A Modular and Configurable Web-Based Frontend Architecture for Educational Modelling Tools

Chaitanya Santosh Tekane

School of Computer Science

McGill University, Montreal, Quebec, Canada

March 18, 2025

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Science

Abstract

Modern software systems are becoming increasingly complex, necessitating structured approaches for design, analysis and reasoning at higher levels of abstraction. Modelling plays a critical role in software engineering, enabling visualization, validation and refinement of system structures and behaviors before implementation. Despite its importance, teaching modelling effectively remains a challenge due to the lack of suitable modelling tools. Existing academic tools are often outdated, unstable or difficult to install, while industrial tools tend to be prohibitively expensive and overly complex for educational use. This gap hinders students from developing practical modelling skills, limiting their ability to apply modelling techniques effectively in software development.

This thesis aims to address these challenges by developing a modern, web-based frontend architecture for modelling tools tailored to education. The objective is to provide a tool that is easily accessible, lightweight and extensible, removing installation barriers and supporting multiple modelling notations. Additionally, the system introduces configurable perspectives, allowing educators to tailor modelling environments based on student's expertise and learning objectives. The core research question explored in this work is: *"How can a modular frontend architecture enable efficient, extensible and*

customizable modelling tools for teaching?"

To achieve this, the research follows a structured methodology. First, a comprehensive set of requirements for modelling tools in education is established, considering usability, adaptability and extensibility. Based on these requirements, a modular frontend architecture is designed using JavaScript, GoJS, WebSockets and a DSL-driven configuration system. The architecture ensures that modelling components are reusable and independent, enabling seamless adaptation for various modelling notations such as class diagrams and state diagrams. The DSL-based configuration allows educators to define modelling concepts and dynamically adjust the available operations based on different pedagogical needs. The system is implemented as a fully web-based tool, ensuring cross-platform accessibility without installation overhead.

The effectiveness of the proposed system is evaluated through a structured validation process, measuring usability, response time, real-time collaboration, modularity and adaptability. Automated testing, runtime behavior analysis and performance benchmarking confirm that the system provides low-latency interactions, seamless real-time synchronization and effective adaptation to different modelling perspectives. The results demonstrate that the tool successfully meets the defined educational requirements, providing a scalable and configurable modelling environment.

Abrégé

Les systèmes logiciels modernes deviennent de plus en plus complexes, nécessitant des approches structurées pour la conception, l'analyse et le raisonnement à des niveaux d'abstraction plus élevés. La modélisation joue un rôle essentiel dans le génie logiciel, permettant la visualisation, la validation et l'affinement des structures et des comportements du système avant la mise en œuvre. Malgré son importance, l'enseignement efficace de la modélisation reste un défi en raison du manque d'outils de modélisation adaptés. Les outils académiques existants sont souvent obsolètes, instables ou difficiles à installer, tandis que les outils industriels ont tendance à être prohibitifs et trop complexes pour un usage pédagogique. Cette lacune empêche les étudiants de développer des compétences pratiques en modélisation, limitant ainsi leur capacité à appliquer efficacement les techniques de modélisation au développement de logiciels.

Cette thèse vise à relever ces défis en développant une architecture frontale moderne basée sur le Web pour des outils de modélisation adaptés à l'éducation. L'objectif est de fournir un outil facilement accessible, léger et extensible, supprimant les barrières d'installation et prenant en charge plusieurs notations de modélisation. De plus, le système introduit des perspectives configurables, permettant aux enseignants d'adapter les

environnements de modélisation en fonction de l'expertise et des objectifs d'apprentissage de l'étudiant. La question de recherche centrale explorée dans ce travail est la suivante: *"Comment une architecture frontale modulaire peut-elle permettre des outils de modélisation efficaces, extensibles et personnalisables pour l'enseignement?"*

Pour y parvenir, la recherche suit une méthodologie structurée. Premièrement, un ensemble complet d'exigences pour les outils de modélisation dans l'éducation est établi, prenant en compte la convivialité, l'adaptabilité et l'extensibilité. Sur la base de ces exigences, une architecture frontale modulaire est conçue à l'aide de JavaScript, GoJS, WebSockets et d'un système de configuration piloté par DSL. L'architecture garantit que les composants de modélisation sont réutilisables et indépendants, permettant une adaptation transparente à diverses notations de modélisation telles que les diagrammes de classes et les diagrammes d'état. La configuration basée sur DSL permet aux enseignants de définir des concepts de modélisation et d'ajuster dynamiquement les opérations disponibles en fonction de différents besoins pédagogiques. Le système est implémenté comme un outil entièrement basé sur le Web, garantissant une accessibilité multiplateforme sans frais d'installation.

L'efficacité du système proposé est évaluée à travers un processus de validation structuré, mesurant la convivialité, le temps de réponse, la collaboration en temps réel, la modularité et l'adaptabilité. Les tests automatisés, l'analyse du comportement d'exécution et l'analyse comparative des performances confirment que le système offre des interactions à faible latence, une synchronisation transparente en temps réel et une adaptation efficace aux différentes perspectives de modélisation. Les résultats démontrent que l'outil répond

avec succès aux exigences pédagogiques définies, en fournissant un environnement de modélisation évolutif et configurable.

Contribution

All chapters of this thesis were researched and written by the student, Chaitanya Santosh Tekane, as a first author. Professor Jörg Kienzle served in a supervisory role, offering conceptual direction, reviewing draft and ensuring the academic rigor of the thesis.

Acknowledgements

First and foremost, I express my deepest gratitude to my supervisor, Jörg Kienzle, for his unwavering support, patience and invaluable guidance throughout my master's thesis. His vast research experience, insightful feedback and encouragement have been instrumental in shaping my work. I am also grateful for his pragmatism and for introducing me to a vibrant research community. Since his departure from McGill, I am grateful to Bettina Kemme for taking over the official supervision and handling the administrative process.

I extend my sincere thanks to Maximilian Schiedermeier, who first introduced me to RESTful APIs and provided valuable insights that supported me throughout my research. I am also thankful to Gunter Mussbacher for his course on Advanced Software Language Engineering and Martin Robillard for his course on Software Privacy, both of which significantly contributed to my understanding of core software engineering concepts. Additionally, I appreciate the contributions of Suraj Van Verma, who worked on the backend and provided essential support.

I would also like to thank Billy Exarhakos for his collaboration on the ECSE539 project, which helped shape Chapter 5 of my thesis. Furthermore, I am grateful to Hyacinth Ali,

who generously shared his time and expertise, providing crucial explanations since we built upon some of his previous work for this project.

Beyond academic and professional support, I am deeply indebted to my family for their unconditional love, encouragement and belief in me. Words cannot capture my gratitude, I simply know I can always count on you.

I would also like to extend my heartfelt appreciation to Aastha, Abhishek, Amit, Aniket, Ankur, Anshita, Bala, Darpan, Dheeraj, Gaurav, Gazal, Hari, Jeffrey, Kevin, Kunal, Lakshya, Manas, Mohaddeseh, Mudit, Nanda, Nikhil, Pavithran, Ravindranath, Rayisha, Rishabh, Romain, Sanket, Shamanth, Shree, Shreya, Siba, Srishti, Vaishnavi, Vishal and Yash. At some point in this journey, each of you has been part of meaningful conversations, shared moments of laughter or offered words of encouragement that kept me going. Your presence has made this experience all the more memorable. Special thanks to my lab mates Aayush, Nelson, Rahma, Shubham and Shwetali for making the research journey enjoyable.

Thank you all.

Contents

List of Figures

List of Tables

List of Code Samples

List of Abbreviations

List of abbreviations used in this thesis:

- **API:** ***A****pplication* ***P****rogramming* ***I****nterface*
- **AOM:** ***A****spect* ***O****riented* ***M****odeling*
- **ACM:** ***A****ssociation for* ***C****omputing* ***M****achinery*
- **AJAX:** ***A****synchronous* ***J****avaScript and* ***X****ML*
- **AR/VR:** ***A****ugmented and* ***V****irtual* ***R****eality*
- **CSS:** ***C****ascading* ***S****tyle* ***S****heets*
- **CORE:** ***C****oncern* ***O****riented* ***R****euse*
- **CI/CD:** ***C****ontinuous* ***I****ntegration and* ***D****eployment*
- **DOM:** ***D****ocument* ***O****bject* ***M****odel*
- **DSL:** ***D****omain* ***S****pecific* ***L****anguage*
- **DSML:** ***D****omain* ***S****pecific* ***M****odeling* ***L****anguage*
- **EMF:** ***E****clipse* ***M****odeling* ***F****ramework*
- **XML:** ***E****xtensible* ***M****arkup* ***L****anguage*
- **FRs:** ***F****rontend* ***R****equirements*
- **GME:** ***G****eneric* ***M****odeling* ***E****nvironment*
- **GSLP:** ***G****raphical* ***L****anguage* ***S****erver* ***P****rotocol*

- **HTML:** ***H****ypertext* ***M****arkup* ***L****anguage*
- **HTTP:** ***H****ypertext* ***T****ransfer* ***P****rotocol*
- **IDEs:** ***I****ntegrated* ***D****evelopment* ***E****nvironments*
- **JSON:** ***J****ava****S****cript* ***O****bject* ***N****otation*
- **JSON-RPC:** ***J****ava****S****cript* ***O****bject* ***N****otation-****R****emote* ***P****rocedure* ***C****all*
- **JSX:** ***J****ava****S****cript* ***X****ML*
- **LSP:** ***L****anguage* ***S****erver* ***P****rotocol*
- **MDA:** ***M****odel* ***D****riven* ***A****rchitecture*
- **MDE:** ***M****odel-****D****riven* ***E****ngineering*
- **M2M:** ***M****odel-to-****M****odel* ***T****ransformation*
- **M2T:** ***M****odel-to-****T****ext* ***T****ransformation*
- **MTT:** ***M****odeling* ***T****ools for* ***T****eaching*
- **NLP:** ***N****atural* ***L****anguage* ***P****rocessing*
- **OCL:** ***O****bject* ***C****onstraint* ***L****anguage*
- **REST:** ***R****epresentational* ***S****tate* ***T****ransfer*
- **Rs:** ***R****equirements*
- **SoC:** ***S****eparation of* ***C****oncerns*
- **SSE:** ***S****erver* ***S****ent* ***E****vents*
- **SOAP:** ***S****imple* ***O****bject* ***A****ccess* ***P****rotocol*
- **SPLE:** ***S****oftware* ***P****roduct* ***L****ine* ***E****ngineering*
- **USE:** ***U****ML-Based* ***S****pecification* ***E****nvironment*
- **UML:** ***U****nified* ***M****odeling* ***L****anguage*
- **UI:** ***U****ser* ***I****nterface*

Chapter 1

Introduction

The increasing complexity of modern software systems necessitates structured approaches to design, analysis and reasoning at higher levels of abstraction. Software engineers must understand and manage intricate system behaviors, interactions and constraints, which cannot be efficiently handled through code alone. *Modelling* provides a structured representation of software systems, enabling engineers to reason about the problem domain, architectural choices and design decisions before implementation.

The Association for Computing Machinery (ACM) curriculum recognizes modelling as a fundamental competency in software engineering education, emphasizing the need for students to develop proficiency in creating, analyzing and refining software models [1, 2]. As modern software development increasingly incorporates model-driven engineering (MDE), students must learn not just theoretical modelling concepts but also how to apply them effectively using appropriate tools.

Several industries have already embraced modelling-based development techniques to manage system complexity. For example, Unreal Engine's *Blueprint* scripting language

provides a graph-based programming approach that allows game developers to visually define logic without writing code, demonstrating how modelling can simplify software design [3]. Similarly, *Simulink* is widely used for graphical modelling of control systems and simulations [4], while Unified Modelling Language (UML)-based modelling tools facilitate code generation and system validation in software engineering. These examples highlight how modelling is no longer just a documentation technique but an integral part of modern software development workflows.

Effective modelling tools are essential for both teaching and practice. The role of modelling tools in education is similar to that of integrated development environments (IDEs) in programming. Just as students do not learn programming by writing code in plain text editors, they should not be expected to learn modelling solely through manual drawing tools. A robust modelling tool enforces syntactic and semantic correctness, preventing students from making invalid constructs. Additionally, modelling tools provide validation, simulation and model execution, which enable deeper understanding and practical application. Without these capabilities, models remain static, disconnected diagrams rather than functional representations of software systems.

However, existing modelling tools pose significant challenges, particularly for educational use. Academic modelling tools, while designed with learning in mind, often suffer from technical limitations such as instability, outdated technology stacks and difficult installation procedures [5]. Many of these tools have been developed as research prototypes rather than fully supported, long-term solutions, leading to maintenance and usability issues. On the other hand, industrial modelling tools, such as enterprise-grade UML tools, tend to be highly complex, expensive and domain-specific, making them unsuitable for introductory modelling courses. As highlighted in discussions that were during the MODELS 2023 and

2024 workshops on Modelling Tools for Teaching (MTT) [5] [6] [7], there is a pressing need for modern, intuitive modelling tools that facilitate teaching while maintaining usability and extensibility.

This thesis addresses these challenges by proposing a modern frontend architecture for modelling tools specifically designed for education. The goal is to develop a web-based modelling tool that runs entirely in the browser, ensuring ease of access without requiring installation and allowing seamless use across different platforms. The architecture will support multiple modelling notations, enabling educators to choose their modelling language based on course objectives. Furthermore, the system will provide customizable modelling languages, allowing teachers to adapt the complexity of a modelling language to the expertise of the students and the teaching objective. By adopting a reusable component-based design, the proposed approach will not only support current modelling needs, but also facilitate future extensions and adaptations, ensuring long-term viability as an educational modelling tool.

1.1 Summary of Contributions

This thesis focuses on the frontend of this modelling tool architecture. Concretely, the thesis makes the following contributions:

1. A comprehensive list of requirements for a frontend architecture for Modelling Tools for Teaching (MTTs), ensuring usability, adaptability and extensibility.
2. The design of a modular and scalable frontend architecture that integrates modern web technologies, enabling the development of intuitive and lightweight modelling tools.
3. Development of reusable frontend components that facilitate the creation of new

modelling editors for various modelling languages without significant reimplementation.

4. Support for customizing modelling tools based on different levels of student expertise and various modelling purposes through:

    - A *Domain-Specific Language* (DSL) for describing modelling languages in terms of concepts and operations, and support for defining modelling purposes
    - A frontend framework that dynamically enables or restricts modelling concepts and operations based on the user's role and teaching context.

1.2 Thesis Outline

The remainder of this thesis is organized into five chapters. Chapter 2 reviews essential background and related work, providing the theoretical and technological context needed to understand our approach. Chapter 3 details the requirements for modelling tools for teaching identified in the literature, outlining the needs of diverse users, and from those a set of requirements for the frontend are derived. Chapter 4 presents our modular frontend architecture with reusable components that serve as the backbone of our system. Chapter 5 describes our DSL-driven approach for specifying modelling language concepts and modelling operations, as well as for tailoring the concepts that are made available to users of the modelling tool. Finally, Chapter 6 concludes the work and outlines future research directions.

Chapter 2

Background and Related Work

This chapter presents the relevant background as well as related work. Section 2.1 introduces key concepts including Model-Driven Engineering, Domain-Specific Languages, and core software design principles such as Separation of Concerns and Planned Reuse. Section 2.2 reviews existing modelling tools for teaching, discussing both specialized modelling tools and language workbenches, as well as the shift toward modern, web-based environments and collaborative efforts in this area. In Section 2.3, we explore architectural and technological foundations for web-based Modelling Tools for Teaching by comparing client-server and microservice architectures, and examining language server protocols for both textual and graphical models. Finally, Section 2.4 delves into the CORE framework, which provide the metamodel basis for our DSL-driven configuration approach.

2.1 MDE, DSL and Software Design Principles

2.1.1 Model-Driven Engineering

Model-Driven Engineering (MDE) [8] [9] views the entire software lifecycle as a process of creating, refining and integrating models. It relies on model transformations and consistency constraints to evolve high-level abstractions into executable artifacts. This approach supports Separation of Concerns (SoC) by isolating distinct system aspects into dedicated models, which in turn facilitates planned reuse and improves maintainability.

2.1.2 Domain-Specific Languages

Domain-Specific Languages (DSLs) [10] are tailored to express domain-specific concepts more naturally than General Purpose Languages. They reduce the semantic gap and accidental complexity, leading to increased productivity and clearer models. When integrated with MDE, DSLs enable automated code generation and model transformations, streamlining the development process and ensuring that models closely reflect the problem domain.

2.1.3 Separation of Concerns

Separation of Concerns (SoC) was popularized by Dijkstra [11] and emphasizes the practice of temporarily focusing on a single development concern, e.g., data modelling, user interface design or performance optimization, while minimizing distractions from unrelated aspects. This strategy helps reduce the cognitive load on developers by compartmentalizing complex systems into more manageable pieces. For example, in a model-driven context, SoC can manifest as distinct models for structure (e.g., class diagrams), behavior (e.g., state machines) and requirements (e.g., use cases). Each model addresses a specific

viewpoint or concern, allowing engineers to work at an appropriate level of abstraction.

2.1.4 Planned Reuse

Planned reuse involves designing software artifacts to be reused deliberately across multiple projects [12]. Unlike opportunistic reuse, which adapts existing code after the fact, planned reuse is built into the design process from the start. This approach minimizes duplication, ensures consistency and allows for more efficient development. By encapsulating distinct concerns into modular, reusable components, planned reuse supports the development of robust systems with lower long-term maintenance costs.

2.2 Modelling Tools for Teaching

Educators have long relied on modelling tools to teach modelling effectively, providing students with an interactive environment to create, validate and refine models. This section first reviews some of the most important modelling tools used in teaching and then presents the common movement towards web-based architectures.

2.2.1 Modelling Tools and Language Workbenches

Modelling tools [13] are essential in educational settings, specifically to teach modelling languages such as the Unified Modelling Language (UML). Students use these tools to learn the syntax, semantics, and structural rules of various modelling languages, often accompanied by a dedicated debugger and other development services. These tools are typically designed to work with one or more specific modelling languages.

In contrast, ***language workbenches*** [14] like the Eclipse Modelling Framework (EMF)

are more generic, allowing users to create and manipulate domain-specific modelling languages. These tools focus on metamodelling and grammar design and are more suitable for students learning to develop custom modelling languages. Although they provide generic services, students often must develop model transformations and code generators themselves.

Our focus in this work is primarily on the frontend, specifically the modelling editor, which forms the core visual and interactive component of the tool. Such a frontend could be used both in modelling tools as well as language workbenches.

2.2.2 Overview of Important Modelling Tools for Teaching

Several academic and industrial modelling tools have been developed to facilitate teaching software modelling. Below, we provide an overview of some of the most prominent tools that have been widely used in educational settings.

1. **GME (Generic Modelling Environment)**: The GME [15] is a domain-specific modelling tool designed for building customized modelling environments. It allows educators and researchers to define new modelling languages through meta-modelling, providing a flexible and extensible infrastructure. GME has been widely used in academic settings for teaching model-driven engineering (MDE) and domain-specific language (DSL) development. It provides an intuitive graphical interface that enables students to design and manipulate models with built-in validation mechanisms. However, GME requires a Windows-based installation and has a steep learning curve, making it less accessible for beginners and limiting its adoption in modern web-based teaching environments.
2. **Atom3/AtomPM**: AToM3 [16] (also known as AtomPM) is another widely used

tool in the modelling community, particularly in educational contexts. It supports multi-paradigm modelling, allowing students to experiment with various modelling languages and formalism transformations within a single environment. AToM3 is designed to enable students to create their own domain-specific modelling languages (DSMLs) [17] using meta-modelling techniques, making it a valuable tool for teaching language design. One of its key strengths is its ability to automatically generate model transformations, which helps students understand the relationships between different modelling approaches. However, AToM3 suffers from aging technology, limited maintenance and an outdated, single-user interface, which hinders its usability in modern teaching environments.

