continuously updated, and these changes update the diagram. However, this results in some challenges. First of all, as control flow structures are supported, most important `if`, all edits to the diagram can cause structural changes to the diagram, and, as a result, changes to the ids of moved elements. Therefore, to keep the manipulation consistent, we determine the locations of textual edits at the beginning of the interaction and do not reevaluate those during the edit. Also, before starting the manipulation, we ensure that textual edits are non-conflicting, meaning that textual edits do not overlap. However, despite these precautions, we do not guarantee that the resulting textual edits will result in the desired changes. As an example, consider the following example:

Listing 5.13 Canvas element where move-edits will not result in the desired output.

```
1 element layout {
2     x = 0
3     if(x > 200) {
4         x = 200
5     }
6     pos = apos(x, 0)
7 }
```

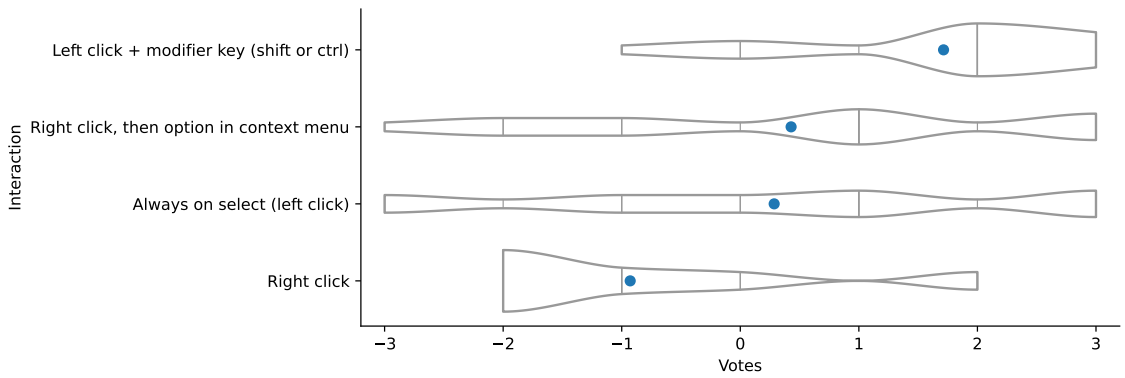

Figure 5.23: Survey results regarding which interaction triggers navigate to source.

When moving `element`, the `x` coordinate in line 1 is updated. However, when exceeding 200, this edit results in no further changes, as in this case, `x` is fixed to the value 200 in line 4. Yet, continuously updating the diagram allows for communicating these limitations to the user.

Canvas points and elements can be moved on the canvas by first selecting them, and then dragging them. First, to be movable it is ensured that the points can be moved consistently. In order to do so, elements and points to move are determined. Overall, the algorithm works in the following steps:

(1) Add points and elements to the set of moved points and elements required to move the initial points and elements. Generally, not all points can be moved, primarily when the location of their definition is unknown or marked to not be edited. If a relative point cannot be moved, its target is added to the moved set, this is done recursively until no further changes occur.

(2) Remove points and elements which are implicitly moved. This affects both line and relative points. A relative point is implicitly moved, if its target is already moved. A line point is implicitly moved, if all points affecting the line are moved. In case the line is the outline of a canvas element, this is the canvas element itself. In case it is a canvas connection, this includes all points which define its segments.

(3) Check that all points and elements remaining can be moved.

(4) Check for consistency: if a line point is in the set of points of elements to be moved, it must be the only entry.

Two different types of moves are implemented: line point moves and translation moves. Line point moves are performed if only a single line point should be moved, translation moves in all other cases. A line point move allows the user to update the segment index, position, and distance of the line point. All three values are replaced completely on edit. In case the distance is not defined, the point can only be moved on the line. Otherwise, arbitrary positions are allowed. Note that depending on the path of the line, not all positions on the canvas can be achieved. In this case, the closest position is chosen. Translation moves modify the position of points and elements by adding or subtracting the change in position. The exact edit depends on the expression: If a coordinate is defined as a number literal, it is replaced completely. However, if another type of expression is used, the increment is added or subtracted, to be as close as possible to the original expression. Of course, if the expression is already an addition or subtraction using the `+` or `-` operator, the added/removed value is modified, instead of adding an additional addition/subtraction. Furthermore, for canvas elements not defining a position at all, an absolute position is added in the `layout` section, or, if no `layout` section exists, a new one is created.

Similarly, to resize one or more elements, the user can drag the edge or corner of a selected element. In case a corner is targetted, the element is resized in both dimensions, otherwise, only one dimension is changed. To give feedback to the user on what type of resize is used, the cursor is updated to an appropriate icon when hovering at the correct spot. Generally, the goal of resizing is to scale the element so that the selected edge or corner is at the position the cursor ends at. However, this can be challenging, as canvas elements can both be rotated, and aligned differently. For example, if an element is center-aligned, the resize has to be double the actual moved distance, as the element effectively grows in both directions. Also, when resizing the side the element is aligned at, the edge will remain at the same location independent of the scale. For this, we identify two options: either, we also modify the position, or resize it by the expected amount, however, the element will grow in the opposite direction. Ultimately, we chose the latter approach, to limit the number of

edits required, and limit the number of undesired consequences. In contrast to move-edits, where values are manipulated additively, resize-edits manipulate values multiplicatively. This provides the benefit of when resizing differently sized elements, their relative size is maintained. Also, it prevents smaller elements from being resized to a negative size, which is not supported.

As shown in Figure 5.22, a rotate icon is displayed for rotatable selectable elements (near (1)). By clicking and dragging the icon, a canvas element can be rotated around its origin. In contrast to move and resize, rotate only affects only the element of which the icon is dragged, and not all selected elements. Also, similar to line point moves, the angle is replaced on edit, and no additive or multiplicate manipulations are applied, to prevent under and overflows of the designated value range (0 to 360).

Last, the graphical view allows modifying the relative position vertical/horizontal part of an axis-aligned canvas connection segment. As described in Section 5.1.1, the pos of the segment can, depending on its sign, both define the relative position of the vertical or the horizontal part of the segment. To support both semantics, the graphical editor allows dragging all parts of the segment. Again, like line move edits and rotate edits, the pos value is replaced with the new value.

6 Architecture & Implementation

The primary goal of this thesis is the development of a framework for modular hybrid diagramming. As part of this thesis, we both developed the framework itself, which can be found at https://github.com/hylimo/hylimo, and a web-based version of the hybrid editor hosted at https://hylimo.github.io. In Section 6.1, we first give an overview of the architecture of the framework, and how it is used to create the web-based hybrid editor. Following, Section 6.2 details some chosen aspects of the implementation.

