TU WIEN Informatics

Developing Sprotty-based Modeling Tools for VS Code

Contribution of a Generic Development Approach and a Tool for ER Modeling

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software- and Information Engineering

eingereicht von

Philipp-Lorenz Glaser
Matrikelnummer 11776175

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork

Wien, 28. Juni 2022

Philipp-Lorenz Glaser	Dominik Bork

TU WIEN Informatics

Developing Sprotty-based Modeling Tools for VS Code

Contribution of a Generic Development Approach and a Tool for ER Modeling

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software- and Information Engineering

by

Philipp-Lorenz Glaser
Registration Number 11776175

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork

Vienna, 28th June, 2022

Philipp-Lorenz Glaser	Dominik Bork

Erklärung zur Verfassung der Arbeit

Philipp-Lorenz Glaser

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 28. Juni 2022

Philipp-Lorenz Glaser

Kurzfassung

Modellierungstools spielen im Bereich des Model-driven Engineering (MDE) eine entscheidende Rolle bei der Erstellung von Abstraktionen eines Systems durch die Verwendung von Modellierungssprachen, die die Spezifikation von Modellen und deren Transformation ermöglichen. Angemessene Toolunterstützung für die Modellierung reduziert nachweislich die Komplexität, verbessert die Verständlichkeit und erhöht die Produktivität der Entwickler. Das Fehlen ausreichender Toolunterstützung wird jedoch oft als ein Hauptproblem für den Erfolg der Einführung von MDE und für die Durchführbarkeit der Modellierung in der Praxis angesehen. Aus diesem Grund und wegen der sich ständig ändernden modernen Anforderungen benötigen Modellierungtools effektive Ansätze für ihre Entwicklung. Mit dem neuesten Trend, Anwendungen auf Web- und Cloud-Plattformen zu verlagern, sind vielversprechende neue Technologien entstanden und verfügbar geworden, die diesen Prozess unterstützen können.

Ein bemerkenswerter Beitrag im Gebiet der Webmodellierung ist die Einführung des Language Server Protocol (LSP) für textuelle Sprachen. Durch die Standardisierung gängiger Sprachfunktionen mit LSP (z.B. Code Completion oder Refactoring) kann eine einzige Implementierung eines sprachspezifischen Servers verwendet werden, um mehreren sprachunabhängigen Clients (d.h. Editoren oder IDEs) Sprachunterstützung hinzuzufügen. Dies reduziert die Komplexität erheblich, aber da LSP in erster Linie auf textuelle Sprachen abzielt, bleibt die Frage offen, ob das Protokoll auch ausreicht, um grafische Sprachen zu unterstützen und ob beiden Arten von Sprachen für heterogene Darstellungen und simultane Bearbeitung kombiniert werden können. In diesem Zusammenhang hat sich das Framework Sprotty bemüht, grafische Modellierung auf Web-Plattformen verfügbar zu machen und bietet eine Erweiterung des LSP für die Erstellung von hybriden Modell-Editoren, die synchronisiertes Editieren zwischen textuellen und grafischen Editoren ermöglichen. In Anbetracht der umfangreichen Funktionalität, der Anpassbarkeit und der verfügbaren Integration mit anderen Technologien (z.B. VS Code oder EMF) ist Sprotty ein interessantes Framework für die weitere Forschung und dient als Grundlage für diese Bachelorarbeit.

In dieser Arbeit stellen wir (*i*) einen generischen Ansatz für die Entwicklung von Sprotty-basierten Modellierungtools für VS Code und (*ii*) das BIGER Tool für die Modellierung von Entity-Relationships (ER) vor, das auf unseren Entwicklungsansatz basiert und gängige Modellierungsfunktionen wie z.B. hybride Modellierung, Rich-Text-Editing oder

(SQL) Code Generierung beinhaltet. Sowohl der Ansatz als auch BIGER integrieren Sprotty Diagramme mit VS Code und einer Xtext-basierten textuellen Sprache, wobei diese durch die webbasierte Architektur des LSP dennoch plattformunabhängig und leicht erweiterbar bleiben.

Abstract

Modeling tools play a critical role in the field of Model-driven Engineering (MDE) to create abstractions of systems through the use of modeling languages which allow specifying models and their transformation. Adequate tool support for modeling has proven to reduce complexity, improve comprehensibility, and increase developer productivity, however, lack of sufficient tooling is often regarded as a key concern in the success of adopting MDE and in making modeling feasible in practice. For this reason, together with continuously changing modern-day requirements, modeling tools require effective approaches for their development and with the recent trend of applications moving to web- and cloud platforms, promising new technologies have emerged and became available that can aid in this process.

A remarkable contribution in the area of web modeling has been the introduction of the Language Server Protocol (LSP) for textual languages. By standardizing common language features through the LSP (e.g., code completion or refactoring), a single implementation of a language-specific server can be used to add language support to multiple language-agnostic clients (i.e., editors or IDEs). This greatly reduces the complexity, but since the LSP primarily targets textual languages, the question remains whether the protocol is also sufficient to support graphical languages, and whether the two can be combined for heterogeneous representations and simultaneous editing. In this context, the diagramming framework Sprotty has made an effort in bringing graphical modeling to web platforms and offers extension of the LSP for the creation of hybrid model editors that enable synchronized editing between textual- and graphical editors. Given the extensive functionality, customizability and available integration with other technologies (e.g., VS Code or EMF), Sprotty becomes an interesting framework to conduct further research on and serves as the foundation for this thesis.

In this work, we contribute (*i*) a generic approach for the development of Sprotty-based modeling tools for the VS Code IDE, and (*ii*) the BIGER tool for Entity-Relationship (ER) modeling which is based on the development approach and includes common modeling features such as, e.g., hybrid modeling, rich-text editing, or (SQL) code generation. Both, the approach and BIGER, integrate Sprotty diagrams with VS Code and an Xtext-based textual language, while still remaining platform-independent and easily extensible, due to its web-based nature and architecture of the LSP.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Problem Statement . . . 3
1.3 Research Objectives . . . 4
1.4 Approach and Contributions . . . 4
1.5 Thesis Structure . . . 5

2 Background **7**
2.1 Model-driven Engineering . . . 7
2.2 Web Modeling with EMF . . . 12
2.3 Entity-Relationship Modeling . . . 19
2.4 Related ER Modeling Tools . . . 24

3 Developing Sprotty-based Modeling Tools for VS Code **27**
3.1 Client-side Sprotty Diagrams . . . 28
3.2 Xtext Language and VS Code Extension . . . 36
3.3 Sprotty Integration with VS Code and Xtext . . . 43
3.4 Hybrid Modeling . . . 48

4 The BIGER Modeling Tool **51**
4.1 Architecture . . . 52
4.2 Feature Showcase . . . 56
4.3 Evaluation . . . 60

5 Conclusion **67**
5.1 Thesis Summary . . . 67
5.2 Outlook . . . 67

A Textual Models **69**
A.1 University Example 69
A.2 Evaluated ER Models 71

List of Figures **75**

List of Tables **77**

Acronyms **79**

Bibliography **81**

CHAPTER 1

Introduction

Model-Driven Engineering (MDE) makes use of models as a fundamental concept throughout the whole software engineering life-cycle with the aim of reducing complexity and improving comprehensibility of complex systems [1, 2]. MDE has shown to bring various benefits when being adopted to development processes, in particular, it can decrease the overall effort in costs and time by increasing development productivity, e.g., through code generation or domain-specific modeling languages [3, 4]. Even though integrating model-based approaches appears to be beneficial, especially for large projects, adoption of MDE in practice has been less prominent and slower than initially expected [5, 6, 7]. A common reason for low adoption is the lack of expertise, as for making MDE feasible it is required to hire domain experts or ensure sufficient training. In practice, this requires a high initial effort and could potentially introduce new risks that do not outweigh the advantages of MDE adoption.

A frequently mentioned drawback of modeling often includes lack of proper tool support [6, 8], with user experience being a substantial factor in making MDE attractive and feasible to users [9]. Modeling tools are generally available in an Integrated Development Environment (IDE) and feature capabilities beyond basic drawing functionality, such as integration, management, and transformation of models [2]. It is often required to support multiple heterogeneous representations of a model with synchronized changes. Usability gains even more importance when introducing non-developers to modeling processes and for improving cross-communication between actors with different levels of expertise [10]. Other relevant factors concerning the quality of modeling tools include, e.g., scalability, performance, interoperability or dealing with the representation of large models [6, 7, 11].

It is self-evident that modeling tools are an integral part of MDE that ask for sound approaches in their development process. Research on the effective development of modeling tools serves as the foundation for this thesis, and in this context we want to further restrict the goal of our work. For this, we first look into key motivational aspects

before we define the problem statement and research objectives. We further discuss the chosen approach together with our contributions, and conclude the introduction with a description of the thesis structure.

1.1 Motivation

Depending on the complexity, there are different risks involved in the development process, arising the question on how to effectively develop modeling tools that fulfill modern-day requirements and help in increasing the adoption of MDE in practice. Recent development has shown trends in IDEs moving away from traditional heavy-weight desktop applications and heading towards more lightweight web-based platforms. Moving modeling to the web offers the potential to address major concerns and drawbacks in traditional modeling tools and close the gap of low MDE adoption [12, 13]. However, traditional tools are relatively feature-rich, requiring efficient approaches to reuse existing modeling functionality in web-based platforms. Alternatively, existing implementations have to be abandoned and re-implemented from scratch.

A remarkable contribution to the area of web modeling has been the Language Server Protocol (LSP)[1] and its adoption for textual modeling languages [14, 15]. Initially introduced by Microsoft, the protocol deals with the $O(n \times m)$ complexity of adding new language support to IDEs by standardizing common language features, such as auto-complete, refactoring, or diagnostics [16]. In theory, significant improvements regarding tool portability can be achieved, since a single implementation of a language-specific server can be reused for delivering language features to multiple language-agnostic clients (i.e., editors). Nowadays, the LSP is supported by many popular IDE platforms and a growing number of tools are implementing language servers using the protocol[2]. However, a major drawback of using the LSP for the development of modeling tools lies in the fact that the protocol primarily targets textual languages, leading to the additional question on how to introduce support for graphical languages. Previous work has proven the potential of adapting the LSP for graphical modeling, including decoupled solutions [17, 18] and the creation of hybrid model editors for textual- and graphical editing [19]. Hybrid or blended modeling allows interactions and modifications within editors to be synchronized between multiple representations of a model, and can greatly improve usability [20] in respect to different quality attributes, namely:

Understandability — Multiple abstraction layers and representations allow models to be more comprehensive across different levels of expertise between tool users.

Learnability — Domain experts can better explain complex models with the help of different graphical representations, simplifying the learning process of modeling.

[1] Official Language Server Protocol Page
[2] List of Tools implementing the Language Server Protocol

Changeability — Model modification can be performed in different views, allowing certain operations to be performed more intuitively, e.g., graphically dragging edges to nodes to create connections between elements, contrary to less intuitive textual changes.

Despite various tools available that offer hybrid modeling approaches, the field is still an active area of research with remaining challenges and opportunities [21]. The diagramming framework Sprotty[3] has made great contribution to the effort of bringing graphical modeling to web-based editors and has proven to aid in the development of hybrid model editors, e.g., for state machines[4], or leveraged in enabling collaborative modeling, e.g., for the Textual Goal-oriented Requirement Language (TGRL) [22]. Sprotty becomes even more interesting as a modeling framework since it offers integration with state-of-the-art IDE platforms, such as VS Code or the Cloud IDE Theia. Furthermore, the framework has the ability to extend the LSP with graphical operations by mapping them to corresponding LSP messages that perform changes on an underlying textual model. Such extension can be implemented for Xtext-based language servers to render diagrams of language artifacts and enabling support for hybrid modeling. All packages of Sprotty are open-source available and actively maintained with various other modeling frameworks built on top of Sprotty, e.g., the Graphical Language Server Platform (GLSP)[5] or Sirius Web[6].

1.2 Problem Statement

Summarizing the motivational aspects of this thesis, LSP has proven efficient reuse of a single language-specific server, delivering its language features to multiple language-agnostic clients. Sprotty as a diagram framework offers approaches in extending the LSP for graphical modeling, thus, opening the potential for the development of hybrid model editors. Matching the move towards web modeling and aiming to contribute to the adoption of MDE in practice, this thesis explores the diagramming capabilities of Sprotty, when used as a framework in the development of modeling tools.

VS Code is a currently popular, web-based IDE and extensions to the platform can make use of its highly extensible API. Furthermore, it fully incorporates the LSP into the platform to create custom textual editors and allows extensions to create web views, which can be used to display arbitrary web-based content, e.g., for rendering Sprotty diagrams. Given the advantages, VS Code appears to be a solid candidate for the deployment of Sprotty-based modeling tools, thus, we further refine the problem of the thesis to explore this exact scenario of deploying hybrid model editors as extensions to the VS Code ecosystem.

[3] Sprotty Project Page
[4] Sprotty Statemachine Example
[5] GLSP Project Page
[6] Sirius Web Project Page

1.3 Research Objectives

Based on the above-mentioned motivational aspects and problem statement, we aim on responding to the following research objectives in the course of this thesis:

RO1 Exploration of the capabilities provided by Sprotty when leveraging the framework for the development of modeling tools for VS Code.

RO2 Contribution of a generic approach for implementing Sprotty-based modeling tools for VS Code, including integration of an Xtext language and enabling hybrid model editing.

RO3 Contribution and evaluation of a Sprotty-based modeling tool for VS Code featuring hybrid textual- and graphical ER modeling and code generation of a relational database schema.

1.4 Approach and Contributions

In this work, we approach the first two established research objectives (RO1, RO2) by contributing a generic approach for developing Sprotty-based modeling tools with integration of VS Code and Xtext. In the contributed approach, we discuss architectural design considerations and provide guidelines for the implementation process in the form of a running example that showcases incremental extensions of a basic modeling tool. The example includes a Sprotty-enhanced language server that integrates Sprotty with a Xtext-based language, together with an extension for deployment to the VS Code ecosystem. As a final step, our contributed approach fully utilizes Sprotty and enables hybrid modeling for selected modeling features.

Besides the approach, we further contribute the bigER modeling tool, used for ER modeling and available as an extension for VS Code. The tool deals with the last research objective (RO3) and fully utilizes Sprotty based on our contributed development approach. Modeling capabilities include hybrid, textual- and graphical model editors to support cross-communication between stakeholders, as well as SQL code generation to integrate with conceptual database design processes. Various other considerations have been established throughout the development process, aiming in fulfilling modern-day requirements to modeling tools such as, e.g., usability, scalability, and portability. An overview of bigER is given in [23] which emerged in the course of this work, our focus in this thesis is on providing an update of the development progress with an initial evaluation, based on concrete modeling cases.

1.5 Thesis Structure

This section concludes the introduction of this work and provides a guideline for the structure of the remaining chapters. In short, we cover the relevant background in Chapter 2, present a generic approach of developing Sprotty-based modeling tools for VS Code in Chapter 3, discuss the BIGER Modeling Tool in Chapter 4 and conclude with a summary and outlook in Chapter 5. In the following, we describe each of the chapters in more detail.

As an entry point for dealing with the problem and research objectives of this thesis, we recapitulate on relevant background information in **Chapter 2**. The chapter introduces foundational concepts of *Model-Driven Engineering* and discusses *Web Modeling with EMF*, including relevant technologies used throughout this work. Since the BIGER modeling tool is a key contribution of this work, we additionally examine principal *ER Modeling* concepts and concluded the background with a comparison of *Related ER Modeling Tools*.

With the relevant background information established, we proceed with **Chapter 3** and our contribution of a generic development approach for Sprotty-based modeling tools. Initially, we discuss the architecture with a focus on the *Client-side Sprotty Diagrams* of the framework and implement a basic diagram for our running example. Then we look into the Xtext language workbench and how to add language support in VS Code. For our running example, we create a *Xtext Language Server* and a *VS Code Language Extension* that communicate with each other through the LSP. The created language server and extension also serve as the basis for our development approach, which we apply as a next step to the example and show the *Integration of Sprotty with Xtext and VS Code*. As a final step, we build on the approach and enable *Hybrid Modeling* with Sprotty to synchronize graphical editing with textual changes.

Our other contribution, the BIGER modeling tool, is presented in **Chapter 4**. We first discuss the *Architecture* of the tool, which is realized based on our development approach, and then we follow with a *Feature Showcase* of the implemented functionality. In the chapter, we also provide an initial *Evaluation* of the tool by recreating and discussing concrete modeling cases. We finish the evaluation of the tool with an overview of supported ER modeling concepts.

At last, in **Chapter 5** we conclude the thesis with a *Summary* and *Outlook* of our work. The summary includes a recap of our contributions and how they deal with the established research objectives, while the outlook reveals current ongoing work and future directions of the BIGER modeling tool.

CHAPTER 2

Background

In this chapter, we cover background information and foundational concepts before we proceed to the contributions of this thesis. First, we introduce MDE (Section 2.1) and give an overview of relevant model-based approaches that are used throughout this work. Next, follows a discussion of web modeling with EMF (Section 2.2), including motivational aspects and underlying technologies of our development approach, namely the LSP, Xtext, Sprotty and VS Code. The last sections become relevant for the BIGER Modeling Tool, where we discuss different ER modeling concepts (Section 2.3), as well as related ER modeling tools (Section 2.4).

2.1 Model-driven Engineering

The discipline of Model-Driven Engineering (MDE), integrates models as a central concept into the software development life-cycle, allowing to focus directly on a concrete problem domain, rather than dealing with underlying, specific technologies. The primary goal of MDE is to reduce complexity and increase comprehensibility of a system by providing additional layers of abstraction and representing software artifacts through the core concepts of *Models* and *Transformations* [1]. In a classical sense, programs can be seen as a combination of algorithms and data structures, leading to the eminent equation of $Algorithms + Data\ Structures = Programs$ [24]. In the context of MDE, this is converted to the equation of $Models + Transformations = Software$, meaning that a combination of models with transformations leads to the result of software [2]. However, the equation alone, does not suffice in describing MDE as a whole. Certain notations, commonly referred to as *Modeling Languages*, are required to represent different concepts. In addition, processes and rules with a certain level of tool support, realized in *Modeling Tools*, are required for sufficient specification of models and transformations. A well acknowledged language in the modeling domain is, e.g., the Unified Modeling Language (UML) [25, 26] with a wide variety of available modeling tools [27].

Apart from MDE, there exist various other terms that use the Model-driven prefix (MD) and might lead to confusion. In general, MDE goes beyond pure development with sound engineering approaches and is a superset of Model-driven Development (MDD). Contrary, Model-based Engineering (MBE) processes are only based on models, as opposed to being used for driving the processes, thus, MBE is a superset of MDE. Another often mentioned term is Model-driven Architecture (MDA), a concrete standard of MDD approaches, managed by the Object Management Group (OMG) [28]. When MDE is applied to software, we particularly speak of Model-driven Software Engineering (MDSE), however, in this thesis we use the two terms interchangeably.

In the following, we look into the basic architecture of MDE and provide a high-level overview of its components. Next we describe modeling languages and metamodeling more specifically, including DSLs, GPLs, the anatomy of languages, and definition of a concrete syntax.

2.1.1 Basic Architecture

The basic architecture of MDE can be seen in Figure 2.1 as taken from [2] with different layers in a vertical- and horizontal space. The vertical aspects correspond to the mapping of models to software artifacts from the modeling- to the realization layer, with the mapping realized through model transformations and code generation in the automation layer. In contrast, the horizontal space deals with the corresponding conceptualization in different abstraction layers. Application-specific models (application layer) are defined using modeling languages that target the specific problem domain (application domain layer). Modeling languages in turn are defined using meta-modeling languages (meta-level). Similarly, model transformations and code generation is defined by transformations, which are further defined using a transformation language. The resulting artifacts, e.g., generated code, can be used in specific platforms, however, often the goal is to create platform-independent artifacts.

2.1.2 Modeling Languages

In the field of software engineering, programming languages encapsulate complex structures and behaviours into specific language concepts that serve the goal of providing abstraction. In the context of MDE, we use the term modeling languages instead, to specify problems as abstract representations in the form of models. Abstraction through models can bring various benefits, e.g., different representation levels can improve cross-communication between stakeholders of different expertise, or the specification of model transformations can enable code generation and automate parts of the development process.

DSL vs. GPL

For the classification of languages, we differentiate between the terms General Purpose Language (GPL) and Domain-specific Language (DSL). In a modeling context, we refer to

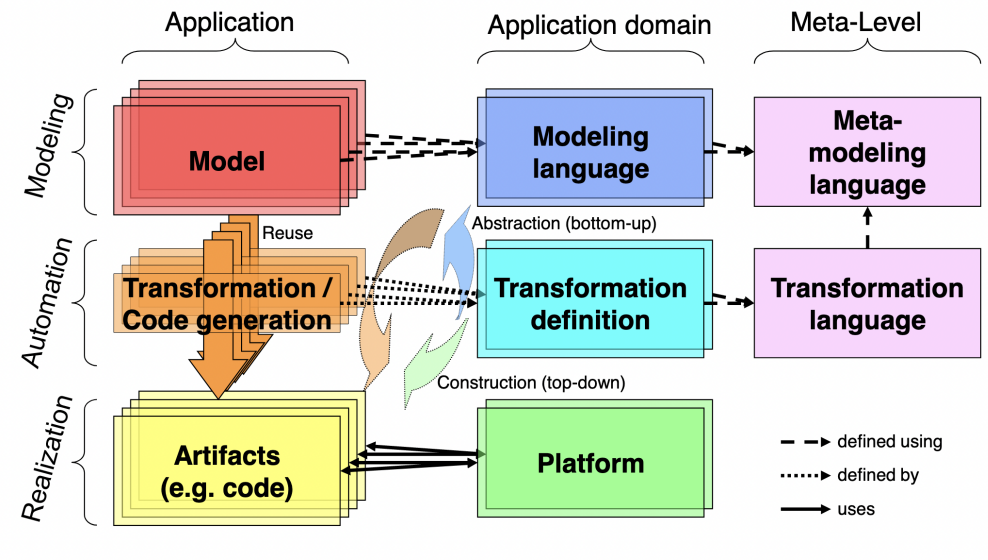


Figure 2.1: High-level Architecture of Model-driven Engineering [2]

languages as General Purpose Modeling Language (GPML) or Domain-specific Modeling Language (DSML). GPLs are designed to solve problems across a majority of domains more generally, and are applicable in a variety of different settings and environments. A well known example for a GPL is the Java language and for a GPML we already mentioned UML, they both can be used for dealing with a wide range of problems. In contrast, DSLs are more specialized and meant to be used for specific problems, with no intended use outside the problem domain. Examples for DSLs include the markup language HTML or the relational database language SQL. Similarly, DSMLs are designed for creating models of a specific domain, and are common practice for modeling, e.g., in the automotive- or aerospace industry. DSLs and DSMLs can greatly reduce complexity and are often convertible to other languages, but, while they can be beneficial when successfully employed into the software engineering process, it should also be stated that they can create new risks when used in the wrong context [29]. The cost of learning a new language can greatly outweigh the limited application uses it has, making it practically useless. In addition, an enormous effort can be spent on the design and implementation of a DSL, such that possible gains of actually using the language do not sufficiently make up the invested development effort.