3. **Umple**: Umple [18] is a hybrid modelling tool that integrates textual and graphical modelling capabilities to support model-driven development. Unlike traditional graphical modelling tools, Umple allows students to define models using a text-based syntax, which is familiar to software developers, while simultaneously providing a graphical representation. This dual approach helps students transition between traditional coding and model-based development more seamlessly. Umple also includes features such as code generation, model validation, and simulation, making it an effective tool for teaching UML, state machines and software design patterns. Moreover, it is web-based, eliminating installation barriers and making it highly accessible for students and educators. Despite these strengths, its textual modelling approach may require additional learning effort for students who are more accustomed to purely graphical environments. Furthermore, Umple is single-user, and hence not capable of online collaborative modelling.
4. **TouchCORE**: TouchCORE [19] is a modelling tool that emphasizes

concern-oriented reuse (CORE), allowing students to create, manipulate and refine models through multi-touch interactions. It is particularly useful for teaching software product line engineering (SPLE) [20], as it enables students to define reusable modelling components encapsulating multiple design variants and instantiate them based on different requirements. A key feature of TouchCORE is its perspective-based modelling approach, which allows educators to define different modelling views depending on the purpose for which a model is being built. TouchCORE is desktop-based, requiring installation. It is also single-user, and lacks web-based support, which can limit its accessibility and scalability for large classroom settings.

5. **USE (UML-based Specification Environment)**: The UML-Based Specification Environment (USE) [21] is a specialized modelling tool designed for validating UML and Object Constraint Language (OCL) specifications. It allows students to define UML class diagrams and formally specify constraints using OCL, enabling them to perform validation checks and detect inconsistencies in their models. USE is particularly valuable in courses that emphasize formal methods, software verification and correctness-by-construction principles. It provides a textual and command-line interface for executing model queries and constraint evaluations, which helps students understand the precise semantics of UML models. However, like many other academic tools, USE suffers from limited usability enhancements and outdated user interface components, making it less appealing for modern interactive teaching scenarios.
6. **Melanie**: Melanie [22] is a multi-level modelling and ontology engineering environment designed to facilitate model management, transformation, validation and

synchronization. It is particularly useful for students and researchers working with model-driven architecture (MDA) and metamodelling techniques. The tool enables users to define model transformations using rule-based mechanisms, making it a powerful platform for teaching model-to-model (M2M) [23] and model-to-text (M2T) [24] transformations. One of Melanie's unique features is its support for dynamic model evolution, allowing users to explore how models change over time and how transformations impact different abstraction levels. It supports multi-level modelling, which extends traditional two-level metamodelling by allowing flexible instantiation chains across different modelling layers. This capability is particularly advantageous for defining domain-specific modelling languages (DSMLs) and ontologies that require hierarchical structuring. Furthermore, Melanie provides graphical modelling capabilities that help users visualize and manipulate models interactively. However, like many academic modelling tools, it has a steep learning curve and requires installation and configuration, which may pose challenges in educational settings.

7. **BIGUML (A GLSP-based Web Modeling Tool)**: BIGUML [25] is a flexible, open-source web-based modelling tool created to facilitate UML diagram creation within modern code editors such as Visual Studio Code [26]. Developed in Austria (primarily by Dominik Bork and Haydar Metin), BIGUML leverages the Graphical Language Server Platform (GLSP) to provide a modular, extensible architecture. This allows users to seamlessly integrate UML diagram editors into various platforms with minimal setup. Unlike conventional rich-client UML tools, BIGUML offers a more lightweight, web-oriented approach that lowers the barrier to entry for students and educators. At present, BIGUML supports both Class Diagrams and Use Case

Diagrams, with plans to iteratively introduce the remaining UML diagram types. Its design emphasizes enhanced usability features such as property views, outline views, and contextual copy-paste operations to accommodate a range of educational or professional use cases. By tapping into GLSP's client-server model, BIGUML ensures that the domain logic (server) and the graphical interface (client) remain cleanly separated, offering extensibility and customizability. Moreover, because BIGUML can run as a VS Code extension, it aligns well with educational settings where students can install, learn and practice UML modeling directly within a familiar development environment.

On the industrial side, enterprise-grade modelling tools, such as Sparx Enterprise Architect [27], MagicDraw [28] and IBM Rational Software Architect [29], introduce a different set of challenges. These tools are highly complex, prohibitively expensive and not tailored for teaching. They lack guided feedback mechanisms and require extensive configuration, making them impractical for beginners [30] [31].

2.2.3 The Shift Towards Web-Based Modelling Tools

We have seen that academic modelling tools tend to suffer from aging technology stacks and lack of active maintenance. Recognizing the limitations of traditional modelling tools, researchers and educators are now shifting toward web-based modelling environments, which provide greater accessibility, ease of maintenance and improved user experience. Unlike locally installed tools, web-based platforms eliminate installation barriers as they allow students to use them directly in their web browsers [32] [33].

A notable example is the Epsilon Playground [34], a cloud-based modelling platform that allows students to experiment with model transformation, validation and code

generation without requiring software installation. Web-based modelling tools offer key advantages such as cross-platform accessibility, real-time collaboration, automatic updates and improved scalability, reducing the maintenance burden on instructors. The problem with the Epsilon playground is that it is aimed at software language engineers, i.e., users that want to create their own modelling languages, and hence is not well-suited for introductory modelling courses.

2.2.4 Collaborative Efforts for a Common MTT Architecture

The increasing need for modern, intuitive and accessible modelling tools for teaching (MTTs) has led to a collaborative effort among researchers and educators to establish a common architecture for MTTs. This movement is co-led by Professor Jörg Kienzle and Steffen Zschaler from King's College London, along with contributions from various institutions engaged in model-driven engineering (MDE) education. The effort aims to create a shared infrastructure that addresses the challenges associated with existing academic and industrial modelling tools, ensuring that future MTTs are scalable, web-based and pedagogically effective. A major milestone in this initiative was the organization of MODELS workshops [35] dedicated to modelling tools for teaching. These workshops provided a platform for researchers and educators to discuss the limitations of current tools and define a roadmap for future MTT development. As a direct outcome of these discussions, a comprehensive study was conducted, culminating in the Requirements for modelling tools for teaching paper [5], which presents a comprehensive set of requirements for MTTs. This paper identifies essential features that MTTs should provide.

In parallel, another significant initiative, the MDENet Education Platform [36], has been developed to support knowledge-sharing and tool integration within the MDE education

community. This platform serves as a hub for researchers, teachers and tool developers, facilitating collaboration and the development of shared resources for teaching modelling effectively.

The collaborative efforts documented in the Requirements for modelling tools for teaching paper [5] and supported by the MDENet Education Platform [36] provide a strong foundation for defining the requirements of modern MTTs. These requirements form the basis of the next Chapter 3, where we systematically derive the necessary functionalities and design considerations for building a flexible and extensible MTT architecture. By leveraging these insights, this thesis aims to contribute to the broader goal of creating an accessible, scalable and reusable modelling tool framework tailored for educational use.

2.3 Architectural and Technological Foundations for Web-Based MTTs

2.3.1 Overview of Architectural Options for MTTs

Web-based Modelling Tools for Teaching (MTTs) can be implemented using a variety of architectural styles, each with its own strengths and trade-offs. Two common approaches are the traditional client-server architecture [37] and the more distributed microservices architecture [38]. For educational tools, however, a typical web-based client-server model is often preferred due to its simplicity, centralized management and ease of deployment.

2.3.1.1 Client-Server Architecture

In a typical client-server MTT, the system is divided into three main layers [39]:

1. **Presentation Layer (Client)**: This layer resides in the user's web browser and is responsible for rendering the user interface, capturing interactions and communicating with the server. The client typically uses HTTP/REST [40] for regular interactions and WebSocket [41] for broadcasting real-time notifications.
2. **Application Layer (Server)**: The server hosts the core business logic and APIs. It processes modelling operations, manages real-time collaboration and coordinates data transformations. The server acts as the central coordinator, ensuring that concurrent updates to the model are handled consistently.
3. **Data Storage Layer**: This layer consists of databases that persist modelling data, user information and configuration settings. The server interacts with the database to store and retrieve model state, ensuring data integrity and consistency.

As shown in Figure 2.1, this client-server model offers a centralized and manageable framework that simplifies maintenance while enabling consistent deployment and updates.

2.3.1.2 Microservice Architecture

An alternative architecture is the microservice architecture, which decomposes the application into a collection of loosely coupled services [42]. Each microservice handles a specific business capability and communicates with others via Application Programming Interfaces (APIs). However, the added complexity of service orchestration, distributed data management and inter-service communication overhead can be significant.

As shown in Figure 2.2, a user requests a resource through a Resource Gateway, which coordinates interactions between microservices. The Auth service validates the user's credentials, ensuring authentication, while the Access service checks permissions for authorization. Finally, the Resource service retrieves the requested modelling resource from

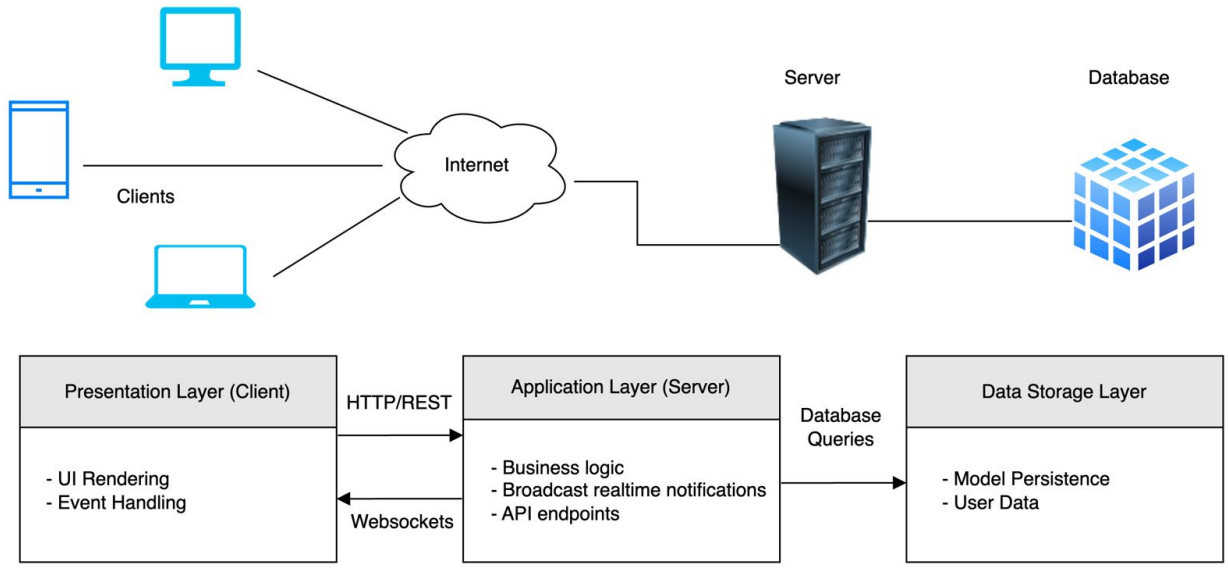

Figure 2.1: Typical Client Server Architecture

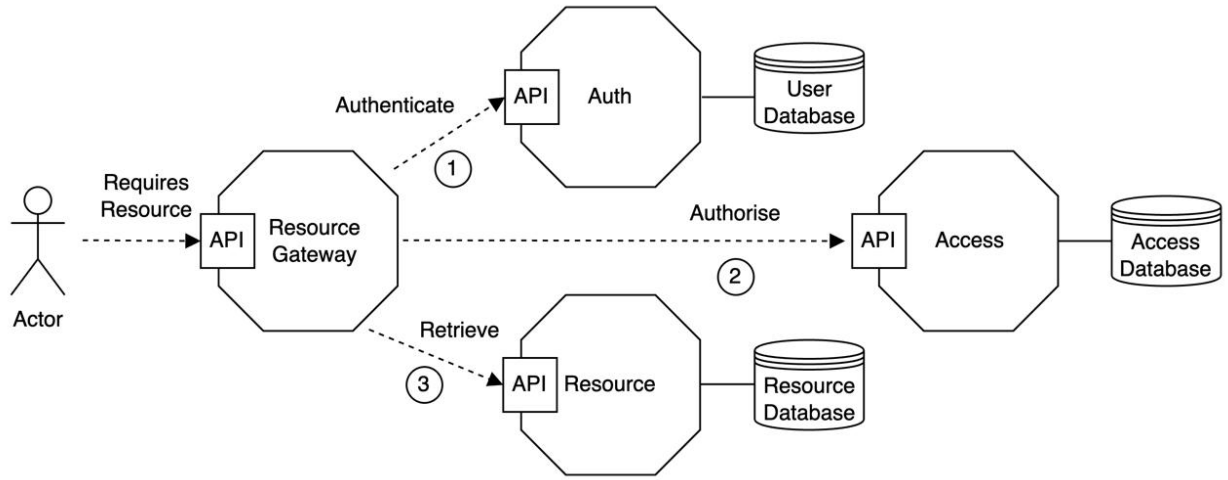

Figure 2.2: Microservice Architecture

the Resource Database.

For our web-based MTT, we adopt a client-server architecture as it offers a clear, layered approach that aligns with our modular frontend design (detailed in Chapter 4).

This architecture supports a robust, interactive user interface in the browser while centralizing business logic and data persistence on the server, making it particularly suitable for educational contexts where ease of use and maintainability are paramount. At the same time, our design leaves room for future evolution toward more distributed models, such as leveraging microservices to provide features like plagiarism detection or other specialized services, if additional scalability or modularity is required.

2.3.2 Language Server Support for Textual and Graphical Models

2.3.2.1 Language Server Protocol (LSP)

Language server protocol (LSP) [43] is a standard communication protocol for code-related services. It is widely used in text-based programming environments of modern IDEs to provide syntax validation, code completion and real-time feedback. One of its key advantages is separation of concerns, as it decouples language-specific logic from the frontend, allowing for better modularity. Additionally, LSP supports multi-client compatibility, enabling integration with various IDEs such as VS Code, Eclipse and IntelliJ. It also facilitates lightweight communication through a standardized JavaScript Object Notation-Remote Procedure Call (JSON-RPC) protocol, ensuring efficient messaging. However, despite these benefits, LSP is designed for text-based code, which means it would work for text-based modelling, but lacks native support for graphical representations, which are essential for diagram-based modelling tools.

2.3.2.2 Graphical Language Server Protocol (GLSP)

Graphical language server protocol (GLSP) [44] extends the principles of LSP to graphical modelling environments, providing server-side processing for diagram-based syntax

validation, layout handling and real-time updates. One of its key advantages is its framework-agnostic support, allowing it to be used across different graphical modelling tools. It also integrates seamlessly with Eclipse Theia and other web-based editors, making it a viable choice for modern development environments. Additionally, GLSP offers scalability, making it well-suited for handling large and complex models.

2.3.2.3 Why LSP and GLSP Were Not Used

Although LSP and GLSP provide robust language-processing capabilities, our system prioritizes lightweight, in-browser execution without additional backend dependencies. While LSP is well-suited for text-based services by decoupling language-specific logic from the client and enabling efficient JSON Remote Procedure Call (RPC) communication, it does not natively address the requirements of graphical modelling, which is a critical aspect of our tool. GLSP extends LSP to provide support for graphical modelling by handling diagram-based syntax validation, layout management and real-time updates. However, integrating GLSP would have significant implications for our setup. It necessitates additional infrastructure for managing custom serialization and deserialization of graphical data, which increases the complexity of the backend. Moreover, aligning GLSP with our client-side visualization engine would require extensive integration efforts and additional configuration layers. These factors collectively introduce considerable development overhead and operational complexity. Therefore, although GLSP is powerful and scalable for handling large, complex models, its integration would compromise our goal of maintaining a lightweight, browser-based modelling tool that is easily configurable. This decision allows us to preserve the simplicity and ease of setup essential for an educational tool while still delivering efficient and robust modelling functionality.

2.4 The CORE Framework

CORE [45] is a modelling language-independent approach that helps software language engineers create modelling languages that include features for streamlining model reuse. At its foundation, CORE defines a metamodel whose primary goal is to capture what model elements within a model are part of its reuse interface [46].

In more recent work, CORE was extended to enable multi-view modelling. To this aim, the CORE metamodel was extended in [47] to reify the concept of modelling language (`CORELanguage`), as well as the operations that a language provides to create and edit models (`CORELanguageAction`) (see Figure 2.3). To use a modelling language within CORE, it suffices to instantiate a `COREExternalLanguage`, and then instantiate `CORELanguageAction` for each one of the model edit operations. For example, for the *Class Diagram* language, the language actions would include *AddClass*, *AddAttribute*, *SetAttributeVisibility*, *AddAssociation*, *SetMultiplicity*, etc.

[47] also proposes the concept of *perspective* (`COREPerspective`). Perspectives can be used to *group a set of languages for a specific modelling purpose*. Perspectives can then expose their own language actions that coordinate the actions of the encapsulated languages to ensure consistent use. For example, one can define a perspective that groups together class diagrams and state diagrams, which can be used to describe the structure and behaviour of a real-time system. In that case, this "real-time system modelling perspective" would expose an *AddClass* language action that would ensure that for each new class that is added to a class diagram, a new state diagram is created at the same time that specifies the behaviour for any instances of the class.

However, manually instantiating these metamodel elements, namely, `CORELanguage`, `CORELanguageAction`, `COREPerspective` and `COREPerspectiveAction` is both tedious and

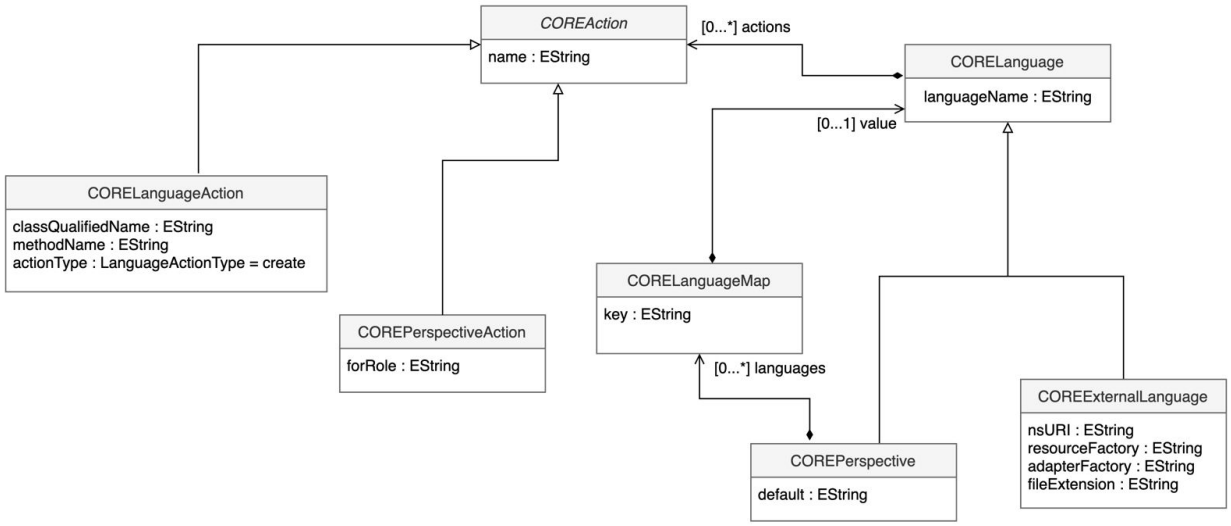

Figure 2.3: Small Exerpt of the CORE Metamodel

error-prone. To address this, [47] introduced a Domain-Specific Language (DSL) that enables language designers to write concise DSL code describing languages and perspectives. A code generator then processes this DSL to produce `CORELanguage` and `COREPerspective` instantiations. This automated process reduces manual effort and ensures consistency across language and perspective definitions. As a result, modelling language developers can focus on the conceptual design of their languages, while the DSL-driven generator handles the repetitive or intricate details of instantiating the CORE metamodel for multi-view modelling.