6.1 Architecture

Figure 6.1 shows the architecture of our web-based hybrid editor. The editor is realized as a client-server architecture, consisting of four main components: textual editor, graphical editor, language client, and language server. The first three components together form the client side of the architecture. As a textual editor, we decided to use the *Monaco Editor*, due to the excellent support for the *Language Server Protocol (LSP)* in the form of the *Monaco Language Client*[1], based on which we implement our language client. It connects through the language client to the language server. First, as defined by the LSP, the language server provides common functionality

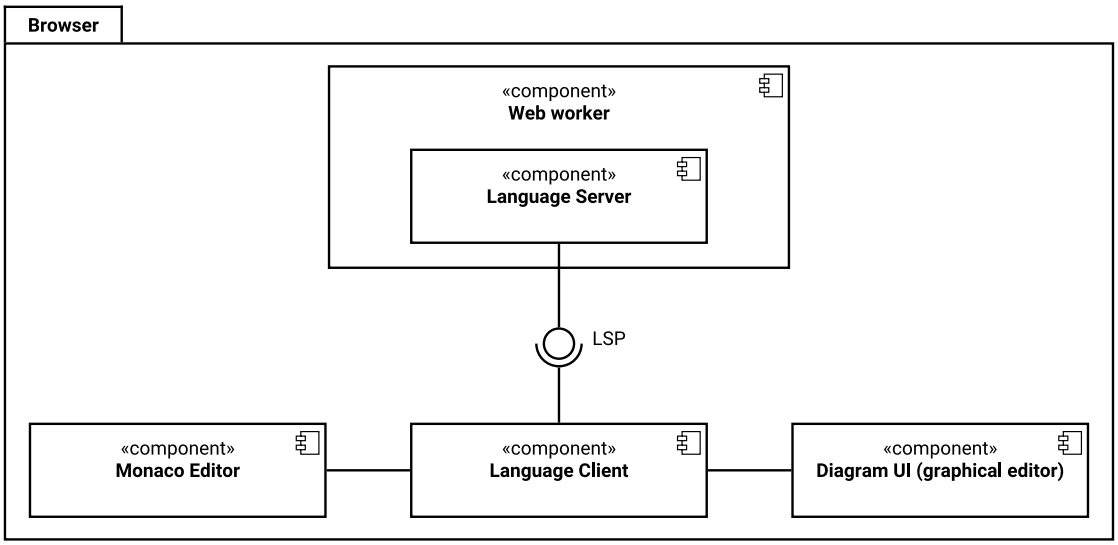

Figure 6.1: Architecture of the web-based hybrid editor.

[1] https://github.com/TypeFox/monaco-languageclient

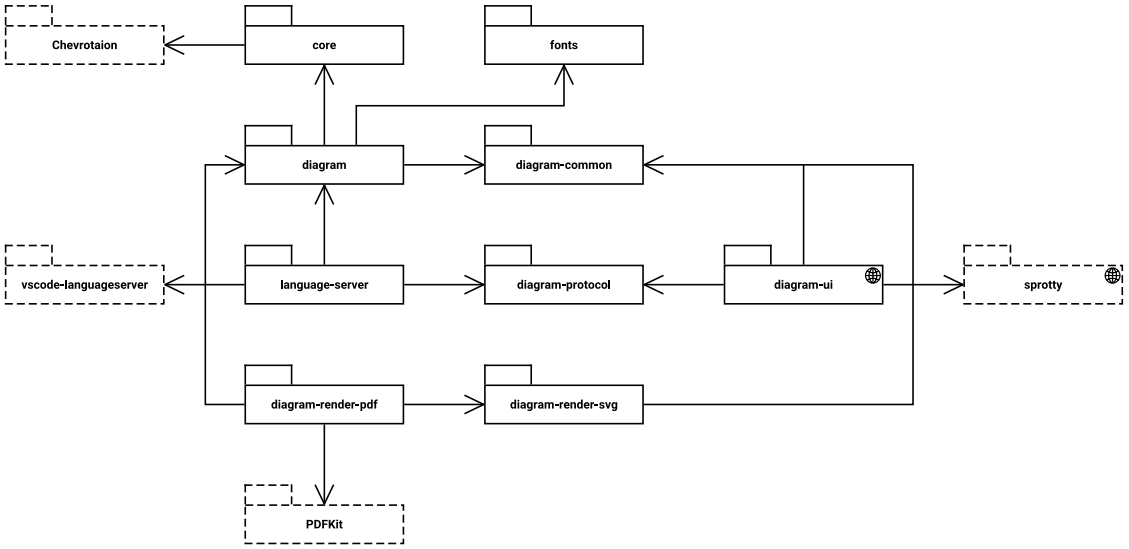

Figure 6.2: Module structure of the HyLiMo framework.

required by textual editors, including formatting, auto-completion, and error markers. Second, it also serves as a diagram server, providing diagrams to connecting clients. The diagrams are generated based on the textual documents registered via LSP. To prevent blocking the UI thread during diagram generation, the language server is executed in its own separate thread using a web worker. The graphical editor is the last major component of our architecture. It displays the diagram provided by the language server and allows modifying the diagram. Similar to the textual editor, it connects via LSP through the language client to the language server. For implementing the graphical editor, we considered two different architectures found in related work: First, when using the approach presented by Glaser [Gla22] for developing *BIGER*, we would implement the graphical editor directly based on *Eclipse Sprotty*[2]. Graphical manipulations would be communicated to the language server, and they are mapped to textual edits. Alternatively, we considered using the higher-level *Graphical Language Server Platform (GLSP)*, as shown by Walsh et al. [WDJ22]. Similar to [WDJ22], here, graphical edits would be persisted in the textual diagram definition by "serializing" the source model provided GLSP[3]. Ultimately, we decided to use the first architecture, as serializing the source model could prove to be challenging, as our textual definition uses an internal DSL with support from general-purpose programming language features. Graphical edits need to modify the textual definition, rather than regenerate it. Also, having an additional source model would result in additional complexity, in particular as defined by our concept, the diagram generated on the language server is layouted already.

Out of these four components, both the language server and graphical editor are provided by our framework, while the textual editor and language client were selected independently for the web-based editor. This architecture allows bringing the hybrid editor to additional environments in the future. In particular, IDE integrations **(RE I.2)**, where the textual editor is typically provided by the IDE, are possible. Yet, the language server-based editor is only effective in environments,

[2] https://github.com/eclipse-sprotty/sprotty
[3] https://www.eclipse.org/glsp/documentation/sourcemodel/

where a textual editor to connect with the language server is present. To simplify the usage of our framework as a library, e.g., for integration into a CLI **(RE I.3)**, we further split our framework into multiple modules. Figure 6.2 gives an overview of available modules, how these modules depend on each other (arrows), and important external dependencies (dashed). Modules marked with a globe symbol can only be used within a browser environment. Overall, our framework consists of the following modules:

core Base module which provides diagram-independent SyncScript functionality, including a parser, runtime, and standard library.

diagram Provides the diagram DSL, and the layout engine used to create layouted diagrams.

language-server Contains the language server responsible for evaluating and layouting diagrams, handling graphical edits, and providing textual editing features.

diagram-ui The graphical editor based on Eclipse Sprotty, can only be used in a web environment.

diagram-common Provides the layouted diagram model used by both diagrams, the graphical editor, and renderers. Also provides common logic for layouting canvas elements and a basic 2D vector math library.

diagram-protocol LSP extensions for communication between the graphical editor (diagram-ui) and the language server.

diagram-render-pdf Renders layouted diagrams to pdf files.

diagram-render-svg Renders layouted diagrams to svg files.

fonts Contains Base64 encoded fonts for Roboto, Open Sans, and Source Code Pro, allowing to use of these fonts in an offline environment.