Anatomy of Languages

It becomes self-evident that modeling languages can be internally complex, and a clear definition of allowed concepts is required to adequately use the language in the specification and validation of models. Even though languages are present in varying nature and are applied in different fields, they still share a certain set of common characteristics. Modeling languages specifications are a composition of *syntax* focusing on the notation of

the language, *semantics* to describe the meaning of language constructs, and pragmatics for the purpose of models [30, 31]. When formalizing a language and ensuring that it is well-defined, we differentiate between the following components [32]:

Abstract Syntax — The Abstract Syntax (AS) corresponds to the internal representation of a modeling language. It formally describes language concepts and their structure, i.e., how language elements can be combined, independent of any concrete form of representation. The AS can be specified using different specification techniques, and they can be represented in different forms, e.g., UML class diagrams or EBNF [33]

Concrete Syntax — The Concrete Syntax (CS) defines concrete representations (notations) for a modeling language and its concepts. It is usually used directly by the modeler and is commonly present textually, graphically or tabular. There can be also be multiple CS representations for a single AS definition, e.g., in a combination of textual and graphical notations [34].

Semantics — Semantics describe the meaning of a language and ensure consistent meaning of the language elements, including their combinations.

Modeling languages commonly also include a Serialization Syntax, e.g., in XML format, for persistent storage of models and to support model exchange between tools. Other important aspects to consider are extension of languages or mapping (transformation) to other languages and domains.

There exist numerous principles and dimensions for the classification of DSL's. Such dimensions include focus, style, notation, internality and execution, as mentioned in [2], additional principles are described in [35]. Guidelines for the design of DSL's can be found in [36]. We now look into basic concepts relevant for the development of modeling tools, with a focus on metamodeling and grammars.

2.1.3 Metamodeling

We defined the AS in a modeling language as a formal description of the language concepts and their structure. There are several ways to define an AS, and in the context of MDE this is commonly achieved through metamodeling. Models in a language are considered valid if they conform to the AS of the language, defined in a metamodel, i.e., each model element is an instance of a metamodel element. Metamodeling languages, also referred to as meta languages, are used to define a metamodel, and are commonly based on object-oriented approaches, e.g., UML derivatives with additional refinement in Object Constraint Language (OCL).

A metamodel can be seen as a model itself and can be defined in even higher level of abstraction, called a meta-metamodel. The recursion of metamodel definitions could go on indefinitely, however, it can be escaped, e.g., by using a meta-circular metamodeling

language such as the Meta Object Facility (MOF). Meta-circularity allows defining language concepts by using the language itself in this process [31]. A commonly used implementation of MOF is present in the Eclipse Modeling Framework (EMF) with the metamodeling language Ecore. We discuss EMF and Ecore in more detail in the next section (see Section 2.2).

Concrete Syntax Definition

In general, the AS is not really intended to be used by humans, as it can include numerous nested references and quickly lead to a cluttered view. As mentioned, the CS deals with the notation of a language and can be used to improve the readability of a model. In addition, it is also possible to define multiple, different CS representations that all conform to a single AS definition, allowing for multi-view [37] and hybrid (or blended) [20] modeling approaches. Representations for the definition of a CS are mostly in the form of a Textual Concrete Syntax (TCS) for textual languages or a Graphical Concrete Syntax (GCS) for graphical languages [38, 31, 2]. Figure 2.2 displays how text and diagram visualize a model and how they both correspond to a respective CS that symbolizes the metamodel, corresponding to the AS.

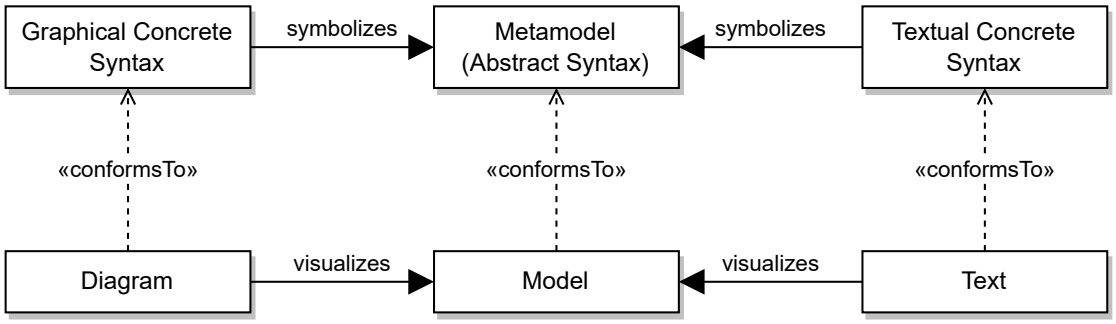


Figure 2.2: Textual- and Graphical Concrete Syntax Definition of Metamodels, adapted from [2]

Textual modeling languages are often defined through a grammar that combines a metamodel with a corresponding TCS. The grammar of a language defines all valid sentences, in contrast to metamodels, which define all valid models. Commonly, the grammar is in Extended Backus Naur Form (EBNF), which allows the specification of context-free grammars through production rules, with a sequence of terminal and non-terminal symbols. When using textual languages, models are defined in simple text files that can lead to higher productivity when making use of features such as, e.g., copy-and-paste, formatting, and version control. A feature-based classification schema of textual syntax mapping approaches can be found in [39].

On the contrary, graphical modeling languages require the specification of a GCS, consisting of graphical symbols, e.g., circles, rectangles or other shapes, compositional rules, e.g., how elements can be nested within each other, with a mapping between the

graphical symbols and the language concepts defined in the AS. For creating graphical model editors, graphical languages additionally specify editing behavior, e.g., creation of elements in a tool palette or handling connections between nodes by dragging edges. Definition of the GCS is commonly based on approaches that include mappings (e.g., Sirius, GMF), annotations (e.g., Eugenia) or APIs (e.g., Graphiti). For more details and an analysis of graphical modeling language notations, we refer to [34].

2.2 Web Modeling with EMF

There are various challenges in MDE that have to be addressed to increase its adoption and feasibility in practice. In regard to tool support, current challenges include modeling of different heterogeneous views, expressing requirements in a human-readable form and scalability in large modeling projects [7]. It is also important to consider the aspect that modeling tools are often integrated into IDEs to blend into the software development process. However, the target platforms are traditionally based on specific technologies that require vendors to implement platform-specific solutions when providing additional tool support in an IDE. This vendor-lock leads to the problem of low portability in a majority of modeling tools, since an enormous effort is required to re-implement existing solutions for different platforms. The mentioned challenges together with modern-day requirements continue to increase the gap for low MDE adoption and the need for high usability, scalability, portability and various other quality attributes in modeling tools.

In recent years, there has been an increased effort in bringing modeling to web- and cloud-based environments such that modeling tools can keep up with modern requirements and in hope of dealing with the drawbacks that lead to low MDE adoption. Also contributing to this effort are IDEs that are following the trend of moving to web platforms and making their API available for extension. Developing extensions to an IDE in a web-based environment opens up a whole new range of technologies that can be leveraged, such as the LSP for textual languages [18] or Scalable Vector Graphics (SVG) for stable rendering of graphical models. Web-based modeling tools have further shown to feature collaboration, high scalability and seamless integration with other tools [13]. While there has been various success in web-based modeling, the challenge of adopting the EMF to web technologies remains an active area of research [18, 40, 41]. Due to tight integration with Eclipse, developing and maintaining EMF-based tools that are deploy-able to web platforms quickly becomes a complex task, with new risks involved throughout the process. Over the years, EMF has seen wide-spread use within the modeling community with various tools relying on it and while alternatives exist, abandonment of EMF means a great amount of modeling functionality is lost and has to be reproduced in a web context. Thus, we consider integration of EMF in web-based solutions substantial for the future success of modeling tools.

The mentioned aspects serve as the underlying motivation for this thesis and lead to the objective of contributing an approach for effectively developing web-based modeling tools that integrate EMF-based languages and fulfill modern requirements. Our contributed

approach leverages the graphical framework Sprotty to enable synchronized modeling of textual- and graphical model representations. Xtext serves as a language workbench in the development process of a modeling language and its TCS. Xtext automatically generates a corresponding language server for the language to allow seamless integration with IDEs and platforms that support the LSP. In our approach, we add language support in VS Code by creating a language extension. Sprotty is used to derive the GCS by integrating with the textual language and generating a corresponding diagram. Sprotty is also used for developing a VS Code extension that ties all the components together, and synchronizes the graphical representation with language artifacts from the textual editor. As a last refinement, we fully utilize Sprotty to enable hybrid modeling approaches, which is achieved through extension of the LSP. In this section, we introduce the most significant technologies in our contributed approach to serve as a foundation for development. First we discuss underlying modeling aspects of EMF, before we proceed with a high-level description of the LSP, Xtext, Sprotty and VS Code. Since the technologies are used throughout the this work, later chapters provide a more detailed description of them (see Chapter 3 and 4).

2.2.1 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is a mature modeling framework and code generation facility that is heavily used in the Eclipse platform. EMF provides a standard for model-based development approaches as part of the Eclipse Modeling Project[1]. Despite a wide area of application, our primary focus in the remainder of this section is on using EMF for the metamodeling process and the design of modeling languages. For a more comprehensive and practical overview, we refer to the book by Steinberg et al. [42], as well as the EMF Programmer's Guide[2].

The OMG defined the MOF as a standard for MDE, in particular, it provides a four-layer modeling architecture with MOF used at the top-layer as a meta-metamodeling language to create metamodels. The created metamodels can then be used for modeling activities and to represent real-world objects. The meta-metamodel of EMF is Ecore, a Java-based implementation that aligns with eMOF. The modeling elements of Ecore are based on the concepts of object-orientation, an overview of concepts and their structure is is shown in Figure 2.3.

EMF supports the import of existing models to Ecore, e.g., from UML, XML or annotated Java, together with generation of a Java infrastructure with an API and editors for the language in the metamodel. Persistence can be enabled through serialization in the XML Metadata Interchange (XMI) format, which is also defined by the OMG. Notable components built-on top of the EMF ecosystem include: servers and storage through model repositories (e.g., EMFStore), graphical- and textual modeling frameworks (e.g., Xtext, Sirius, GMF), model transformation languages (e.g., ATL, Henshin, Xtend) and

[1] Eclipse Modeling Project
[2] EMF Programmer's Guide

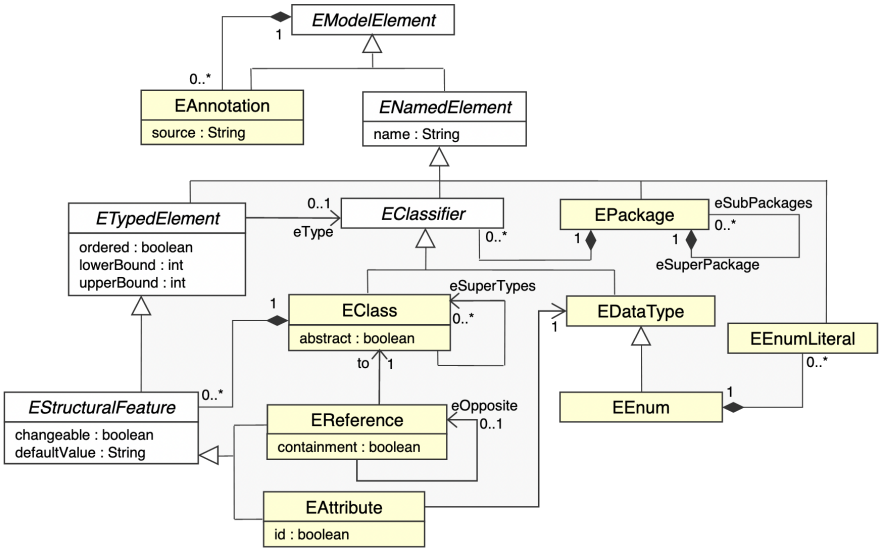


Figure 2.3: Overview of the Ecore Meta-metamodeling Language

UI development tools (e.g., EMF Forms). EMF integrates well with the Eclipse Platform through its OSGi plugin-architecture, however, our target is deployment of tools outside Eclipse. Specifically, we later use Xtext, Sprotty and LSP to integrate EMF-based modeling languages with VS Code and technically any web-based platform supporting the LSP.

2.2.2 LSP

The task of adding language support to different IDEs traditionally involves an enormous implementation and maintenance effort, which can lead to inconsistencies between different integrations and generally is more prone to errors. The effort required in implementing and maintaining n languages for m different editors is in the complexity class of $O(n \times m)$. With IDEs moving to web platforms, Microsoft introduced the Language Server Protocol (LSP) to reduce the effort in supporting different programming languages within VS Code. The protocol allows to decouple the implementation of language-specific servers from language-agnostic clients, i.e., the IDE or editor, thus, reducing the complexity to $O(n + m)$.

The LSP standardizes common language features such as auto-complete, finding references, documentation or hover information and enables inter-process communication through messages between a server and client. Messages are defined in the JSON-RPC[3] format, thus, they can be considered lightweight and stateless. The specification of the protocol is described in [43] and currently features version 3.17.x. The core protocol consists of request-, response- and notification messages, together with utility interfaces and

[3] JSON-RPC Website

-functions, e.g., for specifying a range and position in a text document. An example of request- and response messages for finding a definition is shown in Listing 2.1 and Listing 2.2, respectively. It can be seen that elements in a document are referenced through an Uniform Resource Identifier (URI) with text positions. To know which language features can be communicated between a language server and a client, they both announce their supported capabilities during communication.

```
{
 "jsonrpc": "2.0",
 "id" : 1,
 "method": "textDocument/definition",
 "params": {
  "textDocument": {
   "uri": "file:///Example.java"
  },
  "position": {
   "line": 5,
   "character": 14
  }
 }
}
```

Listing 2.1: Go To Definition - Request

```
{
 "jsonrpc": "2.0",
 "id": 1,
 "result": {
  "uri": "file:///Definition.java",
  "range": {
   "start": {
    "line": 3,
    "character": 7
   },
   "end": {
    "line": 3,
    "character": 15
   }
  }
 }
}
```

Listing 2.2: Go To Definition - Response

Besides VS Code, the LSP is supported in countless other popular IDEs such as Eclipse, Theia or Atom, and it is implemented in numerous language servers[4]. While the LSP has been successful in supporting textual languages [14, 15], it does not address other prominent IDE features such as graphical languages, testing, or debugging, often leading to a mixture of different protocols in a language [44, 45]. Relevant to the context of this work is the question of how the LSP can be reused, to combine textual and graphical languages [17, 19]. Concerns for this approach include differences regarding representation (characters vs shapes), processing (top-down vs graph traversal) and serialization (raw text vs specific formats). In this thesis, we contribute to this area of research by leveraging Sprotty for extending the LSP and to synchronize textual- and graphical model editors.

2.2.3 Xtext

Xtext is a language workbench for developing textual modeling languages and becomes relevant for this thesis due to its integration with EMF, LSP, and Sprotty. Strengths of the framework lie in it being mature and easy to learn, as well as the developed languages offering high quality, optimized performance and support on multiple platforms. In the following, we introduce the key aspects of Xtext, whereas we put the framework into practice in later chapters of the thesis. For more information on Xtext, we refer to the official documentation [46] or the recommended book [47].

[4] LSP Implementors

In Xtext, a language can be simply defined by providing a grammar specification that corresponds to the TCS. The grammar is based on the Extended Backus Naur Form (EBNF) with extensions for type inheritance, type information, containment, and cross-references [48]. From the specified grammar file, the underlying Xtext generator automatically creates a complete infrastructure with components commonly expected to be present in a language. This includes an ANTLR parser[5], a lexer, a derived Ecore model and other components that can be customized through Dependency Injection (DI). Xtext also generates stubs to override the defaults of API methods such as validation of custom constraints, scoping to restrict referable elements in cross-references, value conversion, formatting and numerous more.

Model transformation in the form of Model-to-text (M2T) [49] is supported by implementing a code generator which can further utilize Xtend for readable string concatenation through template expressions. Xtend[6] is a dialect that fully compiles to Java and which claims to be more concise, readable and expressive. Besides template expressions, selected features include, e.g., type inference, lambda expressions or extension methods. In this work we heavily make use of Xtend with most of the language server components implemented with it.

EMF is deeply integrated into the Xtext framework as it is being used to store the Abstract Syntax Tree (AST). The AST is used for various other processing steps, such as validation or code- and diagram generation. In EMF the AST is represented through `EObjects` that map the textual model to an Ecore model. Xtext also generates Java interfaces and implementations for the elements of the language through EMF. Besides a fully-fledged Eclipse editor featuring, e.g., an outline view, syntax highlighting or code completion, Xtext can also generate a generic editor with a language server that supports the LSP. Thus, EMF-based languages implemented in Xtext can be reused to add support in various other editors by communicating with the language server.

2.2.4 Sprotty

Sprotty is a web-based, open-source diagramming framework[7] offering an extensible architecture and a variety of customizable modeling features. Diagrams are embedded in the DOM of a browser, and they provide stable, scalable views of models that are rendered as SVG, styleable with CSS, and can be also be animated. Sprotty offers the choice between two architectural design decisions, a client-only approach with a local model source entirely implemented on the client, or a client-server approach, featuring a remote backend that is responsible for holding a model and delivering representations to the client. The client-side is based on a reactive uni-directional flux architecture for resolving flaws of the traditionally used Model View Controller (MVC) pattern [50, 51]. Sprotty internally defines a JSON protocol for communication between client and server to

[5] ANTLR Website
[6] Xtend Documentation
[7] Sprotty on GitHub

balance the workload of operations by either passing messages to the server or processing them locally.

The client-side of Sprotty is implemented in TypeScript, while the server-side offers a Java and Node.js-based architecture. Configuration of Sprotty, both on the client- and server-side, is done through DI with InversifyJS and Google Guice as a framework respectively. DI allows adding own model elements, figures, behavior, and adapting default implementations. Through the underlying technologies, interoperability of Sprotty is greatly increased, and the framework offers pre-existing integrations with various platforms and tools. An overview of the currently available Sprotty packages and sub-packages is given in Table 2.1. We note that the overview only includes the relevant packages in the context of this thesis and there are additional integrations available that are not listed in the table, e.g., with Theia, Langium, and Eclipse RCP applications. We discuss Sprotty in more detail in later parts of this work (see Chapter 3).

Name	Package	Sub-package	Description
Client	`sprotty`		Client-side code for diagrams
		`protocol`	Code for Node.js-based servers
		`elk`	Client-side layout with ELK
Server	`sprotty-server`		Java bindings for the Sprotty API
		`layout`	Server-side layout with ELK
		`server`	Base library for servers
		`xtext`	Integration with Xtext-based language servers
VS Code	`sprotty-vscode`		Integration with VS Code
		`extension`	Library to create Sprotty VS Code extensions
		`protocol`	Code for communication between extension and webview
		`webview`	Library for implementing webviews with a diagram

Table 2.1: Overview of relevant Sprotty Packages

2.2.5 VS Code

VS Code is currently the most popular IDE in the context of web development according to a recent survey[8] and is mostly open-source, with high extensibility and customizability. Due to it being web-based and lightweight, VS Code can be run on all major hardware and platforms. The cross-platform desktop app can be installed on macOS, Linux, and Windows. Recently, VS Code also became available in the web browser[9], however, with certain limitations compared to the desktop app, e.g., limited file access in certain browsers. Internally, VS Code is based on the application framework Electron and written in TypeScript. Electron is built on top of Chromium and the Node.js runtime, bringing the benefit of standard web APIs to VS Code.

High extensibility is enabled through its extension API[10], allowing extensions to interact

[8] StackOverflow Developer Survey 2019
[9] vscode.dev
[10] VS Code Extension API

```
import * as vscode from 'vscode';

export function activate(context: vscode.ExtensionContext) {

    this.context.subscriptions.push(
        vscode.commands.registerCommand('example.hello', () => {
            vscode.window.showInformationMessage('Hello!');
        }));

    this.context.subscriptions.push(
        vscode.commands.registerCommand('example.newFile', async () => {
            const newDocument = await workspace.openTextDocument({
                content: '\n\n\n\nHello',
                language: 'txt',
            });
            await window.showTextDocument(newDocument)
        }));
}
```

Listing 2.3: Source Code for a VS Code Extension with Command Registration

with VS Code and customize UI components, editors, and themes. An abundance of extensions are freely available in the marketplace[11] and developing extensions is relatively uncomplicated with the thoroughly documented API. VS Code offers common extension capabilities (e.g., commands, configurations, keybindings, storage), theming (e.g., UI colors, editor, file icons), and language extensions (declarative and programmatic language features).

Language extensions are a key aspect in this thesis, and further discussion is provided in later chapters. For now, as an introductory example to the extension API, we discuss the core concept of commands for triggering actions. VS Code extensions always include a manifest that defines metadata together with contribution points. The contribution points include commands that are by default exposed in the command palette and can also be specified to appear, e.g., in context menus or the editor toolbar. Commands are referred to by a name that is defined in the manifest, and they are registered when the extension is activated. Listing 2.3 shows the implementation of two command registrations within the corresponding method that is called upon activation. In line 1 we import the extension API and register a command to display an information message and another command to open a new text file with pre-defined content. Figure 2.4 shows the extension in action with the command palette that contains our two commands. In the background, the file that is created when executing the *New File* command is open, and the information message of the *Hello* command is displayed in the lower-right corner.

[11] VS Code Marketplace

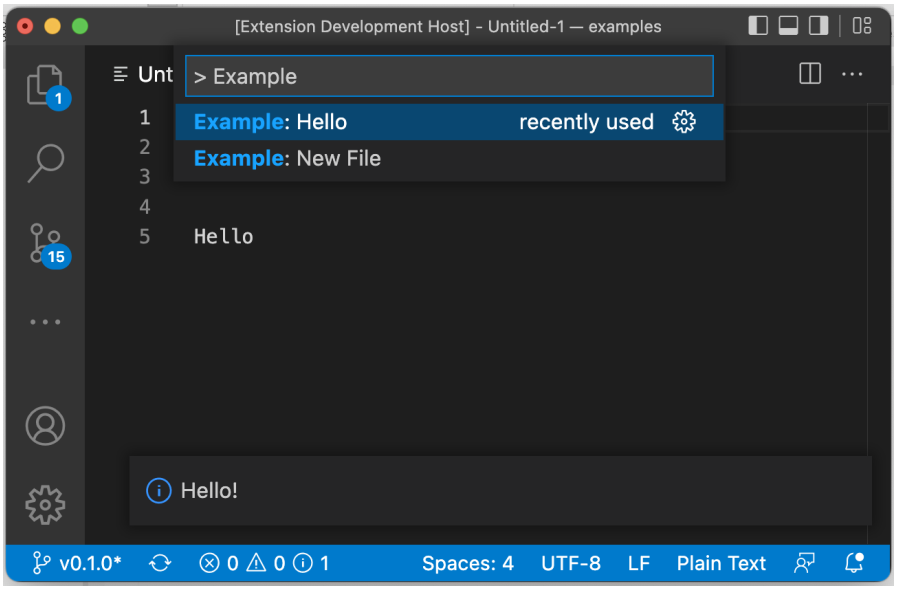


Figure 2.4: VS Code Extension with Commands in Action

2.3 Entity-Relationship Modeling

The Entity Relationship (ER) model is a high-level, conceptual data model, first introduced by Chen [52] in 1976. ER modeling was originally proposed to combine the advantages of previously dominating data models, namely the network model, the relational model and the entity set model. The core modeling concepts include *entities*, *relationships*, and *attributes*, to describe real-world objects and their corresponding properties, together with relationships to other objects. The visual representation of a model is captured in an *ER diagram*, consisting of different shapes and labels that ultimately result in an abstract and unified view of the modeled data. ER modeling is a common practice in the domain of conceptual database design, to visualize the initial database structure in form of ER diagrams, which later serves in the creation process of a logical schema [53]. Conceptual database design with ER models allows for cross-business communication, e.g., between business analysts and database designers, when advancing from the requirement analysis to the logical design of a database [54].