Chapter 3

Requirements

The focus of this chapter is on defining the requirements for the frontend of a web-based modelling tool intended for teaching purpose. Section 3.1 introduces key terminology to establish a common understanding. Section 3.2 presents a comprehensive set of 19 requirements (Rs), distilled from academic research, in particular from the paper "Requirements for Modelling Tools for Teaching" [5], as well as from own insights gathered during the design and development process. Finally, in section 3.3, we analyze these initial requirements to assess their impact on the frontend and derive a set of 16 frontend requirements (FRs) that the frontend we are going to develop needs to fulfill.

3.1 Categories of Modelling Tool Users

We must carefully consider the different types of users who will interact with the tool. Each group of users has specific goals and needs, so we will avoid using the general term "users" and instead refer to the following groups explicitly whenever appropriate:

1. **Modellers/Students:** This group consists mainly of students who use the tool to

create models that are submitted for grading. They focus on efficiently completing exercises, creating models like class diagrams and testing their understanding of the modelling language. Their interaction with the tool is primarily task-oriented, and they benefit from features that streamline modelling tasks and provide feedback.

2. **Educators/Teachers:** Educators use the tool to design exercises, create sample solutions and evaluate student submissions. Their focus is on configuring the tool to align with educational objectives, such as limiting certain modelling language features for pedagogical purposes. For example, a teacher may want to restrict the number of concepts available to students in an exercise or integrate automated grading and feedback mechanisms.
3. **Developers:** Developers are responsible for extending the tool's functionality, adding support for new modelling languages and integrating additional editor features or services. For this category, the architecture of the tool must be flexible and extensible, enabling the seamless addition of new components and plugins.

3.2 Requirements (R)

This section will provide a list of key requirements (R) drawn from the paper "Requirements for Modelling Tools for Teaching" [5] as well as our own observations. These requirements address the needs of the three user categories and form the foundation for the design and functionality of the tool. They are presented here organized into four categories: language and feature-related, user interaction and collaboration-related, education-related and development-related.

3.2.1 Language and Feature-Related Requirements

R1: Multi-Language Support

The system must support multiple modelling languages, enabling the creation and use of different modelling notations such as Class Diagrams, State Diagrams and Sequence Diagrams. This ensures that the modelling tool remains flexible and extensible, allowing different types of models to be defined and visualized. The architecture should allow new modelling languages to be integrated without requiring fundamental changes to the system.

R2: Dual-Mode Support

The tool should provide both graphical and textual user interfaces, catering to different preferences. The graphical interface helps visualize complex relationships (e.g., for *Class Diagrams*), while the textual interface is ideal for Modellers/Students and Educators/Teachers who prefer fast input via typing. Syntax highlighting is essential for textual modelling.

R3: Model-to-Code Synchronization

The tool should enable round-trip synchronization between models and generated code, i.e., any changes in the model are accurately reflected in the code and vice versa. This helps Modellers/Students understand the connection between the visual models and the corresponding code.

R4: Simulation and Execution

The tool should support simulation and execution of behavioural models (e.g., state machines), visualizing the behaviour in real-time. This helps Modellers/Students

understand and Educators/Teachers better explain the semantics of models.

R5: Automatic Validation with Warnings and Errors

The tool must include an automatic validation feature to detect modelling errors. It should also provide warnings. For example, it should inform Modellers/Students when a model element is declared but never used.

R6: Searchable Example Library

A categorized and searchable library of pre-built models and templates should be available for Modellers/Students and Educators/Teachers, serving as a reference and aiding in teaching and learning.

3.2.2 User Interaction and Collaboration-Related Requirements

R7: Intuitive User Interface

The tool must provide a user-friendly, intuitive interface for Modellers/Students and Educators/Teachers to minimize the learning curve and facilitate efficient task completion.

R8: Fast Response Time

The tool must ensure that it executes all operations triggered by students or educators through clicks or menu selections without noticeable delays, providing a seamless and responsive experience, even under high-traffic conditions.

R9: Collaborative Editing with Role-Based Permissions

The tool must support collaborative editing for Modellers/Students and Educators/Teachers both online and offline, with role-based permissions for different access levels (read-only, grade-only, edit and full access). It should include version control and conflict resolution mechanisms to handle simultaneous contributions seamlessly.

R10: Save and Resume Functionality

The tool should support saving intermediary model states, allowing Modellers/Students to pause their work and resume later. It must also support exporting models in a format appropriate for assignment submission.

R11: Integrated User Guide

A comprehensive user manual and a well-structured FAQ section should be integrated within the tool to guide Modellers/Students, Educators/Teachers and Developers in its usage.

R12: Language Customization

The system must allow educators to configure and restrict the available modelling concepts within a language based on learning objectives. This means enabling or disabling specific language elements (e.g., restricting students to only creating classes and attributes initially, while unlocking operations and associations later).

3.2.3 Education-Related Requirements

R13: Gamification

The tool should include a simple reward system, such as earning points or badges for completing modelling tasks. Modellers/Students can unlock small rewards for milestones, like finishing their first model or successfully submitting an assignment.

R14: Flexible Feedback, Grading and Plagiarism Detection

The tool should integrate flexible feedback mechanisms and automated grading for Modellers/Students. It should also include plagiarism detection to ensure the integrity of submitted assignments. These features must accommodate both instant feedback during modelling as well as delayed feedback after submission, with grading tied to both correctness and completeness.

3.2.4 Development-Related Requirements

R15: Modular Architecture

The architecture should be modular, enabling independent updates and extensions to ensure scalability and long-term adaptability for Developers.

R14: CI/CD Support for Development and Deployment

The system should support Continuous Integration and Deployment (CI/CD) to automate testing, maintain code quality and streamline structured deployment processes. This feature is primarily intended for developers working on extending or maintaining the tool. The

CI/CD pipeline should include structured project management, unit testing, automated builds and deployment workflows to ensure stable releases and efficient version control.

R17: Multi-Platform Accessibility

The tool should be accessible across multiple platforms (Windows, macOS, Linux) and devices (Desktop, Tablet), ensuring usability for Modellers/Students, Educators/Teachers and Developers irrespective of their device or operating system.

R18: Simple Installation

The tool should require minimal setup, ideally operating without the need for installation. If installation is necessary, it must provide a streamlined, simple process with clear instructions.

R19: User Interaction Tracking and Feedback Collection

The tool should track interactions from Modellers/Students and Educators/Teachers to continuously improve the user experience. It should also include tools for collecting feedback via surveys to refine the tool's features and performance based on real-world usage.

3.3 Frontend Requirements (FR)

This thesis focuses on the design of a modular and reusable frontend framework for MTTs to be used in a client-server architecture (see section 2.3.1.1). We therefore carefully investigated the requirements (R1–R19) presented in the previous section, determined how they affect the frontend, and based on an in-depth analysis distilled a set of frontend requirements (FR1–FR16). Each FR presented in this section is given a concrete name that reflects its purpose,

which will later be referenced in the subsequent chapters to explain decisions made during the design of the frontend architecture. We also link the FRs to the Rs that they support, and propose ways to measure the success of each FR, where applicable. Again, we present the FRs in this section organized into the same four categories as the global requirements in the previous section.

3.3.1 Language and Feature-Related Requirements

FR1: Graphical and Textual Dual-Mode Interface

The frontend must have a dual-mode interface that integrates both a graphical diagram editor and a textual code editor in a synchronized manner. The graphical interface must allow users to directly manipulate model elements through intuitive drag-and-drop, zoom and pan functionalities. It should present model elements with clear, distinct visual cues and interactive controls to facilitate ease of editing. The textual editor must be embedded alongside the graphical view and provide syntax highlighting, auto-completion, etc. The textual editor must support configurable settings for font size and theme to enhance readability. The editor should provide a simple interface for users to toggle between graphical and textual representations. Finally, changes in one view should be instantly propagated to the other without delay, maintaining the consistency of both model representations.

→ **Supports R2: Dual-Mode Support, R3: Model-to-Code Synchronization**

Measuring the Requirement:

1. **Synchronization Latency:** Measure the time interval between a change made in the graphical view and its reflection in the textual view. This can be done by instrumenting the data-binding events to log timestamps on both sides, then

calculating the average latency over multiple interactions. A lower latency indicates that the interface effectively synchronizes changes, ensuring a seamless dual-mode experience.

2. **Consistency Verification:** Periodically capture snapshots of the underlying data model as displayed in both the graphical and textual views and compare them with what is displayed to verify consistency. This can be achieved using automated tests that simulate updates and then perform a diff analysis between the two views. A high degree of consistency demonstrates that the dual-mode interface maintains an accurate, shared representation of the model across both modes.

FR2: Simulation and Execution Panel

The frontend must have a dedicated simulation and execution panel that allows users to run dynamic models and observe their behavior in real time. This panel must provide essential controls such as play, pause, stop, step-through, and rewind to manage simulation flow. The panel must support both continuous and stepwise execution modes to cater to different teaching scenarios. The design must allow for interactive adjustments, such as altering simulation speed or parameters on the fly. The editor should display the current simulation status clearly, using visual indicators like highlighted active states and transition animations. It should support real-time updates that show how model elements change over time during the simulation.

→ **Supports R4: Simulation and Execution**

Measuring the Requirement:

1. **Control Responsiveness Testing:** Record the response times for simulation panel controls (such as play, pause, stop and step) from when a control is activated to when the corresponding visual update appears on the diagram. This measurement can be

automated through JavaScript event timing logs. Fast response times are crucial for a smooth simulation experience, ensuring that users receive immediate feedback during model execution.

2. **Simulation Accuracy Assessment:** Validate that the simulation panel correctly triggers the expected state transitions by comparing observed outputs with predefined simulation scenarios. This can involve running controlled simulation tests that measure whether the visual indicators (such as highlighted states or animated transitions) match the expected results. High accuracy in these tests indicates that the simulation panel reliably reflects the dynamic behavior of the model.

FR3: Multi-Language Notation Support

The frontend must dynamically adapt to different modelling languages based on predefined configurations, ensuring seamless support for multiple modelling notations. When a specific modelling language (e.g., *Class Diagrams*, *State Diagrams*) is selected, the system must initialize the editor with the correct notation, automatically loading the corresponding elements, symbols and validation rules. The frontend reads language definitions and enforces the correct set of modelling elements and behaviors accordingly. The system must prevent users from freely switching between modelling languages unless explicitly allowed. Instead, it should ensure that a course-specific or assignment-specific language is automatically applied when loading the editor. UI elements such as menus, toolbars and context options must dynamically adjust based on the chosen modelling language, ensuring a consistent and structured editing experience.

→ **Supports R1: Multi Language Support**

Measuring the Requirement:

1. **Correct Language Loading:** Validate that the frontend correctly reads and applies

predefined language configurations. Test different modelling languages (*Class Diagrams*, *State Diagrams*) to ensure that only the relevant elements, rules and UI components are loaded.

2. **Restricted Language Switching:** Ensure that switching between modelling languages is only possible when explicitly enabled in the configuration. Automated UI tests with Jest [48] and Puppeteer [49] should confirm that restricted elements do not appear when switching between predefined modelling contexts.

FR4: Search Interface

The frontend must include an interactive searchable example library accessible directly from the main navigation or toolbar within the tool. This library should be designed as an integrated panel or a modal window that can be opened within the tool's interface. It must feature a categorized structure with distinct sections for different model types, such as "Class Diagrams", "State Machines", etc clearly visible in a sidebar or tab menu. Modellers/Students and Educators/Teachers should be able to search for specific models using a search bar positioned prominently at the top of the library. Search results should be displayed as a grid or list of model thumbnails, each with a brief description visible on hover or click. There should be a clear "Import" button within the preview that allows users to add the selected model to their current project with a single click. Additionally, each model entry should include detailed metadata, such as the model's creator, a short description and tags. The library should also support filtering by tags or categories to streamline model discovery. The design must ensure that the library's interaction is smooth and responsive, providing real-time updates and seamless transitions between different views and actions.

→ **Supports R6: Searchable Example Library**

Measuring the Requirement:

1. **Search and Filter Functionality:** Use browser DevTools (Elements tab) [50] to inspect the DOM updates when applying search queries or category filters. Ensure that the correct models appear without missing or incorrect results and that filtering updates the displayed items instantly without unnecessary reloading.
2. **Model Preview and Import Validation:** Open various example models and check if they correctly display metadata, allow zooming and load without visual glitches. Then, verify that clicking the "Import" button correctly integrates the selected model into the workspace by inspecting the application state or console logs to confirm successful import actions.

3.3.2 User Interaction and Collaboration-Related Requirements

FR5: Intuitive User Interface Layout

The layout of the frontend must be carefully designed so that key functions (e.g., menus) are organized intuitively for the user. This includes designing a clean and uncluttered UI. The placement of UI elements must support efficient workflows for Modellers/Students and Educators/Teachers, allowing them to focus on modelling without unnecessary confusion.

→ **Supports R7: Intuitive User Interface**

Measuring the Requirement:

1. **Click Efficiency:** Track the number of clicks required to perform common tasks (e.g., adding a class, modifying attributes, etc). Fewer clicks and actions indicate that the UI layout is more intuitive and efficient for the user. Use analytics tools like Google Analytics [51] or Hotjar [52] to monitor user click behavior and interaction patterns on

different UI elements. Define a maximum number of clicks or actions for certain tasks (e.g., completing tasks in no more than 3 clicks as suggested in [53]).

2. **First-Click Accuracy:** Measure the percentage of times a user clicks on the correct UI element on their first attempt when performing a specific task. High first-click accuracy indicates that the layout is intuitive and users can quickly find what they are looking for. Tools like Hotjar can track where users first click when given a task. Aim for a target of at least 80-90% first-click accuracy for key tasks, as achieving this threshold ensures that users can navigate efficiently and complete tasks with minimal confusion [54].

FR6: Seamless User Interactions with Minimal Latency

The frontend must be optimized to ensure instantaneous response to all user actions, including clicks, typing, etc. This requires efficient handling of UI events, ensuring that there is no visible delay when elements are moved, resized or edited. The frontend should process user inputs in real-time, ensuring that even complex operations (such as rendering large diagrams or updating relationships) do not introduce lag.

→ **Supports R8: Fast Response Time**

Measuring the Requirement:

1. **Performance Testing:** Use tools like Google Lighthouse [55], WebPageTest [56] or Chrome DevTools [57] to measure the time taken for user inputs (e.g., clicks) to trigger updates in the UI. The goal is to ensure that actions trigger UI updates promptly to maintain a responsive user experience. Aim for response times under 100 milliseconds to ensure an optimal user experience and avoid performance bottlenecks [58].
2. **User Feedback:** Gather user feedback through usability tests, focusing on perceived

lag during normal and high-load operations. Any reported delays should be investigated, with concrete metrics established for reducing response times.

FR7: Embedded User Guide with Searchable FAQ

The frontend must integrate a user guide and a searchable FAQ section that can be accessed directly within the tool's interface without navigating away from the application. A "Help" button or icon should be consistently visible (e.g., top-right corner) for Modellers/Students, Educators/Teachers and Developers. When clicked, a popup or side panel or new web page will open, displaying the user guide and FAQ. The FAQ section will include a search bar to quickly find relevant topics. The guide will support text, images and collapsible sections to facilitate easy navigation.

→ **Supports R11: Integrated User Guide**

Measuring the Requirement:

1. **Search Query Response Time:** Log the time taken from when a user enters a search query in the FAQ search bar to when results are displayed. The goal is to ensure smooth searching by keeping response times as quick as possible. This can be measured using browser DevTools [57] or JavaScript performance timers [59].
2. **Help Button Click Functionality:** Automate a test script using Selenium or Puppeteer to verify that clicking the Help button consistently opens the user guide or FAQ section without delays or errors. This ensures accessibility from all relevant pages.

FR8: User-Friendly Setup and Responsive Model Canvas

The frontend must ensure that Modellers/Students, Educators/Teachers and Developers can access the tool directly through a web browser, ideally without any installation required.

If installation is unavoidable, it should be minimal and straightforward, with clear, user-friendly instructions. The tool's modelling canvas must dynamically resize and adjust based on the size of the user's screen, whether it is running on a desktop or tablet. The modelling elements must remain readable and selectable regardless of resolution changes. The UI should be adaptive, so no elements overlap or become inaccessible when the window is resized.

→ **Supports R17: Multi-Platform Accessibility, R18: Simple Installation**

Measuring the Requirement:

1. **Installation and Setup Time Measurement:** If installation is required, measure the total time taken from downloading to the first successful launch. Use a simple script or manual stopwatch timing. The goal is to ensure that the setup is completed promptly with minimal steps.
2. **Responsive UI and Canvas Behavior Testing:** Open the tool on different screen resolutions (e.g., 1920×1080, 1366×768, 1024×768, tablet mode) and verify that the canvas resizes dynamically, no UI elements overlap, and all modelling elements remain readable and interactive. This can be quickly tested using browser DevTools [57].

FR9: Real-Time Online Collaboration with Role-Based Permissions

The frontend must support real-time online collaboration, enabling multiple users to work on the same model concurrently. It must implement a system where changes made by one user are instantly synchronized across all connected clients. The interface should visually differentiate contributions from different users through color-coding or user tags. It must enforce role-based permissions that dynamically adjust the available UI controls based on the user's role, such as read-only, editor or administrator. It should incorporate visual indicators that show active collaboration, such as highlighting elements that are being edited by others. The system must have mechanisms to prevent conflicting edits and resolve them if they

occur, displaying conflict alerts to users. It must allow educators to configure user roles and permissions directly within the interface, ensuring that unauthorized actions are disabled. The design should ensure that the collaboration features are integrated seamlessly with the core model editing functionality. It must support both individual and group editing sessions without sacrificing performance. The frontend should be capable of displaying a history of changes for transparency. It must also provide feedback mechanisms that alert users to new updates from collaborators. The role-based interface adjustments should be clearly visible and intuitive, ensuring that users are aware of their permissions at all times.