6.2 Implementation

In order to use the framework in a web-based environment without relying on server-provided capabilities, we decided to implement all modules of the framework using web technologies. As our main programming language, we chose TypeScript (TS) for good integration with the JS ecosystem, while providing strong typing for increased code quality and improved development experience. To simplify the implementation, we rely on several open-source libraries, with notable dependencies being shown in Figure 6.2. Table 6.1 gives an overview of used dependencies by module. The remainder of this section focuses on some chosen aspects of the implementation, starting with SyncScript followed by the language server, with a focus on performance improvements. Last, we describe challenges in layouting and point projection.

dependency	usage	module
chevrotain	lexer and parser for SyncScript	core
sprotty	diagramming library used for the graphical editor	diagram-ui
prettier	formatter for SyncScript implemented as plugin	langauge-server
vscode-languageserver	language server library for JS and TS	language-server
pdfkit	PDF rendering of diagrams	diagram-render-pdf
fontkit	TrueType and OpenType implementation used for text layouting	diagram
linebreak	line break opportunity identification in text layouting using Unicode Line Breaking Algorithm	diagram
bezier-js	bounding box and point projection for bezier curves	diagram-common
svgpath	SVG path simplification and transformation	diagram-common
transformation-matrix	calculation of transformation matrices as the composition of translation and rotation	diagram-common

Table 6.1: Important dependencies used by each module.

SyncScript

For implementing SyncScript, we considered two different approaches: First, by using a language workbench, in particular, *langium*[4], and second, without any language workbench and instead using underlying libraries, e.g., for parsing and formatting, directly. Primarily, a language workbench reduces development effort by providing an environment for developing the language. However, SyncScript is a highly dynamic and flexible language, thus, most functionality provided by the language workbench could either not be used or needs to be highly customized. Consequently, we decided to directly use underlying libraries, as we expect additional complexity introduced by the language workbench to outweigh the functionality we could use. For implementing the lexer and parser, *chevrotain*[5] was chosen, both for its excellent performance[6], and ease of use. Primarily, due to using a JS DSL, parser code generation was not necessary, and debugging was simplified. As chevrotain only provide a CST, we generate the AST using the visitor pattern. Compared to the CST, the AST abstracts from details required for an unambiguous grammar. However, source code location tracking is required for our hybrid concept to work. Each AST node represents an expression in the program. Location tracking allows replacing said expression in the source code while guaranteeing maintained syntactical correctness. In order to execute the program, the AST is transformed into an executable AST. Here, each expression provides an evaluation function, which recursively evaluates the expression. An executable module therefore consists of a list of expressions, which are executed in order. While user programs can be created this way, internal modules also need to provide intrinsic functions, for example for the addition of numbers. We

[4] https://langium.org/
[5] https://chevrotain.io/
[6] https://chevrotain.io/performance/

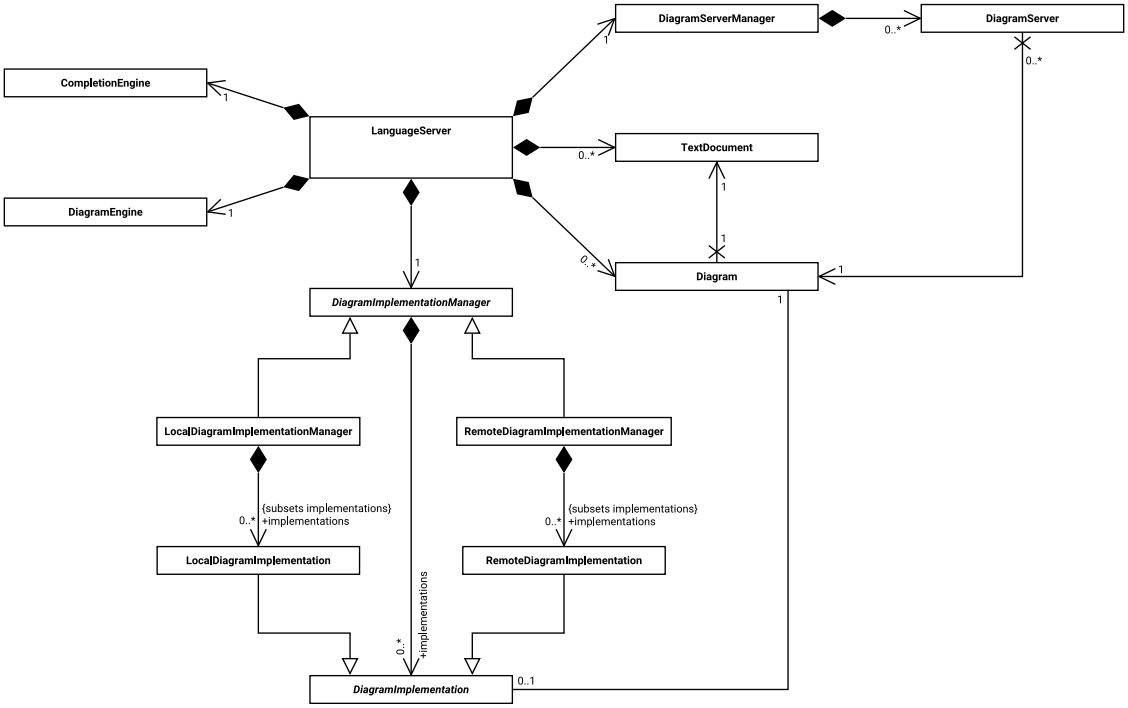

Figure 6.3: Class diagram of the HyLiMo language server.

implement those directly as executable expressions, without an underlying AST node. To simplify the creation of such expressions, we define an internal TS DSL. In particular, defining functions as TS functions is supported. At runtime, location tracking is performed by treating all values as a tuple of the actual value, and the defining AST expression. The corresponding AST expression is determined by the executable expression responsible for creating the value. Note that as not all executable expressions are based on code the user writes, a (modifiable) AST expression is not always available, resulting in some graphical changes being unavailable. Also, to improve user experience, not all expressions change the source part of the tuple. For instance, when accessing a field (`test.x`), if the field already defines a source AST node, the tuple is directly returned. As a result, in most cases, the actual original definition is modified, instead of just replacing the assignment, as shown in the following example: When modifying this absolute position defined in line 3 the variables in lines 1 and 2 are updated, instead of replacing the variables in line 3.