With its long history, there have been various adaptions of the underlying, generic ER model to other domains, featuring varying syntax and semantics, e.g., the temporal- [55], probabilistic- [56] or leveled [57] models. The Extended Entity Relationship (EER) model is presumably the most prominent extension, and includes concepts such as generalization or specialization [58, 59]. A comparative analysis of the different notations can be found in [60]. ER models consist of simple, yet expressive concepts to describe conceptual models. In their basic form, conceptual models comprise entities, relationships and attributes, however, additional refinement of these elements allows the expression of constraints. In the remaining of this section, we successively introduce the fundamental concepts of

the ER model, whereas we mainly focus on generic concepts (as originally introduced by Chen [52]). However, we further back our description by additional literature of the database domain [53, 54, 61] and for a formal definition, including extension of the ER model, we refer the reader to [62].

2.3.1 Entities and Attributes

An *entity* represents a real-world object from a specific problem domain and is distinguishable from other objects. The existence of the represented objects can be of physical- (e.g., person, building, vehicle) or conceptual nature (e.g., company, school). Entities are identified by a corresponding name and a list of *attributes*. Attributes are used to describe individual properties and characteristics of an entity. ER models generally allow for further classification of attributes, depending on how the data (representing an entity's property) should be stored. The most common classifications include differentiation between:

Composite or Simple — Sometimes an attribute can be further subdivided into multiple components, e.g. a name consisting of a first name and a surname. In such cases, we refer to them as *composite* (also referred to as complex). In contrast, if an attribute can not be further subdivided, we speak of *simple* attributes.

Multi-valued or Single-valued — As the name suggests, *multi-valued* attributes, can have multiple values to describe a property, e.g. a person can have multiple addresses. Attributes that only contain a single value, are called *single-valued*.

Derived or Stored - If the value of an attribute can be computed from another attribute, we speak of a *derived* attribute, as opposed to a *stored* attribute. For example, an age property of a person can be calculated based on the birthday, assuming birthday is a stored attribute.

Required or Optional — In some cases, instances of attributes do not have concrete values associated to them, i.e. containing NULL values. It is in general considered good practice to denote such attributes as *optional*. Contrarily, we refer to attributes as *required*, when it is necessary to have concrete values associated to them, i.e., not allowed to contain NULL.

To be able to uniquely identify entities, it is required to define a minimal set of attributes as a candidate key. The selected candidate key is called a *primary key*. Keys in ER models are called identifying attributes, as opposed to descriptive attributes. In many cases, the candidate key consists of a single attribute (often in the form of an artificially defined ID) but can also be composed of multiple attributes, referred to as a *composite key*. Sometimes the concept of foreign keys is also modeled in ER diagrams, however, foreign keys are generally not a concern of the conceptual design layer and are treated in a layer that is closer to the actual database application, such as the logical layer.

In Figure 2.5 we provide examples for representing different types of attribute classifications in a common graphical notation. The `student_id` attribute serves as the primary key for the entity `Student`. The entity contains the composite attribute `name`, which composes a `first_name` and a `last_name`. Furthermore, we have the multi-valued attribute `phone_number`, since a student can have multiple telephone numbers associated to them. Last, the entity has the derived attribute `age`, which can be calculated from the simple (also stored, single-valued and required) attribute `birthday`. We also want to note that the question, whether an object should be modeled as an entity or an attribute, is not always straight-forward and can be influenced by factors such as the modeling context or requirements.

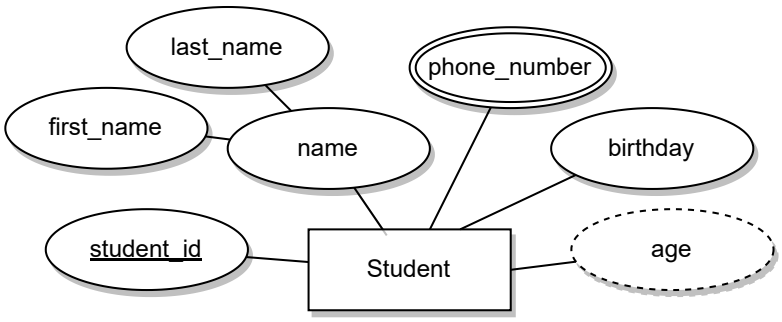
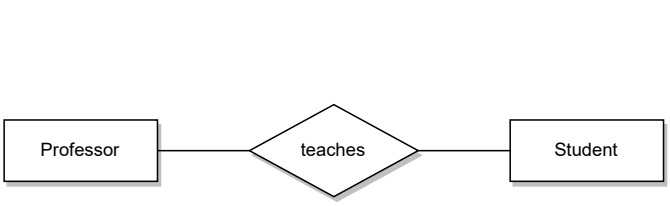


Figure 2.5: Entity with different types of Attributes

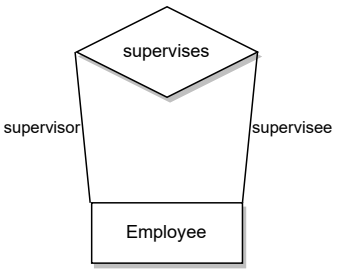
2.3.2 Relationships and Constraints

The ER model defines connections between objects (i.e. entities) with the concept of *relationships*. Similarly to entities, relationships are referred to by a name and can include attributes, however, with the exception of only descriptive attributes being allowed. A relationship additionally defines a set of associated entities, which we refer to as participating entities. In a mathematical sense, a relationship R is defined as the Cartesian product of participating entities $E_1, E_2, ..., E_n$, such that $R \subseteq E_1 \times E_2 \times, ... \times E_n$.

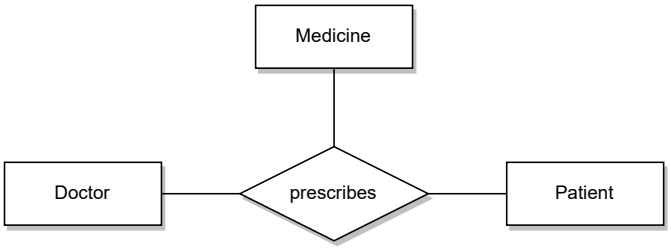
We define n as the degree, denoting the amount of participating entities in a relationship. *Binary relationships* have a degree of $n = 2$ and are most commonly used in ER models, e.g. a relationship between a professor that teaches students (Figure 2.6a). A degree of three (i.e. $n = 3$) is called a *ternary relationship*, e.g. a relationship between a doctor that prescribes medicine for a patient (Figure 2.6c). Higher-order relationships, beyond ternary, rarely occur in ER models, as they often add unnecessary complexity to models and are not covered further in this section. Participating entities generally play a certain role in a relationship, which can be denoted through *role names* and is commonly present in *recursive relationships*. Recursive (or self-referencing) relationships are a special case of a binary relationship R, where $E_1 = E_2$. Figure 2.6b shows this case and how role names are used to differentiate between the roles of a `supervisor` and `supervisee` within employee entities.



(a) Binary Relationship



(b) Recursive Relationship with Roles

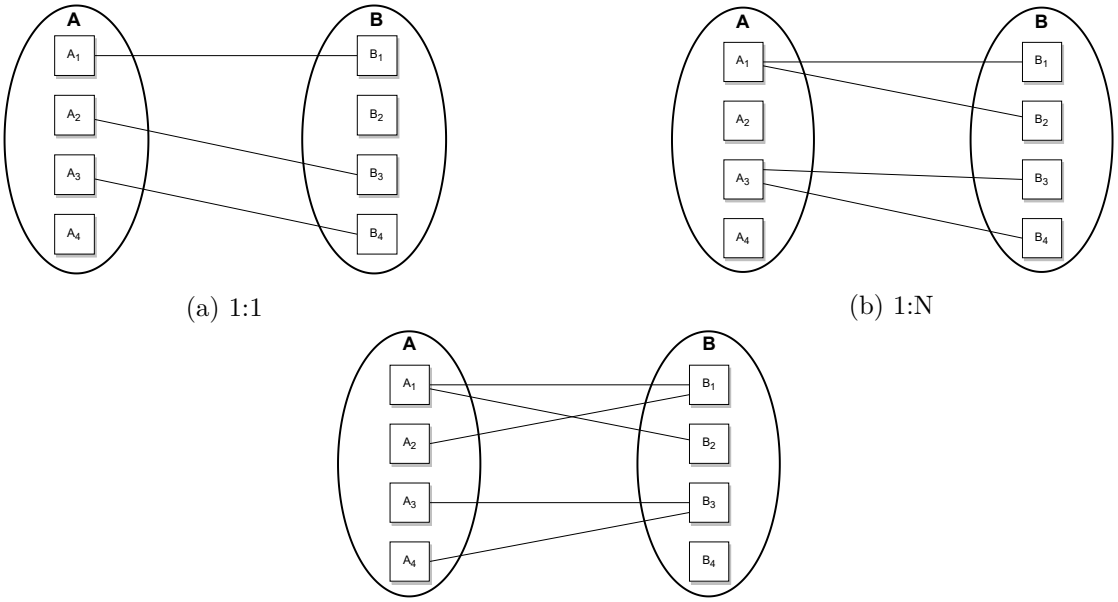


(c) Ternary Relationship

Figure 2.6: Degree of Relationships

Besides the degree, ER models may also specify structural constraints on relationships to limit the number of possible occurrences of entity instances within the relationship. Structural constraints are commonly defined in terms of *cardinality*, mainly differentiating between 1:1 (One-to-One), 1:N (One-to-Many) and N:M (Many-to-Many) relationships. Cardinality can be visualized in the form of mathematical relations between sets to aid in understanding the relationship constraints and how participating entity instances are allowed to relate to each other. In Figure 2.7 we show such representation of two entity sets, including relations between instances. Considering the modeling case from Figure 2.6a, we slightly extend the model to include cardinality constraints. By stating that one professor teaches zero or more students and a student can not be taught by more than one professor, we express a 1:N relationship (Figure 2.7c). We could also restructure the sentence to express a 1:1 relationship instead, such that a professor teaches at most one student and a student is taught by at most one professor (Figure 2.7a). Last, in case of an N:M relationship, we can state that one professor teaches zero or more students and one student is taught by zero or more professors (Figure 2.7b).

Sometimes cardinality is further broken down and expressed more detailed, in the form of a min-max notation, also referred to as *multiplicity*. This more refined notation allows specification of concrete values for the lower- and upper-bound of entity occurrences. Multiplicity is essentially the combination of cardinality- and participation constraints. Cardinality describes the maximum number of allowed relationship occurrences for participating entities, thus, corresponding to the upper-bound in the multiplicity range. In contrast, *participation* constraints, determine whether all instances of an entity occur



(a) 1:1

(b) 1:N

(c) N:M

Figure 2.7: Cardinality Constraints represented as Relations between Sets

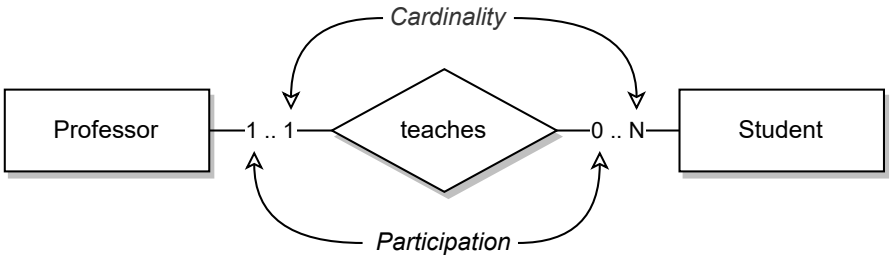


Figure 2.8: Difference between Cardinality and Participation Constraints

in a relationship (total/mandatory participation) or only some of them (partial/optional participation). This corresponds to the lower-bound in the multiplicity range. Figure 2.8 shows the differences between the two, with the model describing the case of exactly one professor teaching zero or more students.

We further note that definitions and representations of the mentioned modeling concepts and especially constraints can vary greatly when reading through different literature. For example, multiplicity is sometimes read from opposite directions (e.g. in the UML notation) or often only cardinality is expressed through values, while participation is either modeled as a single edge (partial participation) or a double edge (full participation). For thorough understanding of ER models, it is essential to be aware of the concrete notations and definitions used for the concepts.

2.3.3 Weak Entities

The final concept we cover are weak entities. So far, we only considered regular entities, containing a key attribute that is used for unique identification. ER models further allow the classification of a *weak entity* that can not be uniquely identified alone, and its existence depends on another strong entity. In this case, the dependency between the two entities is modeled in the form of an identifying relationship, with the weak entity defining a partial key attribute. The primary key then becomes a combination of the partial key together with the key attribute defined in the identifying, strong entity. Furthermore, the identifying relationship is constrained to be either 1:1 or 1:N, with N and a total participation constraint on the weak side. A weak entity and its identifying relationships are commonly modeled with a thicker border compared to the regular counterparts, this graphical representation is exemplified in Figure 2.9. In the figure, a chapter only contains an attribute for the section number, and thus, can not be uniquely identified alone. To express a dependency to the strong entity `Book` (containing the key `isbn`), we specify an identifying relationship between the two and declare `Chapter` as weak, with `section_nr` serving as a partial key.

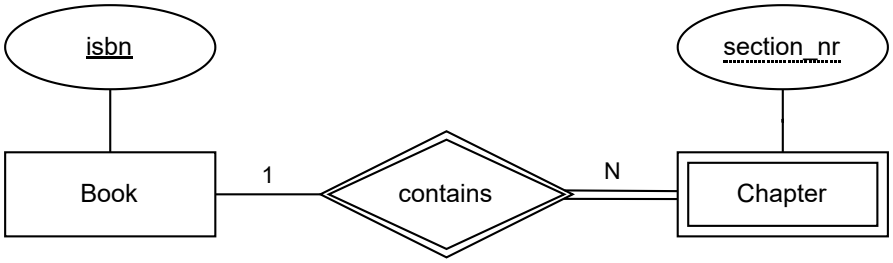


Figure 2.9: Weak Entity and Identifying Relationship

2.4 Related ER Modeling Tools

With ER modeling being around for a long period of time, there have been numerous contributions in providing tool support to this field. Traditional tool support ranges from educational use-cases that focus on the validation of ER models to complete data modeling applications for the design of databases. There is even support for ER modeling in general drawing applications, but they often disregard the underlying modeling semantics and are solely intended for graphical representations. Given the abundance of available tool support and for alignment with the context of this thesis, we restrict our focus in this section on recently developed ER modeling tools which are related and comparable to the BIGER Modeling Tool. We found a handful of tools that fulfill our restriction criteria and provide a comparison in respect to selected tooling aspects in Table 2.2. The basis for these aspects serves the BIGER Modeling Tool which is a web-based VS Code extension with the code being open-source available on GitHub[12]. It offers hybrid modeling support

[12] BIGER - GitHub Repository

based on the LSP with a textual- and graphical editor. Additional key features include code generation of database tables (in SQL) and model validation.

	dbdiagram	**ERDPlus**	**MIST**	**ERtext**	**vuerd**	**ERD Preview**	**bigER**
Open Source			✓	✓	✓	✓	✓
Web-based	✓	✓		✓	✓		✓
IDE Integration			Eclipse	Eclipse	VS Code, Atom	VS Code	VS Code
Textual Editor (DSL)	✓		✓	✓		✓	✓
LSP Implementation	✓			✓			✓
Diagram Rendering	✓	✓	✓	✓	✓	✓	✓
Graphical Editor	✓	✓	✓		✓		✓
Hybrid Modeling	✓	✓					✓
Code Generation	✓	✓	✓	✓	✓		✓
Model Validation	✓		✓	✓			✓

Table 2.2: Comparison of Tools and VS Code Extensions for ER Modeling

While the tooling aspects are biased to align with the strengths of the bigER Modeling Tool, the comparison serves in underlining the tool's unique contribution factor. It is the first real modeling tool deployed to the VS Code ecosystem that supports the LSP, making it relatively simple to realize additional support for other IDEs. Due to its web-based nature, the bigER Modeling Tool can technically also be embedded in web pages.

In the following, we discuss each of the selected tools for comparison in more detail and elaborate on their supported features:

- **dbdiagram** is available for web browsers[13] with a textual- and graphical editor that also supports hybrid modeling. The tool uses a language server of the DBML - Database Markup Language for creating database tables and their references textually, as well as for rendering the diagrams. The tool supports import of existing databases from, e.g., rails (schema.rb) or SQL (PostgreSQL, MySQL, SQL Server) as well as export of created models to, e.g., PNG images, PDF documents or SQL code. The UI has a modern look and feel with dark and light themes or custom table colors for premium users, as well as collaboration support in the form of sharing and version history.
- **ERDPlus** is also available for web browsers[14], however, ER models are created graphically together with form inputs, and it does not offer a textual editor with a language. It also supports creating relational- and star schemas with transformation from ER to relational schemas, generation of SQL DDL statements and export to PNG images. Usage of ERDPlus in the context of database design is described in [63].
- **MIST** is the official abbreviation of the Multi-Paradigm Information System Modeling Tool, which includes a DSL specifically designed for EER modeling

[13] dbdiagram Website
[14] ERDPlus Website

[64]. Through its underlying EER metamodel, it offers a bidirectional (graphical and textual) approach for conceptual modeling and model transformation into a relational data model, or a class model. The abstract syntax is specified in Ecore while the textual- and graphical concrete syntax is defined using Xtext and Eugenia respectively. MIST can be used in Eclipse and despite using Xtext, the tool is realized[15] with an earlier version that does not support the LSP yet.

- **ERtext** implements a Xtext-based DSL for conceptual modeling of relational databases [65, 66]. The code is open-source available on GitHub[16] and while technically being web-based through the LSP the tool is so far only supported within Eclipse. Textual models offer validation as well as support for generation of diagrams (PlantUML), logical schemas and SQL code (MySQL and PostgreSQL). The generated diagrams can not be graphically edited, thus, also lack hybrid modeling support.

- **vuerd** is a web-based and purely graphical ER modeling tool, with the code being open-source available[17]. The tool is available in two IDEs (VS Code, Atom) and it also offers a playground in a web browser. The graphically created ER models (stored as JSON) do not offer a corresponding textual editor, however, the models can be used to generate SQL code.

- **ERD Preview** is also available as a VS Code extension[18] offering a textual DSL for the specification of ER models. The DSL (erd-go) translates a plain text description of a relational database schema to a graphical ER diagram. However, the language does not implement the LSP and there is no support for graphical editing in the diagrams.

[15] MIST Github Repository
[16] ERtext Github Repository
[17] vuerd Github Repository
[18] ERD Preview VS Code Marketplace

CHAPTER 3

Developing Sprotty-based Modeling Tools for VS Code

In this chapter we deal with the first research objectives of this thesis, in particular, we contribute an approach for the development of modeling tools based on Sprotty. The approach is illustrated with a running example that is iteratively extended with new modeling capabilities over the course of this chapter. The example features a reasonably simple modeling language to focus on the principal aspects of the development process and with the aim to provide an approach that is as generic as possible. Despite its simplicity, the result is a full-fledged modeling tool, including a textual editor with rich text editing features and a synchronized graphical representation that supports hybrid modeling. Code of the running example is provided on GitHub[1] and structured according to each section.

The remaining of this section is structured as follows. First, we look into the architecture of Sprotty with a particular focus on the client-side and the key features. We implement a basic client-only diagram for our running example and discuss further customization options (Section 3.1). In the next section, we shift the focus to developing the underlying modeling language with Xtext, create a VS Code extension, and connect the language with the extension for textual editing support (Section 3.2). We then integrate Sprotty with Xtext, the LSP and VS Code, and implement a diagram corresponding to the textual editor (Section 3.3). In the last step, we fully utilize Sprotty and enable hybrid modeling support between the two representations for selected editing features (Section 3.4).

[1] GitHub Repository of the Running Example

3.1 Client-side Sprotty Diagrams

A high-level overview of Sprotty is provided in the previous chapter (see Section 2.2.4), we now draw on the initial introduction and describe the technical aspects of the framework more in depth. The primary focus in this section is on the client-side of Sprotty such that we can later leverage the framework in a client-server scenario and add diagram support to a textual language with a server. For this, we first look into the base architecture and the concepts needed to create and manipulate diagrams. We then make use of the established architecture in our running example and implement a client-side diagram in the browser. The example is relatively simple and intended to highlight the generic implementation steps, thus, we provide hints on further customization as well.

3.1.1 Architectural Overview

Before we look into the actual components that are part of the core architecture, we first introduce certain underlying concepts that are relevant for creating and manipulating Sprotty diagrams.

SModel, Actions and Commands

Sprotty uses an internal representation for models called the *SModel*, which is basically a composition of JSON objects containing different properties, e.g. ID, position or size. The `SModelElement` interface serves as the base for all other elements with the required properties for a type and ID, as well as an optional list of child elements and CSS classes. Other elements inherit this class and define further properties. A class diagram of all the predefined elements, including their extensions, is shown in Figure 3.1. Sprotty also offers an extension of the SModel that can be used for representing graph-like models called an *SGraph*. An SGraph represents the root of the graph structure and consists of child nodes (`SNode`) that are connected by edges (`SEdge`). An edge requires a source and target property, which is set in the form of an ID from either a node or port (`SPort`). There are different ways to route edges and nodes can further define compartments (`SCompartment`) to include elements such as labels (`SLabel`) and other nested nodes.

Interactions and manipulations on the diagram are encapsulated as *actions*. An action in its most basic form consists of a `kind` property that is used to declaratively describe an operation on the underlying model. More complex operations include additional properties to further describe the behavior. An example for the `SelectAction`, used to select and deselect certain elements, is given in Listing 3.1. The information, which elements should be selected and deselected, is provided through additional properties that contain a list of strings, corresponding to the ID's of the elements. Actions can be represented as simple JSON objects, thus, they also serve in forwarding protocol messages to a remote server in a client-server application scenario.

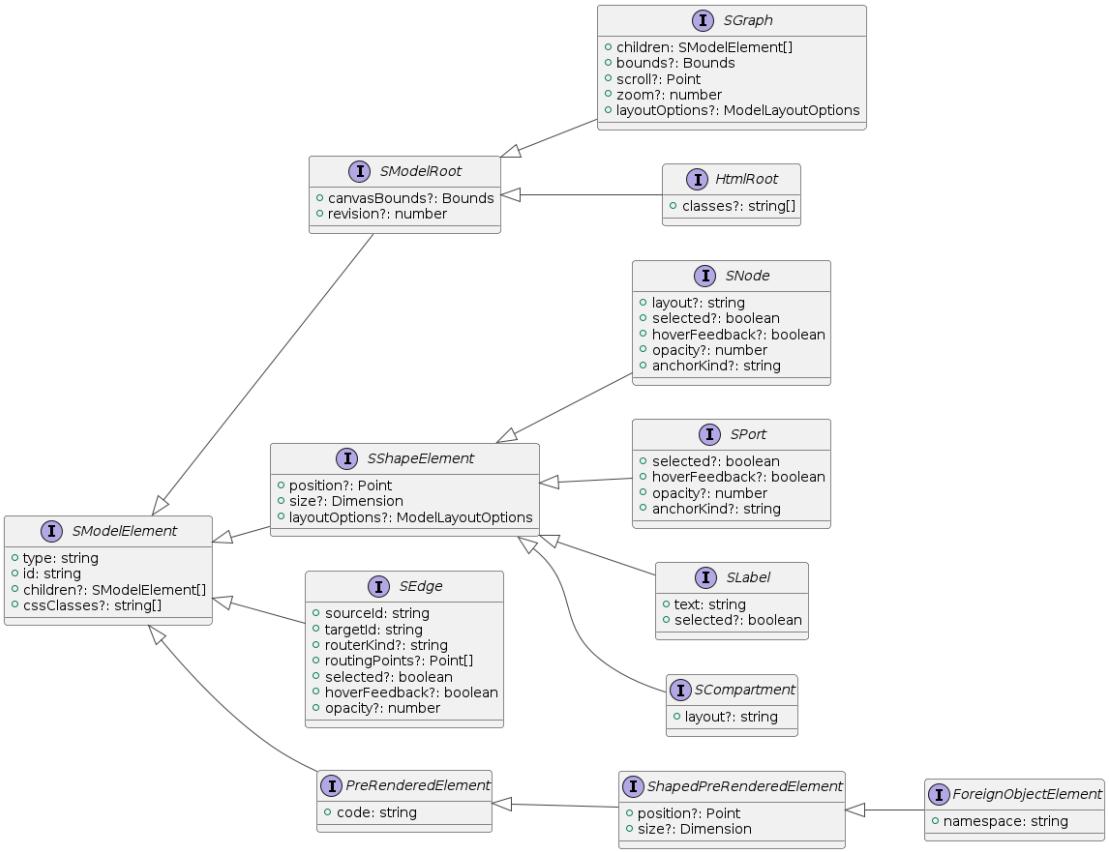


Figure 3.1: SModel Class Diagram

```
export interface SelectAction {
    kind: typeof SelectAction.KIND
    selectedElementsIDs: string[]
    deselectedElementsIDs: string[]
}
```

Listing 3.1: Select Action Example

To perform an operation on the diagram, actions are converted to *commands* that implement the actual behavior of the operation. Each command implements the methods `execute()`, `undo()`, and `redo()` with each of the methods receiving the context, containing the current model, as a parameter. Within the methods, the respective behavior for an action is implemented and an updated SModel representation is returned for further processing.

