TU WIEN Informatics

Using GLSP to create an OntoUML editor

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Benjamin Wieser
Matrikelnummer 11919177

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dr. Dominik Bork

Wien, 31. August 2025

Benjamin Wieser	Associate Prof. Dr. Dominik Bork

TU WIEN Informatics

Using GLSP to create a OntoUML editor

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Wirtschaftsinformatik

by

Benjamin Wieser
Registration Number 11919177

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dr. Dominik Bork

Vienna, August 31, 2025

Benjamin Wieser	Associate Prof. Dr. Dominik Bork

Erklärung zur Verfassung der Arbeit

Benjamin Wieser

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

Wien, 31. August 2025

Benjamin Wieser

Danksagung

An dieser Stelle möchte ich all jenen Personen meinen aufrichtigen Dank aussprechen, die mich während der Entstehung dieser Bachelorarbeit unterstützt und begleitet haben.

Mein besonderer Dank gilt meinem Betreuer, Associate Prof. Dr. Dominik Bork, für die hervorragende Betreuung, sein engagiertes Feedback und unerschöpfliche Geduld während der gesamten Bearbeitungszeit. Seine konstruktiven Anregungen waren maßgeblich für das Gelingen dieser Arbeit.

Ebenso möchte ich Dipl.-Ing. Haydar Metin meinen herzlichen Dank für seine tatkräftige Unterstützung während der Entwicklungsphase der Arbeit aussprechen. Seine Hilfsbereitschaft und sein technischer Rat waren von unschätzbarem Wert.

Schließlich danke ich meiner Familie und meinen Freunden für ihre Motivation und ihr Verständnis in den letzten Monaten.

Kurzfassung

Die konzeptuelle Modellierung spielt eine entscheidende Rolle bei der Entwicklung und dem Verständnis komplexer Informationssysteme. Während die Unified Modeling Language (UML) weit verbreitet ist, mangelt es ihr an inhärenten ontologischen Unterscheidungen, die für die robuste Ontologieerstellung notwendig sind. OntoUML, eine Erweiterung von UML, die auf der Unified Foundational Ontology (UFO) basiert, begegnet dieser Einschränkung durch die Einführung von Stereotypen und Constraints. Bestehende OntoUML-Editoren leiden oft unter Plattformabhängigkeit, was Flexibilität und Interoperabilität behindert. Diese Bachelorarbeit präsentiert bigOntoUML, ein neuartiges, plattformunabhängiges Modellierungswerkzeug für OntoUML. Durch die Nutzung des Graphical Language Server Protocol (GLSP) verwendet bigOntoUML eine Client-Server-Architektur, die die Kernmodellierungslogik und Validierung in einem auf Visual Studio Code basierenden Language Server trennt. Dies ermöglicht die potenzielle Interaktion mehrerer Client-Anwendungen mit derselben OntoUML-Modellierungs- und Validierungs-Engine. Die Arbeit beschreibt detailliert die Implementierung von Schlüsselfunktionen, einschließlich der Unterstützung des OntoUML-Metamodells, einer benutzerfreundlichen grafischen Oberfläche, der Modellvalidierung basierend auf OntoUML-Constraints, der Implementierung von OntoUML-Mustern (insbesondere des Phasenpartitionsmusters) sowie Modellimport- und -exportfunktionen. Eine Evaluierung der implementierten Funktionen demonstriert die Machbarkeit und das Potenzial von bigOntoUML als flexible und interoperable Plattform für die Erstellung und Validierung ontologisch fundierter konzeptueller Modelle. Diese Arbeit legt den Grundstein für zukünftige Entwicklungen, die darauf abzielen, die Fähigkeiten des Werkzeugs weiter zu verbessern und seine Anwendbarkeit zu erweitern.

Abstract

Conceptual modeling plays a crucial role in the development and understanding of complex information systems. While the Unified Modeling Language (UML) is widely adopted, it lacks inherent ontological distinctions necessary for robust ontology creation. OntoUML, an extension of UML grounded in the Unified Foundational Ontology (UFO), addresses this limitation by introducing stereotypes and constraints. Existing OntoUML editors often suffer from platform dependency, hindering flexibility and interoperability. This bachelor thesis presents BigOntoUML, a novel platform-independent modeling tool for OntoUML. By leveraging the Graphical Language Server Protocol (GLSP), bigOntoUML adopts a client-server architecture, separating the core modeling logic and validation into a Visual Studio Code-based Language Server. This enables the potential for multiple client applications to interact with the same OntoUML model and validation engine. The thesis details the implementation of key features, including support for the OntoUML metamodel, a user-friendly graphical interface, model validation based on OntoUML constraints, the implementation of OntoUML patterns (specifically the Phase Partition Pattern), and model import/export functionalities. An evaluation of the implemented features demonstrates the feasibility and potential of bigOntoUML as a flexible and interoperable platform for creating and validating ontologically sound conceptual models. This work lays the foundation for future development aimed at further enhancing the tool's capabilities and expanding its applicability.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**