Listing 6.1 Creation of an absolute point using two variables.

```
1 x = 10
2 y = 10
3 apos(x, y)
```

Language Server

The language server is implemented using *vscode-languageserver*, the by far most popular language server framework for JS/ TS. In contrast to its name, the resulting language server can be used independently of *VS Code*. As shown in Figure 6.3, the language server maintains a list of TextDocuments and Diagrams, each diagram references the TextDocument it is based on. Diagrams are provided to connecting editors using the DiagramServer, each DiagramServer serves a single diagram to a single client, each Diagram can be served by any amount of DiagramServers, thus, allowing multiple clients to show the same diagram. The DiagramServerManager manages the DiagramServers and handles connecting and disconnecting clients. Latency is an important concern, especially when performing continuous graphical edits, e.g., dragging a diagram element on the canvas. Several measures were taken in order to improve performance: First, several language servers can be connected via LSP to distribute the workload. As can be seen in Figure 6.3, `Diagram` references the `DiagramImplementation` for evaluating and layouting diagrams. Two types of diagram implementations exist: local and remote ones. While `LocalDiagramImplementation` performs all actions on the primary language server, `RemoteDiagramImplementation` delegates these requests using the LSP to secondary language servers, not blocking the primary one. By connecting to secondary language servers using LSP, the developer using our framework can decide if, where, and how many secondary language servers to deploy. While this prevents the primary language server from blocking, still, latency is high, as for each update, the code needs to be executed and the diagram layouted. To decrease latency and improve framerate, we use predictions while performing graphical continuous edits. Figure 6.4 shows how this process is implemented. While performing a graphical edit, updates are repeatedly sent to the language server. For each update, a resulting diagram with the update applied, the so-called prediction, is immediately sent to the client. Simultaneously, a full reevaluation of the diagram is started asynchronously. When the full reevaluation is completed, the result is sent to the client. However, this diagram is older than the predicted updates sent in between. Therefore, to prevent "jumping" in the graphical editor due to seemingly outdated data, all updates received since the full reevaluation are also applied to the resulting diagram similar to how predictions are computed. Predicted updates are always based on the newest available full reevaluation, also, to prevent latency increases, only one full update is performed at any given time. To avoid invalid graphical states, the final update of a graphical interaction always causes a full reevaluation. However, prediction-based updates are not supported if the IDs of diagram elements change, as in this case, we cannot safely determin to which element to apply the prediction. This is, for example, the case when adding a position to an element previously not defining a position. In this case, only the results of full reevaluations are shown. To improve the editing experience in this case, we need to speed up the evaluation and layouting process. Based on tracing results, we identify layouting text as the main problematic area. Therefore, we added a cache for both layouting texts and paths.

Layouting

With the context of the HyLiMo framework, layouting is the process of generating layouted diagrams based on diagrams. While for most elements, said process is quite trivial, texts and paths prove to be challenging to layout. First, in most browser-based diagramming frameworks, including *Mermaid*, *diagrams.net*, and Eclipse Sprotty, text is layouted natively by the browser. However, as our framework should also work in a non-browser environment without requiring a headless

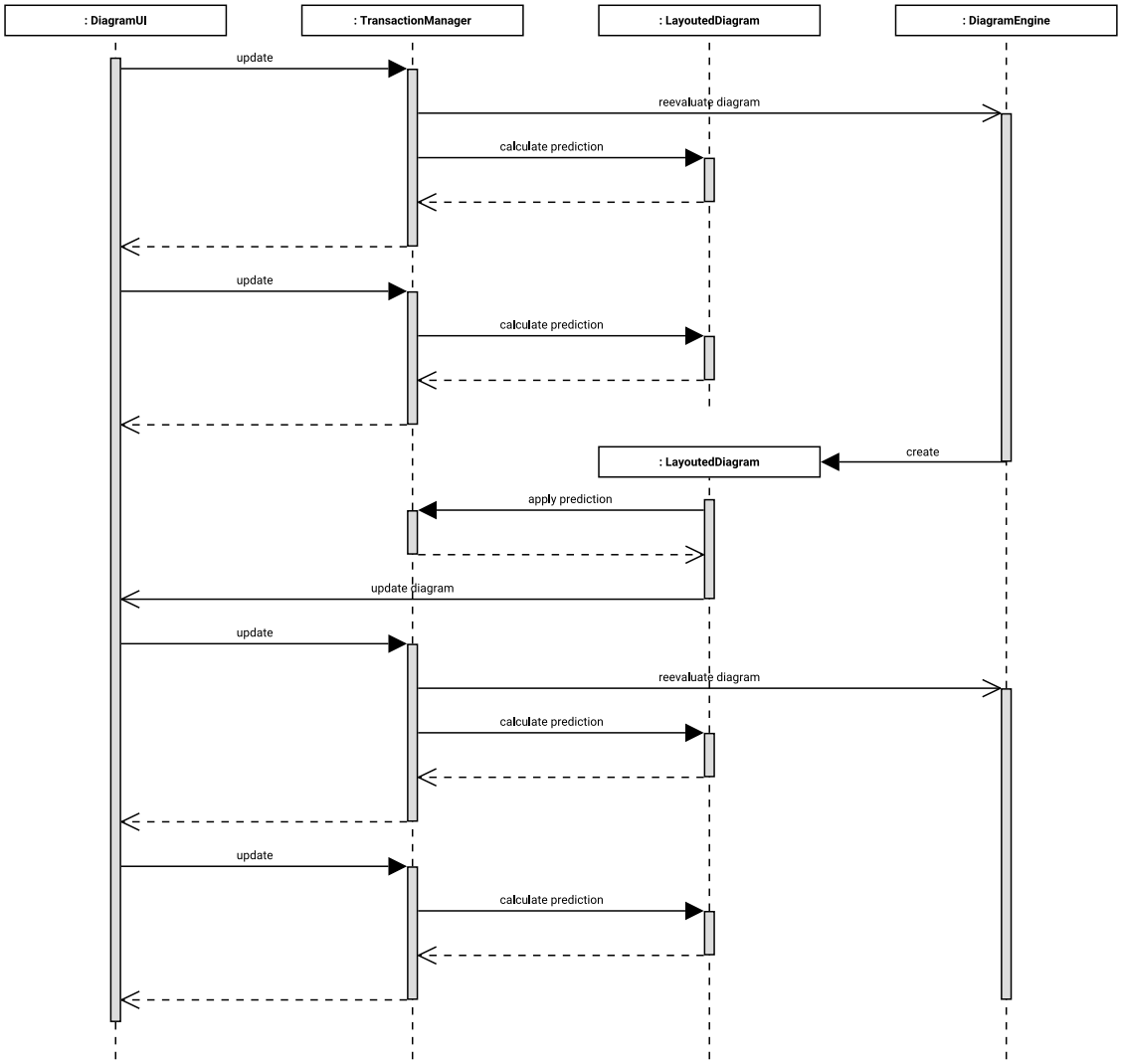

Figure 6.4: Sequence diagram of how a continuous graphical edit is handled by the language server

browser, we decided to layout text manually. As we did not succeed in finding a ready-to-use library that supports *fontkit* fonts, we implemented a custom algorithm. It layouts the text using fontkit, and determines breakpoints based on the available width. Paths are layouted using an iterative algorithm: First, an algorithm calculates the bounding box of an existing path. While libraries exist for this use case, most importantly *svg-path-bbox*, all identified libraries assume a stroke width of 0. However, with a positive stroke width, line cap, line join and miter limit also influences the dimensions of the minimal bounding box. To take these values into account, we adopted the algorithm used by svg-path-bbox to consider these metrics. Next, the path is scaled in both directions to fit the desired dimensions. Depending on the configured stretch style attribute, the path is either scaled uniformly in both directions or to fill the available space. These two steps are repeated until the fit is precise enough, or the maximum amount of iterations is exceeded. Note that an iterative algorithm is required, as scaling non-uniformly can result in a different new bounding box than expected, as the miter length of miter tips largely depends on the angle two lines meet.