Core Components

The core architecture of Sprotty is shown in Figure 3.2, displaying how the previously introduced concepts are passed around different components. The architecture implements a unidirectional cyclic event flow that is commonly present in reactive flux applications [51] and is an improvement to the traditional Model View Controller (MVC) pattern [67].

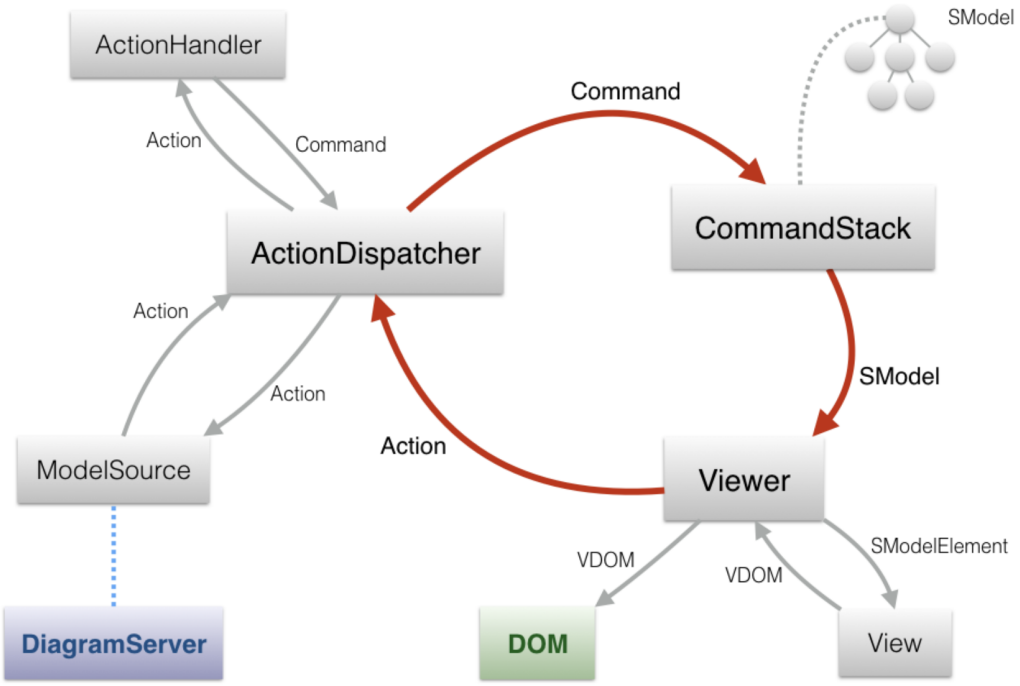


Figure 3.2: Client-side Architecture of Sprotty [67]

A `ModelSource` serves the model to the event cycle and is the entry point for the client-side of Sprotty. In a client-only scenario, a `LocalModelSource` is used to generate a model representation, while in a client-server scenario a remote model source is used that is connected to a `DiagramServer`. All form of actions are received in the `ActionDispatcher` and delegated to a respective `ActionHandler`, where actions are converted to commands. The converted commands are then passed by the dispatcher to the `CommandStack` to update the underlying model. Commands are chained and merged on a stack for supporting undo and redo operations, as well as for animating changes in the diagram. The updated model is passed to the `Viewer`, where the diagram is rendered in a virtual Document Object Model (DOM). The viewer maps each model element to a corresponding view by looking up its ID in the view registry. Event listeners and animations are added to the DOM through decorators, allowing to create new actions in the viewer and passing them to the dispatcher, where the cycle continues.

Sprotty can be customized using Dependency Injection (DI) through the lightweight

inversion of control container InversifyJS[2]. Diagrams are implemented by creating a DI container where bindings to services and the mapping of SModel elements to their view classes are defined. Sprotty provides several default container modules with bindings that can be reused by loading them in an own diagram container, and the injected dependencies can then be used with the `@inject` keyword. The framework is built to be highly modular and customizable, thus, all the default components can be customized by rebinding them to custom implementations. A list and description of the provided default modules is shown in Table 3.1 with each module corresponding to a certain feature.

Feature Module	**Description**
bounds	Bounds (position, size), layout (hbox, vbox, stack) and alignment
button	Button handler and SButton model element
command-palette	Default command palette (invoked with Ctrl + Space)
context-menu	Default context menu (invoked with right-click)
decoration	Decorate elements, e.g., issue Markers
edge-intersection	Handle intersections for edges
edge-layout	Handle edge layouting
edit	Create, delete and edit elements
expand	Expand elements
export	Export the diagram, e.g., to SVG
fade	Fade animation of elements
hover	Hover support for elements
move	Move elements in the diagram
nameable	Naming support for elements
open	Open elements
projection	Display projection markers on borders of the diagram
routing	Different routing styles for edges
select	Actions to select/deselect (all) elements or get current selection
undo-redo	Undo/redo operations, e.g., invoked by keybinding
update	Update the model
viewport	Viewport manipulation, e.g., zoom, scroll, center
zorder	Render elements in front of others

Table 3.1: Default Feature Modules provided by Sprotty

3.1.2 Implementation

By understanding the architecture of Sprotty we can now make use of the framework and go into the implementation procedure. Before we can use Sprotty, we need to set up a basic web development project to be able to display the diagrams in a web environment, such as in a browser. Sprotty is available as an npm package[3], as such, it makes sense

[2] InversifyJS on GitHub
[3] npm Website

to use npm or a similar package manager for managing Sprotty and any additional dependencies. For the remaining of this thesis, we use yarn[4] as a package manager. For the initial project structure, we merely created an HTML file for the diagram container and defined the dependencies. The dependencies mainly include `sprotty` (v0.11.1), for client-side diagrams and `http-server`, for running the application on localhost.

With the project set up and able to be run in a web browser, the diagrams are ready to be implemented. The official documentation of Sprotty provides a good starting point, and the page for getting started[5] provides a basic implementation procedure. The procedure is roughly split into four stages and centered on specifying the following components:

1. **Model** — Definition of a model for the internal SModel representation.
2. **Views** — Definition of views (using JSX, SVG) and styling (CSS) for rendering the model.
3. **Configuration** — Configuration of Sprotty in a diagram container using Dependency Injection.
4. **Model Source** — Connecting a remote- or local model source to the configured diagram container.

In the following, we describe the individual stages in more detail by incorporating the procedure in our running example and creating a basic diagram.

Model

The first step is to choose a schema for the model representation defined through SModel elements and forming a tree structure. As mentioned, the interface `SModelElement` provides a generic representation that can be reused or arbitrarily extended to capture additional behavior and properties. The root for a model is defined through an `SModelRoot` element with interfaces provided for HTML content and graph structures (consisting of nodes, edges, ports, compartments and labels). Interfaces extending `PreRenderedElement` can be used for HTML and SVG code, e.g., for complex figures, computing the view on the server, or including existing shapes. In our example, we do not need to override any of the provided model elements and simply reuse the existing `SGraph` model with `CircularNode` and `RectangularNode` to represent the elements.

Views

Sprotty uses the type property of model elements to map them to corresponding views and rendering the diagram in a virtual DOM using the snabbdom library[6]. Views implement

[4] yarn Website
[5] Sprotty - Getting Started
[6] snabbdom on GitHub

the base interface `IView` with the `render()` method that returns a virtual node (`VNode`) element for a corresponding SModel element and rendering context taken as a parameter. JSX and SVG is used for specifying the view representation, and additional styling can be applied through CSS. Views commonly return an SVG container element (`<g>`) containing shapes such as rectangles (`<rect>`), circles (`<circle>`) or text (`<text>`). Based on the passed model element, the shapes can further specify additional classes and properties, e.g., to define different styling whether the element is selected or to set the position and size. Sprotty provides defaults for views of various elements that can be reused and further extended. For our example, we again reuse provided views for each of the element schemas, namely `SGraphView`, `RectangularNodeView` and `CircularNodeView`.

Configuration

The next step is the configuration of Sprotty by creating a DI container with the InversifyJS framework. As listed in Table 3.1, various default modules are provided for certain features that can be loaded individually or all at once into a container. Sprotty components are denoted in the `TYPES` constant with respective symbols that can be used to add new bindings or rebind their defaults to custom implementations. Custom bindings are defined in a new container module, together with a registration of the model elements and their configuration for mapping them to views. In the example, we create an empty container that loads all the default features of Sprotty using the provided method `loadDefaultModules()` and override the defaults by loading a custom container module. Within the custom container module, we specify the mapping of the model elements to views, using the `configureModelElement()` method that combines registration and configuration. Last, we declare paths to required CSS files and return the loaded DI container in a function.

Model Source

The last step is to declare the model source that is responsible for generating a representation of our model and set values for the required properties of the elements. Since we primarily focus on the client-side of Sprotty in this section, we use a local model source and generate the model locally by creating concrete instances of SModel elements. Opposed to the local approach, a client-server scenario would use a remote model source with a diagram server to generate a JSON representation with the structure of the defined model on the server-side. The `LocalModelSource` class is the default implementation for the local approach, which can be used by binding the class to the model source symbol in the DI container. For displaying the diagram, the final application has to create an instance of the DI container, the action dispatcher and the model source to generate, set and update the model. In the example, we combine this process and generate an `SGraph` element that contains child nodes with specified properties for the position and size. To add basic user interaction, we add buttons with event listeners to create elements by making use of the `CreateElementAction`. The example is finalized by encapsulating

the procedures of a local model source in a method which is called in the main entry point of the web application. Figure 3.3 shows the result of the implemented example as displayed in the browser, with additional nodes created by the respective buttons, and two of the nodes selected.

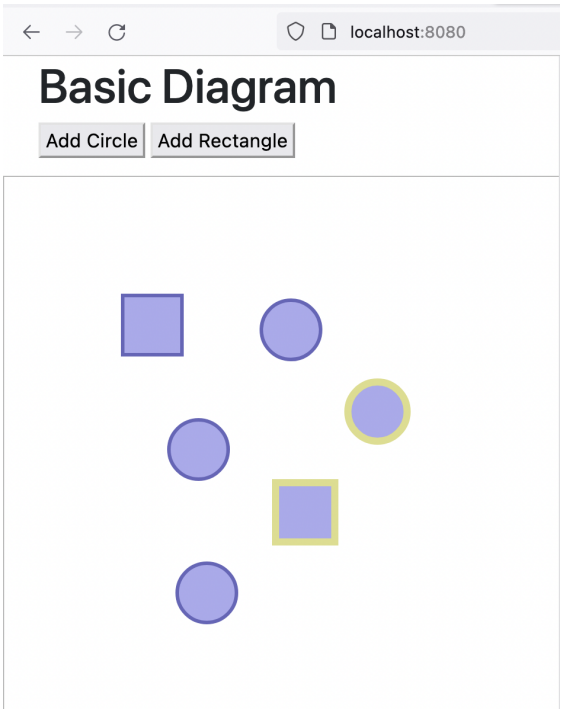


Figure 3.3: Result of the implemented Example Sprotty Diagram

3.1.3 Further Customization

The example only covers the implementation of Sprotty diagrams in a basic manner, and there are loads of customizable functionalities left untouched. Thus, we introduce a small subset of selected features to give additional hints on how Sprotty can be further customized. Naturally, new features can always be implemented from scratch, however, the effort is significantly decreased by adapting the provided defaults. Features provided by Sprotty target a wide variety of common diagramming use-cases, and it is often advantageous to look into the available components of modules, as they often contain supplementary components that are not loaded by default. Official examples in the repository[7] can serve as reference implementations for different diagram types. In the following, we discuss two of the examples in more detail and put additional capabilities of the framework to the foreground.

The first example, shown in Figure 3.4, generates a total amount of 50 nodes and 100 edges, with each node containing a label and the source and target of edges chosen

[7] Sprotty Examples

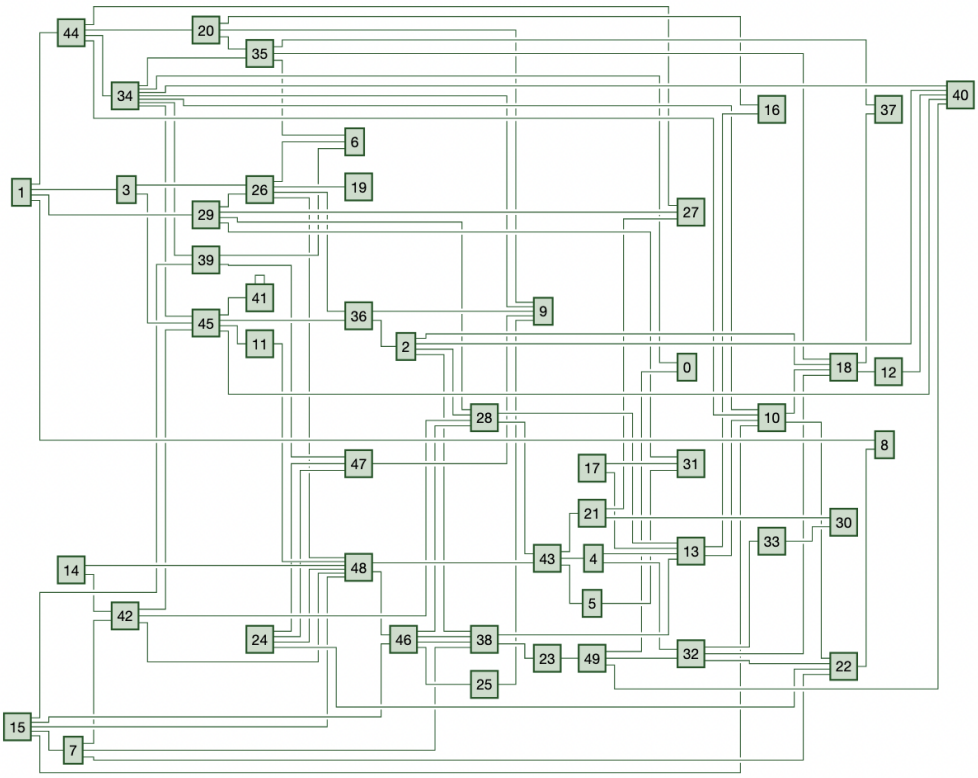


Figure 3.4: Sprotty Example - Randomly generated Graph with Automatic Layout

randomly. With the amount of elements, the model is presumably large, and the diagram requires the implementation of an advanced layout mechanism to aid in understanding the representation. Sprotty integrates the Eclipse Layout Kernel (ELK) to provide advanced layout algorithms for graph visualizations, that can be used both on the client-side and the server-side. ELK ships with different standard layout algorithms that can be reused and further customized. In this case, the example takes advantage of the standard layered algorithm and additionally uses edges with gaps on intersections to display an appealing visualization of the graph.

The second example (Figure 3.5) showcases a simple class diagram with different kind of model elements and various available interactions. The model consists of nodes for classes that can be contained within a package node. Class nodes have a header containing an icon, and their attributes can be expanded or collapsed through the respective button. The example also shows three different types of edges, namely: a curved Bézier edge, a standard straight edge and a straight edge with Manhattan routing. Labels are positioned on different locations and the routing handles can be changed for all the edges in the diagram. When hovering over classes, a popup shows up, enabled by a custom `PopupModelProvider`. The popup model defines an additional `SModelRoot`, besides the one in diagram, that can also include other model elements, such as pre-rendered

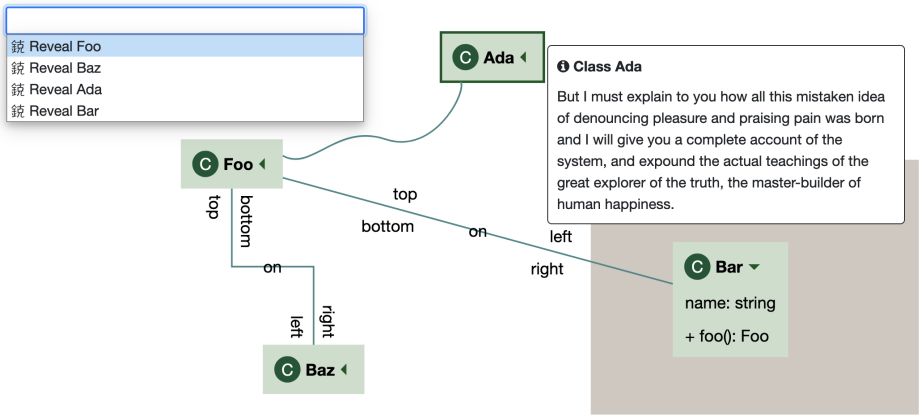


Figure 3.5: Sprotty Example - Class Diagram

HTML as shown in the example. Regarding user interaction, the example allows renaming all the labels with additional label validation, e.g., when the name is empty, and also comprises a command palette for centering a specific class node.

3.2 Xtext Language and VS Code Extension

Besides a diagram allowing only graphical interactions with a model, the basic editor we implemented does not utilize the full potential of Sprotty yet. We want to connect the diagram to a textual editor and take advantage of rich text editing support in VS Code. For this, we require a textual language with a language server that can communicate with VS Code through LSP messages. The language workbench Xtext is a good fit for this scenario, as it allows us to quickly create a DSL with a language server. As the server communicates through the LSP, the language can be supported within VS Code by creating a language extension.

In the following, we describe the process of creating a textual modeling language and adding support in VS Code. First, we show how to create a language with Xtext and generate a language server. Then we cover the creation of a language extension for VS Code, including customization of declarative language features. Last, we add support for programmatic language features by connecting the language server with a VS Code language client.

3.2.1 Xtext Language Server

Creating a language from scratch is not an easy task, and there are numerous complex components to consider during the development process. It is crucial in being aware of underlying aspects when creating a programming language. Two major components are the lexer for lexical analysis and the parser for syntactic analysis. The lexer converts a sequence of characters into tokens by using regular expressions, while the parser makes

sure the structure of statements is valid. After successful lexing and parsing of a program, commonly an in-memory representation is created for semantic analysis. The internal representation is usually in the form of a tree structure, such as the Abstract Syntax Tree (AST). Only after these steps, a program can be interpreted and generation enabled.

It is self-evident that implementing such language features quickly becomes complex and can easily be prone to errors if not done correctly. When trying to integrate a language, e.g., into an IDE, this process becomes even more unmanageable and for this reason the Xtext language workbench comes into play for efficiently creating a DSL. Xtext requires only the specification of a grammar for generating a whole language infrastructure. Internally, ANTLR is used for parsing and EMF serves in representing the AST. Languages created with Xtext can thus be integrated with existing modeling tools that are also based on EMF. Furthermore, the created languages can be easily reused in different IDEs as it is one of the first language workbenches supporting the LSP. All the generated language components can be customized through Dependency Injection with Guice.

In the following, we cover the required steps for creating a basic language for our running example, including a language server and LSP support. Since the implementation of Xtext languages is not our primary focus in this thesis, we do not cover additional customization of language components and instead refer to the official documentation [46] as well as the comprehensive book [47].

Project Creation

The easiest way to automatically generate a language server is by creating a new project and selecting the corresponding options in the creation process. For that, we open an Eclipse workspace and create a new Xtext Project (`File→New→Other→Xtext Project`). Alternatively, for existing EMF-based languages, the wizard also supports creating a new project from an existing Ecore model. Within the project wizard, we enter a name for the project and language, together with an extension that defines the type of files where our language should be used. At this stage, it is important to not finish the creation process and instead go to the next page for advanced configuration. Here we define the facets to allow Xtext to generate the packages for the language server, Figure 3.6 shows the specified options. For a language server, we require generic IDE support and a regular build. In addition, but not required, we add testing support and choose Gradle as the build system with a Maven/Gradle source layout.

Once the project creation is finished, the wizard creates the packages for the selected facets. Namely, this consists of the root package (`org.example.basic.parent`) with submodules for the language (`org.example.basic`) and the generic IDE components (`org.example.basic.ide`). The next step is to specify a grammar and customize the generated defaults.

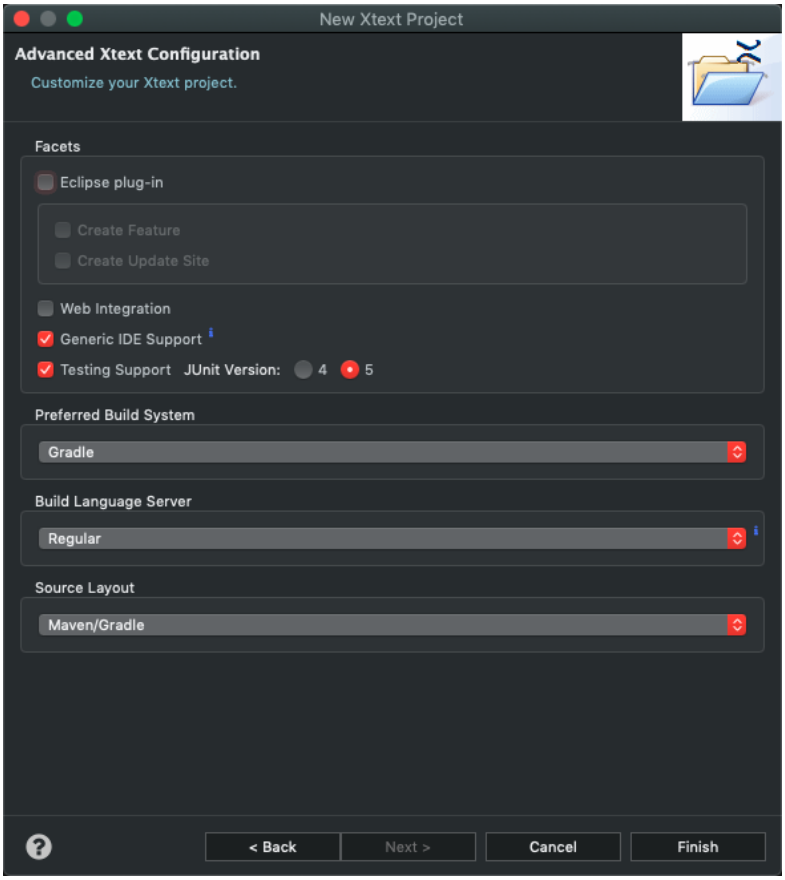


Figure 3.6: Advanced Xtext Configuration Page

Grammar

The grammar for the language is specified in the `.xtext` file located in the language submodule of the created project. We specify a grammar for a basic modeling language that allows creating shapes as shown in Listing 3.2. As the official documentation already offers a comprehensive overview of the language used for specifying Xtext grammars[8], we assume the reader to be familiar with the basic concepts and do not further elaborate on the grammar.