2 Foundations **3**
2.1 Visual Studio Code . . . 3
2.2 Language Server . . . 3
2.3 Eclipse GLSP . . . 4
2.4 BigUml . . . 4
2.5 Eclipse UML2 . . . 4
2.6 OntoUML . . . 5

3 Related Work **11**
3.1 OntoUML for Visual Paradigm . . . 11
3.2 OpenPonk . . . 12
3.3 A Modeling Infrastructure for OntoUML . . . 13
3.4 Tonto . . . 13
3.5 OntoGrapher . . . 13
3.6 OLED . . . 13
3.7 Summary . . . 13

4 Implementation **15**
4.1 Requirements . . . 15
4.2 OntoUML Metamodel . . . 16
4.3 User Interface . . . 18
4.4 Model Validation . . . 20
4.5 OntoUML Patterns . . . 26
4.6 Phase Partition Pattern . . . 27
4.7 Import Model . . . 27
4.8 Export Model . . . 28

4.9 Evaluation 28

5 Future work **31**
5.1 Integrate OntoUML server 31
5.2 Anti pattern validation 31
5.3 Multible diagram support 31
5.4 Model actions on specific elements 32

6 Conclusion **33**

Overview of Generative AI Tools Used **35**

List of Figures **37**

List of Tables **39**

List of Algorithms **41**

Bibliography **43**

CHAPTER 1

Introduction

Conceptual Modeling is an important part of developing and understanding information systems. Modern information systems rely on huge amounts of interconnected data and logic. To make these systems comprehensible to humans, it is necessary to create models that structure the information and logic contained in information systems. The process of conceptual modeling involves the identification of key entities within a system, the definition of the relationships that exist between these entities, and the specification of any constraints that govern their interactions. By simplifying complex ideas into more understandable representations, conceptual models facilitate communication among stakeholders [Tha11] and provide a guide to implementation.

The Unified Modeling Language (UML) is a well established language for describing conceptual models. Most professionals in computer science have at least some knowledge about it, and have learned about it at university [RLR14]. However, UML lacks inherent ontological distinctions necessary for creating robust ontologies. Therefore, the OntoUML extension to the UML language was created. It allows to extend UML classes and associations with stereotypes that are grounded in the Unified Foundational Ontology (UFO) [GBF+22] . Additionally, it places constraints on how the stereotypes can be applied to the model, allowing additional validation of the models.

There are already some editors available for OntoUML, most notably an extension of Visual Paradigm [FSV+21]. However, they are all tied to a specific platform. To make operations on the models as well as validations of the model more independent, bigOntoUML relies on the Graphical Language Server Protocol (GLSP) [Foub]. GLSP is a project from the Eclipse Foundation that enables Language Servers for Graphical Applications. This enables bigOntoUML to separate model operations into the server, while graphical representation is implemented in the client. By utilizing this concept, it is possible to create multiple client applications that utilize the same server [BL23]. bigOntoUML implements its server as a Visual Studio Code extension. However,

it would be possible to create clients for other platforms like the Web or as a Visual Paradigm plugin.

This thesis aims to integrate the OntoUML meta-model in the existing BigUML software. By relying on the VS Code platform, we show that a web-based editor for OntoUML can be created. By utilizing the GLSP framework, we show that an OntoUML editor can be separated into a user interface and a model server.

The chapter Foundations describes core concepts that are used throughout this thesis. The related work chapter looks at previous attempts to create OntoUML editors, their advantages and shortcomings. In the implementation chapter, we go into how the BigOntoUML editor is built. In future work, we have a look at what can still be improved about BigOntoUML and how OntoUML editors can be improved going forward.

CHAPTER 2

Foundations

2.1 Visual Studio Code

Visual Studio Code (VS Code) is an Open Source text editor developed by Microsoft. It is built on Electron, a framework that allows building desktop applications with JavaScript. This allows VS Code to be available for many operating systems like Linux, Mac and Windows. Due to its extension-based architecture, it allows for heavy customization. Therefore, it is easy to use VS Code with many different programming languages. This also allows bigOntoUML to create a custom editor for OntoUML diagrams. The reliance on Electron allows extension developers to create custom views built on HTML, CSS and JavaScript. Due to this, it is possible to use web development frameworks that are widely used and understood by many developers. bigOntoUML takes advantage of this by building its user interface in React.