→ **Supports R9: Collaborative Editing with Role-Based Permissions**

Measuring the Requirement:

1. **Synchronization Delay Measurement:** Measure the time taken for a change made by one user to appear on the screens of all other collaborating users. This can be achieved by logging timestamps at the moment of change and when it is reflected across clients via WebSockets. Consistently low delays are indicative of efficient real-time synchronization in the collaborative environment.
2. **Effectiveness of Role-Based Permissions:** Simulate collaborative scenarios with users assigned different roles and monitor the system's enforcement of permissions. This can include measuring the number of unauthorized attempts and verifying that UI elements corresponding to restricted actions are disabled or hidden. A robust role-based system should prevent unauthorized changes and maintain a clear separation of privileges.

FR10: Offline Collaboration and Conflict Resolution

The frontend must support offline collaboration by enabling users to continue working even when network connectivity is lost. The interface should display clear visual indicators that

inform users when they are offline and that changes are being cached locally. Upon reconnection, the system must automatically synchronize the locally cached changes with the backend while detecting any conflicts that arise. The frontend must include a conflict resolution interface that allows users to compare the local version and the server version side by side. This interface should highlight differences clearly and offer simple options for merging or discarding changes. It must enforce role-based permissions during conflict resolution to prevent unauthorized modifications. The offline collaboration feature should provide real-time feedback on the synchronization process, such as progress indicators or status messages. The system must ensure that offline edits are seamlessly integrated with the live model once connectivity is restored. It should also log conflict events and provide users with a summary of actions taken during synchronization. The design must guarantee that data integrity is maintained, even when users switch between online and offline modes.
→ **Supports R9: Collaborative Editing with Role-Based Permissions, R10: Save and Resume Functionality**
Measuring the Requirement:

1. **Offline Editing and Synchronization Testing:** Measure the system's ability to handle offline edits by simulating a network disconnect during active modelling. Manually record the state of the model before and after disconnecting, then monitor the automatic synchronization upon reconnection using logging or internal timestamps. The goal is to ensure that the locally cached changes merge seamlessly with the backend model, and that any conflicts are detected and resolved correctly.
2. **Conflict Resolution Interface Evaluation:** Simulate concurrent edits by two users on the same model element to force a conflict scenario. Evaluate the conflict resolution interface by verifying that it clearly highlights the differences between the

local and server versions and that it offers intuitive options for merging or discarding changes. The objective is to confirm that the interface enforces role-based permissions and maintains data integrity during the conflict resolution process, as evidenced by the final, consistent state of the model.

FR11: Adaptive Frontend Modelling Interface

The frontend must dynamically adapt to language customizations done by an educator/teacher, ensuring that the modelling environment aligns with specific teaching goals. The educator can decide which modelling elements and operations are available within a given session, and the UI should enable or disable the corresponding visualizations and menu items, limiting the complexity of available modelling constructs. The educator should be able to specify the customization without having to modify the tool's implementation.

→ **Supports R12: Language Customization**

Measuring the Requirement:

1. **Customization Enforcement and UI Adaptation:** To validate that the frontend dynamically adapts based on the customization specified by the educator, we need to verify that only the allowed modelling elements, operations, and UI components are available for a given customization. This can be done by applying multiple predefined customizations and systematically checking that the frontend correctly enables or disables the related UI elements. Automated UI testing should be conducted using Jest and Puppeteer to simulate switching customizations and confirm that restricted elements are properly hidden, disabled or blocked. Additionally, DevTools DOM inspection should be performed to ensure that restricted components cannot be accessed by manually modifying the HTML.

3.3.3 Education-Related Requirements

FR12: External Gamification Integration Interface

The system must incorporate an external gamification integration interface that seamlessly connects the modelling tool with a dedicated gamification platform. Hence, on the frontend, the integration must be designed to display minimal, non-intrusive notifications, such as small toast messages that inform users when points are awarded or badges are unlocked. The user interface must include a clearly visible link or button that redirects users to the external dashboard, where detailed gamification data, such as leaderboards, badges and rewards history is managed and presented. Additionally, the frontend should maintain a clean and focused modelling environment by limiting the display of gamification elements within the tool itself. Instead, it should only present essential cues that prompt users to check the external platform for more comprehensive information.

→ **Supports R13: Gamification**

Measuring the Requirement:

1. **Engagement Metrics Analysis:** Instrument the frontend to log all interactions with gamification elements (e.g., toast notifications and redirects to the external dashboard). Analyze these logs to measure user engagement frequency, such as how often users trigger synchronization or view gamification notifications. Higher engagement rates can indicate that the gamification features are effectively integrated and motivating. This quantitative analysis helps assess if users are interacting with the gamification components as intended.
2. **User Satisfaction and Usability Testing:** Conduct structured usability tests and surveys focused on the gamification interface. Collect qualitative feedback on the clarity, responsiveness, and overall satisfaction with the minimal notifications and

integration flow. Ask users whether the gamification cues and the external dashboard access are intuitive and add value without cluttering the core modelling interface. Comparing the survey responses with the logged usage data can validate the effectiveness of the integration from a user experience perspective.

FR13: Visual Annotations and Overlays

The frontend must support visual annotations and overlays that provide immediate, in-context feedback to users during the modelling process. It must display error, warning and information overlays directly on the diagram using distinct colors and icons. These overlays should appear dynamically as model elements are edited, highlighting issues such as syntax errors, invalid relationships or inconsistencies. The frontend must implement tooltips or pop-up panels that offer detailed explanations for each annotated issue. It should allow educators to annotate models with grading comments or validation feedback that can be toggled on and off. The design must ensure that overlays do not obstruct the main diagram, using transparent layers or side panels as appropriate. It must provide a consistent style for annotations across different modelling languages to enhance clarity. The system should support real-time updating of overlays as model changes occur, ensuring that feedback stays up to date. It must include a mechanism for validating model elements against predefined rules and displaying the results visually. It must allow users to click on an overlay to access further details or to dismiss it if the issue has been resolved. The annotations should be integrated into the overall user interface so that they are part of the natural editing workflow. The interface must also provide an option for educators to configure which types of annotations are active.

→ Supports R5: Automatic Validation with Warnings and Errors, R14: Flexible Feedback, Grading and Plagiarism Detection

Measuring the Requirement:

1. **Overlay Update Time Measurement:** Instrument the overlay system to capture the delay between the occurrence of an event (e.g., a validation error) and the appearance of the corresponding visual annotation on the diagram. This metric can be measured using performance logging and is critical for ensuring that feedback is provided in near real time. A minimal delay ensures that users receive prompt and useful feedback.
2. **User-Centric Evaluation of Feedback Effectiveness:**

   Conduct user studies where participants model and then use the overlay system to resolve errors. Measure both the time taken for users to identify and correct errors and the accuracy of these corrections. This method assesses the practical effectiveness of the visual overlays and provides insights into how well the system supports learning through immediate, contextual feedback.

3.3.4 Development-Related Requirements

FR14: Frontend Modular Component System

The frontend of the modelling tool should follow a Modular Component System that allows Developers to independently manage, update and extend different aspects of the tool without affecting orthogonal functionality. This modular approach should structure the frontend into distinct, self-contained components, each responsible for specific features. Each component should be designed as an isolated unit, following a clear interface or contract, ensuring that it can be easily integrated into the overall architecture. For example, a diagram editor responsible for visual modelling by Modellers/Students should

function independently of a text-based interface designed for rapid input by Educators/Teachers. The components should communicate through well-defined APIs or events, ensuring that modifications or extensions in one part of the system (e.g., adding support for a new modelling language) do not disrupt other components. Developers should be able to introduce new capabilities, such as additional diagram types or updated simulation engines, without requiring modifications to existing components. To facilitate maintainability, the architecture should ensure a clear separation of concerns, such as distinguishing UI logic from backend interactions. This approach is intended to make debugging and upgrading individual features more efficient while preserving system integrity. Additionally, a well-structured modular system should support future-proof extensions, such as collaborative features or enhanced visualization modes, by simply adding or modifying specific frontend components. By adopting this Modular Component System, the frontend architecture should remain scalable and adaptable, allowing Developers to extend the tool in response to evolving requirements while maintaining performance, usability and long-term sustainability.

→ **Supports R15: Modular Architecture**

Measuring the Requirement:

1. **Component Independence Verification:** Modify or replace a specific component (e.g., the diagram editor or textual interface) and test whether the rest of the system continues functioning without errors. Use browser console logs to ensure no unintended dependencies or breakages occur.
2. **Component Integration and Extensibility Test:** Add a new mock component (e.g., a simple UI panel or visualization module) following the modular architecture guidelines and verify if it integrates smoothly without modifying existing components.

Check for proper event communication using DevTools Event Listeners or API calls to ensure data flows correctly between components.

FR15: CI/CD Integration for Frontend Development

Each frontend modification must pass automated unit and integration testing using Jest, ensuring that UI components and interactions function correctly. Additionally, static analysis tools should validate JavaScript modules, CSS styles and other frontend assets to detect errors before deployment. The CI/CD pipeline should automatically compile, bundle and minify the frontend code to optimize performance and ensure compatibility with the backend system. To prevent unexpected UI regressions, the system should support pre-release staging environments, allowing developers to preview frontend modifications before production deployment. Structured version control should ensure that all updates are properly documented, with rollback mechanisms available for quick restoration in case of failures. The pipeline must also integrate logging and error tracking, providing developers with real-time insights into build failures, UI test results and deployment status. By implementing a comprehensive CI/CD workflow, this requirement ensures that the frontend development process remains automated, efficient and scalable, supporting long-term maintenance and continuous system improvement.

→ **Supports R16: CI/CD Support for Development and Deployment**

Measuring the Requirement:

1. **Automated Testing and Build Validation:** Introduce controlled errors into the code (e.g., syntax mistakes or deliberate API deprecations) and observe whether the CI/CD pipeline detects these errors using tools such as Jest [48] for unit testing, Puppeteer [49] for UI regression tests, and ESLint [60] or SonarQube [61] for static code analysis. The pipeline should fail the build immediately when any test or static

analysis check fails. This validation is confirmed by reviewing CI logs and automated test reports that clearly indicate the presence of the injected errors. This method demonstrates that every update is rigorously tested, ensuring that no faulty code is merged into the main codebase.

2. **Staging Deployment and Rollback Testing:** Simulate a deployment failure by deliberately misconfiguring a component or introducing a runtime error in the staging environment, then verify that the rollback mechanism is triggered automatically. Tools such as a pre-release staging server and automated deployment scripts (e.g., using Jenkins [62] or GitLab CI/CD [63]) should log the error and initiate a rollback. The rollback process is validated by confirming that the system reverts to the previous stable version and that integration tests on the staging environment pass with the restored version. This approach provides concrete evidence that the CI/CD pipeline not only detects errors during the build and test phases but also safeguards system stability through an effective rollback process.

FR16: Interaction Tracker and Feedback Module

The frontend must implement an Interaction Tracker and Feedback Module that actively monitors and records user interactions within the modelling tool. This module is designed to capture granular details of each modelling action performed by Modellers/Students and Educators/Teachers, such as adding or modifying classes, editing diagrams and toggling between graphical and textual views. It must collect data on the frequency and duration of feature usage, tracking how long users interact with specific UI components. This information should be stored locally in the browser and optionally synchronized with a central server for further analysis. In addition, the module must incorporate a non-intrusive feedback prompt that appears at key moments, for example, immediately after a model is completed

or when a session ends. This prompt should invite users to rate their experience and offer optional comments about specific tool features. The feedback mechanism must be designed to minimize disruption to the user's workflow by appearing only during natural pauses. The frontend should also support a dedicated "Feedback" section where users can access and submit detailed surveys or questionnaires, tailored to different user roles. This section must be built with an intuitive UI, ensuring that feedback collection is as straightforward as possible.

→ **Supports R19: User Interaction Tracking and Feedback Collection**

Measuring the Requirement:

1. **Quantitative Log Analysis and Performance Metrics:** Measure the frequency and duration of user interactions by instrumenting the module to log all events with timestamps. Analyze these logs to determine the average response time of the module, the total number of interactions per session, and the usage distribution of various features. Additionally, assess the performance overhead introduced by the tracker by comparing system latency with and without the module enabled.
2. **User Experience Surveys and Dashboard Accuracy:** Conduct user surveys and structured usability studies with both students and educators to evaluate the clarity and usefulness of the visual feedback provided by the dashboard. Collect qualitative feedback on the intuitiveness of the feedback prompts and the overall satisfaction with the interaction tracking. In parallel, validate the accuracy of the gathered metrics by comparing logged interaction data with user-reported activity, ensuring that the module accurately reflects actual usage patterns.

3.4 Summary

This chapter focused on analyzing the literature and our practical experiences to distill a comprehensive set of requirements for Modelling Tools for Teaching (R1–R19). From these requirements, we derived a set of Frontend Requirements (FR1–FR16) that directly address usability, collaboration, educational and development aspects specific to the frontend. The mapping between FRs and Rs is summarized in table 3.1. These FRs form the basis for the modular, reusable frontend architecture presented in Chapter 4.

Table 3.1: Mapping of Frontend Requirements (FRs) to Supported Requirements (Rs)

Language and Feature Related	
FR1: Graphical and Textual Dual-Mode Interface	R2: Dual-Mode Support; R3: Model-to-Code Synchronization
FR2: Simulation and Execution Panel	R4: Simulation and Execution
FR3: Multi-Language Notation Support	R1: Multi-Language Support
FR4: Search Interface	R6: Searchable Example Library
User Interaction and Collaboration Related	
FR5: Intuitive User Interface Layout	R7: Intuitive User Interface
FR6: Seamless User Interactions with Minimal Latency	R8: Fast Response Time
FR7: Embedded User Guide with Searchable FAQ	R11: Integrated User Guide
FR8: User-Friendly Setup and Responsive Model Canvas	R17: Multi-Platform Accessibility; R18: Simple Installation
FR9: Real-Time Online Collaboration with Role-Based Permissions	R9: Collaborative Editing with Role-Based Permissions
FR10: Offline Collaboration and Conflict Resolution	R9: Collaborative Editing; R10: Save and Resume Functionality
FR11: Adaptive Frontend Modelling Interface	R12: Language Customization
Education Related	
FR12: External Gamification and Integration Interface	R13: Gamification
FR13: Visual Annotations and Overlays	R5: Automatic Validation; R14: Flexible Feedback, Grading and Plagiarism Detection
Development Related	
FR14: Frontend Modular Component System	R15: Modular Architecture
FR15: CI/CD Integration for Frontend Development	R16: CI/CD Support for Development and Deployment
FR16: Interaction Tracker and Feedback Module	R19: User Interaction Tracking and Feedback Collection

Chapter 4

Modular Frontend Architecture with Reusable Components

This chapter presents a structured approach to designing a modular frontend architecture for a web-based graphical modelling tool, emphasizing reusability, extensibility, scalability and maintainability. The goal is to enable seamless integration of new modelling notations while minimizing development effort by reusing code. To achieve this, the system is designed with generic modules handling core functionalities and language-specific modules. This separation ensures that new editors can be added without modifying core modules. Additionally, the architecture supports real-time collaboration, ensuring consistency across multiple users.

In developing this architecture, we have focused on implementing the core functionalities that are essential for the effective operation of the editor. Specifically, we have implemented the following key frontend requirements listed from Chapter 3:

- FR3: Multi-Language Notation Support
- FR5: Intuitive User Interface Layout

- FR6: Seamless User Interactions with Minimal Latency
- FR8: User-Friendly Setup and Responsive Model Canvas
- FR14: Frontend Modular Component System

The frontend requirement, which we have also implemented, FR11: Adaptive Frontend Modelling Interface, related to perspective-based customization, is not covered in this chapter. It will be addressed in the next Chapter 5, where the DSL for Language Configuration is introduced to dynamically control language-specific operations based on user roles and teaching objectives.

Besides these core requirements, our architecture also accommodates additional frontend requirements outlined in Chapter 3, as well as non-core features, which are considered add-ons and fall outside the scope of my thesis. These features, although not fully implemented in this work, are designed to integrate seamlessly into the system in the future. These add-on requirements include:

- FR1: Graphical and Textual Dual-Mode Interface
  - Although our current implementation provides a robust graphical modelling interface, a synchronized textual editor with real-time code updates and error feedback is considered an add-on enhancement; this extension can be seamlessly integrated into our existing architecture.
  - By developing this dual-mode functionality as an add-on, the system remains flexible and modular, allowing the textual view to be added or upgraded independently, thereby enhancing the overall user experience; however, its full implementation will require additional development effort from content developers and educators to design the integrated code editor and ensure robust synchronization.

- FR2: Simulation and Execution Panel
  - Our frontend currently renders static diagrams and the addition of a dedicated simulation and execution panel with controls for play, pause, step-through and rewind, is viewed as an add-on that extends the basic visualization into dynamic model simulation.
  - This enhancement will integrate real-time visual feedback, such as animations and state highlights, into the existing UI, leveraging our modular design to add simulation capabilities without altering core functionality; however, building a fully featured simulation panel will require further development effort from UI designers and educators to create an intuitive simulation control interface and robust feedback mechanisms.
- FR4: Search Interface
  - While our system currently offers basic functionality for accessing example models, a fully interactive and searchable example library that is well-categorized remains a planned enhancement.
  - This add-on would require further development to curate content, design an intuitive search interface and integrate it into the tool, tasks that would likely involve substantial input from educators and content creators.
- FR7: Embedded User Guide with Searchable FAQ
  - The current implementation includes a basic help feature; however, a detailed embedded user guide with a fully searchable FAQ section is an add-on feature.
  - This enhancement would improve usability by offering comprehensive documentation and troubleshooting support, but it demands additional content

creation and interface design work, which goes beyond the core functionality.

- FR9: Real-Time Online Collaboration with Role-Based Permissions
  - While basic real-time synchronization is implemented, a fully featured online collaboration module with dynamic role-based UI adjustments is considered an add-on; this feature will further refine user interactions by clearly differentiating contributions and enforcing editing permissions.
  - The modular architecture supports the integration of enhanced collaboration tools, ensuring that advanced features such as conflict resolution and personalized UI elements can be added as separate modules without affecting the core real-time update mechanism; however, implementing these advanced functionalities will require additional development work from the collaborative systems team and educators.
- FR10: Offline Collaboration and Conflict Resolution
  - The current system supports basic model persistence and real-time updates, but comprehensive offline collaboration with local caching and a user-friendly conflict resolution interface is planned as an add-on enhancement.
  - By designing offline collaboration as an add-on, our architecture ensures that modules for local data storage and synchronization can be integrated later without requiring changes to the core online collaboration framework; however, this enhancement will demand extra development effort from both frontend and backend teams to implement reliable offline caching and an intuitive conflict resolution mechanism.
- FR12: External Gamification and Integration Interface

  - Integration with an external gamification platform featuring leaderboards, badges and progress tracking is considered an optional enhancement rather than a core requirement.
  - This feature would involve interfacing with third-party APIs to transmit modelling activity data and display gamification elements externally, and it would require extra development effort and coordination with external system providers.

- FR13: Visual Annotations and Overlays

  - The core system already supports basic visual annotations, but a comprehensive module for advanced feedback including customizable overlays for grading, validation and plagiarism detection is regarded as an add-on enhancement.
  - Developing this add-on leverages our modular frontend design, enabling advanced annotations to be implemented and updated independently, thereby enhancing the overall user experience; however, delivering a fully detailed annotation system will require additional work from content developers and UX designers to define comprehensive feedback criteria and ensure seamless integration with the existing model rendering.