With the specification of our language finished, we can now generate the corresponding infrastructure, by right-clicking on the file and selecting the option for generating Xtext artifacts. The Xtext code generator then uses an MWE2 workflow to generate artifacts and derive an ANTLR parser with an Ecore metamodel for the grammar. EMF is used for representing the AST and it is also leveraged by Xtext for generating Java interfaces with implementations for each rule of the specified language concepts.

[8] Xtext Grammar Documentation

```
grammar org.example.basic.DSL with org.eclipse.xtext.common.Terminals

generate dSL "http://www.example.org/basic/DSL"

Model:
    shapes+=Shape*
    connections+=Connection*;

Shape:
    Circle | Rectangle;

Circle:
    'Circle' name=ID;

Rectangle:
    'Rectangle' name=ID;

Connection:
    source=[Shape] '->' target=[Shape];
```

Listing 3.2: Basic Xtext Grammar for the Shape Language

Once the generation process is finished, the language can quickly be tested in an Eclipse editor with various IDE features available out of the box. The Xtext generator also created an implementation for a language server with LSP support. We cover how to integrate the language server component with an editor in a later step, when adding support for VS Code. More complex languages generally require customization of the generated components. Commonly this includes adding validation and scoping or implementing a code generator.

3.2.2 VS Code Language Extension

With the textual modeling language ready to be used in an IDE, we first require a VS Code extension that can support the language. There is no built-in language support within VS Code, but instead different APIs are available to incorporate language features into the IDE. VS Code differentiates between two types of language features:

Declarative Language Features that can be supported in form configuration files. This commonly includes, e.g., syntax highlighting, bracket matching, code snippets (templates) or toggling comments. In general, such features are declaratively defined such as through regular expressions, and are generally more primitive than the programmatic counterparts.

Programmatic Language Features include more complex functionality and are often available in a dedicated language server that utilizes a powerful programming language such as Java. These features become available through LSP messages and can include formatting, refactoring, diagnostics (validation) or hover information.

For now, we do not go into programmatic language features as they become relevant in a later stage, when connecting our language server with the extension. The current focus is on creating a new language extension for VS Code consisting of purely declarative language features. In the following, we explain how to scaffold an extension skeleton and customize the initial components.

Scaffold Extension

As with other previously used technologies, we first show the initial project creation steps required to create a new language extension for VS Code. To get started, we recommend using yeoman[9] with the VS Code extension generator for scaffolding a new extension project. The generator templates a `package.json` file and creates the base folder structure with relevant files, to get started with the development of extensions. The following command installs both of these two packages globally: `npm install -g yo generator-code`. Note that Node.js is required to be installed on the system to execute the command.

Once the installation is complete, the command `yo code` starts the generator. Within the newly opened prompt, we select *New Language Support* as the desired extension type and the remaining information should match with the existing language, but can be changed afterwards as well. Based on the provided information, the generator creates a corresponding project skeleton including all the necessary files needed for a language extension. The resulting folder structure, as opened in VS Code, is shown in Figure 3.7.

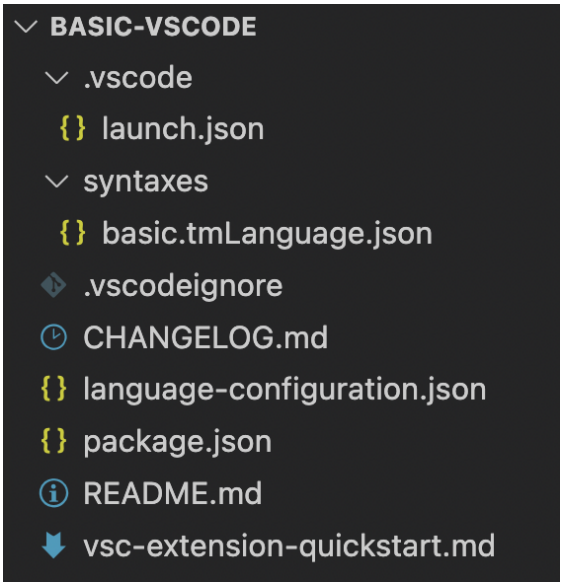


Figure 3.7: Folder Structure of the newly generated VS Code Language Extension

[9] yeoman Website

Most importantly, the generated project includes an *Extension Manifest* (`package.json`), with a *TextMate Grammar* (`syntaxes/erd.tmLanguage.json`) and *Language Configuration* (`language-configuration.json`). Also generated are markdown files for the project (`README.md` and `CHANGELOG.md`) and a configuration file to exclude certain files during packaging (`.vscodeignore`). The extension can be launched with the provided launch configuration (`.vscode/launch.json`) and for a quick start guide, one can refer to the corresponding markdown file (`vsc-extension-quickstart.md`).

Extension Manifest and Declarative Language Features

The extension manifest is required for all VS Code extensions and is located in the root directory of the extension. The file includes extension metadata (e.g. author, description) and application-specific fields. VS Code extensions run as a packaged Node.js application, so the package includes npm fields, such as scripts and dependencies. New functionality can be added to VS Code in the form of *Contribution Points* specified in the `contributes` field. The generator already added two contribution points to the extension, a language configuration and a grammar for syntax highlighting. Both of the respective fields should link to the corresponding files. The `engines` field declares which version of VS Code is compatible with the extension. At this stage, no further adjustments to the manifest are required, however, we want to note its importance for developing extensions. Documentation on the available fields is available in the extension manifest reference[10] as well as the npm package.json reference[11].

VS Code supports different components to customize the syntax highlighting for a language. The most basic form is *tokenization*, powered by a TextMate grammar. On-top of TextMate, VS Code also supports styling the tokens with *theming* and a more advanced tokenization approach with the use of a semantic token provider. As a first step, we modify the TextMate file to highlight keywords of the new language. The grammar uses regular expressions to map the contents of the text to tokens, so we define individual patterns to tokenize the corresponding keywords.

Additional declarative language features can be specified in the language configuration file. This includes features for the editor such as toggling block comments or bracket matching. The provided defaults in the file are sufficient for most languages, and in our case we keep them as originally generated.

Running the Extension

With the appropriate files defined for the declarative language features, the extension requires a main entry point to be run in VS Code. For this, we first define a new activation event in the `package.json`. The extension should be activated whenever a file in our language is opened in the editor, this behavior can be specified in the `activationEvents` field by adding `onLanguage:basic`. As a next step, we create a

[10] Extension Manifest Reference
[11] npm package.json Reference

main entry source file that implements the functions `activate()` and `deactivate()`. To initially test whether the activation events work, we use a function from the extension API for showing an information message in the activate function.

To finish the set-up, we create a webpack[12] bundle and define additional scripts for building the extension. Once the set-up is complete and successfully built, we can run the extension within an extension host of VS Code. The running behavior is defined in the `launch.json` file, and our running VS Code extension is shown in Figure 3.8. When opening a `.basic` file, the extension is automatically activated, as indicated by the information message. Furthermore, the declarative language features become available, e.g., syntax highlighting for specified keywords or commands for toggling comments in the command palette.

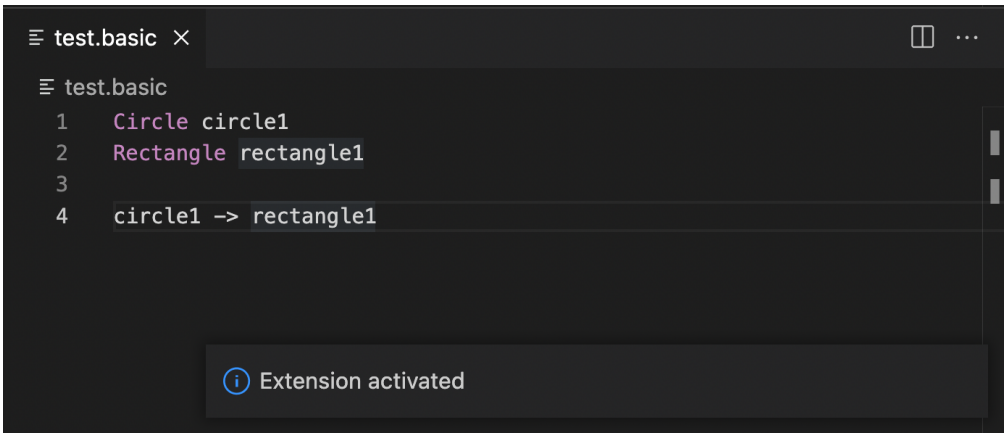


Figure 3.8: Basic Language Extension running in VS Code

3.2.3 Connecting the Language with the Extension

We successfully created a textual language with Xtext as well as a VS Code extension to provide declarative language features in the editor. The goal now, is to connect the two components to enable full language support in the editor and allow the language server to communicate programmatic language features to the extension.

The first step is making the language server available for the extension such that the VS Code language client can communicate with the server through the LSP. By default, when building the language, the Gradle build creates a zip file that contains the language server in the `distributions` folder of the `ide` package's build directory. In our case, the path to this folder is `org.example.basic.ide/build/distributions/`, and we could simply copy the zip file to the extension and unzip it to make it available. However, we automate this task, so that whenever the language is built the server automatically gets unzipped and copied to the extension.

[12] webpack Website

Now that the language server is available for the extension, we need to implement a language client. The VS Code API provides a language client, which we can create in the `activate()` method in our main extension file. We pass additional client and server options during creation for launching the executable of the server and can then start the client.

With the language server available and the language client implemented, the extension is ready to be rebuilt and tested. If we now open a file in our language, various new features become available in the editor. Xtext automatically provides generic implementations for programmatic language features, such as, e.g., auto-complete, rename refactoring or validation. Depending on the complexity of the language, the features provided by Xtext can vary. Figure 3.9 shows the provided validation feature of our language with a problem view, error markers and hover information.

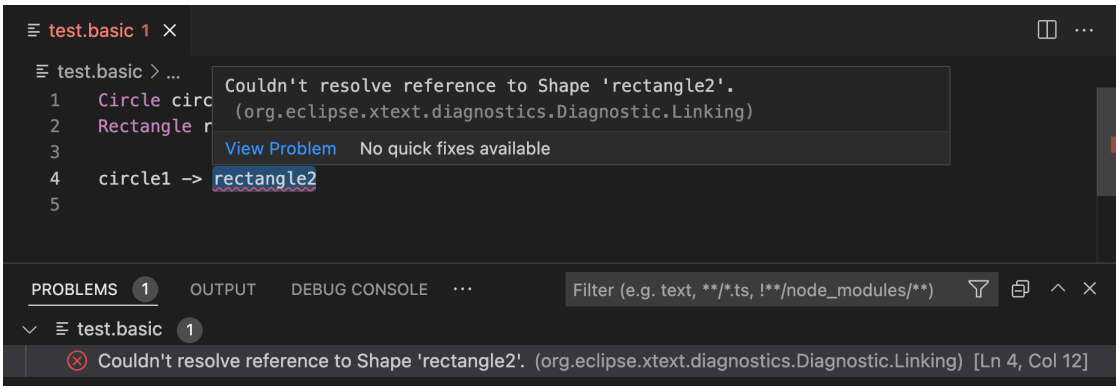


Figure 3.9: Validation Language Feature of the Example

3.3 Sprotty Integration with VS Code and Xtext

With the implementation of the textual language and editing features finished, we now leverage Sprotty to integrate with VS Code and Xtext, and implement a graphical view for the modeling language. Apart from the client-side package, Sprotty provides additional packages to integrate diagrams with other technologies. We use `sprotty-server`[13] to add graphical support to Xtext language servers, as well as `sprotty-vscode`[14] to create Sprotty VS Code extensions and display a graphical view in a panel next to the textual editor.

In this section, we first provide an architectural overview of our approach and then proceed with a description of the implementation process for integrating Sprotty, based on our running example.

[13] sprotty-server on GitHub
[14] sprotty-vscode on GitHub

3.3.1 Architectural Overview

Figure 3.10 provides a high-level overview of the architecture when integrating Sprotty with VS Code and Xtext. The architecture is realized in the form of a client-server application, with the client-side consisting of the extension and a web view, while the server-side includes a Sprotty-enhanced language server that extends the Xtext language server. In the following, we describe each of the components in more detail.

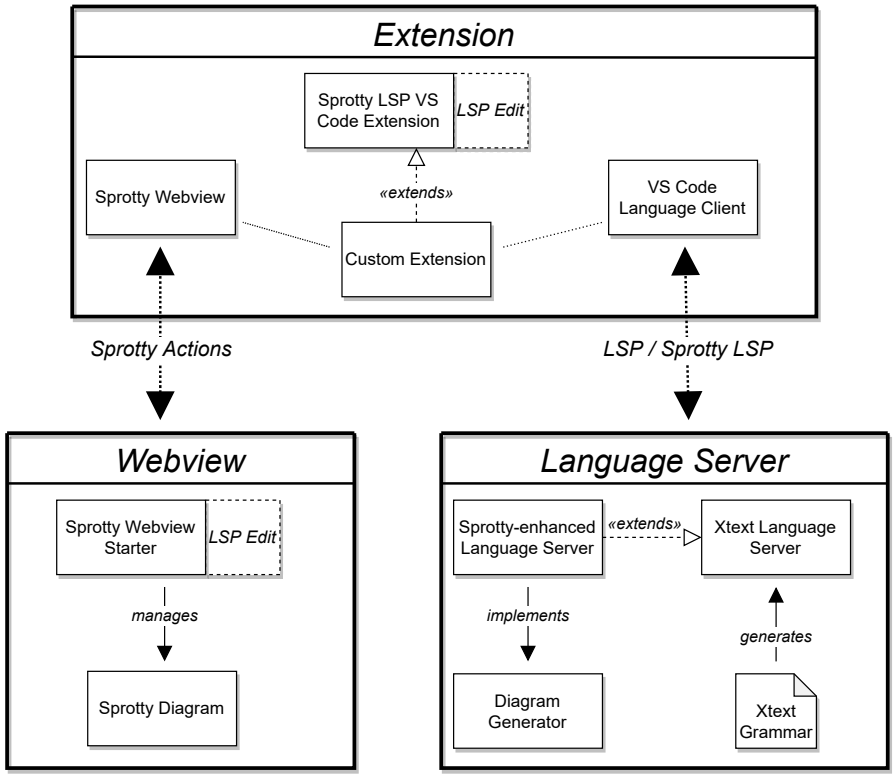


Figure 3.10: Architecture of Sprotty integrated with VS Code and Xtext

The extension contains the entry point and serves as the central component for communication between the web view and the language server. When the extension is activated, the respective activate method creates an instance of our custom extension. The custom extension extends the `SprottyLspVsCodeExtension` class provided by Sprotty, in case of adding editing support, the respective LSP edit class is used, as we show in the next section, when we enable hybrid modeling. The abstract base class for the extension already provides various functionality, e.g., registering commands or accepting messages from a language server. To glue the components to the extension, we merely need to implement the abstract methods `getDiagramType()`, `createWebView()`, and `activateLanguageClient()`. The web view creation method loads the bundled JavaScript file of our web view package, and the extension communicates with the diagram in the web view through Sprotty actions. The language client activation method

launches the executable of our language server, communication is established through LSP messages together with Sprotty LSP extension messages.

The **Webview** component is part of the client as well, but is separated from the extension in its own npm package. As mentioned, the extension accesses the web view component by loading a single JavaScript file, which is bundled, e.g., through webpack. The main entry point of the web view extends the provided abstract class `SprottyStarter` or `SprottyLSPEditStarter`. The starter classes set up communication with the extension, together with additional bindings for VS Code. The diagram is created in the abstract `createContainer()` method, in this method we simply return an instance of our DI container that contains the Sprotty configuration and defines the bindings of our model elements.

The server-side contains a Sprotty-enhanced language server that adds a diagram module to the language and generic IDE components of Xtext. An implementation of a *Diagram Generator* is required to generate a model representation on the server through SModel elements. Various diagram components are bound by default in the diagram module, and we further discuss the underlying components of the server-side in the implementation process next.

3.3.2 Sprotty-enhanced Language Server

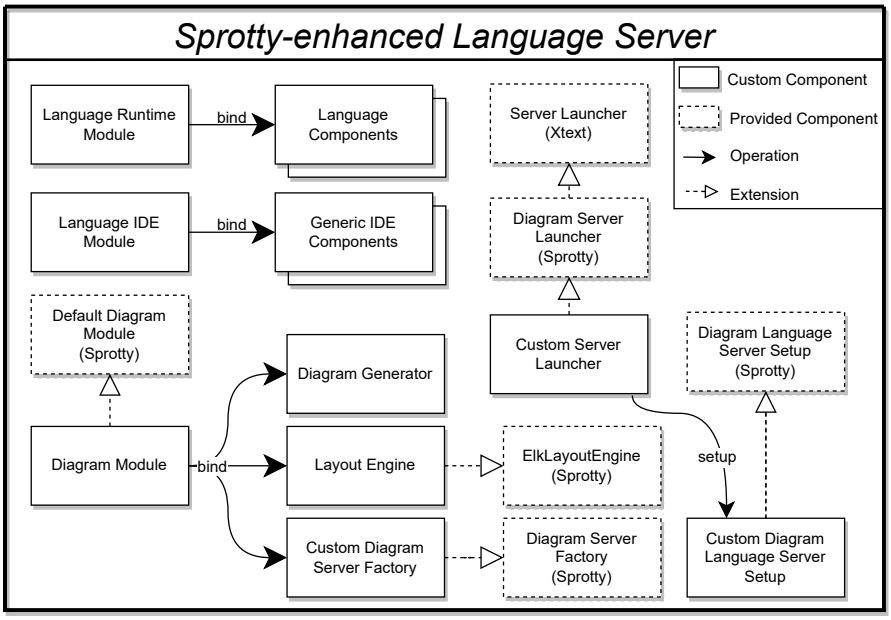


Figure 3.11: Sprotty-enhanced Language Server

As the server-side is slightly more complex, we provide a more detailed overview of the language server components in Figure 3.11. First, we create a Sprotty-enhanced language server by integrating the server-side of Sprotty into the language server and enhance

the LSP with Sprotty actions. For the implementation, we add `sprotty-server` with `xtext` submodule to the Gradle build of the `ide` package. The generated Xtext language server contains a setup class that creates Guice injectors for the language's runtime and IDE module. In the setup, we add a diagram module to the injectors, which extends the default module provided by Sprotty.

In the diagram module, we add bindings to the diagram generator that is responsible for transforming the language concepts to SModel elements, based on the derived Ecore metamodel. A Java API for the SModel elements is available in the server package of Sprotty. In our example we map the root to an SGraph, together with the rectangle and circle concepts of the language to SNodes and connections to SEdges. We further make use of server-side layout by using the provided layout engine, which internally is implemented with ELK. Alternatively, we could define a specific position and size property when generating the model elements, and make the client-side responsible for the layout. At last, we extend the default diagram server factory with a custom implementation, and define a diagram type that will be used for creating a diagram server for a client.

With the necessary components of the diagram module implemented, we additionally create a custom server launcher by using the provided diagram server launcher by Sprotty which further extends the server launcher generated by Xtext. In the launcher, we create a custom language server setup for registering ELK and Sprotty actions. The setup is based on a predefined class from Sprotty and replaces the language server with a diagram language server that hooks the diagram generator into the Xtext life-cycle. Sprotty provides both, a standard and a synchronized diagram server module to optionally synchronize selections of the text editor and diagram. For properly enabling the synchronization we add trace information to the generated graphical model that can also be used for displaying issue markers and popups, e.g., for validation errors, in the respective diagram elements.

3.3.3 Sprotty VS Code Extension and Webview

The Sprotty-enhanced language server is now ready to be connected with VS Code. For this, we have to implement two new components, a Sprotty LSP extension and a Sprotty webview. The extension connects to the language server and is responsible for managing the webview. This allows the extension to act as the central point for communication by tunneling Sprotty actions through the LSP and the webview protocol. Glue code to implement both, the extension and the webview, is provided in the `sprotty-vscode` package.

First, we implement the webview component that contains the diagram container. We can reuse most of the example we developed in the first section of this chapter that is using the client-side of Sprotty. Since the model is now generated on the server-side, we do not require a local model source anymore and adapt some bindings in the DI container. Generated model elements from the server are exchanged as JSON objects

by the extension to the webview, thus, the bindings of the elements should match with the types we defined on the server. The container is created in the main entry point of the webview and is handled by extending the class `SprottyStarter`. The abstract starter class includes additional bindings to VS Code, and we only have to implement a simple method that returns the customized Sprotty DI container. The final step for implementing the webview involves bundling all our code and dependencies into a single file such that it can be run by the extension. For bundling, we again use webpack, with a configuration file and a package script to copy the bundle into the extension.

Now, we move to implementing the extension component. The `sprotty-vscode` package provides abstract classes to create Sprotty extensions with different levels of language integration. Since we want to connect the diagram with our language in the editor, we use the provided class for an LSP extension and implement the required methods. There are three methods we have to implement: (*i*) a method to get the diagram type that aligns with the one defined in the server, (*ii*) a method to create an instance of a `SprottyLspWebview` that runs the script of our bundled webview and contains the Sprotty client with the diagram container, and (*iii*) a method to activate the language client for which we can simply reuse our existing implementation.

Next, we adapt the main entry file of the extension to create an instance of the LSP extension in the activation method. We also adapt the deactivation method to deactivate the language client with the provided method. As a final step, we make changes in the `package.json` file and define commands by reusing the default ones provided by Sprotty. Most importantly, this includes a command for opening the diagram that we expose in the command palette, explorer and the editor. With an additional activation event, we define that the extension should also be activated when opening the diagram.

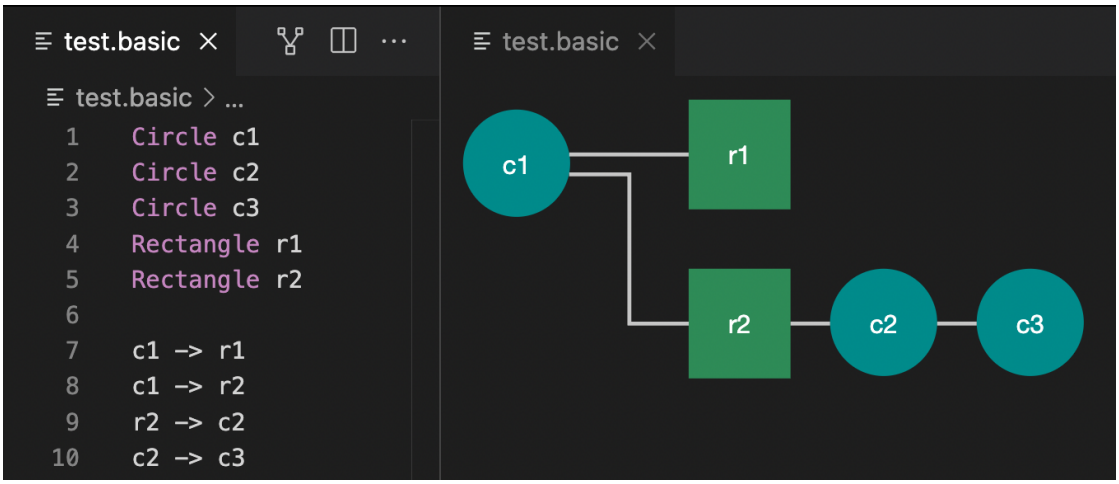


Figure 3.12: Sprotty VS Code LSP Extension Example

The integration of Sprotty with VS Code and Xtext can now be tested by running the extension and opening a model in our language. The final result of our example can be

seen in Figure 3.12 with the textual model on the left and the corresponding graphical representation on the right side. The diagram can be opened with the command we specified in the contribution field of our extension. Both views are fully synchronized on selection changes, meaning, if we select an element in the textual editor the diagram gets focused at the equivalent element in the diagram and the same behavior applies vice versa. When modifying the textual model, the diagram automatically gets updated and also displays validation errors. While the elements can be moved around in the diagram, the positions are not saved and reset on textual changes, since we use the layout engine in the server for automatically laying out the graphical model.