Point Projection

When graphically moving points on lines, primarily canvas connections and outlines of canvas elements, we need to determine the point on the line closest to the cursor position. This process is called point projection. For straight lines, this process is trivial, for cubic bezier curves, the functionality is provided by *bezier-js*. However, elliptical curves prove to be more challenging, as the trivial approach, using the intersection point with the line connecting the point to project with the center of the ellipse results in wrong results. As a solution, we use a modified version of the Wijewickrema-Esson-Papliński method presented in [CW13], with the implementation highly inspired by their C++ version[7].

[7] https://people.cas.uab.edu/~mosya/cl/C++projections/

7 Evaluation

This chapter details how we evaluated our work. The evaluation consists of two parts: First, in Section 7.1, we evaluate if our framework meets the conceptual requirements laid out in Section 4.2.1. Following, Section 7.2 gives an overview of two case studies conducted using our web-based hybrid editor. Last, Section 7.3 concludes this section with threads to validity.

7.1 Conceptual Compilance

In Section 4.2.1, we defined nine conceptual requirements which formalize our concept or work. Following, we analyze to which extent we met those requirements.

CR 1: Hybrid editor As seen without web-based implementation, our framework supports a hybrid editor. While the graphical editor is provided by the HyLiMo framework and based on *Eclipse Sprotty*, the textual editor can be chosen based on environmental requirements and is connected using the LSP. As required, users can manipulate the diagram using both editors. However, some limitations exist: while every aspect of the diagram can be manipulated using the textual editor, currently, the graphical editor only supports changing positions, resizing canvas elements, and rotating canvas elements.

CR 2: Diagram DSL For defining the diagram, our framework provides a custom DSL. It provides general-purpose diagramming features and a higher-level DSL for UML class diagrams.

CR 3: Programming Language Features The diagram DSL is implemented as internal DSL in our custom GPL SyncScript. SyncScript is a Turing-complete general-purpose programming language, providing common programming-language features including (1) variables, (2) custom functions, (3) multiple data types (strings, numbers, objects), and (4) a standard library and module system.

CR 4: Code as Single Source of Truth When defining diagrams using our framework, the diagram is defined using a textual concrete syntax. While a graphical editor as part of the hybrid editor concept exists, graphical edits are persisted as textual changes. Furthermore, all information is contained in a single file. Yet, diagrams may have external dependencies. First, fonts are defined as Uniform Resource Locator (URL), to support external font files. To allow defining diagrams not requiring an internet connection, our framework provides a set of default fonts included as data URLs. Next, external information can be injected into the global scope before execution of the DSL code. Currently, this is used for theming, allowing injecting a `theme` variable.

CR 5: Live-Synced Editing While performing graphical edits, the textual definition is continuously updated. Also, textual edits immediately cause a rerendering of the diagram, resulting in a live-synced experience. However, for technical reasons, both processes are not immediate, depending on the diagram size and complexity, we observe delays of up to one second for reasonably sized diagrams. While the framework provides mechanisms mitigating these delays described in Section 6.2, those are not applicable in all situations.

CR 6: Manual Layouting Top-level diagram elements, e.g., UML classes, are typically created using canvas elements on a canvas. The positions of canvas elements and connections between them are defined using absolute and relative points, allowing for manual layouting. In particular, to graph-based auto-layout algorithm is used when layouting a diagram. While our framework provides some elements which allow layouting their contents, including horizontal and vertical stacks (vbox and hbox), these are intended for creating and layouting the contents of diagram elements, e.g., the fields of a UML class.

CR 7: Precise Layouting Positions are defined as either absolute or relative using 64-bit floating point numbers, allowing for precise layouting.

CR 8: Styling Each element supports a set of style attributes affecting its style. Depending on the type of element, different attributes are available, e.g., while a span supports fontSize and fontFamily, a rect does not. Style attributes can either be set directly on the element or using SCSS-inspired styles which set attributes on all elements matching some selectors. These functionalities allow users to affect the style of both single elements, and the whole diagram.

CR 9: Modular Extensible Framework Diagram types are defined using SyncScript modules. To allow using multiple diagram type modules at once, diagrams should always be defined in a trailing lambda function call, to define the used diagram type. Currently, we only provide a UML class diagram module. Also, while our framework supports providing external diagram type modules, our web-based editor currently does not due to security concerns.

Summarizing these results, the HyLiMo framework completely fulfills conceptual requirements 2, 3, 6, 7, and 8. Requirements 1, 5, and 9 are fulfilled to a large extent, however, future work is needed is required to further improve the provided functionality. Similarly, requirement 4 is fulfilled to a large extent. However, we realized that some external dependencies, in particular theming support, can significantly improve user experience, and thus decided against completely omitting any external dependencies.

7.2 Case Studies

While Section 7.1 shows that implementing a framework according to our concept is viable, we still need to prove that using the framework, in particular the web-based editor, results in an acceptable diagramming experience in real-life use cases. Therefore, we conducted two case studies. As at the point of the case studies publicly available documentation was spare, we provided both users with an example class diagram and answered resulting questions on how to realize graphical features. Before starting to create the diagrams, both users were asked to share information on problems, feature requests, and general information about which aspects of the framework and editor work well and which do not.