- FR15: CI/CD Integration for Frontend Development

  - Although our architecture currently supports manual testing, the integration of automated build, test and deployment pipelines is not realized.
  - This CI/CD enhancement would streamline development, improve code quality and prevent regressions, but it is considered an add-on that requires further investment in automation tools and infrastructure.

- FR16: Interaction Tracker and Feedback Module

- A comprehensive module for tracking user interactions and gathering real-time feedback is envisioned as an add-on. Such a module would provide detailed analytics to help refine the system and guide instructional improvements.
- However, its implementation would require significant additional work to integrate with the existing architecture and to design a user-friendly dashboard for educators and developers.

This chapter begins with Section 4.1, which covers the goals and principles behind the frontend architecture. Section 4.2 then explains the technologies used and the rationale behind their selection. Section 4.3 provides an architecture overview, presenting the high-level structure and key components of the system. Section 4.4 discusses the reusability and extensibility of the frontend, focusing on common utility modules and the process of creating a new language. Finally, In Section 4.5, we validate the core frontend functionalities.

4.1 Goals and Principles

Our architecture of the modelling tool revolves around six core principles:

4.1.1 Support for Multiple Modelling Languages

A key requirement is the ability to handle various modelling languages (e.g., Class Diagrams, State Diagrams or Sequence Diagrams) without demanding an extensive development effort. Each language can be added as a self-contained set of modules. Modules of one language will have no dependencies on modules of another language. This modular approach ensures that introducing new modelling languages does not impact the already existing ones.

4.1.2 Reusability of UI Components

Rather than creating specialized elements for every language separately, out framework provides generic UI components that implement common functionality required by editors, such as different shapes for representing nodes, different configurations for displaying links between shapes, and shared logic for drag-and-drop, selection or context menus. By crafting these elements to be reusable, developers can maintain consistency across the diagram types of different modelling languages and reduce code duplication.

4.1.3 Extensibility for Developers

The design allows new modelling languages and features to be integrated without disrupting existing functionality. Instead of modifying existing modules, developers can create fresh ones or import from shared libraries. This “create, don’t edit” mindset fosters parallel development, clear version control and minimal risk of breaking older features.

4.1.4 Maintainability and Scalability

A modular structure ensures that updates or expansions such as adding new modelling languages and diagram types or refining existing ones can occur with minimal impact on the rest of the system. Over time, as the tool grows to accommodate larger user bases or additional modelling features, this clean separation of concerns helps keep maintenance overhead low and performance stable.

4.1.5 Real-Time Collaboration

Because the tool must handle multiple simultaneous users, real-time synchronization is central. Any model edits by one user are quickly propagated to all others, enabling interactive and efficient teamwork. Incremental diagram updates and lightweight broadcasts help keep collaboration smooth, avoiding disruptive full refreshes or out-of-date local views.

4.1.6 Security in Architecture

Security underpins the entire design. Only structured, authorized modifications are allowed, preventing invalid or malicious changes to the model. Each user action is controlled by predefined operations, ensuring that the system maintains data integrity and model consistency at all times. By default, unauthorized direct modifications or bypassing of the tool's logic are not permitted, supporting both reliable multi-user sessions and long-term stability of models.

4.2 Technologies and Selection Reasons

The system is primarily built using the following technologies, each chosen for its efficiency, scalability and seamless integration within our web-based modelling tool:

1. **GoJS:** A JavaScript library for interactive diagrams, providing essential graph-based visualization and event-driven interactions [64].

   Reasons for Selection: We chose GoJS over alternatives like D3.js and JointJS because it provides built-in support for diagramming elements such as nodes, links

and hierarchical structures, significantly reducing development effort [65]. Unlike D3.js, which requires extensive custom implementations for each diagram type, GoJS offers a structured API with predefined templates for classes, associations, and enumerations, aligning with our reusability and extensibility principles. A key advantage of GoJS is its model-view architecture, where a single underlying data model[1] can be visualized by multiple views. This allows for dynamic updates, ensuring that any modification to the data model is instantly reflected in all associated views, reducing manual synchronization overhead. Additionally, GoJS provides efficient real-time updates, built-in undo/redo support and interactive event handling, making it a robust choice for an intuitive, user-friendly modelling tool.

2. **JavaScript:** Ensures maintainability and type safety, reducing runtime errors and improving development efficiency.

Reasons for Selection: We chose JavaScript over TypeScript due to its simpler setup, native browser execution and flexibility in handling dynamic UI elements [66]. TypeScript's static typing adds overhead with an additional compilation step, while JavaScript enables faster prototyping and real-time updates without dependencies. Given our tool's focus on interactive modelling, JavaScript's dynamic nature allows seamless integration with GoJS and WebSockets, avoiding compatibility issues. Unlike Java or C++, JavaScript eliminates installation barriers, ensuring cross-platform accessibility and efficient asynchronous operations, making it the most suitable choice for a web-based modelling tool.

[1]Not to be confused with the models that we are talking about in model-driven engineering.

Performance Considerations: JavaScript's event-driven model is inherently well-suited for real-time updates and UI interactions, while Java and C++ are better suited for backend and system-level applications where memory management and performance-intensive computations are critical. While TypeScript provides better maintainability with static type checking, it introduces an additional compilation step, which could add delays in development iterations. In contrast, JavaScript's dynamic execution eliminates the need for pre-compilation, ensuring faster execution cycles and immediate feedback during development.

Developer Efficiency and Modularity: JavaScript provides a large ecosystem of libraries and frameworks, which enhances productivity and reusability. A Java or C++ frontend would require significantly more boilerplate code for UI rendering and interactivity, whereas JavaScript enables a concise, modular structure with built-in support for asynchronous operations and WebSockets. While TypeScript enforces stricter type definitions, reducing runtime errors, JavaScript's dynamic flexibility allows faster iteration cycles and seamless debugging without needing to recompile.

3. **Websockets:** Enables real-time synchronization, ensuring that modifications are instantly reflected across multiple active sessions.

Reason for Selection: WebSockets were chosen over AJAX polling and Server-Sent Events (SSE) because they provide bi-directional, low-latency communication, essential for real-time updates in collaborative modelling [67].

Comparison with AJAX Polling: AJAX requires frequent HTTP requests to fetch updates, which increases server load and response time, making it inefficient for

real-time interactions. WebSockets, on the other hand, maintain an open connection, ensuring that only the required data is transmitted, reducing bandwidth usage and improving performance.

Comparison with SSE: Server-Sent Events (SSE) only support one-way communication (server to client), which is insufficient for our needs, as users must also send updates when modifying diagrams.

4. **Representational State Transfer (REST) API:** REST APIs play a crucial role in our architecture by enabling structured, stateless communication between the frontend and backend, ensuring efficient data retrieval.

   Reasons for Selection: We selected REST APIs over alternatives like GraphQL, gRPC and SOAP due to their simplicity, scalability and seamless integration with our JavaScript-based frontend [68]. REST's stateless communication ensures efficient handling of multiple concurrent users. Unlike GraphQL, which introduces query complexity and over-fetching risks, REST follows a predictable, endpoint-based approach, ensuring optimized communication between the frontend and backend. Compared to gRPC, which requires strict schema enforcement and binary encoding, REST's JSON-over-HTTP format is more human-readable and widely supported across web environments. SOAP, on the other hand, has significant XML overhead and complexity, making REST the lightweight and efficient choice for our use case.

5. **HTML and CSS:** Provides a structured, responsive and efficient UI design.

   Reasons for Selection: We opted for a custom component-based design in vanilla JavaScript rather than using frameworks like React, Angular or Vue, because this

approach offers full control over the UI while keeping the system lightweight and framework-independent. By using plain JavaScript, we are able to directly access the full power of GoJS for diagramming, which comes with built-in, reusable UI components that facilitate the creation and manipulation of complex diagrams [69].

Comparison with React/Angular/Vue: While React and Angular provide built-in component structures, they introduce framework dependencies that require additional learning and performance overhead. Since our architecture is highly interactive but not state-heavy, using custom JavaScript components ensures better maintainability and direct browser execution without a virtual DOM overhead. Unlike React's JSX syntax, we use standard HTML elements and JavaScript event-driven updates, making it framework-agnostic and extensible.

4.3 Architecture Overview

4.3.1 High-Level Structure

Figure 4.1 illustrates the layered design of the frontend, showing how the application is split into an Application Layer (top), a Visualization & Interaction Layer (middle) and a Communication Layer (bottom). Each module's color indicates whether it is generic (pink), language-specific (green) or part of the Perspective Configuration (blue), which allows modelling languages to be configured for different teaching purposes. We mention perspective configuration here in the architecture at a high level. All perspective-related details will be discussed in Chapter 5.

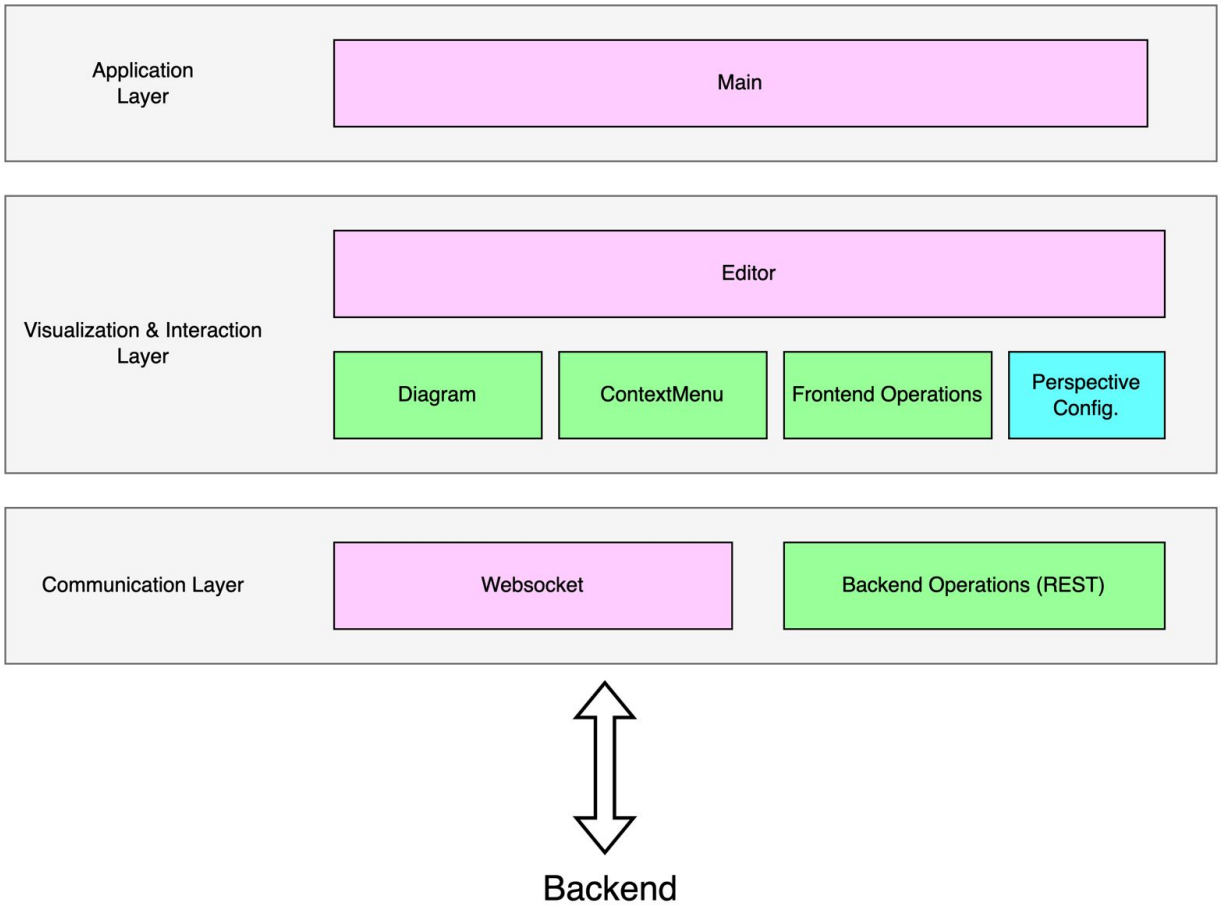

Figure 4.1: High-Level Frontend Architecture

4.3.1.1 Application Layer

The *Application Layer* appears at the top and contains the generic *Main* module, which is highlighted in pink. This *Main* module coordinates the tool's startup process by initializing the environment when the user launches the application.

4.3.1.2 Visualization & Interaction Layer

Beneath the *Application Layer*, the *Visualization & Interaction Layer* houses both generic and language-specific modules. The *Editor* (pink) is generic, and responsible for capturing user events (clicks, drags, right-clicks) and makes use of the language-specific modules. The *Diagram* (green) defines how nodes and links appear for a given language (e.g., classes, states), while the *ContextMenu* (green) provides language-specific actions (e.g., "Add Class", "Add State") that the *Editor* can display when the user right-clicks on a model element or on the background of the editor. Additionally, *Frontend Operations* (green) handle language-specific logic on the client side that react to notifications received from the backend (e.g., creating or deleting model elements), ensuring that each diagram type can define its own ways of reacting to notifications without the need to modify the *Editor* or *Main* modules. Although the Perspective Configuration is also language-specific and loaded generically; it simply contains references to language-specific concepts and model edit operation names, which are elaborated in Chapter 5.

4.3.1.3 Communication Layer

At the bottom we have the *Communication Layer* which ensures real-time synchronization and model persistence. The *Websocket* module (pink) is generic. It listens for broadcast notifications from the backend and forwards them to the *Editor* so that every connected client sees the latest model changes visualized in their frontend. The *Backend Operations* (green) module provides language-specific REST calls. By isolating these operations in a language-specific module, the architecture accommodates a variety of modelling languages while preserving the same overall flow for user interactions and real-time updates.

4.3.2 Main Interaction Flows

This section focuses on how the modules interact at runtime. We divide the interactions into three main flows: *Initialization*, *User Actions* and *Real-Time Updates*.

4.3.2.1 Initialization Flow

Figure 4.2 illustrates how the system prepares the modelling environment when the user first opens the editor. The process begins with the user launching the tool, prompting the *Main* (generic) module to call an `init()` function on the *Editor*. The *Editor* then requests a diagram template from the *Diagram* (language-specific) module. In response, the *Diagram* module creates the `GoJS templates` and applies them to *GoJS* (e.g., by calling `applyTemplate(...)`), which defines how nodes and links will be rendered. Next, the *Editor* retrieves model data by calling `getModel()` on the *Backend Operations* (language-specific) module, which makes a REST request to the Backend to fetch the stored diagram state. Once the *Editor* receives the model, it instructs the *Diagram* module to `setupModel(...)`. The *Diagram* then calls `setupGoJsGraph(...)` on *GoJS* to populate the actual nodes and links in the browser. By separating each responsibility, such as visual template creation, model fetching and final rendering, this initialization process ensures that the *Editor* and *Main* modules remain generic, while language-specific details are cleanly encapsulated in language-specific modules.

4.3.2.2 User Actions Flow

Figure 4.3 illustrates how the system handles user interactions, particularly how a right-click on the diagram triggers a language action. First, the user right-clicks on the diagram and *GoJS* captures this event. It then calls the generic *Editor's* `handleClick()` method to

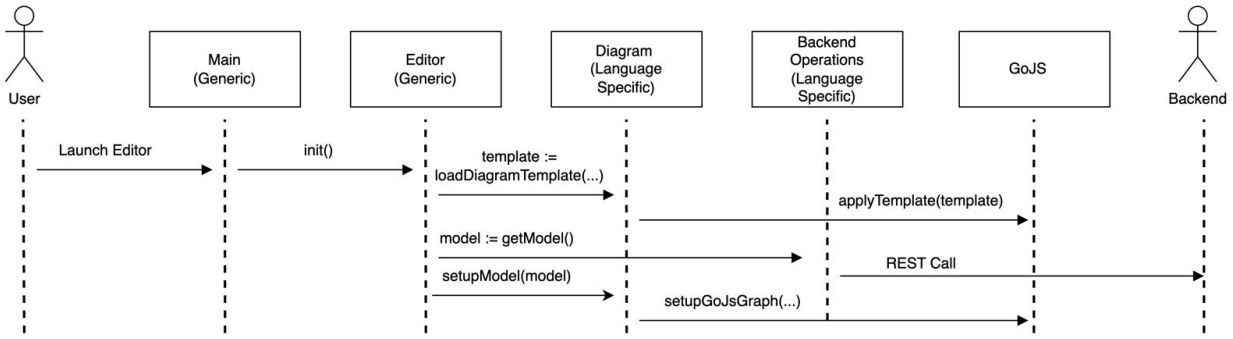

Figure 4.2: Initialization Flow

initiate the process. The *Editor*, in turn, invokes the language-specific *ContextMenu module* (e.g., by calling `getMenu()`) to retrieve a list of menu items associated with the current diagram context. Once the *Editor* instructs *GoJS* to display these items, the user selects an action, such as "Add Class". At that moment, *GoJS* triggers the callback function registered in the *ContextMenu* module for the chosen menu item. Unlike the initial event, which passed through the *Editor*, this subsequent action bypasses the Editor and flows directly to the *ContextMenu's* callback. The callback then calls a function in the Backend Operations module (e.g., `addState(stateName)`), which finally sends a stateless REST request to the backend to update the model.

4.3.2.3 Real-Time Update Flow

Figure 4.4 illustrates how the system synchronizes model changes across all connected clients when the *Backend* updates its model (e.g., by adding a new class or state). First, the *Backend* detects a change in a model, and broadcasts the to each connected client, effectively calling something like `broadcastChange()` on every *Websocket* (generic) instance that corresponds to a client session linked to the model that was modified. On the frontend, the *WebSocket* then notifies the *Editor* (generic) with a message such as

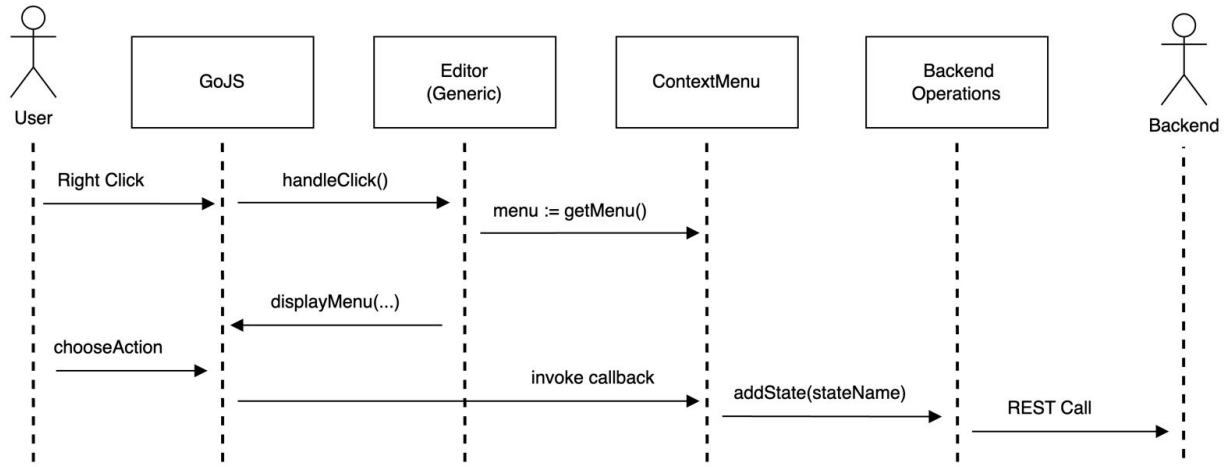

Figure 4.3: User Actions Flow

"classAdded". The *Editor*, upon receiving this notification (e.g., via a method like `notify("classAdded")`), invokes an operation provided by *Frontend Operations* (language-specific) function, for instance, `processWebSocketMessage("classAdded")` to interpret the message data and determine how the diagram should be updated. Finally, the *Frontend Operations* module calls a method like `updateGoJSGraph()` in *GoJS* to add or modify the relevant nodes and links. By keeping the *WebSocket* and *Editor* generic, the system accommodates any modelling language, while each language-specific *Frontend Operations* module applies the precise logic required to incorporate the new or changed elements into the local diagram.