3.4 Hybrid Modeling

The VS Code extension we implemented throughout the last sections already provides a powerful modeling tool. However, support for an even better modeling experience can be achieved by fully utilizing Sprotty and enabling hybrid modeling [20]. Currently, the diagram only features selection changes that update the selection in the textual editor. However, by fully utilizing Sprotty we can enable hybrid modeling to add support for graphical editing in the diagram that also updates the underlying textual model if needed. Sprotty can enable hybrid model editors by mapping the graphical changes to textual modifications, mostly in the form of LSP messages, and notifying the server of the updates. For this approach, it is important to only update the diagram when the textual changes have occurred to avoid any inconsistencies between the two editors.

In the following, we show an approach on how to enable hybrid model editing with Sprotty by discussing which components have to be modified and then show selected editing operations. For discussing the implementation process, we extend the example developed in the previous sections.

3.4.1 Enabling Hybrid Modeling

As mentioned in the previous section, Sprotty offers different levels of extensions to VS Code. In the most basic form, an extension merely shows a diagram in a webview panel and is not connected to a language. The next level adds LSP support and synchronizes the diagram with the text editor, as implemented in the previous section. Now we leverage the last level of Sprotty extensions and make use of its LSP edit components.

Since we already created a Sprotty LSP extension and a Sprotty-enhanced language server, we can reuse the existing implementation and only have to apply minor changes. We enable hybrid modeling by now using the abstract classes `SprottyLspEditExtension` on the extension-side and `SprottyLspEditStarter` on the webview-side. No additional changes are required, however, since the approach for hybrid modeling is realized in form of LSP messages we either have to implement the graphical editing features on our own or reuse the ones provided by Sprotty.

3.4.2 Rename Labels

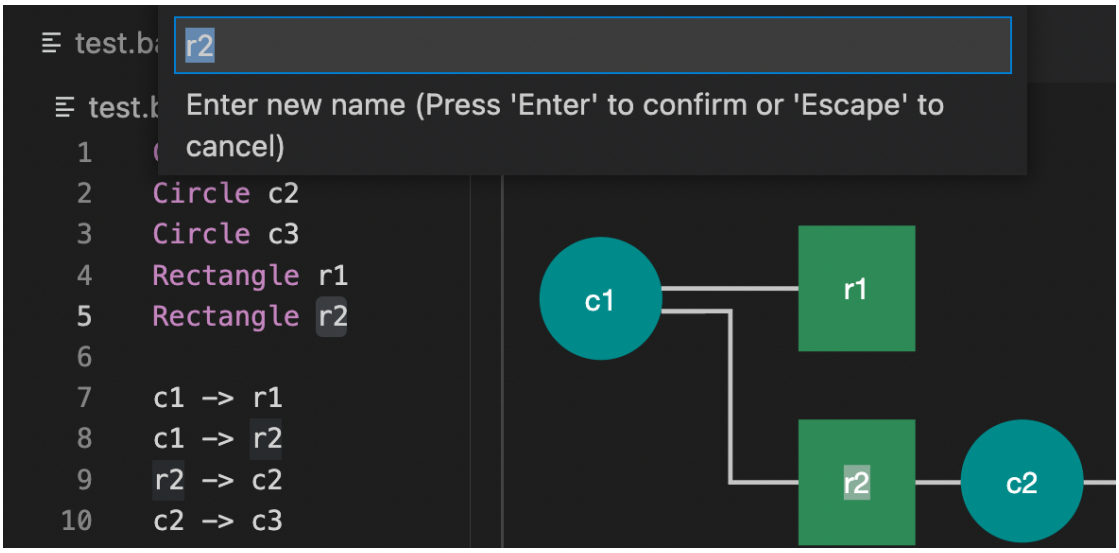


Figure 3.13: Graphical Editing Support for Renaming Labels

A graphical editing feature that can be added relatively easy is support for renaming labels. To enable this with Sprotty, the label has to include trace information of the corresponding structural feature in the language, which can then be used to apply a workspace edit to the respective textual occurrence in the document. In our case, we already added tracing to names of shapes in the diagram generator. We additionally add action handlers for workspace- and label-edit actions in the extension, and enable the feature for the label model element in the DI config of the webview. We further exclude the UI module for label editing when loading the default modules of Sprotty since we use the UI of VS Code instead. Figure 3.13 shows the realized renaming support for labels in our example. When entering a new value, the text occurrence of the name in the defining element as well as all the references are updated.

3.4.3 Create Elements

Creation of elements in the graphical editor can be enabled by mapping the behavior to LSP messages in form of *Code Actions*. A code action allows editing the textual document by specifying textual updates and at which position they should occur in the document. Normally this would be used in the textual editor to provide users with a solution to fix errors or apply a refactoring. However, we can also execute code actions in the diagram and to our advantage, `sprotty-vscode` provides a popup palette that requests code actions from the server and renders them when hovering over the diagram. From the popup palette, a code action can then be executed which notifies the server to apply the edits in the textual model and generate a new diagram that contains the changes.

In our example, we reuse the provided popup palette and start by specifying code actions that can create the individual model elements. We implement this in a new class that implements `ICodeActionService2` on the server, for creating both, circles and rectangles, since the behavior is almost identical. We also bind the new code action service in the IDE module and configure new model elements in the webview to be able to display the palette. Figure 3.14 shows the realized code action popup palette with a new rectangle created in line 1. The palette becomes available when hovering over the diagram and contains the two actions for creating the elements.

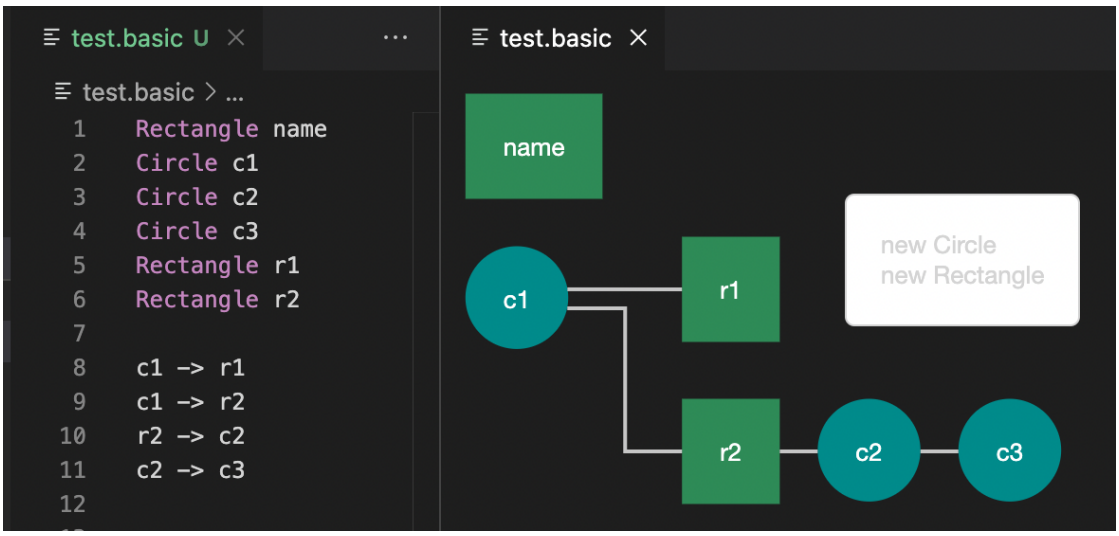


Figure 3.14: Graphical Editing Support for Creating Elements

CHAPTER 4

The BIGER Modeling Tool

The BIGER modeling tool is an extension to the VS Code IDE for modeling ER diagrams and includes both, a textual and a graphical editor. The extension supports hybrid modeling within both editors, i.e., modifications to the model in either view, update the underlying representation and the changes are automatically synchronized. This allows users to choose between creating ER models graphically or textually, and can be further utilized to allow cross-communication between stakeholders, e.g., business analysts with no technical experience can initially create an ER diagram for business requirements by using the graphical editor, while a database engineer refines the model in the textual editor. The tool also allows generating SQL code out of the models, which can be used to automatically map conceptual ER models into a logical (relational) database schema.

BIGER is a key contribution of this thesis and is based on the development approaches and architectural considerations, described previously (see Chapter 3). In the course of this work, we also published a paper, providing a high-level introduction to the tool [23]. After publication, more refinements have been realized to improve the tool and in this thesis we update on our research progress. The extension can be downloaded from the VS Code marketplace[1] and the source code is open-source available on GitHub[2].

In this chapter, we start by giving an overview of the tool's realized architecture and discuss specific components in more detail (Section 4.1). Next, we showcase the currently available modeling features (Section 4.2). As a final part, we evaluate how well different ER modeling concepts are supported, based on models presented in TU Wien's database course, and finish the chapter with results of the evaluation (Section 4.3). Current ongoing development is discussed in the conclusion (see Chapter 5).

[1] BIGER - VS Code Marketplace
[2] BIGER - GitHub Repository

4.1 Architecture

The architecture of the bigER modeling tool is realized based on our contributed generic development approach of Sprotty-based modeling tools for VS Code (see Chapter 3). In this section, we first describe the general architecture of the VS Code extension and then take a detailed look into the underlying modeling language.

4.1.1 VS Code Extension

Figure 4.1 shows the architecture of the bigER modeling tool VS Code extension. At its core, the tool consists of three separate main components, namely the *extension*, *webview* and *language server*. The architecture is realized as a client-server application with the extension and webview corresponding to the client-side and the language server on the server-side.

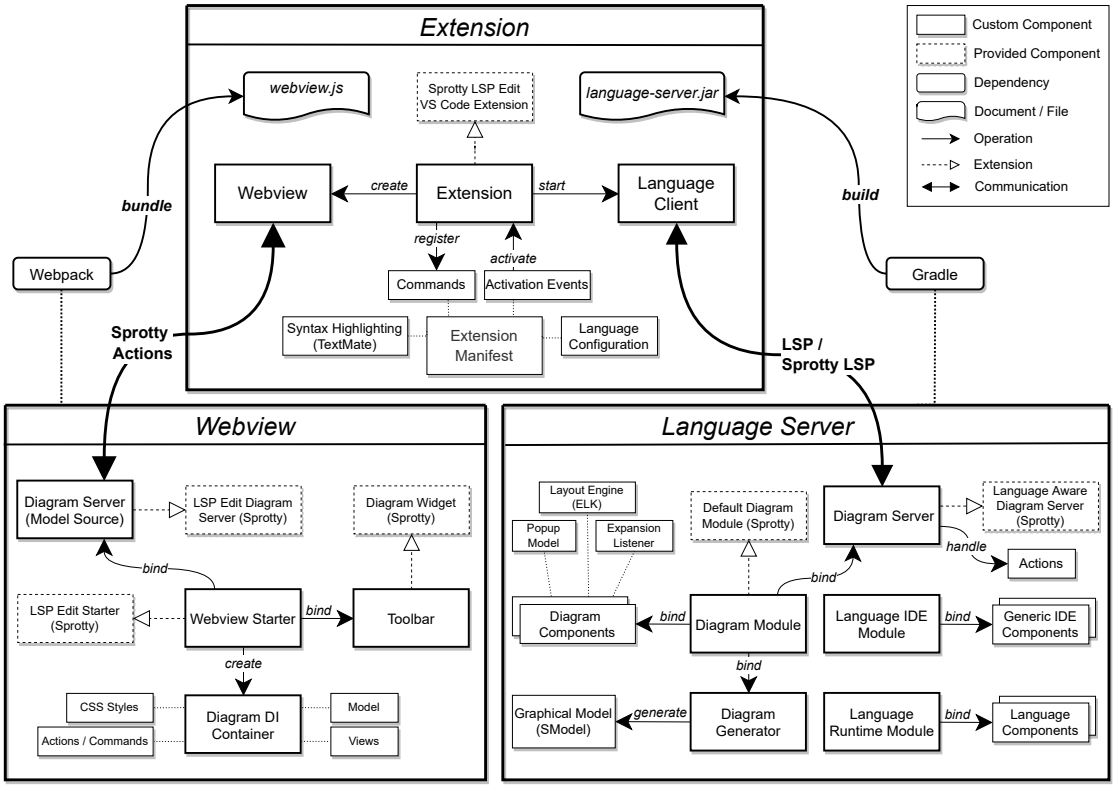


Figure 4.1: Architecture of the bigER Modeling Tool

The language server is responsible for providing language-specific functionality of the ER modeling language to the client. Textual language and generic IDE components are implemented with the Xtext language workbench that generates a complete language infrastructure and binds the respective components in the *Language IDE Module* and *Language Runtime Module*. The server is enhanced with graphical language features

through the Sprotty framework, adding an additional *Diagram Module*. The diagram module binds various default implementations provided by Sprotty together with custom components such as the *Layout Engine* (based on ELK) to automatically layout diagrams, a *Popup Model* when hovering over elements and the *Expansion Listener* to expand and collapse elements. Sprotty requires the implementation of a *Diagram Generator* that generates the *SModel* elements of the graphical model representation. Communication with the client-side is realized in the *Diagram Server* that extends the *Language Aware Diagram Server* implementation provided by Sprotty to handle custom *Actions* for additional operations in the diagram. The language server is built as a `.jar` file and made available to the client using the Gradle build system.

In the extension, the *Language Client* executes the available binary of the language server and handles communication by sending and receiving LSP and extended Sprotty LSP messages for both textual- and graphical language features. The base of all VS Code extensions is an *Extension Manifest* in a `package.json` file that declares metadata and extension-specific functionality. Declarative language features of the BIGER modeling tool include a TextMate grammar for *Syntax Highlighting* and a *Language Configuration*. The manifest defines activation points to specify that the extension should be activated when opening `.erd` files. Upon activation, an instance of a Sprotty VS Code extension is created that registers commands, starts the language client and initializes the webview.

The Sprotty diagrams of the graphical model are rendered in a webview panel of VS Code, as it can display arbitrary web content within an iframe HTML element. Sprotty provides a starter class for webviews, used in a *Sprotty Webview Starter* to create the diagram with additional VS Code bindings. Configuration of the diagram is done through DI in the *Diagram DI Container* including custom actions, commands, the model elements with corresponding views and CSS styles. The webview starter additionally binds a *Toolbar* to execute operations on the diagram and a *Diagram Server* for communicating with the extension. The diagram server in the webview acts as a model source and is responsible for handling operations in the diagram that are sent and received from and to the extension through *Sprotty Actions*.

4.1.2 Modeling Language

A high-level overview of the modeling language and its components in the BIGER modeling tool are provided in Figure 4.2. ER models conform to a metamodel and are defined in XMI files. The ER models are visualized in form of a textual- and graphical notation, which both conform to a respective CS definition that symbolizes the AS. The grammar serves as the base for the language and specifies the TCS for the underlying textual ER models. Xtext derives a corresponding metamodel from the grammar and generates the infrastructure for the language, such as a parser, serializer and various other defaults. The derived metamodel, defined in the Ecore meta-metamodeling language, specifies the AS of the language and is used in representing the AST of ER models in XMI files. The AST is then used by the diagram generator for specifying the GCS of graphical models by mapping the model elements to diagram representations (Abstract Syntax-to-

Concrete Syntax (AS2CS) mapping). The language additionally features a code generator implemented in Xtend to transform the conceptual ER models and generate a relational schema in the form of SQL `CREATE TABLE` statements.

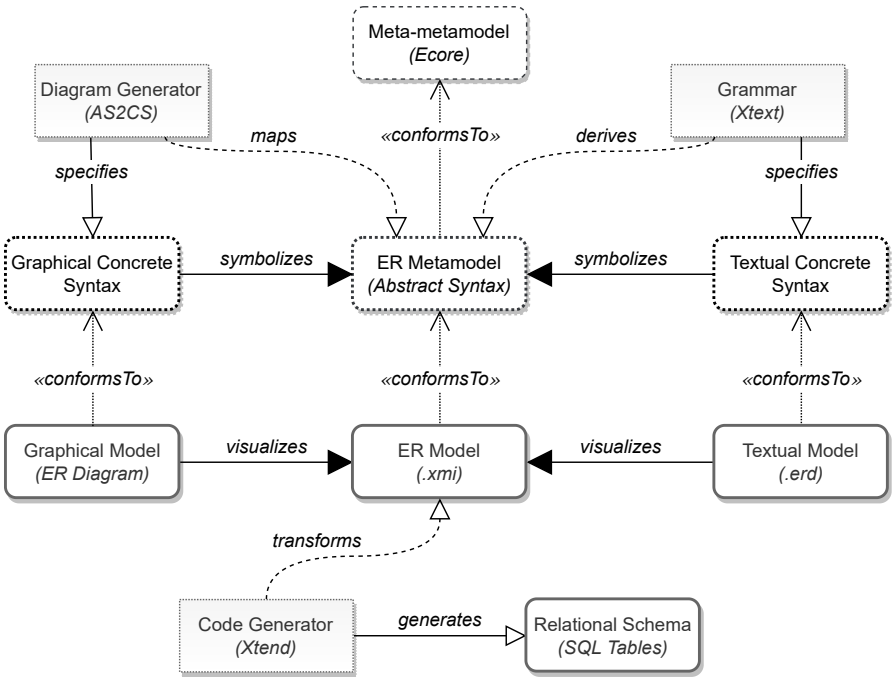


Figure 4.2: Overview of the Modeling Language in the bigER Modeling Tool

The grammar of the bigER modeling tool is listed in Listing 4.1 and shows the available language concepts and their structure. Each grammar rule including its relevant properties is mapped to a respective element in Ecore that constructs the corresponding metamodel. The derived metamodel for the grammar is shown in Figure 4.3, mostly consisting of `EClass` elements with `EAttributes` and connected with each other through `EReferences`. From the metamodel, EMF also generates a Java API for the language that is used, e.g., for model transformation in the diagram generator or the code generator.

```
grammar org.big.erd.EntityRelationship with org.eclipse.xtext.common.Terminals

generate entityRelationship "http://www.big.org/erd/EntityRelationship"

Model:
    ('erdiagram' name=ID)?
    generateOption=GenerateOption?
    (entities+=Entity | relationships+=Relationship)*;

GenerateOption:
    'generate' '=' generateOptionType=GenerateOptionType;

Entity:
    (weak?='weak')? 'entity' name=ID ('extends' extends=[Entity])? '{'
```

```
        (attributes += Attribute)*
    '}';

Relationship:
    (weak?='weak')? 'relationship' name=ID '{'
        (source=RelationEnd (('->' target=RelationEnd)
        ('->' ternary=RelationEnd)?)?)?
        (attributes += Attribute)*
    '}';

RelationEnd:
    target=[Entity] ('['
        (cardinality=CardinalityType | customMultiplicity=STRING)
    ']')?;

Attribute:
    name=ID (':' datatype=DataType)? (type=AttributeType)?;

DataType:
    type=ID ('(' size=INT ')')?;

enum AttributeType:
    NONE = 'none' | KEY = 'key' | PARTIAL_KEY = 'partial-key' |
    OPTIONAL = 'optional' | DERIVED = 'derived' | MULTIVALUED = 'multivalued';

enum CardinalityType:
    ONE = '1' | MANY = 'N';

enum GenerateOptionType:
    OFF = 'off' | SQL = 'sql';
```

Listing 4.1: Xtext Grammar of the ER Language

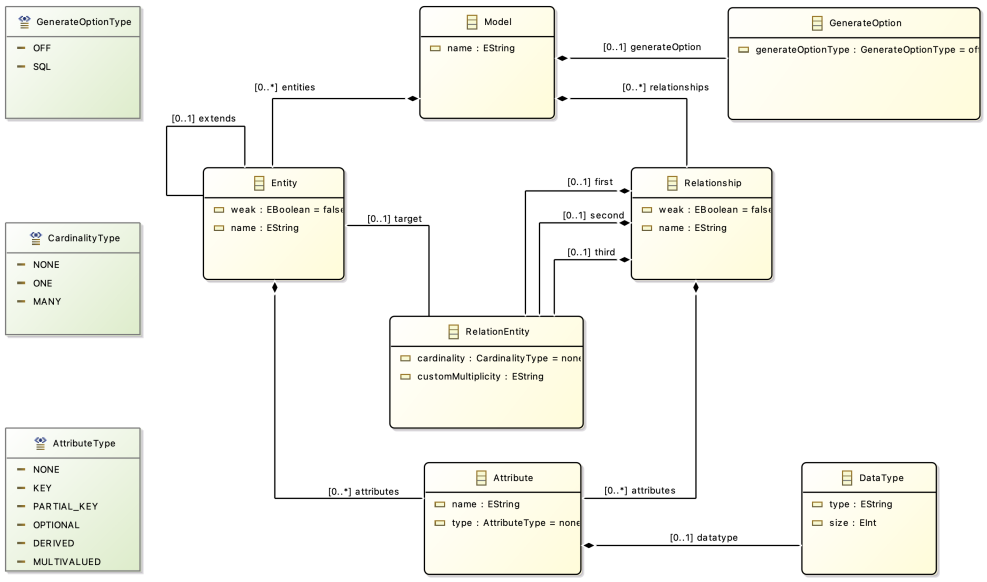


Figure 4.3: Metamodel of the ER Modeling Language as derived by Xtext

4.2 Feature Showcase

Once the bigER modeling tool is downloaded from the marketplace, the extension activates itself within VS Code when opening files ending in `.erd`. An `.erd`-file contains the complete specification of an ER model, however, such a textual view is quite cluttered and might discourage less technically experienced users from using the tool. Thus, besides the textual editor, the tool includes additional components to visualize, modify and transform models, while also adding improvements to usability. The core components of the bigER modeling tool consist of the following components:

Textual Editor — For the specification of an ER model in a textual DSL including various rich text editing features, e.g., syntax highlighting, auto-complete, model validation.

Diagram View — Renders an ER diagram that is synchronized with a corresponding textual model. The diagram view can be partly used as a graphical model editor, e.g., for adding, renaming or deleting model elements.

Code Generator — Generates SQL code by transforming a conceptual ER model to a logical schema for a relational database. Code is automatically generated on textual changes and can be toggled in the diagram or specified in the header of a textual model.