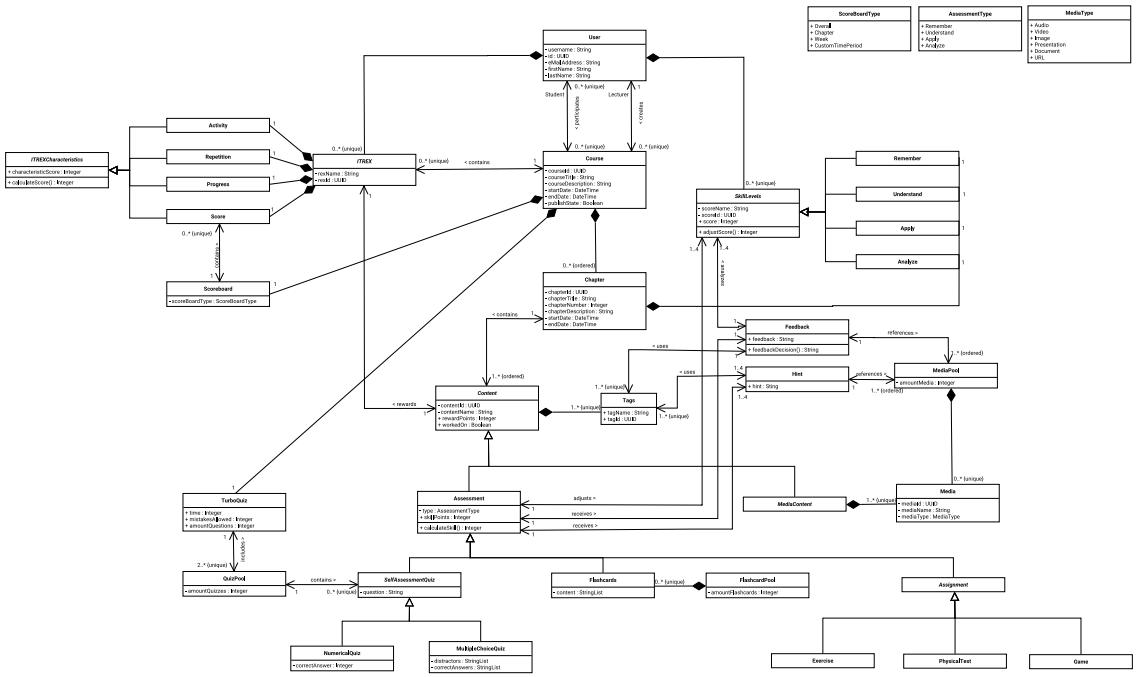

Figure 7.1: Case study: new domain model for IT-REX.

The first case study involves the creation of a new and improved domain model for *IT-REX* [MSB23; SBB+22] by a doctoral researcher experienced in several diagramming and modeling tools, which is shown in Figure 7.1. The diagram is defined by a 607 line long textual definition and includes 34 classes and three enumerations. Used features include (1) classes, including properties and operations, (2) keywords on classes for enumerations, (3) generalizations, (4) compositions, (5) associations with various navigatabilities, (6) labels on associations, (7) line and axis-aligned connection segments, and (8) absolute and relative positioning. In particular, the researcher used lists and loops to equally space elements and create connections that are joined at some point in the diagram. Apart from marking classes as abstract, neither styling nor theming was used. However, layouting, in particular using graphical edits, was extensively used. Received feedback primarily consisted of feature requests:

(1) Marking classes as abstract

(2) Download the diagram definition when entering Ctrl + S

(3) Saving diagrams to GitHub

(4) Documentation

(5) Making text in the pdf export searchable

Due to being low-effort features, we implemented the first two requested features immediately upon request.

Our second case study involves an undergraduate student in the field of software engineering creating two UML class diagrams as part of his Bachelor's thesis on an improved issue synchronization for *Gropius* [SBB20]: The first diagram is shown in Figure 7.2 and gives an overview of the

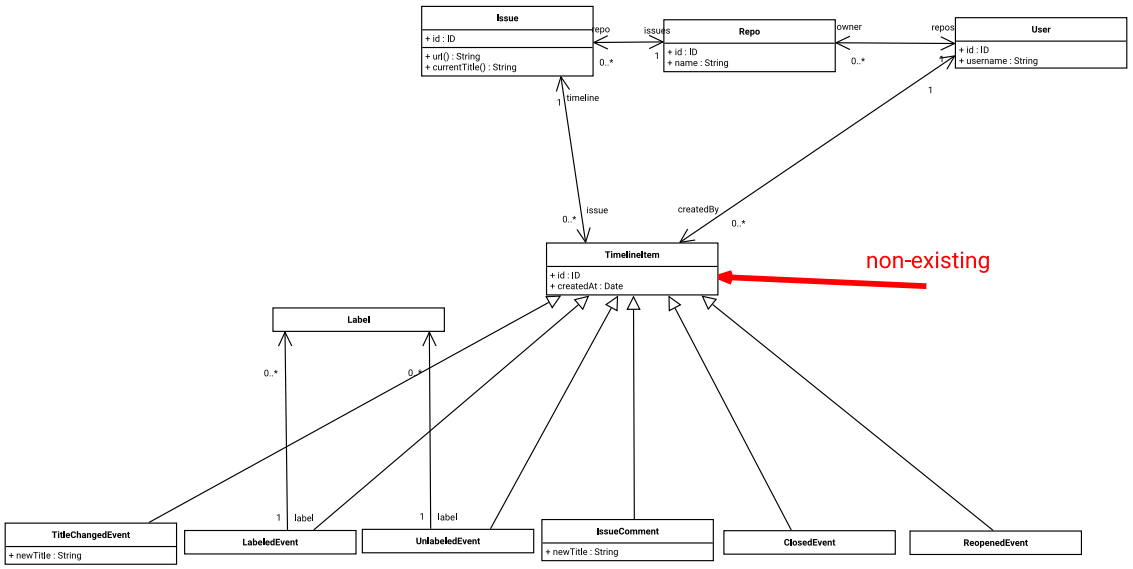

Figure 7.2: Case study: class diagram of a subset of GitHub's GraphQL API.

shortcomings of GitHub's GraphQL support for issues, e.g., bug reports. Second, a diagram for the current domain model of Gropius, an improved version of the domain model presented at [SBB21], to cover newer requirements as described by Speth et al. [SBK+23], was created. Overall, the used features are similar to our first case study, with the addition of styling which was used for the arrow and text shown in Figure 7.2. While the first diagram was manually created, the second diagram was partially generated based on the GraphQL API of Gropius. While the content of the diagram, including classes and their relations, was generated, layouting was performed manually and graphically. As our framework allows for separating content and layout information, the content of the diagram could be regenerated and inserted without losing layouting information. Similar to the first case study, received feedback primarily consisted of feature requests, including:

(1) Snapping **(RE IV.10)**

(2) Rounding numbers inserted by graphical edits

(3) Labels with absolute distance to arrowhead

(4) Modifier key for moving elements only on x-axis/y-axis

(5) Copy&Paste of classes **(RE IV.16)**

Summarizing these results, we show that our framework, in particular the web-based editor, can be used to create UML class diagrams. However, future work is required for improving existing, and implementing additional features.

7.3 Threads to Validity

In this section, we describe threads to validity. According to Runeson et al. [RH09, pp. 153-154], four aspects of validity can be distinguished: construct, internal, and external validity, and reliability.

First, regarding external validity, our modularity concept may be insufficient for other diagram types. As part of this thesis, we showed how UML class diagrams can be supported. However, while these results likely transfer to similar diagram types, e.g., UML object diagrams, this might not be the case for non-graph-based diagram types, in particular in the area of interactive graphical features. Also, our modularity concept might not support more complex graphical interactions, in particular creating new diagram elements, and diagram type-specific interactions. For collecting initial requirements, we conducted expert interviews. As both experts have an academic background, as either current or former researchers, there is a risk that the collected requirements do not generalize to other groups of users who might use our framework.

Furthermore, experts were interviewed consecutively. While we tried to keep interviews as independent and comparable as possible, in particular by using the same demonstration and not incorporating newly developed features, there is an inherent risk that previous interviews affected the interview process, posing a threat to construct validity.