4.4 Reusability and Extensibility of the Frontend

A principal objective of the frontend architecture is to enable new modelling languages to be integrated or existing ones to be extended without altering the core modules such as *Main*, *Editor* and *Websocket* or affecting already existing language-specific modules. The

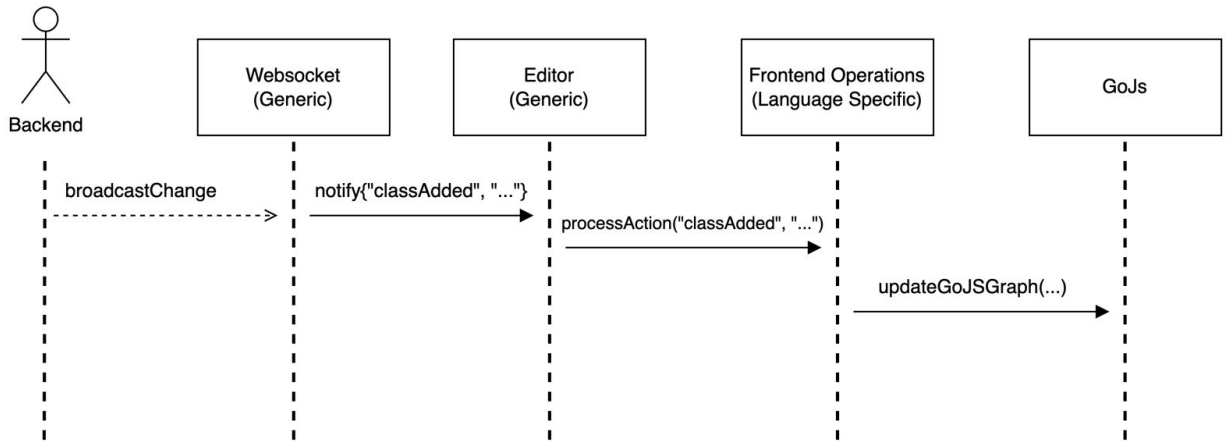

Figure 4.4: Real Time Updates Flow

architecture provides common utility modules for *Diagram*, *ContextMenu*, *Frontend Operations* and *Backend Operations* ensuring developers can reuse core logic rather than duplicating it. Whenever a new language is introduced, developers create language-specific modules that can import these common utilities. This section provides common utilities for each major module type and provides a step-by-step look at how a new language is typically introduced.

4.4.1 Common Utility Modules

The system includes shared utility modules that centralize repeated code and ensure consistency across all languages. For instance, a developer has access to:

4.4.1.1 Diagram Utilities

This module holds standard shape definitions, node styling and color palettes. It includes functions like `createDefaultNodeStyle(options)` that configure uniform styling.

4.4.1.2 Context Menu Utilities

This module provides generic functions to build menu items and prompt user input, such as `createMenuItem(label, action)`.

4.4.1.3 Frontend Operations Utilities

This module centralizes client-side logic, e.g., local model transformations or validations. These common utilities centralize shared code so that each language-specific module imports them, ensuring consistency and reducing maintenance overhead.

4.4.1.4 Backend Operations Utilities

This module handles standardized HTTP requests with a function such as `safeFetch(url, options)` that checks response statuses and parses JSON uniformly.

These utilities are imported by the new language-specific modules, so the developer does not have to duplicate common logic.

4.4.2 Integrating a New Modelling Language

To integrate a new modelling language into the frontend, the developer does not modify any of the generic modules (*Main*, *Editor*, *Websocket*). Instead, the developer creates the *Diagram*, *ContextMenu*, *FrontendOperations* and *BackendOperations* modules for the new language, importing any code needed from the common utility modules. The interfaces for the modules are shown in Figure 4.5.

In the following subsections, we illustrate this process using a concrete example of adding support for a state diagram language.

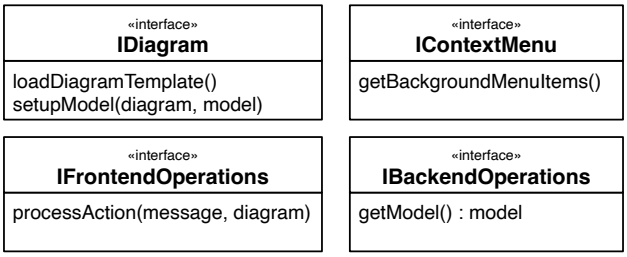

Figure 4.5: Interfaces for Integrating a New Modelling Language

4.4.2.1 Diagram Module Example

In the *Diagram* module (see code sample 4.1) for the *State Diagram* language, the interface function `loadDiagramTemplate()` is designed to provide the *Editor* with the necessary visual templates to render the diagram. The module begins by importing GoJS and a shared utility function, `createDefaultNodeStyle()`, from the common utilities module `diagramCommonUtils`, which guarantees that all diagrams maintain a consistent look and feel. Within `loadDiagramTemplate()`, a shorthand alias $ is created for `go.GraphObject.make`, a factory function provided by GoJS that simplifies the creation of diagram objects. The code then calls `createDefaultNodeStyle()` with an option to set the fill color to light blue, which returns a standard style object used in the node template. The node template is defined using a call to $ with parameters that specify an automatic layout `("Auto")` and include the default style. In this context, the "Auto" layout ensures that each state node is arranged efficiently, while the default style renders it as a rounded rectangle with a text block inside, where the text is bound two-way to the node's `stateName` property. This binding ensures that any changes made to the state name in the diagram immediately update the underlying data model and vice versa.

For transitions, the link template is defined using bezier-based routing instead of orthogonal routing, which better captures the fluidity of state transitions. The template

specifies a stroke width for the link shape and uses GoJS's built-in arrowhead, commonly known as the `"Standard"` arrowhead, which is appropriate for indicating transitions between states in a state diagram. Finally, the function returns an object containing both the node and link templates.

The second interface function, `setupModel()`, takes the configured diagram and a model data object (comprising arrays of nodes and links) and initializes the diagram's model by creating a `new go.GraphLinksModel`. This step integrates the current state of the model data retrieved from the backend into the visual display, completing the initialization process for the State Diagram.

Code Sample 4.1: Diagram Module for State Diagram

```
import * as go from "gojs";
import { createDefaultNodeStyle } from "../common/diagramCommonUtils";
export function loadDiagramTemplate() {
  const $ = go.GraphObject.make;
  const defaultNodeStyle = createDefaultNodeStyle({ fill: "lightblue" });
  const nodeTemplate = $(
    go.Node,
    "Auto",
    defaultNodeStyle,
    $(
      go.TextBlock,
      { margin: 8, editable: true },
      new go.Binding("text", "stateName").makeTwoWay()
    )
  );
  const linkTemplate = $(
    go.Link,
    { routing: go.Link.Bezier, curviness: 20 },
    $(go.Shape, { strokeWidth: 2 }),
    $(go.Shape, { toArrow: "Standard" })
  );
  return { nodeTemplate, linkTemplate };
}
export function setupModel(diagram, modelData) {
  diagram.model = new go.GraphLinksModel(modelData.nodes, modelData.links);
}
```

4.4.2.2 ContextMenu Module Example

This language-specific module defines the context menu for the *State Diagram* language in the interface function *getBackgroundMenuItems()*. It returns an array of menu items, each with an action callback that *GoJS* invokes when the user selects an item. In the code sample 4.2, one of these items allows a user to add a new state to the diagram. Upon selection, the callback prompts the user for a state name, then calls the corresponding *Backend Operation* to persist the new state. By encapsulating these definitions within a dedicated module, each modelling language can specify its own actions and callbacks, leaving the generic *Editor* unaltered.

Code Sample 4.2: ContextMenu Module for State Diagram

```
1  import { getUserInput } from "../common/contextMenuCommonUtils";
2  import * as BackendOps from "./operationsState.js";
3  export function getBackgroundMenuItems() {
4    const items = [];
5    items.push({
6      label: "Add State",
7      action: async () => {
8        const stateName = await getUserInput("Enter state name:");
9        if (stateName) {
10         await BackendOps.addState(stateName);
11       }
12     },
13   });
14   return items;
15 }
```

4.4.2.3 Frontend Operations Module Example

In our real-time update flow, once the backend updates its model, it broadcasts a change (e.g., “stateAdded”) to all connected clients. On the frontend, the generic *WebSocket* receives this broadcast and notifies the *Editor* using a function such as

`notify("stateAdded")`. The *Editor*, which remains generic and unaware of language-specific details, then calls the interface function `processAction()` from the language-specific *Frontend Operations* module (refer code sample 4.3), passing along the message and the current GoJS diagram instance. This function serves as the central handler for processing real-time notifications. It examines the type of the incoming message: if the message type is "stateAdded", it calls `insertStateNode()`, which incrementally adds the new state to the diagram model using `diagram.model.addNodeData()`; if the message is "stateUpdated", it calls `updateNodeInDiagram()`, a shared utility that updates the properties of an existing node based on the provided data; and if the message is "stateDeleted", it calls `removeStateNode()`, which finds and removes the node with the corresponding ID from the model.

Code Sample 4.3: Frontend Operations Module for State Diagram

```
import { updateNodeInDiagram } from "../common/frontendOpsCommonUtils";
export function processAction(message, diagram) {
  switch (message.type) {
    case "stateAdded":
      insertStateNode(diagram, message.data);
      break;
    case "stateUpdated":
      updateNodeInDiagram(diagram, message.data);
      break;
    case "stateDeleted":
      removeStateNode(diagram, message.data.id);
      break;
    default:
      console.warn("Unhandled message type:", message.type);
  }
}
function insertStateNode(diagram, stateData) {
  diagram.model.addNodeData(stateData);
}
function removeStateNode(diagram, stateId) {
  const nodeData = diagram.model.nodeDataArray.find(nd => nd.id === stateId);
```

```
    if (nodeData) {
      diagram.model.removeNodeData(nodeData);
    }
  }
```

4.4.2.4 Backend Operations Module Example

This *Backend Operations* module is responsible for all REST calls related to the *State Diagram* language. It imports a shared function, `safeFetch`, from a common utility module that standardizes error handling and JSON parsing (see code sample 4.4). The function `addState(stateName)` constructs a POST request to the endpoint `/api/stateDiagram/addState`, sending the new state's name as a JSON payload. When the backend successfully creates the new state, it returns a JSON response typically including the new state's ID which this function then returns to the caller. Similarly, `deleteState(stateId)` issues a DELETE request to remove a state identified by its ID from the backend model. The module also includes a `getModel()` function, which is critical during the initialization flow. When the *Editor* needs to load the existing diagram, it calls `getModel()` from this module, which sends a GET request to `/api/stateDiagram/model`. The backend responds with the complete model data, allowing the *Editor* to set up the diagram accordingly by calling `setupModel()` on the language-specific *Diagram* module.

Code Sample 4.4: Backend Operations Module for State Diagram

```
import { safeFetch } from "../common/operationsCommonUtils";
export async function addState(stateName) {
  const response = await safeFetch("/api/stateDiagram/addState", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ stateName }),
  });
  return response;
}
```

```
10    export async function deleteState(stateId) {
11      const response = await safeFetch(`/api/stateDiagram/deleteState/${stateId}`, {
12        method: "DELETE",
13      });
14      return response;
15    }
16    export async function getModel() {
17      const response = await safeFetch("/api/stateDiagram/model", {
18        method: "GET",
19        headers: { "Content-Type": "application/json" },
20      });
21      return response;
22    }
```

4.5 Validation

In this section, we evaluate the core frontend functionalities implemented in our modular architecture. Our validation approach combines performance measurements, design-by-construction arguments and minimal empirical testing where feasible. Each evaluation is tailored to demonstrate that the frontend provides a responsive, user-friendly experience while maintaining modularity and extensibility.

4.5.1 Experimental Setup

All validation tests were performed under a controlled environment to ensure accurate and repeatable results. The following specifications outline the test setup:

4.5.1.1 Hardware & System Specifications

- **Device**: MacBook Air (M2)
- **Processor**: Apple M2 (8-core CPU, 10-core GPU)
- **Memory**: 8GB Unified RAM

- **Storage**: 256GB SSD
- **Operating System**: macOS Sequoia 15.3
- **Network Connection**: Stable 500 Mbps Fiber Connection

4.5.1.2 Software & Testing Tools

- **Browser**: Google Chrome 133.0.6943.126 (Latest Stable Release)
- **Developer Tools Used**:

    1. **Chrome DevTools** (Performance profiling, network logs, local storage verification)
    2. **WebPageTest** (Load time and resource consumption analysis)
    3. **JavaScript Performance Timers** (`performance.now()`)
    4. **Selenium & Puppeteer** (Automated UI testing)
    5. **Jest** (Unit testing and snapshot testing for verifying UI component integrity and preventing unintended changes)

All tests were conducted on a local development environment. The REST API was hosted locally and WebSockets were tested using a dedicated test server to simulate multi-user collaboration.

4.5.2 Validation of FR3: Multi-Language Notation Support

To validate this, we adopted a design-by-construction approach. To develop the frontend we implemented full support for the *Class Diagram* language. To validate FR3, we additionally implemented a prototype of the *State Diagram* modules. A thorough code review confirms that language-specific modules are isolated from core components. Visual

templates for nodes and links are rendered exactly as specified by each language module. We verified this by comparing the rendered output against predefined templates, confirming complete consistency. Performance logs show stable resource usage during language switches. The system does not require any modifications to the *Editor* when new language modules are integrated. Each language module loads and operates independently, ensuring that additional languages can be added seamlessly.

4.5.3 Validation of FR5: Intuitive User Interface Layout

We instrumented the UI to measure the number of clicks required to perform essential operations. In our controlled tests with over 50 trials, the average click count for performing operations was found to be no more than three, which confirms that the interface minimizes interaction overhead [53]. The grouped arrangement of controls and the established visual hierarchy were verified through internal code reviews, ensuring that related functions are easily accessible. Our observations show that each click effectively triggers the corresponding action without unnecessary delays or redundant steps. Overall, our findings confirm that the intuitive UI layout is effective in minimizing user effort, while a broader user study for further validation remains outside the scope of this thesis.

4.5.4 Validation of FR6: Seamless User Interactions with Minimal Latency

We instrumented the event handling code to measure the round-trip delay from when a user clicks “Create Class” to when the update is rendered in the GoJS diagram. Over 50 iterations, our measurements yielded an average delay of 75 milliseconds with a standard deviation of 10 milliseconds, which clearly demonstrates that our asynchronous processing and data-binding

mechanisms are optimized for responsiveness. Tests under simulated concurrent interactions confirmed that the delay remains below 90 milliseconds, ensuring that multiple simultaneous events do not affect performance. These quantitative results provide concrete proof that our frontend design meets the requirement for seamless, low-latency interactions.

4.5.5 Validation of FR8: User-Friendly Setup and Responsive Model Canvas

The frontend's user-friendly setup and responsive model canvas are validated by simulating dynamic browser resizing and observing the layout's stability. Controlled tests confirm that the canvas automatically adjusts its layout, with all UI elements repositioning correctly without overlap. The design uses flexible CSS grids and media queries, ensuring consistent scaling across browser sizes. Performance logs indicate that the canvas redraws smoothly with no noticeable delays during resizing. These results provide concrete proof that the responsive design meets our intended specifications.

4.5.6 Validation of FR14: Frontend Modular Component System

This requirement is validated by analyzing the code structure to ensure that core modules remain independent from language-specific modules. We evaluate this by reviewing module dependency diagrams that clearly show minimal coupling between generic components and language-specific modules. The modular architecture is further validated by successfully adding a new language-specific modules, i.e., the ones for *State Diagrams*, without modifying the generic core. Performance profiling indicates that the modular design introduces minimal overhead, ensuring that system efficiency is maintained.

4.6 Summary

In this chapter, we developed a modular frontend architecture for our web-based modelling tool, supporting the core frontend requirements and the additional add-on frontend requirements, as outlined at the beginning of this chapter. This architecture not only meets these frontend requirements, but also supports the seamless integration of the add-on features mentioned in future work section 6.2. Our validation experiments confirm that the core functionalities perform as intended. In the next chapter, we will explore the DSL for Language Configuration, which will further enhance the system by allowing educators to dynamically adapt a language to a specific teaching context.

Chapter 5

Language Customization through a Domain-Specific Language

This chapter proposes a Domain-Specific Language (DSL) for adapting the frontend of a modelling language to a specific teaching context / teaching needs. It hence addresses one of the key functional requirements from chapter 3, namely FR11: Adaptive Frontend Modelling Interface, and of course does so in a modular way in order to comply with FR14: Frontend Modular Component System. This chapter begins with Section 5.1, which reviews the CORE framework, the CORE DSL for specifying languages and perspectives, and its limitations when applied in a teaching context; Section 5.2 then explains how we extended the CORE metamodel to capture high-level language concepts and link them to language actions; Section 5.3 describes our extension of the CORE metamodel and DSL grammar for both language definitions and perspective configurations; Section 5.4 presents concrete DSL examples for a *Class Diagram* language as well as for a *Domain Modelling* and a *Design Modelling* perspective; Section 5.5 details the code generation process that

instantiates the augmented metamodel based on the DSL; Section 5.6 shows how the frontend architecture adapts its visual templates and context menus according to the generated perspective configuration and finally, Section 5.7 presents our validation.

5.1 Review of CORE and Limitations in Teaching

5.1.1 CORE Review

As detailed in Chapter 2, the CORE framework was originally developed to help modelling language creators add reusability features to their modelling languages. CORE defines a metamodel that focuses on capturing the essence of a modelling language. The developer of a specific modelling language can instantiate a *CORELanguage* and define the model edit actions that the language offers by creating corresponding *CORELanguageAction* objects. Moreover, CORE supports multi-view modelling by extending the metamodel with additional elements, such as *COREPerspective* and *COREPerspectiveAction*, which allow different modelling languages to be coordinated for a given purpose. This design enables a modelling language developer to not only specify the operations available within a language, but also to define how those operations interact within a unified perspective. However, while this action-based focus has proven effective for technical modelling, it does not fully address the unique needs of a teaching context.