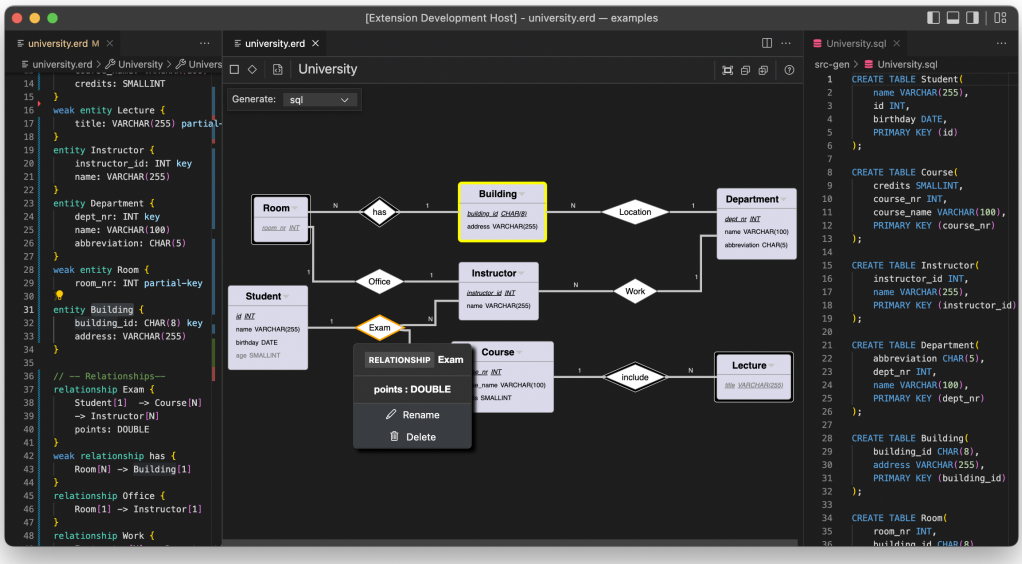


Figure 4.4: Main Components of the bigER Modeling Tool in VS Code

ER Concept	Textual Concrete Syntax
Entity	`entity`
Weak Entity	`weak entity`
Relationship	`relationship`
Weak Relationship	`weak relationship`
Inheritance (EER)	`entity A extends B`
Binary Relationship	`A -> B`
Recursive Relationship	`A -> A`
Ternary Relationship	`A -> B -> C`
Cardinality	`A[1] -> A[N]`
Multiplicity / Role	`A["0..1"] -> A["1..N"]`
Attribute and Data Type	`attribute: datatype`
Primary key	`key`
Partial key	`partial-key`
Optional / Nullable	`optional`
Derived	`derived`
Multi-valued	`multi-valued`

Table 4.1: Textual Concrete Syntax of BIGER

Figure 4.4 shows the components in VS Code with the textual editor on the left, the diagram view in the center, and SQL code generated by the code generator on the right. The figure features a basic example that is a simplified ER model of a university database. In the example, students take exams of courses that are graded by instructors with multiple lectures included in courses. Instructors work in a department and have their office in a room of a department's building. In the remaining of this section, we further look into each of the above-mentioned components and showcase their modeling features based on the university example.

4.2.1 Textual Editor

The complete textual ER model of the university example in the figure is provided in the appendix (see Section A.1.1). In BIGER, textual models are specified in a DSML with the syntax of the language constructs listed in Table 4.1 and documentation provided in the GitHub Wiki[3]. As mentioned, the textual editor supports different language features through the LSP. Besides syntax highlighting, bracket matching, quick fixes (lightbulb) and selection, which can be seen in Figure 4.4, the tool also supports validation code completion, hover information, renaming and finding references. A basic example of code completion (i.e. auto-complete feature) is displayed in Figure 4.5. Code completion provides suggestions of keywords and language constructs at the current position, and as

[3] BIGER GitHub Wiki - Language Documentation

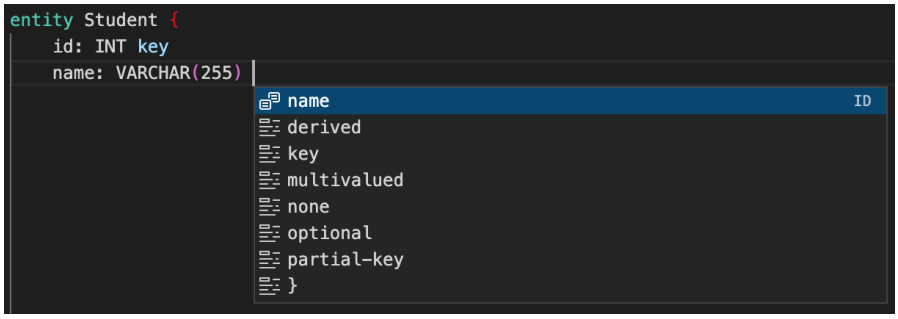


Figure 4.5: Code Completion in the Textual Editor of bigER

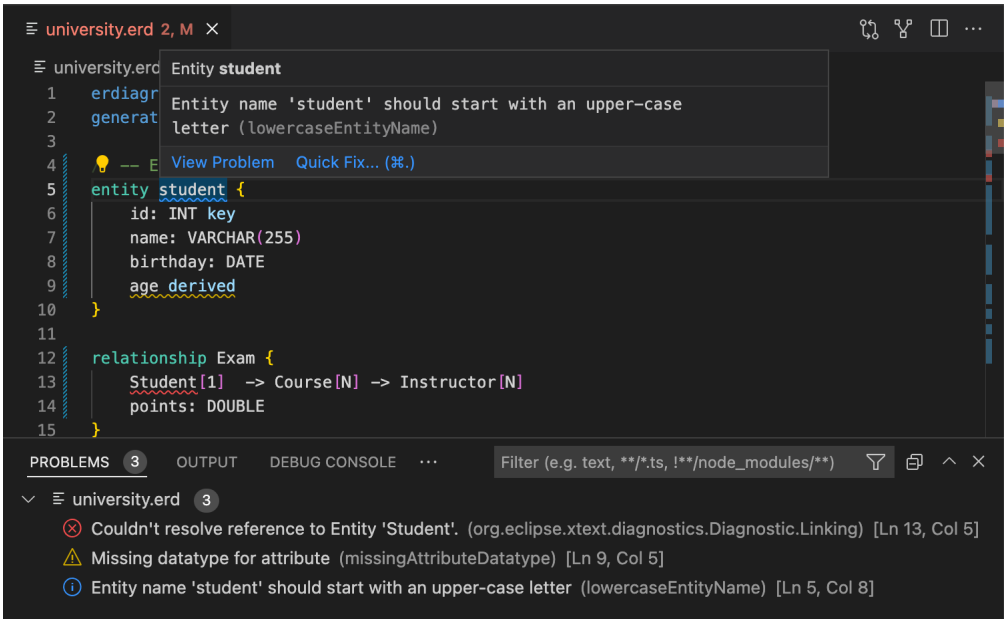


Figure 4.6: Validation in the Textual Editor of bigER

seen in the figure this includes adding a new attribute (name), keywords for types, or an enclosing curly bracket to finish the specification of an entity.

Validation in the textual editor is shown in Figure 4.6, including an error, warning and information message. Validation messages are underlined at the affected position in the editor and become visible on hover or can be seen in the problems view. As shown in the figure for the lower-case entity `student`, certain validation messages contain suggested quick fixes for fixing the corresponding mistake. In the case of the figure, the first letter of the entity's name would be fixed to be upper-case instead. When the code generator option is enabled, the validation becomes stricter, as not all constructs can be properly transformed to SQL code. Validation is also present in the diagram view, as shown in Figure 4.7, and described in the next subsection.

4.2.2 Diagram View

From the textual ER model, specified in the textual editor, the diagram view shows a corresponding graphical representation of the model. Figure 4.7 shows various features that are available in the diagram view.

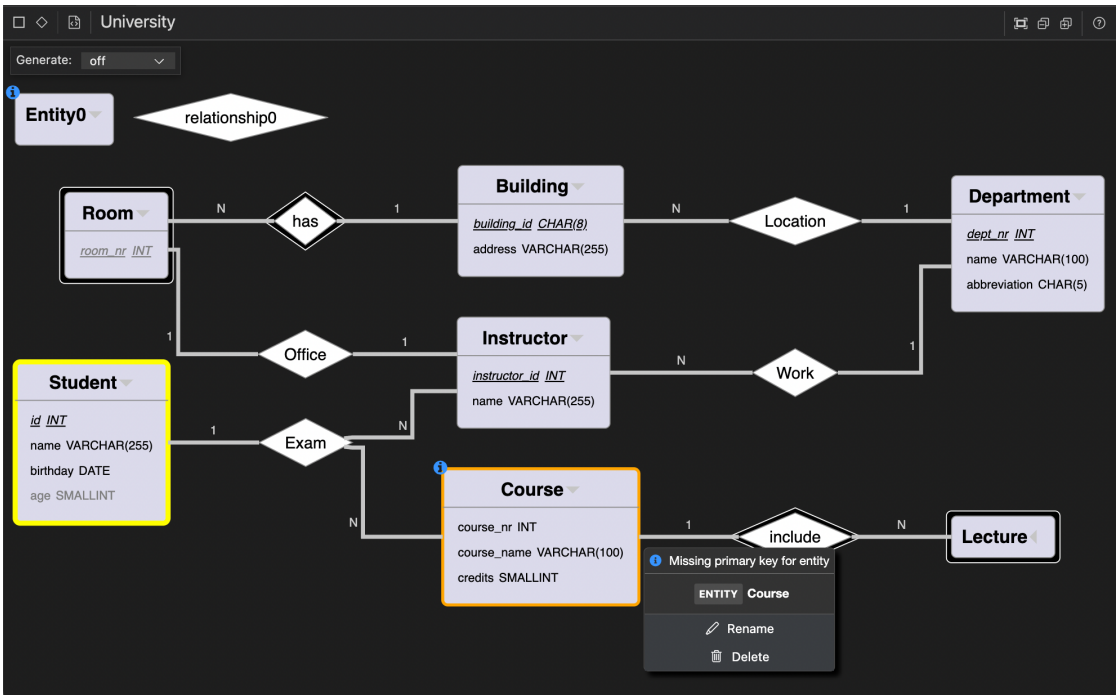


Figure 4.7: Diagram View of BIGER

The toolbar, located at the top, contains the name of the model in the center and various buttons for interactions with the diagram. On the left side of the toolbar, new entities or relationships can be created. Newly created model elements are shown in the figure, namely, `Entity0` and `relationship0`, and the name is automatically incremented, depending on elements that are already available in the model, i.e., `Entity1` is created when `Entity0` is already present. Also on the left side is a button to toggle the code generator panel, as shown above the newly created elements with the option `off` selected to disable the generator. On the right side the diagram can be centered, all elements collapsed or expanded and there is a button that links to the wiki for help.

Entities in the diagram can be individually expanded or collapsed, e.g., the entity `Lecture` is collapsed. Model elements can be selected, which also synchronizes the selection with the textual editor (e.g., `Student`), or hovered, which opens a popup for the element (e.g., `Course`). In the popup, the type and name of the element is displayed, for relationships descriptive attributes are also displayed, and there are actions to rename or delete the hovered element. If a validation message is present, the respective element is highlighted in the diagram and the corresponding message is available in the popup.

Besides renaming through the popup, entities, relationships and attribute names can also be renamed when double-clicking on the element.

4.2.3 Code Generator

At last, we look into the code generator component of the tool. Currently, only generic SQL code can be generated, however, with plans to support code generation for concrete databases in the future, e.g., MongoDB or SQL Server. The generated SQL code from the textual ER model of the university example is also provided in the appendix (see Section A.1.2). By default, the code generator is disabled and can be enabled textually in the editor, by adding `generate=sql` to the header, or graphically in the diagram through the button in the toolbar. The transformation from ER models to a relational schema follows a generic mapping described, e.g., in [54, 53]. Supported ER concepts for transformation are listed in Table 4.2 (see Section 4.3.2).

4.3 Evaluation

One of the primary use-cases of the bigER modeling tool is its usage for education. The goal of this section is to evaluate the tool based on the modeling of different ER concepts and identify any drawbacks. Based on the evaluation, we can then improve certain weak spots if needed and eventually be able to recommend the tool to database courses that cover ER modeling. First, we evaluate different ER concepts and how well they can be modeled. In a further step, we discuss the recreated models and conclude the evaluation with a summary of the results.

4.3.1 ER Concepts

For the initial evaluation, we compiled a list of concepts to evaluate, based on the ER modeling lecture from the database systems course taught at TU Wien. The course material is primarily based on the book [68]. Both, the book and the slides from the course contain ER models in German language, thus, for consistency purposes, we recreate the models in German as well. This approach allows us to better compare the original models with the recreated ones, and is irrelevant in respect to the quality of the evaluation. The concepts we evaluate include:

1. Entities and Relationships
2. Attributes and Keys
3. Roles and Recursive Relationship
4. Cardinality
5. Ternary Relationship

6. Multiplicity (min-max Notation)
7. Weak Entities

We now separately evaluate the concepts from the list above. For each concept we show its representation as originally introduced in the course/book as well as the recreated models in the graphical editor of the BIGER modeling tool. We then compare the representations and follow up with a discussion to identify any flaws. The corresponding textual models are listed in the appendix (see Section A.2).

Basic Entities and Relationships

The first concepts we evaluate are basic entities and relationships. The original representation (Figure 4.8a) models two basic entities `StudentIn` and `Vorlesung`, connected by the relationship `hoeren`. The recreated model (Figure 4.8b) is equivalent to the original model. Note that we included a key attribute in both entities of the textual model to avoid the missing key info message and collapsed the compartments.

(a) Original Model (b) Recreated Model

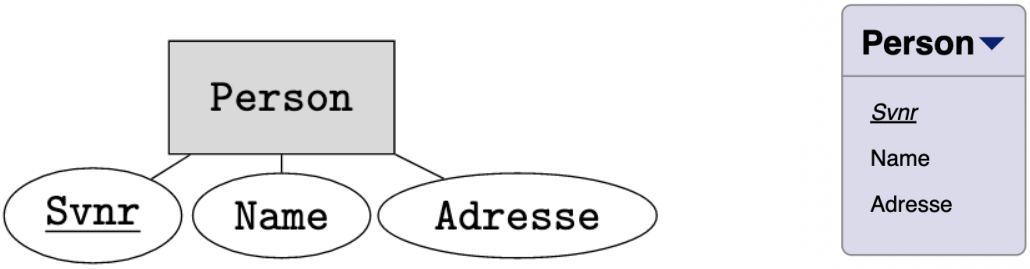
Figure 4.8: Evaluation - Basic Entities and Relationships

Attributes and Keys

The original representation (Figure 4.9a) contains the entity `Person` with the key attribute `Svnr` together with the simple attributes `Name` and `Address`. The model can be recreated equivalently (Figure 4.9b), containing all the attributes and with the key being underlined as well. However, the recreated model has a slightly different representation, as all attributes are contained within the entity. We decided to use this notation instead, based on the fact that large models can quickly become cluttered when using the original notation of ellipses.

Roles and Recursive Relationship

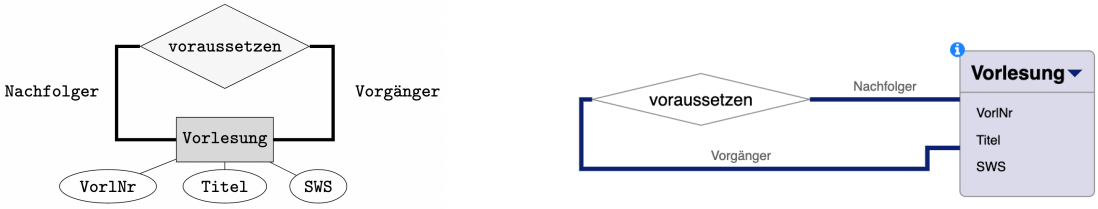
Roles are commonly present in recursive relationships, thus, we evaluate the concepts together. The model contains the entity `Vorlesung` with three attributes and the recursive relationship `voraussetzen`. Roles are denoted as labels on the edges of the relationship, namely `Vorgänger` and `Nachfolger`. The model can be recreated, however, roles are only partially supported in the tool by using a custom notation. Roles on relationships do not capture any semantics and, as we see later, they replace cardinality or multiplicity. Thus, there is no form of validation and the code generator does not support the concept either. Furthermore, the layout of recursive relationships is not



(a) Original Model (b) Recreated Model

Figure 4.9: Evaluation - Attributes and Keys

very attractive and is one of the key aspects to be improved. The model is recreated to align with the original one which does not specify a key attribute and for this reason the information icon appears on the entity.



(a) Original Model (b) Recreated Model

Figure 4.10: Evaluation - Roles and Recursive Relationship

Cardinality

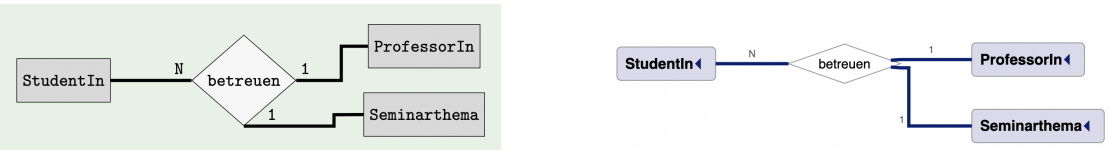
The cardinality concept can be specified in the form of 1:1 (One-to-One), 1:N (One-to-Many), N:1 (Many-to-One) or N:M (Many-to-Many) relationships with values on the edges corresponding to the respective entities on the same side of the relationship. The original model specified a 1:N relationship for `R` and the entities `E1` and `E2`. Cardinality is fully supported in the bigER Modeling Tool including validation and code generation. However, we note that N:M relationships are expressed as N:N in the tool, which semantically is not fully valid since N refers to a value and N:N would mean that both values are equal.

(a) Original Model (b) Recreated Model

Figure 4.11: Evaluation - Cardinality

Ternary Relationship

Opposed to binary relationships connected by two different entities, a ternary relationship connects three different entities. The ternary relationship in the original model consists of the entities `StudentIn`, `Professorin` and `Seminarthema` connected by the relationship `betreuen` with cardinality N:1:1. None of the entities contain any attributes. In BIGER the ternary relationship can be recreated with only the layout being slightly different. However, relationships with an arity greater than three are not supported, but planned as a future feature.

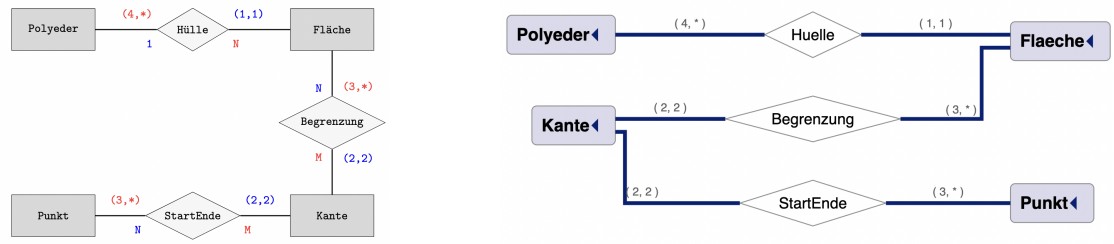


(a) Original Model (b) Recreated Model

Figure 4.12: Evaluation - Ternary Relationship

Multiplicity

Relationships can be further refined with the multiplicity concept, denoting a minimum and maximum value. The original model includes both, cardinality and multiplicity on the respective edges. BIGER currently only allows either the cardinality or the custom notation to be specified for relationships, with no option to declare both. Similar to roles, multiplicity can be expressed in form of the custom notation, however, also with only partial support in the tool. Since we already evaluated the cardinality concept, we recreated the model by choosing to denote the multiplicity now. The layout of the original model can not be recreated, as the layered layout algorithm in the tool does not connect entities at the top or bottom-side.



(a) Original Model (b) Recreated Model

Figure 4.13: Evaluation - Multiplicity

Weak Entity

The last concept we evaluate are weak entities. In the original model (see Figure 4.14a) the strong entity `Gebäude` with the primary key `GebNr` is connected to the weak entity

Raum with the partial key RaumNr. The recreated model (see Figure 4.14b) contains the same information and both, the weak entity and the relationship are colored with a stronger border to indicate the dependency to the strong entity. Also, the partial key is highlighted differently in the recreated model, however, colored in gray and with no dashed underline as opposed to the original model. We also note that the participation concept is not supported in bigER, thus, the partial participation edge is drawn the same as any other edge of a relationship in the tool.

(a) Original Model (b) Recreated Model

Figure 4.14: Evaluation - Weak Entity

4.3.2 Result and Discussion

As part of the initial evaluation of the bigER modeling tool, we evaluated whether the tool offers the capability for modeling different ER concepts as taught in the database course at TU Wien. Based on the result we have shown that all the concepts from the course can be textually- and graphically recreated, however, the evaluation has also shown to include certain limitations in the modeling process. Limitations include the following points:

- **Partial support for roles and multiplicity**. For the two concepts, we mention that there is only partial support in the tool, meaning that they can only be expressed by providing a custom concept that does not correspond to ER and does not capture the correct semantics. For this reason, they can also not be properly validated and are not supported in the code generator.
- **Roles and Multiplicity as a replacement for Cardinality**. In the evaluation we were not able to denote both, multiplicity and cardinality on the same edge. The same applies for the roles and, thus, modelers have to choose between one of the concepts that are represented and disregard the others.
- **Layout limitations**. The layout of the graphical models can be customized, however, when updating or reopening models, the layout is reset. Furthermore, when considering the automatic layout, the representation is not always optimal, with edges only flowing left and right, leading to not always optimal representations, e.g., when modeling recursive relationships.

A primary goal of the bigER modeling tool is to provide a versatile and high-quality modeling experience for ER modeling, thus, we plan on providing improvements to

the above-mentioned limitations in future releases. Furthermore, even though our tool supports validation of ER models and mapping to a relational schema, the two aspects were not considered in the evaluation. The reason for this is that the underlying language is still relatively experimental and expected to change, especially after this initial evaluation. We definitely consider validation and transformation of models crucial aspects in regard to the overall usability of the tool, thus, they are both planned to be evaluated in future work. Furthermore, even though we explicitly mentioned that our initial goal is to only evaluate concepts from the classical ER model, we also consider the EER model to be an important data model and plan on supporting and evaluating its features in the near future.

To conclude this section and summarize the results, we regard the initial evaluation of the BIGER modeling tool a success, highlighting the potential for its use in education. Nevertheless, when reading through additional database literature, often additional concepts come up that were not considered in the evaluation. While the course at TU Wien discusses some of them, there is no adequate model available in the course material that can be recreated and used for evaluation. Since the goal is to properly showcase the current capabilities of the BIGER modeling tool, we provide an overview of the supported concepts. The result of the ER concepts we discovered in literature and the corresponding support in tool is summarized in Table 4.2.

ER Concept	**Textual**	**Graphical**	**Code Generator**
Regular Entity	✓	✓	✓
Weak Entity	✓	✓	✓
Binary Relationship	✓	✓	✓
Recursive Relationship	✓	✓	
Ternary Relationship	✓	✓	✓
Higher-Order Relationship			
Multiplicity	≈	≈	
Cardinality	✓	✓	✓
Participation			
Roles	≈	≈	
Entity Attributes	✓	✓	✓
Relationship Attributes	✓	✓	✓
Datatype	✓	✓	✓
Key	✓	✓	✓
Composite			
Multi-valued	✓		
Derived	✓	✓	✓
Optional	✓	✓	✓

Table 4.2: Supported ER Concepts in the BIGER Modeling Tool

CHAPTER 5

Conclusion

In conclusion of this thesis, we summarize the results of our research and contributions. In addition, we provide an outlook regarding the BIGER modeling tool, including future work and potential research directions.

5.1 Thesis Summary

In the introduction (see Chapter 1) we introduced the context, motivational aspects, and problem statement of this thesis, and based on this information we further established the research objectives (RO1, RO2, RO3). After a description of the relevant background (see Chapter 2), we presented the contributions of this work to provide answers for the research objectives. Summarizing the research outcomes of this thesis, we specifically contribute the following:

- An approach for the development of Sprotty-based Modeling Tools for VS Code **(RO2)**
- The BIGER modeling tool for ER modeling as an extension to the VS Code ecosystem **(RO3)**

A combination of both contributions deals with **RO1**, as various capabilities of Sprotty are showcased throughout this work when leveraging the framework for the development of modeling tools for VS Code.

5.2 Outlook

With completion of this thesis, we release version 0.1.0 of the BIGER modeling tool, however, we continue to actively work on new releases. Current ongoing work includes,

improvements to quality management of the project, together with testing of the modeling functionality provided in the tool. Furthermore, another bachelor thesis is currently in progress that focuses on different notations of ER modeling, and incorporates the notations to bigER to support different representations of models.