Considering internal validity, there is the risk we overlooked important aspects of hybrid diagramming. Next, due to time constraints, we conducted our case studies while the framework was still in development. Thus, some features, in particular navigating to source, were not available to the participating users. Also, due to missing documentation, we provided information, help, and example diagrams to participants. Without these, users might not have been able to create the diagrams or use our framework. In particular, we cannot guarantee that existing documentation would have been sufficient to use the framework and editor.

8 Conclusion

This section concludes this thesis, by first summarizing it in Section 8.1. Following, Section 8.2 goes over its main benefits, while Section 8.3 highlights its limitations. Last, we outline future work in Section 8.4.

8.1 Summary

The main objective of this thesis was the implementation of a framework for modular hybrid diagramming. In order to do so, we first collected the requirements of such a framework. Initial requirements were identified based on related work, and a rudimentary version of our hybrid editor, showcasing the concept of hybrid textual-graphical manipulations, was created. Based on these, we performed expert interviews to collect further requirements and design questions. To evaluate the collected requirements and decide on design questions, we conducted a survey with the same experts. Based on the collected requirements, we created a detailed concept of our modular hybrid diagramming framework. Within this concept, a textual diagram DSL is used to primarily define diagrams. We decided to implement our diagram DSL as internal DSL in our custom-designed general-purpose programming language SyncScript, allowing for the use of programming language constructs when defining diagrams. SyncScript is designed for maximum flexibility, in particular implementing internal DSLs, but also for syntactic simplicity to support our hybrid concept. This DSL allows creating diagrams, which can be layouted and displayed in the graphical editor. To increase flexibility, the framework also provides styling support. The graphical editor allows, manipulating the textual definition using graphical interactions. As part of this thesis, moving, scaling, and rotating diagram elements on the canvas are supported. To create a modular framework, different diagram types are supported using DSL modules, using a common lower-level graphics framework. Currently, a module to define UML class diagrams exists. We implemented our concept as a JS library, which can be found on GitHub[1]. By using the *Language Server Protocol (LSP)*, we ensured supporting different environments and created a web-based version of our editor[2]. We evaluated this editor with two case studies, here, two participants created multiple UML class diagrams. This evaluation showed that our framework allows for hybrid diagramming for UML class diagrams, however, it also showed future work is required to further improve graphical editing features. Thus, regarding **RQ 1**, we can conclude that implementing a framework for modular live-synced hybrid diagramming is technically feasible. The case studies also showed that programming features, most important `for`-loops, help with layouting elements manually and

[1] https://github.com/hylimo/hylimo
[2] https://hylimo.github.io

precisely (**RQ 2**), in particular, when joining multiple associations. As we implemented the module for UML class diagrams as internal DSL using the lower-level diagramming modules, we showed that such features indeed improve extendability, answering **RQ 3**.

8.2 Benefits

Primarily, our framework brings the benefits of hybrid/blended modeling to diagramming. Compared to existing diagramming frameworks, this approach has multiple benefits: First, compared to purely graphical frameworks, the textual DSL allows for a more efficient definition of diagram elements. Also, programming language features improve extensibility and allow automation of repetitious tasks, e.g., by defining custom functions and variables. Compared to textual tools with auto-layout, manual layouting gives the user more flexibility and control over the appearance of the created diagram, which is required for some use cases, e.g., scientific publications. Compared to textual tools with manually defined positions, the hybrid approach allows intuitively manipulating positions, dimensions, and rotations graphically. Furthermore, the graphical view can be used to navigate the diagram more efficiently by jumping to the corresponding position in the textual definition. Last, our framework supports both browser and console environments, allowing for improved tool support in the future. To summarize, while none of the features provided by our framework is particularly new, the hybrid combination of textual and graphical editing is novel in the area of diagramming and allows for an improved diagramming experience, in particular, if manual layouting is required. As a result, main user groups profiting from these benefits include researchers, when creating diagrams for publications, and developers/architects, when creating diagrams for documentation. These users profit both from expedited creation of diagrams and improved maintainability due to the use of a textual concrete syntax.

8.3 Limitations

Three major limitations exist: First, currently, only the web-based editor is implemented. Other environments, including desktop, IDE and CLI support are currently not implemented, forcing users to use the web-based version. Second, only a subset of identified graphic editing features is implemented. In particular, no editing features involving creating new elements, or diagram type-specific editing features are implemented. Last, only UML class diagrams are currently supported. For all other diagram types, users either need to use the lower-level graphic feature, potentially implementing some DSL functionality themselves. Also, several minor limitations exist, including no import/export functionality, and missing text search in generated PDFs.

8.4 Future Work

Multiple areas for future work exist. First, the framework needs to be improved, so that the not yet implemented requirements will be supported. In particular, more sophisticated graphical editing features should be supported. This includes modifying connections, editing text and styles, copying&pasting elements, and creating new elements. Also, additional diagram types, including

non-graph-based diagram types could be implemented. Next, all identified environments should be supported. With these changes, a full evaluation can be performed. Such an evaluation could compare the concept of hybrid diagramming with other approaches, with a focus on diagramming efficiency, and maintainability of created diagrams to answer **RQ 4**, which has not been evaluated as part of this thesis due to missing functionality in the framework. Collaboration features are another interesting aspect. As a textual definition is used as a single source of truth, collaborative diagramming could be supported by synchronizing the textual definition. However, simultaneous graphical edits will impose challenges that need to be solved in future work. Another area of interest lies in seamless integration into existing online platforms, e.g., GitHub. Such integrations may be possible via browser extensions, or using extension features provided by the platform.

Bibliography

[ACLP17] L. Addazi, F. Ciccozzi, P. Langer, E. Posse. "Towards Seamless Hybrid Graphical–Textual Modelling for UML and Profiles". In: *Modelling Foundations and Applications*. Ed. by A. Anjorin, H. Espinoza. Cham: Springer International Publishing, 2017, pp. 20–33. ISBN: 978-3-319-61482-3 (cit. on pp. 16, 17).

[ATB03] Auer, Tschurtschenthaler, Biffl. "A flyweight UML modelling tool for software development in heterogeneous environments". In: *2003 Proceedings 29th Euromicro Conference*. 2003, pp. 267–272. DOI: 10.1109/EURMIC.2003.1231600 (cit. on pp. 1, 18).

[BD90] B. Barsky, T. DeRose. "Geometric continuity of parametric curves: constructions of geometrically continuous splines". In: *IEEE Computer Graphics and Applications* 10.1 (1990), pp. 60–68. DOI: 10.1109/38.45811 (cit. on p. 43).

[BVJ+16] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, J. Siegmund. "Efficiency of Projectional Editing: A Controlled Experiment". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 763–774. ISBN: 9781450342186. DOI: 10.1145/2950290.2950315 (cit. on p. 12).