5.1.2 Limitations for Teaching

In educational settings, the primary focus is on gradually introducing students to the underlying concepts of modelling languages rather than focusing on language operations alone. Teachers typically prefer to start with the basic concepts of a modelling language,

e.g., *Classes* and *Attributes* for *Class Diagrams*, and then progressively introduce more complex elements, e.g., Visibility and Associations, as students gain proficiency. However, the original CORE framework, with its emphasis on *LanguageAction* and *PerspectiveAction*, does not offer a mechanism for selectively hiding or revealing specific language concepts (for example, *Operation*, *Visibility* or *Navigability*). This limitation means that the framework lacks the flexibility needed to tailor the modelling environment to different teaching stages. In other words, while CORE is effective at enabling multi-view modelling and reusability, it falls short when educators require a simplified interface that gradually exposes new concepts and features of a modelling language. This shortcoming has motivated our extension of the CORE metamodel to include explicit *language concepts*, allowing for the creation of perspectives that can hide or expose specific features according to pedagogical requirements. Such an approach supports incremental learning and aligns the tool more closely with the needs of educators, ensuring that students are not overwhelmed by advanced features before they are ready.

5.2 Extending the CORE Metamodel with Language Concepts

5.2.1 Metamodel Extension

Our approach augments the original CORE metamodel by explicitly capturing *language concepts* in addition to language actions, thereby enabling more teacher-friendly definitions of modelling languages. As shown in Figure 5.1, we introduce a new class called *CORELanguageConcept* (highlighted in green) to represent language concepts. In the

original metamodel, the focus was on instantiating *CORELanguageAction* and *COREPerspectiveAction* to support multi-view modelling, but this approach did not readily allow educators to selectively expose or hide specific modelling concepts. By adding an association labeled *hasConcepts* between *CORELanguage* and *CORELanguageConcept*, a single language can now define multiple concepts, while another association, *conceptActionLink*, connects each language concept to the operations (*CORELanguageAction*) that act upon it or use the concept in some way. This extension ensures that a modelling language developer can declare both the actions *and* the fundamental concepts relevant to a language.

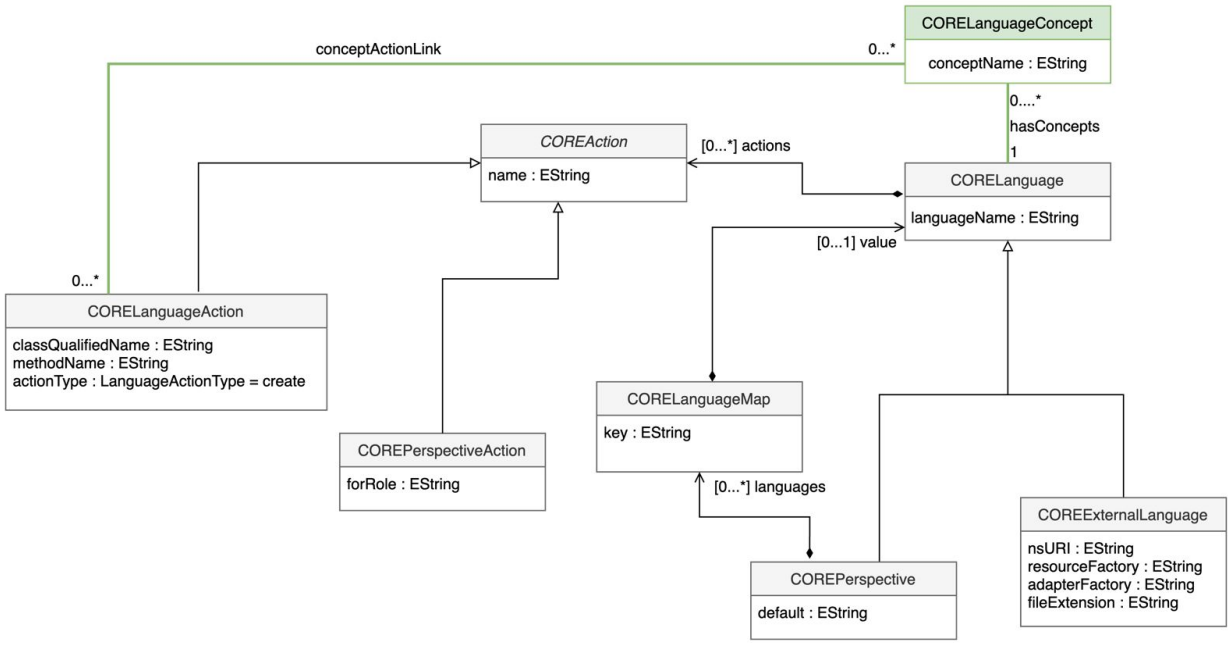

Figure 5.1: Metamodel Extension

5.2.2 Benefits of the Metamodel Extension

The primary benefit of augmenting the CORE metamodel with language concepts is that it enables both language designers and educators to tailor the modelling environment to their needs more precisely. Since *COREPerspective* inherits from *CORELanguage*, a teacher can now define a perspective that specifies which concepts are available (or hidden) in a particular perspective. For example, in an introductory course on class modelling, the teacher might choose to hide advanced concepts such as *Navigability* or complex *Association Classes* by simply not associating those concepts with the perspective. Conversely, for an advanced design course, all concepts can be exposed to provide a richer set of modelling features. This selective exposure not only simplifies the learning curve for beginners but also allows for a gradual introduction of complexity as students become more proficient.

5.3 Adapting the DSL

In this work, we extended the CORE DSL to support the *explicit declaration of high-level language concepts* as well as the *configuration of teaching perspectives*. This extension allows language designers to not only define the operational actions of a language, but also to declare its fundamental concepts without having to specify their internal details. In our DSL, the language definition now includes a dedicated block for concepts, and the language actions reference the concepts they use. This makes it possible for educators to later tailor the modelling environment by hiding advanced concepts, and hence also all linked language actions, when necessary.

5.3.1 DSL Grammar for Language Definitions

The DSL grammar for language definitions has been updated to include a `concepts` section as shown in Code Sample 5.1 on lines 11 to 13. The `actions` block is also extended with an `associated concepts` clause (see lines 31 to 33), linking each operation to its relevant high-level concept. This structure ensures that the code generator can instantiate the augmented CORE metamodel with both *language actions* and *language concepts*. The clear separation of these elements simplifies the development process and improves consistency across the language definition.

Code Sample 5.1: Xtext Grammar for Language Definitions

```
1   Language:
2       language name=ID {
3           rootPackage rootPackage=STRING;
4           packageClassName packageClassName=ID;
5           nsURI nsURI=STRING;
6           resourceFactory resourceFactory=STRING;
7           adapterFactory adapterFactory=STRING;
8           weaverClassName weaverClassName=STRING;
9           fileExtension fileExtension=ID;
10          modelUtilClassName modelUtilClassName=STRING;
11          concepts {
12              (concepts += Concept)*
13          }
14          language elements {
15              (elements += LanguageElement)*
16          }
17          (actions {
18              (actions += Action)*
19          })?
20      }
21  ;
22  Concept:
23      concept conceptName=ID {
24          // Additional configuration parameters can be included here if needed.
25      }
26  ;
27  Action:
28      create methodNameAndParameters=STRING {
29          metaclass : metaclass=ID;
```

```
30            classQualifiedName : classQualifiedName=STRING;
31            (associated concepts {
32                (associatedConcepts += Concept)*
33            })?
34        }
35    ;
```

5.3.2 DSL Grammar for Perspective Definitions

The perspective grammar is similarly extended to allow teachers to configure which language concepts are hidden. As seen in Code Sample 5.2, the `perspective` block includes a `hidden concepts` section (see lines 13 to 15) within each referenced language. This enables educators to create tailored perspectives that limit the modelling environment to only those concepts that should be used by the students for the specific modelling activity.

Code Sample 5.2: Xtext Grammar for Perspective Definitions

```
1     Perspective:
2         perspective {
3             name: name=STRING;
4             (default : isDefault=ID;)?
5             languages {
6                 (languages += LanguageRef)*
7             }
8         };
9     LanguageRef:
10        existing language name=ID {
11            "roleName" roleName=ID;
12            "rootPackage" rootPackage=STRING;
13            (hidden concepts {
14                (hiddenConcepts += HiddenConcept)*
15            })?
16            (actions {
17                (actions += PerspectiveAction)*
18            })?
19        };
20    HiddenConcept:
21        name=ID;
22    ;
```

5.4 DSL Examples

This section presents concrete examples that demonstrate how the extended DSL is used to define a *Class Diagram Language* and two distinct perspectives: a *Domain Modelling Perspective* and a *Design Modelling Perspective*. These examples illustrate how language designers declare core concepts and link them to operations, and how educators can customize the modelling environment by hiding undesired concepts.

5.4.1 Defining the Class Diagram Language

In this example, the DSL is used to define a *Class Diagram Language*. The language definition includes both traditional properties and a set of high-level language concepts. As shown in Code Sample 5.3 lines 10 to 19, the language declaration comprises a `concepts` section where fundamental concepts such as `Class`, `Attribute` and several advanced concepts (like `Operation`, `Visibility`, `Navigability`, `Association`, `AssociationClass` and `NaryAssociation`) are declared. The actions block then defines operations that are associated with these concepts. This comprehensive language definition serves as the basis for subsequent perspective configurations.

Code Sample 5.3: Class Diagram Language DSL

```
1   language classDiagram {
2       rootPackage "com.example.classdiagram";
3       packageClassName classDiagram;
4       nsURI "http://com.example/classDiagram";
5       resourceFactory "com.example.ClassDiagramResourceFactory";
6       adapterFactory "com.example.ClassDiagramAdapterFactory";
7       weaverClassName "com.example.ClassDiagramWeaver";
8       fileExtension cd;
9       modelUtilClassName "com.example.ClassDiagramModelUtil";
10      concepts {
11          concept Class;
12          concept Attribute;
```

```
            concept Operation;
            concept Visibility;
            concept Navigability;
            concept Association;
            concept AssociationClass;
            concept NaryAssociation;
        }
        language elements {
            // Additional language elements (omitted for brevity)
        }
        actions {
            create "createClass(String name)" {
                metaclass : Class;
                classQualifiedName : "com.example.Class";
                associated concepts { Class }
            }
            create "createAttribute(String name, String type)" {
                metaclass : Attribute;
                classQualifiedName : "com.example.Attribute";
                associated concepts { Attribute, Class }
            }
            create "createAssociation(String role)" {
                metaclass : Association;
                classQualifiedName : "com.example.Association";
                associated concepts { Association, Class }
            }
        }
    }
```

This snippet clearly demonstrates how language concepts and operations are declared. The language designer defines core concepts and then links specific actions to these concepts using the *associated concepts* clause. This DSL code is then processed by the code generator to instantiate the augmented CORE metamodel.

5.4.2 Defining a Domain Modelling Perspective

During early software development phases, it is a common activity to analyze the structure of a problem domain by specifying relevant domain concepts alongside their properties and relationships. While in theory any structural modelling language can be used for domain

modelling, practically most software development processes that incorporate domain modelling use a variant of class diagrams to express domain models.

Most of the time, though, only a subset of the class diagram language is used. For example, since in domain models the focus isn't on behaviour, classes in domain models typically don't contain operations, and associations do not show navigability, since classes are not assigned any behavioural responsibilities, and hence don't need to interact with other classes.

With our extension to CORE, a teacher can now define a *Domain Modelling Perspective* that exposes only the core concepts, such as `Class` and `Attribute`, and features such as `Operation`, `Visibility` and `Navigability` are hidden. Code Sample 5.4 shows how the DSL is used to define this perspective (concepts are hidden in lines 7 to 11).

Code Sample 5.4: Domain Modelling Perspective DSL

```
1  perspective {
2      name : "DomainModelling";
3      languages {
4          existing language classDiagram {
5              "roleName" domainRole;
6              "rootPackage" "com.example.classdiagram";
7              hidden concepts {
8                  Operation;
9                  Visibility;
10                 Navigability;
11             }
12         }
13     }
14 }
```

5.4.3 Defining a Design Modelling Perspective

Further down the development process of a system, class diagrams are commonly used to prescribe or document the implementation solutions of a system. This is typically called

design modelling. Design models should be comprehensive abstractions of a system's implementation. The modeller at this phase clarifies how domain-level relationships are implemented, assigns responsibilities to classes and as a result distributes the system's behaviour over the design classes.

In a *Design Modelling Perspective* therefore concepts like `Class`, `Operation` and `Association` are exposed, and so are `Visibility`, `Navigability`, etc. On the other hand, concepts that is hard to correlate with implementation solutions, such as `NaryAssociation` and `AssociationClass` and related language actions are intentionally omitted, as shown in Code Sample 5.5 lines 7 to 10.

Code Sample 5.5: Design Modelling Perspective DSL

```
1   perspective {
2       name : "DesignModelling";
3       languages {
4           existing language classDiagram {
5               "roleName" designRole;
6               "rootPackage" "com.example.classdiagram";
7               hidden concepts {
8                   AssociationClass;
9                   NaryAssociation;
10              }
11          }
12      }
13  }
```

5.5 Code Generation for Concept-Based Perspectives

The code generation process is a crucial component of our approach, as it transforms the high-level DSL definitions into concrete instantiations of the augmented CORE metamodel, i.e., *CORELanguage* and *COREPerspective*. This process ultimately produces a complete perspective configuration that the frontend uses at runtime to control which operations and

visual elements are active.

5.5.1 Process Overview

The code generator begins by parsing the DSL input, which now includes both language concept declarations and perspective directives. It identifies all the declared concepts (for example, `concept Class;` and `concept Attribute;`) and then examines the `hiddenConcepts` array specified within each perspective block. For every declared concept that is not listed as hidden, the generator creates an instance of a corresponding `LanguageConcept` element within the augmented CORE metamodel. At the same time, it establishes associations via `LanguageConceptActionLink` objects that link these concepts to the relevant language actions as defined in the `actions` block. As a result, the generator produces a comprehensive perspective configuration that reflects the high-level definitions provided by language designers and educators. This configuration, which is automatically generated from the DSL, encapsulates all allowed language concepts and their corresponding actions, ensuring that only the intended features are enabled at runtime.

5.5.2 Implementation Details

The implementation of the code generator involves several key steps. First, the generator tokenizes and parses the DSL input to extract both concept declarations and the perspective directives, including the `hiddenConcepts` specification. During the next phase, for each concept that is declared and not marked as hidden, the generator instantiates a `LanguageConcept` element. Simultaneously, it creates `LanguageConceptActionLink` associations to bind each instantiated concept with the language actions it is meant to influence. This association ensures that operations such as `createClass`,

`createAttribute` and others are properly linked to their respective concepts, as defined in the DSL.

Once all relevant concepts and associations are instantiated, the generator produces a perspective configuration file. This file contains flags that enable operations for the allowed concepts and disable those associated with hidden ones. The final perspective configuration is then loaded at runtime by the frontend modules, which use it to determine which visual templates are active, and which language edit operations should be displayed in the context menus. This automated instantiation process significantly reduces manual effort, minimizes errors and ensures consistency in how language and perspective definitions are generated.

5.6 Frontend Adaptation

We now review the main interaction flows already presented in subsection 4.3.2, but now extended in such a way that allows a teacher to customize a modelling language for a specific teaching purpose.

As shown in figure 5.2, when the user launches the editor, the *Main* module invokes the *Editor* to start up. The *Editor* then loads the *Perspective Configuration* (highlighted in green), which determines which language concepts and operations are allowed or hidden. In particular, the *Main* module either retrieves a default perspective or processes a user selection to determine which configuration the *Editor* should load. Using this configuration, the *Editor* calls the *Diagram* module to generate or modify GoJS templates. Next, the Editor requests the current model data via *Backend Operations*, which performs a REST call to the Backend. Finally, the *Editor* instructs the *Diagram* module to set up the diagram

with the retrieved data, and the *Diagram* applies the final updates in *GoJS*.

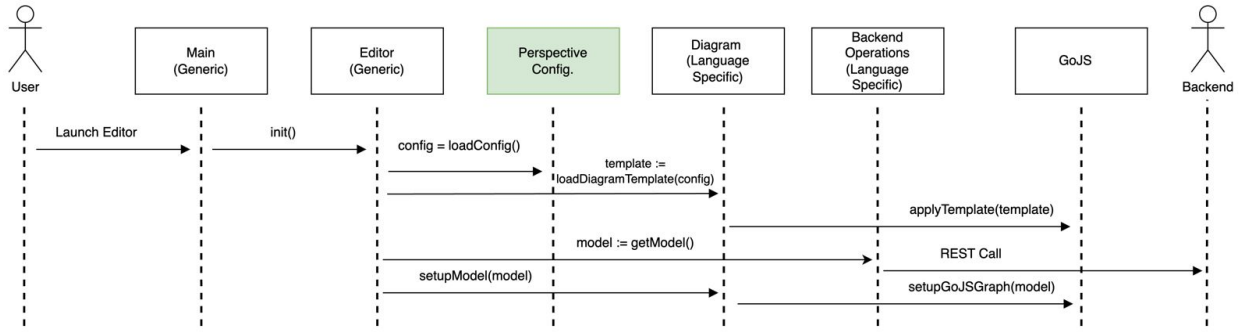

Figure 5.2: Dealing with Perspectives during the Initialization Flow

During the user actions flow as shown in figure 5.3, the *Editor* consults the *Perspective Configuration* to determine which actions should be displayed in the context menu. When the user right-clicks on the diagram, *GoJS* notifies the *Editor*, which in turn requests a filtered menu from the *ContextMenu* module using the perspective. Once the user selects an action, *GoJS* invokes the corresponding callback in *ContextMenu*, which call the appropriate function in *Backend Operations*.

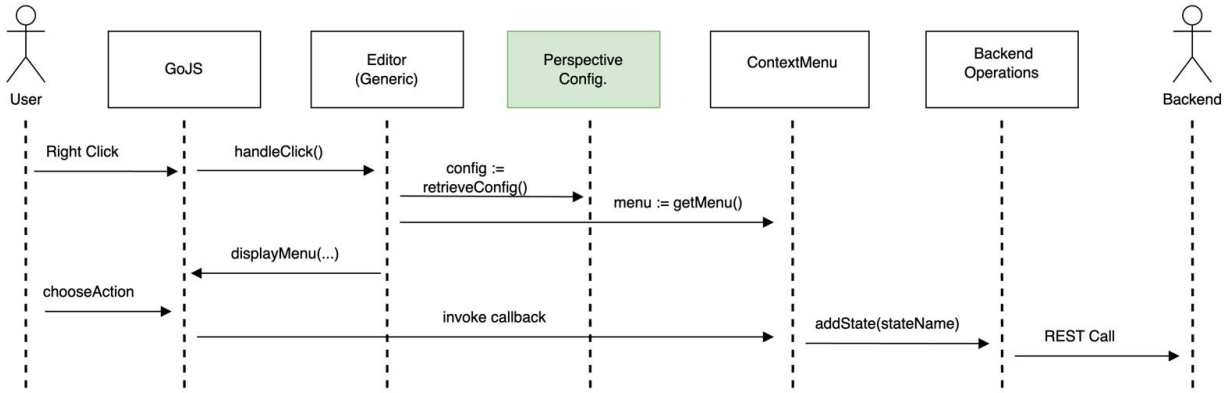

Figure 5.3: Dealing with Perspectives during the User Actions Flow

In the real-time update flow, the approach remains the same as in Chapter 4 because

perspective constraints do not alter the core notification sequence. Any concept not permitted by the active perspective is simply not displayed, so the overall process for broadcasting and handling model changes does not change. If a user's perspective hides certain concepts, those elements are never rendered for that user, while another user's view may display them if allowed. Consequently, we reuse the same figure 4.4 without modifications, as the real-time update logic remains unaffected by perspective rules.