In addition to ongoing work, we plan on adding support for the following features in the modeling tool:

- Additional hybrid modeling functionality with the aim to support most of the textual features, graphically as well, e.g., routing edges in the diagram to create relationships graphically.
- Full support of all ER concepts and additional EER concepts, based on the results of the evaluation in this thesis.
- Improvements to the current layout and support for additional layout algorithms, since the evaluation has shown the drawbacks of current layout options.
- Import of existing ER models, e.g., from other tools, and additional export options, e.g., SQL code generation for concrete database technologies or export to other files, such as images.

APPENDIX A

Textual Models

A.1 University Example

A.1.1 Textual ER Model

```
erdiagram University
generate=sql

entity student {
    id: INT key
    name: VARCHAR(255)
    birthday: DATE
    age: SMALLINT derived
}
entity Course {
    course_nr: INT key
    course_name: VARCHAR(100)
    credits: SMALLINT
}
weak entity Lecture {
    title: VARCHAR(255) partial-key
}
entity Instructor {
    instructor_id: INT key
    name: VARCHAR(255)
}
entity Department {
    dept_nr: INT key
    name: VARCHAR(100)
    abbreviation: CHAR(5)
}
weak entity Room {
    room_nr: INT partial-key
}
entity Building {
    building_id: CHAR(8) key
    address: VARCHAR(255)
}
```

```
relationship Exam {
    student[1]  -> Course[N]
    -> Instructor[N]
    points: DOUBLE
}
weak relationship has {
    Room[N] -> Building[1]
}
relationship Office {
    Room[1] -> Instructor[1]
}
relationship Work {
    Instructor[N] -> Department[1]
}
weak relationship include {
    Course[1] -> Lecture[N]
}
relationship Location {
    Building[N] -> Department[1]
}
```

Listing A.1: University Example - Textual Model

A.1.2 Generated SQL Code

```
CREATE TABLE student(
        name VARCHAR(255),
        birthday DATE,
        id INT,
        PRIMARY KEY (id)
);

CREATE TABLE Course(
        course_name VARCHAR(100),
        course_nr INT,
        credits SMALLINT,
        PRIMARY KEY (course_nr)
);

CREATE TABLE Instructor(
        instructor_id INT,
        name VARCHAR(255),
        PRIMARY KEY (instructor_id)
);

CREATE TABLE Department(
        dept_nr INT,
        abbreviation CHAR(5),
        name VARCHAR(100),
        PRIMARY KEY (dept_nr)
);

CREATE TABLE Building(
        address VARCHAR(255),
        building_id CHAR(8),
        PRIMARY KEY (building_id)
);

CREATE TABLE Room(
        room_nr INT,
```

```
        building_id CHAR(8),
        PRIMARY KEY (room_nr, building_id),
        FOREIGN KEY (building_id) references Building ON DELETE CASCASE
);

CREATE TABLE Lecture(
        title VARCHAR(255),
        course_nr INT,
        PRIMARY KEY (title, course_nr),
        FOREIGN KEY (course_nr) references Course ON DELETE CASCASE
);

CREATE TABLE Exam(
        id INT references student(id),
        course_nr INT references Course(course_nr),
,       instructor_id INT references Instructor(instructor_id),
        points DOUBLE,
        PRIMARY KEY (id, course_nr, instructor_id)
);

CREATE TABLE Office(
        room_nr INT references Room(room_nr),
        instructor_id INT references Instructor(instructor_id),
        PRIMARY KEY (room_nr, instructor_id)
);

CREATE TABLE Work(
        instructor_id INT references Instructor(instructor_id),
        dept_nr INT references Department(dept_nr),
        PRIMARY KEY (instructor_id, dept_nr)
);

CREATE TABLE Location(
        building_id CHAR(8) references Building(building_id),
        dept_nr INT references Department(dept_nr),
        PRIMARY KEY (building_id, dept_nr)
);
```

Listing A.2: University Example - Generated SQL Code

A.2 Evaluated ER Models

A.2.1 Basic Entities and Attributes

```
erdiagram Example1

entity StudentIn {
    id key
}
entity Vorlesung {
    id key
}
relationship hoeren {
    StudentIn -> Vorlesung
}
```

Listing A.3: Basic Entities and Attributes

A.2.2 Attributes and Keys

```
erdiagram Example2

entity Person {
    Svnr key
    Name
    Adresse
}
```

Listing A.4: Attributes and Keys

A.2.3 Roles and Recursive Relationship

```
erdiagram Example3

entity Vorlesung {
    VorlNr
    Titel
    SWS
}
relationship voraussetzen {
    Vorlesung["Vorgaenger"] -> Vorlesung["Nachfolger"]
}
```

Listing A.5: Roles and Recursive Relationship

A.2.4 Cardinality

```
erdiagram Example4

entity E1 {
    id key
}
entity E2 {
    id key
}
relationship R {
    E1[1] -> E2[N]
}
```

Listing A.6: Cardinality

A.2.5 Ternary Relationship

```
erdiagram Example5

entity StudentIn {
    id key
}
entity ProfessorIn {
    id key
}
entity Seminarthema {
    id key
}
```

```
relationship betreuen {
    StudentIn[N] -> ProfessorIn[1] -> Seminarthema[1]
}
```

Listing A.7: Ternary Relationship

A.2.6 Multiplicity

```
erdiagram Example6

entity Polyeder {
    id key
}
entity Flaeche {
    id key
}
entity Kante {
    id key
}
entity Punkt {
    id key
}
relationship Huelle {
    Polyeder["(4,*)"] -> Flaeche["(1,1)"]
}
relationship Begrenzung {
    Kante["(2,2)"] -> Flaeche["(3,*)"]
}
relationship StartEnde {
    Kante["(2,2)"] -> Punkt["(3,*)"]
}
```

Listing A.8: Multiplicity

A.2.7 Weak Entity

```
erdiagram Example7

entity Gebaeude {
    GebNr key
    Hoehe
}
weak entity Raum {
    RaumNr partial-key
    Groesse
}
weak relationship liegt_in {
    Gebaeude[1] -> Raum[N]
}
```

Listing A.9: Weak Entity

List of Figures

2.1 High-level Architecture of Model-driven Engineering [2] 9
2.2 Textual- and Graphical Concrete Syntax Definition of Metamodels, adapted from [2] 11
2.3 Overview of the Ecore Meta-metamodeling Language 14
2.4 VS Code Extension with Commands in Action 19
2.5 Entity with different types of Attributes 21
2.6 Degree of Relationships 22
2.7 Cardinality Constraints represented as Relations between Sets 23
2.8 Difference between Cardinality and Participation Constraints 23
2.9 Weak Entity and Identifying Relationship 24

3.1 SModel Class Diagram 29
3.2 Client-side Architecture of Sprotty [67] 30
3.3 Result of the implemented Example Sprotty Diagram 34
3.4 Sprotty Example - Randomly generated Graph with Automatic Layout . 35
3.5 Sprotty Example - Class Diagram 36
3.6 Advanced Xtext Configuration Page 38
3.7 Folder Structure of the newly generated VS Code Language Extension . . 40
3.8 Basic Language Extension running in VS Code 42
3.9 Validation Language Feature of the Example 43
3.10 Architecture of Sprotty integrated with VS Code and Xtext 44
3.11 Sprotty-enhanced Language Server 45
3.12 Sprotty VS Code LSP Extension Example 47
3.13 Graphical Editing Support for Renaming Labels 49
3.14 Graphical Editing Support for Creating Elements 50

4.1 Architecture of the bigER Modeling Tool 52
4.2 Overview of the Modeling Language in the bigER Modeling Tool 54
4.3 Metamodel of the ER Modeling Language as derived by Xtext 55
4.4 Main Components of the bigER Modeling Tool in VS Code 56
4.5 Code Completion in the Textual Editor of bigER 58
4.6 Validation in the Textual Editor of bigER 58
4.7 Diagram View of bigER 59
4.8 Evaluation - Basic Entities and Relationships 61

4.9 Evaluation - Attributes and Keys 62
4.10 Evaluation - Roles and Recursive Relationship 62
4.11 Evaluation - Cardinality 62
4.12 Evaluation - Ternary Relationship 63
4.13 Evaluation - Multiplicity 63
4.14 Evaluation - Weak Entity 64

List of Tables

2.1 Overview of relevant Sprotty Packages 17
2.2 Comparison of Tools and VS Code Extensions for ER Modeling 25

3.1 Default Feature Modules provided by Sprotty 31

4.1 Textual Concrete Syntax of BIGER 57
4.2 Supported ER Concepts in the BIGER Modeling Tool 65

Acronyms

API Application Programming Interface. 3, 12, 16–18, 39, 42, 43, 46, 54

AS Abstract Syntax. 10–12, 53

AS2CS Abstract Syntax-to-Concrete Syntax. 53, 54

AST Abstract Syntax Tree. 16, 37, 38, 53

CS Concrete Syntax. 10, 11, 53

DI Dependency Injection. 16, 17, 30–33, 37, 45–47, 49, 53

DOM Document Object Model. 16, 30

DSL Domain-specific Language. 8, 9, 25, 26, 36, 37, 56

DSML Domain-specific Modeling Language. 9, 57

EBNF Extended Backus Naur Form. 10, 11, 16

EER Extended Entity Relationship. 19, 25, 26, 57, 65, 68

ELK Eclipse Layout Kernel. 17, 35, 46, 53

EMF Eclipse Modeling Framework. 5, 11–16, 37, 38, 54

ER Entity Relationship. 4, 5, 7, 19–26, 51–54, 56, 57, 59, 60, 64, 65, 67, 68

GCS Graphical Concrete Syntax. 11–13, 53

GLSP Graphical Language Server Platform. 3

GPL General Purpose Language. 8, 9

GPML General Purpose Modeling Language. 9

IDE Integrated Development Environment. 1–3, 12–15, 17, 25, 26, 37, 39, 45, 46, 50–52

LSP Language Server Protocol. 2, 3, 5, 7, 12–16, 25–27, 36, 37, 39, 42, 44–49, 53, 57

M2T Model-to-text. 16

MBE Model-based Engineering. 8

MDA Model-driven Architecture. 8

MDD Model-driven Development. 8

MDE Model-Driven Engineering. 1–3, 5, 7, 8, 10, 12, 13

MDSE Model-driven Software Engineering. 8

MOF Meta Object Facility. 11, 13

MVC Model View Controller. 16, 30

OCL Object Constraint Language. 10

OMG Object Management Group. 8, 13

SQL Structured Query Language. 4, 25, 26, 51, 54, 56–58, 60, 68

SVG Scalable Vector Graphics. 12, 16, 31–33

TCS Textual Concrete Syntax. 11, 13, 16, 53

UML Unified Modeling Language. 7, 9, 10, 13, 23

URI Uniform Resource Identifier. 15

XMI XML Metadata Interchange. 13, 53

Bibliography

[1] J. Bézivin, "Model driven engineering: An emerging technical space," in *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers* (R. Lämmel, J. Saraiva, and J. Visser, eds.), vol. 4143 of *Lecture Notes in Computer Science*, pp. 36–64, Springer, 2005.

[2] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice, Second Edition*. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, 2017.

[3] P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernandez, B. Nordmoen, and M. Fritzsche, "Where does model-driven engineering help? experiences from three industrial cases," *Software & Systems Modeling*, vol. 12, no. 3, pp. 619–639, 2013.

[4] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Science of Computer Programming*, vol. 89, pp. 144–161, 2014.

[5] R. V. D. Straeten, T. Mens, and S. V. Baelen, "Challenges in model-driven software engineering," in *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers* (M. R. V. Chaudron, ed.), vol. 5421 of *Lecture Notes in Computer Science*, pp. 35–47, Springer, 2008.

[6] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE Software*, vol. 31, no. 3, pp. 79–85, 2013.

[7] A. Bucchiarone, J. Cabot, R. F. Paige, and A. Pierantonio, "Grand challenges in model-driven engineering: an analysis of the state of the research," *Software and Systems Modeling*, vol. 19, pp. 5–13, Jan. 2020.

[8] J. Whittle, J. E. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "Industrial adoption of model-driven engineering: Are the tools really the problem?," in *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*

(A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. J. Clarke, eds.), vol. 8107 of *Lecture Notes in Computer Science*, pp. 1–17, Springer, 2013.

[9] S. Abrahão, F. Bourdeleau, B. Cheng, S. Kokaly, R. Paige, H. Stöerrle, and J. Whittle, "User experience for model-driven engineering: Challenges and future directions," in *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 229–236, IEEE, 2017.

[10] H. Bruneliere, E. Burger, J. Cabot, and M. Wimmer, "A feature-based survey of model view approaches," *Software & Systems Modeling*, vol. 18, no. 3, pp. 1931–1952, 2019.

[11] D. S. Kolovos, R. F. Paige, and F. Polack, "The grand challenge of scalability for model driven engineering," in *MoDELS (Workshops)*, vol. 5421 of *Lecture Notes in Computer Science*, pp. 48–53, Springer, 2008.

[12] J. Gray and B. Rumpe, "The evolution of model editors: browser-and cloud-based solutions," *Software & Systems Modeling*, vol. 15, no. 2, pp. 303–305, 2016.

[13] S. Popoola, J. C. Carver, and J. Gray, "Modeling as a service: A survey of existing tools," in *MoDELS (Satellite Events)*, vol. 2019 of *CEUR Workshop Proceedings*, pp. 360–367, CEUR-WS.org, 2017.

[14] H. Bünder, "Decoupling language and editor-the impact of the language server protocol on textual domain-specific languages.," in *MODELSWARD*, pp. 129–140, SciTePress, 2019.

[15] H. Bünder and H. Kuchen, "Towards multi-editor support for domain-specific languages utilizing the language server protocol," in *MODELSWARD (Revised Selected Papers)*, vol. 1161 of *Communications in Computer and Information Science*, pp. 225–245, Springer, 2019.

[16] N. Gunasinghe and N. Marcus, "Understanding the language server protocol," in *Language Server Protocol and Implementation*, pp. 11–22, Springer, 2022.

[17] R. Rodriguez-Echeverria, J. L. C. Izquierdo, M. Wimmer, and J. Cabot, "Towards a language server protocol infrastructure for graphical modeling," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pp. 370–380, ACM, 2018.

[18] R. Rodríguez-Echeverría, J. L. C. Izquierdo, M. Wimmer, and J. Cabot, "An LSP infrastructure to build EMF language servers for web-deployable model editors," in *MoDELS (Workshops)*, vol. 2245 of *CEUR Workshop Proceedings*, pp. 326–335, CEUR-WS.org, 2018.

[19] L. Walsh, J. Dingel, and K. Jahed, "Toward client-agnostic hybrid model editor tools as a service," in *Proceedings of the 23rd ACM/IEEE International Conference on*

Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 1–1, 2020.

[20] F. Ciccozzi, M. Tichy, H. Vangheluwe, and D. Weyns, "Blended modelling-what, why and how," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 425–430, IEEE, 2019.

[21] I. David, M. Latifaj, J. Pietron, W. Zhang, F. Ciccozzi, I. Malavolta, A. Raschke, J.-P. Steghöfer, and R. Hebig, "Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study," *Software & Systems Modeling To appear*, 2022.

[22] R. Saini and G. Mussbacher, "Towards conflict-free collaborative modelling using vs code extensions," in *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 35–44, IEEE, 2021.

[23] P.-L. Glaser and D. Bork, "The bigER tool-hybrid textual and graphical modeling of entity relationships in vs code," in *2021 IEEE 25th International Enterprise Distributed Object Computing Workshop (EDOCW)*, pp. 337–340, IEEE, 2021.

[24] N. Wirth, *Algorithms and data structures*. Prentice Hall, 1985.

[25] M. Fowler, *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[26] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, and R. Pretorius, "Empirical evidence about the uml: a systematic literature review," *Software: Practice and Experience*, vol. 41, no. 4, pp. 363–392, 2011.

[27] L. Khaled, "A comparison between UML tools," in *ICECS*, pp. 111–114, IEEE Computer Society, 2009.

[28] A. G. Kleppe, J. B. Warmer, J. Warmer, and W. Bast, *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[29] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[30] D. Harel and B. Rumpe, "Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff," 2000.

[31] B. Combemale, R. France, J.-M. Jézéquel, B. Rumpe, J. Steel, and D. Vojtisek, *Engineering modeling languages: Turning domain knowledge into tools*. CRC Press, 2016.

[32] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[33] D. Bork, D. Karagiannis, and B. Pittl, "A survey of modeling language specification techniques," *Information Systems*, vol. 87, 2020.

[34] D. Bork, D. Karagiannis, and B. Pittl, "Systematic analysis and evaluation of visual conceptual modeling language notations," in *2018 12th International Conference on Research Challenges in Information Science (RCIS)*, pp. 1–11, IEEE, 2018.

[35] R. F. Paige, J. S. Ostroff, and P. J. Brooke, "Principles for modeling language design," *Information and Software Technology*, vol. 42, no. 10, pp. 665–675, 2000.

[36] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel, "Design guidelines for domain specific languages," *arXiv preprint arXiv:1409.2378*, 2014.

[37] A. Cicchetti, F. Ciccozzi, and A. Pierantonio, "Multi-view approaches for software and system modelling: a systematic literature review," *Software and Systems Modeling*, vol. 18, no. 6, pp. 3207–3233, 2019.

[38] F. Fondement, "Concrete syntax definition for modeling languages," 2007.

[39] T. Goldschmidt, S. Becker, and A. Uhl, "Classification of concrete textual syntax mapping approaches," in *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings* (I. Schieferdecker and A. Hartman, eds.), vol. 5095 of *Lecture Notes in Computer Science*, pp. 169–184, Springer, 2008.

[40] F. Rani, P. Diez, E. Chavarriaga, E. Guerra, and J. de Lara, "Automated migration of eugenia graphical editors to the web," in *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings* (E. Guerra and L. Iovino, eds.), pp. 1–7, ACM, 2020.

[41] F. Bedini, R. Maschotta, and A. Zimmermann, "A generative approach for creating eclipse sirius editors for generic systems," in *IEEE International Systems Conference, SysCon 2021, Vancouver, BC, Canada, April 15 - May 15, 2021*, pp. 1–8, IEEE, 2021.

[42] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[43] "Language Server Protocol Website." https://microsoft.github.io/language-server-protocol/, accessed: 2022-06-07.

[44] J. K. Rask, F. P. Madsen, N. Battle, H. D. Macedo, and P. G. Larsen, "The specification language server protocol: A proposal for standardised LSP extensions," in *Proceedings of the 6th Workshop on Formal Integrated Development Environment, F-IDE@NFM 2021, Held online, 24-25th May 2021* (J. Proença and A. Paskevich, eds.), vol. 338 of *EPTCS*, pp. 3–18, 2021.

[45] P. Jeanjean, B. Combemale, and O. Barais, "IDE as code: Reifying language protocols as first-class citizens," in *ISEC 2021: 14th Innovations in Software Engineering Conference, Bhubaneswar, Odisha, India, February 25-27, 2021* (D. P. Mohapatra, S. Mishra, T. Clark, A. Dubey, R. Sharma, and L. Kumar, eds.), pp. 1–5, ACM, 2021.

[46] "Xtext Documentation." https://www.eclipse.org/Xtext/documentation/, accessed: 2022-06-07.

[47] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[48] J. Yue, "Transition from EBNF to xtext," in *Joint Proceedings of MODELS 2014 Poster Session and the ACM Student Research Competition (SRC) co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 28 - October 3, 2014* (S. Sauer, M. Wimmer, M. Genero, and S. Qadeer, eds.), vol. 1258 of *CEUR Workshop Proceedings*, pp. 75–80, CEUR-WS.org, 2014.

[49] L. M. Rose, N. D. Matragkas, D. S. Kolovos, and R. F. Paige, "A feature model for model-to-text transformation languages," in *Proceedings of the 4th International Workshop on Modeling in Software Engineering, MiSE 2012, Zurich, Switzerland, June 2-3, 2012* (J. M. Atlee, R. Baillargeon, R. B. France, G. Georg, A. Moreira, B. Rumpe, and S. Zschaler, eds.), pp. 57–63, IEEE Computer Society, 2012.

[50] C. Gackenheimer, "Introducing flux: An application architecture for react," in *Introduction to React*, pp. 87–106, Springer, 2015.

[51] A. Boduch, *Flux architecture*. Packt Publishing Ltd, 2016.

[52] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, p. 9–36, Mar. 1976.

[53] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems, 3rd Edition*. Addison-Wesley-Longman, 2000.

[54] R. Ramakrishnan and J. Gehrke, *Database Management Systems, 3rd Edition*. McGraw-Hill, 2003.

[55] H. Gregersen and C. S. Jensen, "Temporal entity-relationship models-a survey," *IEEE Transactions on knowledge and data engineering*, vol. 11, no. 3, pp. 464–497, 1999.

[56] D. Heckerman, C. Meek, and D. Koller, "Probabilistic entity-relationship models, prms, and plate models," *Introduction to statistical relational learning*, vol. 2007, pp. 201–238, 2007.

[57] M. Gandhi, E. L. Robertson, and D. V. Gucht, "Leveled entity relationship model," in *International Conference on Conceptual Modeling*, pp. 420–436, Springer, 1994.

[58] T. J. Teorey, D. Yang, and J. P. Fry, "A logical design methodology for relational databases using the extended entity-relationship model," *ACM Computing Surveys (CSUR)*, vol. 18, no. 2, pp. 197–222, 1986.

[59] B. Thalheim, "Extended entity-relationship model," in *Encyclopedia of Database Systems* (L. Liu and M. T. Özsu, eds.), pp. 1083–1091, Springer US, 2009.

[60] I.-Y. Song, M. Evans, and E. K. Park, "A comparative analysis of entity-relationship diagrams," *Journal of Computer and Software Engineering*, vol. 3, no. 4, pp. 427–459, 1995.

[61] S. Bagui and R. Earp, *Database design using entity-relationship diagrams*. Auerbach Publications, 2003.

[62] B. Thalheim, *Entity-relationship modeling: foundations of database technology*. Springer Science & Business Media, 2013.

[63] N. Jukic, S. Vrbsky, S. Nestorov, and A. Sharma, *Database systems: Introduction to databases and data warehouses*. Pearson, 2014.

[64] M. Celikovic, V. Dimitrieski, S. Aleksic, S. Ristic, and I. Lukovic, "A DSL for EER data model specification," in *Information Systems Development: Transforming Organisations and Society through Information Systems - Proceedings of the 23rd International Conference on Information Systems Development, ISD 2014, Varaždin, Croatia, September 2-4, 2014* (V. Strahonja, N. Vrcek, D. P. Vukovac, C. Barry, M. Lang, H. Linger, and C. Schneider, eds.), Association for Information Systems, 2014.

[65] J. Lopes, M. Bernardino, F. Basso, and E. Rodrigues, "Textual approach for designing database conceptual models: A focus group," in *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2021, Online Streaming, February 8-10, 2021* (S. Hammoudi, L. F. Pires, E. Seidewitz, and R. Soley, eds.), pp. 171–178, SCITEPRESS, 2021.

[66] J. Lopes, M. Bernardino, F. Basso, and E. Rodrigues, "Empirical evaluation of a textual approach to database design in a modeling tool," in *Proceedings of the 23rd International Conference on Enterprise Information Systems, ICEIS 2021, Online Streaming, April 26-28, 2021, Volume 1* (J. Filipe, M. Smialek, A. Brodsky, and S. Hammoudi, eds.), pp. 208–215, SCITEPRESS, 2021.

[67] "Sprotty Wiki." https://github.com/eclipse/sprotty/wiki/, accessed: 2022-06-07.

[68] A. Kemper and A. Eickler, *Datenbanksysteme - Eine Einführung, 8. Auflage*. Oldenbourg, 2011.