[CBR+17] S. Cook, C. Bock, P. Rivett, T. Rutt, E. Seidewitz, B. Selic, D. Tolbert. *Unified Modeling Language (UML) Version 2.5.1*. Standard. Object Management Group (OMG), Dec. 2017. URL: https://www.omg.org/spec/UML/2.5.1 (cit. on pp. 6–8, 66).

[CK19] J. Cooper, D. Kolovos. "Engineering Hybrid Graphical-Textual Languages with Sirius and Xtext: Requirements and Challenges". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019, pp. 322–325. DOI: 10.1109/MODELS-C.2019.00050 (cit. on p. 16).

[CTVW19] F. Ciccozzi, M. Tichy, H. Vangheluwe, D. Weyns. "Blended Modelling - What, Why and How". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019, pp. 425–430. DOI: 10.1109/MODELS-C.2019.00068 (cit. on pp. 1, 15).

[CW13] N. Chernov, S. Wijewickrema. "Algorithms for projecting points onto conics". In: *Journal of Computational and Applied Mathematics* 251 (2013), pp. 8–21 (cit. on p. 82).

[DLP+22] I. David, M. Latifaj, J. Pietron, W. Zhang, F. Ciccozzi, I. Malavolta, A. Raschke, J.-P. Steghöfer, R. Hebig. "Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study". In: *Software and Systems Modeling* (2022), pp. 1–33 (cit. on pp. 15, 16).

[DVMV17] I. Dejanović, R. Vaderna, G. Milosavljević, Ž. Vuković. "TextX: A Python tool for Domain-Specific Languages implementation". In: *Knowledge-Based Systems* 115 (2017), pp. 1–4. ISSN: 0950-7051. DOI: 10.1016/j.knosys.2016.10.023. URL: http://www.sciencedirect.com/science/article/pii/S0950705116304178 (cit. on p. 11).

[ESV+13] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, J. van der Woning. "The State of the Art in Language Workbenches". In: *Software Language Engineering*. Ed. by M. Erwig, R. F. Paige, E. Van Wyk. Cham: Springer International Publishing, 2013, pp. 197–217. ISBN: 978-3-319-02654-1 (cit. on pp. 11, 51).

[Fow10] M. Fowler. *Domain-specific languages*. Pearson Education, 2010 (cit. on p. 10).

[GB21] P.-L. Glaser, D. 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)*. IEEE. 2021, pp. 337–340 (cit. on p. 16).

[GKR+08] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel. "Monticore: a framework for the development of textual domain-specific languages". In: *Companion of the 30th international conference on Software engineering*. 2008, pp. 925–926 (cit. on p. 11).

[Gla22] P.-L. Glaser. *Developing Sprotty-based Modeling Tools for VS Code*. 2022 (cit. on pp. 16, 17, 76).

[GW14] J. Gibbons, N. Wu. "Folding domain-specific languages: deep and shallow embeddings (functional pearl)". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 2014, pp. 339–347 (cit. on p. 12).

[HKR21] K. Hölldobler, O. Kautz, B. Rumpe. *MontiCore Language Workbench and Library Handbook*. 2021 (cit. on p. 12).

[Lim] S. Limited. *Diagramming vs modelling*. URL: https://structurizr.com/help/modelling (cit. on p. 5).

[MHS05] M. Mernik, J. Heering, A. M. Sloane. "When and how to develop domain-specific languages". In: *ACM computing surveys (CSUR)* 37.4 (2005), pp. 316–344 (cit. on pp. 10, 13).

[Mog22] T. Æ. Mogensen. *Programming Language Design and Implementation*. Springer Nature, 2022 (cit. on pp. 9, 10).

[MSB23] N. Meissner, S. Speth, U. Breitenbücher. "An Intelligent Tutoring System Concept for a Gamified e-Learning Platform for Higher Computer Science Education". In: *Proceedings der SEUH 2023*. Gesellschaft für Informatik, Bonn, 2023, pp. 105–111 (cit. on p. 85).

[Ozk19] M. Ozkaya. "Are the UML modelling tools powerful enough for practitioners? A literature review". In: *IET Software* 13.5 (2019), pp. 338–354. DOI: https://doi.org/10.1049/iet-sen.2018.5409. eprint: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-sen.2018.5409. URL: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-sen.2018.5409 (cit. on p. 17).

[Por13] R. Porst. *Fragebogen: Ein Arbeitsbuch*. Springer, 2013 (cit. on p. 33).

[RH09] P. Runeson, M. Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical software engineering* 14 (2009), pp. 131–164 (cit. on p. 86).

[SBB+22] S. Speth, S. Becker, U. Breitenbücher, P. Fuchs, N. Meißner, A. Riesch, D. Wetzel. "IT-REX — A Vision for a Gamified e-Learning Platform for the First Semesters of Computer Science Courses". In: *Software Engineering im Unterricht der Hochschulen (SEUH 2022)*. Ed. by V. Thurner, B. Kleinen, J. Siegeris, D. Weber-Wulff. Gesellschaft für Informatik, Bonn, 2022, pp. 43–48. DOI: 10.18420/SEUH2022_05 (cit. on p. 85).

[SBB20] S. Speth, U. Breitenbücher, S. Becker. "Gropius — A Tool for Managing Cross-component Issues". In: *Software Architecture*. Ed. by H. Muccini, P. Avgeriou, B. Buhnova, J. Camara, M. Caporuscio, M. Franzago, A. Koziolek, P. Scandurra, C. Trubiani, D. Weyns, U. Zdun. Cham: Springer International Publishing, 2020, pp. 82–94. ISBN: 978-3-030-59155-7 (cit. on p. 85).

[SBB21] S. Speth, S. Becker, U. Breitenbücher. "Cross-Component Issue Metamodel and Modelling Language." In: *CLOSER*. 2021, pp. 304–311 (cit. on p. 86).

[SBK+23] S. Speth, U. Breitenbücher, N. Krieger, P. Wippermann, S. Becker. "Integrating Issue Management Systems of Independently Developed Software Components". In: *Proceedings of 24th International Conference on Agile Software Development (XP23)*. Springer, 2023 (cit. on p. 86).

[TO] TypeFox, Obeo. *Xtext / Sirius - Integration*. URL: https://www.obeodesigner.com/resource/white-paper/WhitePaper_XtextSirius_EN.pdf (cit. on p. 16).

[VP12] M. Voelter, V. Pech. "Language modularity with the MPS language workbench". In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012, pp. 1449–1450. DOI: 10.1109/ICSE.2012.6227070 (cit. on p. 12).

[WB23] A. Wąsowski, T. Berger. *Domain-specific Languages: Effective Modeling, Automation, and Reuse*. Springer Nature, 2023 (cit. on pp. 11–13).

[WDJ22] L. Walsh, J. Dingel, K. Jahed. "A General Architecture for Client-Agnostic Hybrid Model Editors as a Service". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '22. Montreal, Quebec, Canada: Association for Computing Machinery, 2022, pp. 749–754. ISBN: 9781450394673. DOI: 10.1145/3550356.3563131 (cit. on pp. 16, 76).

All links were last followed on May 10, 2023.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Ohmden, 10.05.2023 Krieger

place, date, signature