Hence, the integration of the perspective configuration into our modular frontend architecture allows the system to dynamically adjust visual templates and context menus. This ensures that the modelling interface aligns with the educational goals set by the teacher, providing a progressively detailed environment that supports both novice and advanced users. This approach leverages the modular design described in Chapter 4 and the DSL extensions to deliver a truly customizable modelling tool.

5.7 Validation

This section validates the language customization capabilities enabled by our DSL and the seamless integration of the modular component system.

5.7.1 Validation of FR11: Adaptive Frontend Modelling Interface

The language customization interface is validated by comparing the DSL input with the generated perspective configuration. For instance, in the *Domain Modelling Perspective*, the DSL explicitly omits advanced concepts such as *Operation*, *Visibility* and *Navigability*. The resulting configuration file correctly contains flags (e.g., "hideOperation": true,

"hideVisibility": true, "hideNavigability": true), confirming that only the intended, fundamental elements are enabled. Similarly, the *Design Modelling Perspective* excludes complex constructs like *AssociationClass* and *NaryAssociation*, and the generated output reflects these omissions accurately. Our evaluation confirms that the DSL-generated configuration instantiates the augmented CORE metamodel precisely as specified by the DSL code. Code inspections and controlled tests demonstrate that the language customization interface accurately maps the teacher's DSL definitions to the corresponding perspective settings. This direct mapping offers concrete proof that our approach supports flexible, teacher-friendly customization.

5.7.2 Validation of FR14: Frontend Modular Component System

The modular component system is validated by confirming that the DSL-generated perspective configuration is seamlessly integrated into the frontend. Our evaluation shows that generic modules correctly load the perspective configuration, which is then propagated to language-specific modules. Detailed code analysis confirms that these modules dynamically adjust their visual templates and menu items according to the configuration flags. For example, when a perspective flag indicates that a certain operation is hidden, the corresponding UI element is omitted from the rendered diagram. Controlled integration tests verify that these customized settings are applied consistently across both the *Domain* and *Design Modelling* perspectives. This modular design ensures that any new language-specific module can be integrated without altering the generic core, proving the system's extensibility. Our observations demonstrate that the configuration-driven adaptation is implemented by design, offering robust and verifiable proof of modular integration.

5.8 Summary

In this chapter, we extended the CORE framework to better serve teaching by augmenting its metamodel with explicit language concepts and linking these concepts to their associated actions. Our enhanced DSL now allows language designers to define not only the operational aspects of a modelling language but also its high-level constructs, enabling educators to tailor the user interface by selectively hiding language features. We presented concrete DSL examples for a *Class Diagram* language and two perspectives, a *Domain Modelling Perspective* that omits concepts like *Operation*, *Visibility* and *Navigability*, and a *Design Modelling Perspective* that excludes *AssociationClass* and *NaryAssociation*. Furthermore, our code generator instantiates the augmented CORE metamodel based on these DSL definitions, and our frontend dynamically adapts its visual templates and context menus accordingly. Our validation also confirms that these contributions function as intended, providing a robust foundation for dynamic, teacher-friendly customization. Overall, these contributions facilitate a dynamic, teacher-friendly customization of the modelling tool, empowering educators to progressively introduce complexity and align the tool with their pedagogical objectives.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we developed a robust and teacher-friendly web-based frontend architecture for Modelling Tools for Teaching that meets a comprehensive set of requirements for usability, adaptability and extensibility. We began by formulating a detailed list of frontend requirements and designing a modular architecture that clearly separates concerns across the Application, Visualization & Interaction and Communication layers. Our work led to the development of reusable UI components, which facilitate the rapid creation of new modelling editors for diverse modelling languages without significant reimplementation. In addition, we extended the CORE framework by augmenting its metamodel with explicit high-level language concepts and we enhanced the Domain-Specific Language of the CORE framework to allow language designers to define these concepts along with their associated operations. This DSL-driven approach empowers educators to customize the modelling environment dynamically by hiding language

concepts and/or language operations in the editor, which is useful to gradually introduce modelling complexity as students progress, thus aligning the tool closely with pedagogical objectives. From the DSL, the adapted code generator produces CORE language and CORE perspective definitions that are then read by our generic frontend components to adapt the visual templates and context menus according to the teacher's needs.

While the contributions of this thesis do not address all the functional requirements for Modelling Tools for Teaching identified in [5], the framework architecture and design lay a solid foundation for future enhancements.

6.2 Future Work

Although we have implemented the core functionalities in Chapters 4 and 5, our architecture also supports several additional frontend requirements (FRs) mentioned at the beginning of Chapter 4, which are yet to be implemented as add-ons. The first future work task will be to implement these add-on requirements. Beyond these, we propose several innovative, teacher-centric enhancements that build on our current work and offer creative avenues to support educators in teaching modelling more effectively.

6.2.1 Customizable Themes and Accessibility Options

To further enhance the user experience, the system could be extended with advanced UI customization and accessibility features. Educators and students could select from a variety of themes, including high-contrast modes, adjustable font sizes and customizable color schemes, to suit their individual preferences or accommodate accessibility needs. Additionally, incorporating voice command support and screen reader compatibility would

make the tool more inclusive. These enhancements could be implemented as configuration options within the generic *Editor* module, allowing the user to dynamically switch themes and accessibility settings without affecting the underlying functionality.

6.2.2 Collaborative Peer Review and Annotation

To foster collaborative learning, an integrated peer review and annotation module could allow students to comment on and evaluate each other's models directly within the tool. For example, a student reviewing a class diagram might annotate a specific class with suggestions like, "Consider adding a superclass to consolidate common attributes". This module would enable inline annotations, threaded discussions and a rating system, with all feedback synchronized across clients using the existing *WebSocket* based real-time collaboration framework. By incorporating a dedicated collaboration component into the *Communication Layer*, the tool would provide a structured environment for peer feedback, enabling educators to gather diverse insights and track the evolution of student models over time.

6.2.3 Context-Aware Interactive Tutorials

Another direction is the development of context-aware, interactive tutorials that provide step-by-step guidance while students work on their models. For instance, when a student is in the process of creating a class, the system could detect incomplete or incorrect model elements and automatically launch an interactive tutorial overlay. This overlay would walk the student through best practices for class design, demonstrating how to add attributes, specify visibility and create associations. Integrated within our modular architecture, these tutorials could be implemented as additional components that hook into the *Editor*'s event-

handling system. When the *Editor* receives a specific trigger, e.g., an error notification or a detected prolonged period of inactivity on a particular element, then the interactive tutorial module would activate, ensuring that guidance is provided in context and seamlessly without disrupting the overall workflow.

6.2.4 Natural Language Processing (NLP) Integration

Integrating NLP capabilities offers a novel way to lower the entry barrier for modelling. In this enhancement, students would be able to describe a model in natural language using simple sentences to generate initial model elements automatically. For instance, a student could type "Create a class named Student with attributes name and age" and the system would parse this description to generate the corresponding class diagram elements. This NLP module could be integrated as an additional input component within the *Editor*, providing a conversational interface that complements the graphical and textual editors. Such an approach would not only support beginners, but also enable intuitive interaction for all users.

6.2.5 Augmented and Virtual Reality (AR/VR) Integration

One promising future direction is the integration of augmented and virtual reality technologies to create immersive educational experiences. Although AR/VR is well established in industry, for example, enabling workers to interact with machines via virtual controllers and visualize state machines in real time, its potential for teaching modelling lies in transforming abstract diagrams into tangible, spatial experiences. This immersive visualization can enhance student's understanding by providing a natural sense of scale and relationship among model components, thereby improving spatial reasoning and clarifying

complex interactions that are often difficult to grasp in 2D representations. By allowing students to navigate around a model, zoom in on detailed areas, and observe dynamic state changes, AR/VR can bridge the gap between theoretical concepts and practical, real-world applications. However, it is important to conduct further research to determine the optimal way to integrate these technologies into a teaching context, ensuring that the additional complexity truly contributes to improved learning outcomes.

References

[1] A. Tucker, *A model curriculum for k–12 computer science: Final report of the acm k–12 task force curriculum committee.* ACM, 2003.

[2] T. J. T. F. on Computing Curricula, "Curriculum guidelines for undergraduate degree programs in software engineering," New York, NY, USA, Tech. Rep., 2015.

[3] M. Romero and B. Sewell, *Blueprints visual scripting for Unreal engine: The faster way to build games using UE4 blueprints.* Packt Publishing Ltd, 2019.

[4] D. K. Chaturvedi, *Modeling and simulation of systems using MATLAB and Simulink.* CRC press, 2017.

[5] J. Kienzle, S. Zschaler, W. Barnett, T. Sağlam, A. Bucchiarone, S. Abrahão, E. Syriani, D. Kolovos, T. Lethbridge, S. Mustafiz *et al.*, "Requirements for modelling tools for teaching," *Software and Systems Modeling*, vol. 23, no. 5, pp. 1055–1073, 2024.

[6] M. Tools-Teaching, "Modelling tools for teaching 2023," 2023. [Online]. Available: https://modellingtoolsforteaching.github.io/mtt2023.html

[7] M. T. for Teaching, "Modelling tools for teaching," 2023. [Online]. Available: https://modellingtoolsforteaching.github.io

[8] S. Kent, "Model driven engineering," in *International conference on integrated formal methods*. Springer, 2002, pp. 286–298.

[9] D. C. Schmidt *et al.*, "Model-driven engineering," *Computer-IEEE Computer Society-*, vol. 39, no. 2, p. 25, 2006.

[10] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[11] E. W. Dijkstra, E. W. Dijkstra, E. W. Dijkstra, E.-U. Informaticien, and E. W. Dijkstra, *A discipline of programming*. prentice-hall Englewood Cliffs, 1976, vol. 613924118.

[12] C. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.

[13] F. Ciccozzi, M. Famelis, G. Kappel, L. Lambers, S. Mosser, R. F. Paige, A. Pierantonio, A. Rensink, R. Salay, G. Taentzer *et al.*, "How do we teach modelling and model-driven engineering? a survey," in *Proceedings of the 21st ACM/IEEE international conference on model driven engineering languages and systems: Companion proceedings*, 2018, pp. 122–129.

[14] M. Fowler, "Language workbenches: The killer-app for domain specific languages," 2005.

[15] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The generic modeling environment," in *Workshop on Intelligent Signal Processing, Budapest, Hungary*, vol. 17, no. 01, 2001, p. 2001.

[16] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin, "Atompm: A web-based modeling environment," in *Joint proceedings of MODELS'13*

Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013): September 29-October 4, 2013, Miami, USA, 2013, pp. 21–25.

[17] G. Kardas, Z. Demirezen, and M. Challenger, "Towards a dsml for semantic web enabled multi-agent systems," in *Proceedings of the International Workshop on Formalization of Modeling Languages*, 2010, pp. 1–5.

[18] T. C. Lethbridge, A. Forward, O. Badreddin, D. Brestovansky, M. Garzon, H. Aljamaan, S. Eid, A. H. Orabi, M. H. Orabi, V. Abdelzad *et al.*, "Umple: Model-driven development for open source and education," *Science of Computer Programming*, vol. 208, p. 102665, 2021.

[19] M. Schöttle, N. Thimmegowda, O. Alam, J. Kienzle, and G. Mussbacher, "Feature modelling and traceability for concern-driven software development with touchcore," in *Companion Proceedings of the 14th International Conference on Modularity*, 2015, pp. 11–14.

[20] K. Pohl, G. Böckle, and F. Van Der Linden, *Software product line engineering: foundations, principles, and techniques*. Springer, 2005, vol. 1.

[21] M. Gogolla, F. Büttner, and M. Richters, "Use: A uml-based specification environment for validating uml and ocl," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 27–34, 2007.

[22] C. Atkinson and R. Gerbig, "Melanie: Multi-level modeling and ontology engineering environment," in *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*, 2012, pp. 1–2.

[23] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger, "Reuse in model-to-model transformation languages: are we there yet?" *Software & Systems Modeling*, vol. 14, pp. 537–572, 2015.

[24] L. M. Rose, N. Matragkas, D. S. Kolovos, and R. F. Paige, "A feature model for model-to-text transformation languages," in *2012 4th International Workshop on Modeling in Software Engineering (MISE)*. IEEE, 2012, pp. 57–63.

[25] H. Metin and D. Bork, "Introducing biguml: a flexible open-source glsp-based web modeling tool for uml," in *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 2023, pp. 40–44.

[26] "Visual studio code documentation," https://code.visualstudio.com/docs.

[27] Sparx Systems, "Model-based systems engineering (mbse)," 2025.

[28] E. Planas and J. Cabot, "How are uml class diagrams built in practice? a usability study of two uml tools: Magicdraw and papyrus," *Computer Standards & Interfaces*, vol. 67, p. 103363, 2020.

[29] D. Leroux, M. Nally, and K. Hussey, "Rational software architect: A tool for domain-specific modeling," *IBM systems journal*, vol. 45, no. 3, pp. 555–568, 2006.

[30] J. Chimiak-Opoka, B. Demuth, D. Silingas, and N. Rouquette, "Requirements analysis for an integrated ocl development environment," *Electronic Communications of the EASST*, vol. 24, 2009.

[31] L. Compagna, D. Massacci, and N. Zannone, "A comparison between uml tools," *ResearchGate*, 2007.

[32] N. Laranjeiro and A. M. Pinto, "Onda: Online database architect," *arXiv preprint arXiv:2401.16552*, 2024.

[33] J. Chacon, H. Vargas, G. Farias, J. Sanchez, and S. Dormido, "Ejs, jil server, and labview: An architecture for rapid development of remote labs," *IEEE Transactions on Learning Technologies*, vol. 8, no. 4, pp. 393–401, 2015.

[34] D. Kolovos and A. Garcia-Dominguez, "The epsilon playground," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2022, pp. 131–137.

[35] M. 2024, "Models 2024 workshops," 2024. [Online]. Available: https://conf.researchr.org/track/models-2024/models-2024-workshops

[36] S. Zschaler, W. Barnett, A. Boronat, A. Garcia-Dominguez, and D. Kolovos, "Move your mde teaching online: The mdenet education platform," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, 2024, pp. 6–10.

[37] D. T. Dewire, *Thin Clients: Web-based Client/Server Architecture and Applications*. McGraw-Hill Professional, 1998.

[38] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[39] W. Eckerson, "Three tier client/server architecture: Achieving scalability, performance and efficiency in client server applications," *Open Information Systems*, vol. 10, no. 1, 1995.

[40] P. Adamczyk, P. H. Smith, R. E. Johnson, and M. Hafiz, "Rest and web services: In theory and in practice," *REST: from research to practice*, pp. 35–57, 2011.

[41] I. Fette and A. Melnikov, "The websocket protocol," Tech. Rep., 2011.

[42] M. Waseem, P. Liang, M. Shahin, A. Di Salle, and G. Márquez, "Design, monitoring, and testing of microservices systems: The practitioners' perspective," *Journal of Systems and Software*, vol. 182, p. 111061, 2021.

[43] D. Bork and P. Langer, "Language server protocol: An introduction to the protocol, its use, and adoption for web modeling tools," *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, vol. 18, pp. 9–1, 2023.

[44] H. Metin and D. Bork, "A reference architecture for the development of glsp-based web modeling tools," *Software and Systems Modeling*, pp. 1–27, 2025.

[45] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-oriented software design," in *Model-Driven Engineering Languages and Systems*, A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 604–621.

[46] J. Kienzle, G. Mussbacher, O. Alam, M. Schöttle, N. Belloir, P. Collet, B. Combemale, J. Deantoni, J. Klein, and B. Rumpe, "Vcu: the three dimensions of reuse," in *Software Reuse: Bridging with Social-Awareness: 15th International Conference, ICSR 2016, Limassol, Cyprus, June 5-7, 2016, Proceedings 15*. Springer, 2016, pp. 122–137.

[47] H. Ali, G. Mussbacher, and J. Kienzle, "Perspectives to promote modularity, reusability, and consistency in multi-language systems," *Innovations in Systems and Software Engineering*, vol. 18, pp. 5–37, 2022.

[48] G. Georgiev, I. Balabanova, P. Kogias, S. Sadinov, and S. Kostadinova, "Jest," *Journal of Engineering Science and Technology Review*, vol. 11, no. 3, pp. 128–132, 2018.

[49] D. Kondratiuk, *UI Testing with Puppeteer: Implement end-to-end testing and browser automation using JavaScript and Node. js.* Packt Publishing Ltd, 2021.

[50] M. Frisbie, "Devtools pages," in *Building Browser Extensions: Create Modern Extensions for Chrome, Safari, Firefox, and Edge.* Springer, 2022, pp. 269–299.

[51] J. L. Ledford, J. Teixeira, and M. E. Tyler, *Google analytics.* John Wiley and Sons, 2011.

[52] R. Paananen, "Establishing guidelines for usability testing when using web analytical tools," 2023.

[53] J. Porter, "Testing the three-click rule," *User Interface Engineering*, 2003.

[54] MeasuringU, "First click times on websites versus images," 2024. [Online]. Available: https://measuringu.com/first-click-times-on-websites-versus-images/

[55] T. McGill, O. Bamgboye, X. Liu, and C. S. Kalutharage, "Towards improving accessibility of web auditing with google lighthouse," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC).* IEEE, 2023, pp. 1594–1599.

[56] M. Kinnunen, "Evaluating and improving web performance using free-to-use tools," Master's thesis, M. Kinnunen, 2020.

[57] G. C. Developers, "Chrome devtools: Web performance analysis and debugging," Google Chrome, 2025. [Online]. Available: https://developer.chrome.com/docs/devtools/

[58] R. Lucas and L. Rojas, "Performance evaluation of web applications: Using tools for performance testing," *Journal of Web Engineering*, vol. 19, no. 7, pp. 545–563, 2020.

[59] M. Contributors, "High resolution time api (performance.now())," Mozilla Developer Network (MDN), 2025.

[60] V. Bhutani, F. G. Toosi, and J. Buckley, "Analysing the analysers: An investigation of source code analysis tools," *Applied Computer Systems*, vol. 29, no. 1, pp. 98–111, 2024.

[61] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed? a closer look at realistic usage of sonarqube," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 209–219.

[62] J. Smart, *Jenkins: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[63] A. Decan, T. Mens, P. R. Mazrae, and M. Golzadeh, "On the use of github actions in software development repositories," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 235–245.

[64] Y. Arroyo, A. I. Molina, A. M. Torres, J. Mateo, and M. Á. Redondo, "Cross-platform collaborative graphical editors in engineering education," in *2024 International Symposium on Computers in Education (SIIE)*. IEEE, 2024, pp. 1–6.

[65] V.-D. Vogt, "An open source browser-based software tool for graph drawing and visualisation," 2014.

[66] P. G. A. N. Gowda, "Typescript vs. javascript: A comparative analysis."

[67] P. Murley, Z. Ma, J. Mason, M. Bailey, and A. Kharraz, "Websocket adoption and the landscape of the real-time web," in *Proceedings of the Web Conference 2021*, 2021, pp. 1192–1203.

[68] J. Sayago Heredia, E. Flores-García, and A. R. Solano, "Comparative analysis between standards oriented to web services: Soap, rest and graphql," in *Applied Technologies: First International Conference, ICAT 2019, Quito, Ecuador, December 3–5, 2019, Proceedings, Part I 1*. Springer, 2020, pp. 286–300.

[69] M. Levlin, "Dom benchmark comparison of the front-end javascript frameworks react, angular, vue, and svelte," 2020.