2.2 Language Server

Language Server is a concept originally introduced by Microsoft in VS Code [BL23]. It allows extension developers to separate language specific logic like code completion, validation and syntax highlighting from the more general logic of the editor. This allows creators of code editors to support many different languages without having to specifically implement the logic for the specific language into their editor. On the other hand, this allows creators of programming languages to create one language server that can then be integrated into many different code editors. The Language Server Protocol (LSP) is built on JSON-RPC, a protocol that describes how to exchange messages between two systems based on JSON formatted messages. The LSP does not define how information gets exchanged between the client and the server. It is possible that the server and the client run in the same process and exchange JSON information through function calls. It is also possible for them to run on different processes and exchange information with

network sockets. The Language Server works by having clients exchange messages with the server for every user interaction. For example, opening a file, inserting a character or requesting automatic code formatting. The server then responds with changed file content or an error message.

2.3 Eclipse GLSP

Eclipse Graphical Language Server Protocol applies the concepts of LSPs to graphical languages [MB23b]. This could be UML, ER or OntoUML diagrams. In GLSP the server is responsible for managing the source model and transforming it into the graphical model that is defined by GLSP. The graphical model contains some core building blocks like Nodes, Edges and Labels. This graphical model is serialized to JSON and sent to the client. In addition, the server is responsible for providing actions by which the model can be edited. This includes things like creating nodes and edges, moving elements or transformations, that can be applied to the model. Additionally, the server provides model validation such as error and warning messages. The client is responsible for transforming the graphical model into a user interface. In a web based setting, this is usually done by rendering it to SVG. It also provides a user interface for applying the actions that the server provides. When the server applies an action the client usually renders the model, to display the applied changes.

2.4 BigUml

BigUml [MB23a] is a VS Code extension developed by the BIG Research Group at TU Wien that utilizes GLSP. It supports different UML diagrams such as class diagram, activity diagram or state machine diagram. To handle model transformation as well as serialization of the model on the server, it uses the Eclipse UML2 library. bigOntoUML is built on top of BigUml and relies on it for the core UML functionality.

2.5 Eclipse UML2

Eclipse UML2 [Foua] is a project within the Eclipse Modeling Tools (MDT) that provides an implementation of the Unified Modeling Language (UML) 2.x metamodel. Built upon the Eclipse Modeling Framework (EMF), UML2 offers a foundational framework for developing UML-based modeling tools and applications within the Eclipse environment. It aims to facilitate the creation, manipulation, and exchange of UML models by providing a structured and standardized representation of the UML metamodel, along with supporting functionalities for model persistence and interoperability through XML. However, it can also be used as a standalone library to work with UML models. bigOntoUML relies on UML2 for model serialization and basic UML model actions and validations.

2.6 OntoUML

OntoUML is an ontology-driven conceptual modeling language [GBF+22]. Its primary goal is to support the creation of conceptual models that are not only intuitive and understandable but also formally well-grounded and semantically precise according to established ontological principles. It is specifically designed to analyze and represent the structure and relationships within a particular domain of interest, often used in requirements engineering, domain modeling for software development, and knowledge representation.

2.6.1 Relationship to UFO (Unified Foundational Ontology)

OntoUML is fundamentally based on the Unified Foundational Ontology (UFO). UFO is a scientifically grounded, top-level ontology that provides a set of basic categories and relations used to describe reality. It makes crucial distinctions about the nature of entities, such as:

- **Endurants vs. Perdurants:** Things that exist in full at any instant they exist (objects) vs. things that unfold over time (processes, events). OntoUML primarily focuses on modeling endurants.
- **Substantials vs. Moments:** Independently existing entities (like a person or a car) vs. entities that depend existentially on others (like a person's skills, a car's color, or a marriage).
- **Rigidity:** Whether a property is essential to an entity (e.g., being a Person is rigid) or contingent (e.g., being a Student is anti-rigid, as a person can cease to be a student without ceasing to be a person).
- **Sortals vs. Non-Sortals:** Types that provide a principle of identity for their instances (e.g., Person, Apple) vs. types that do not (e.g., Red Thing, Physical Object). UFO provides the semantic foundation for OntoUML. The specific modeling constructs available in OntoUML directly map to these ontological distinctions from UFO. This grounding ensures that OntoUML models adhere to principles that reduce ambiguity and prevent common conceptual errors.

2.6.2 Relationship to UML

OntoUML extends the UML, specifically its Class Diagrams. It uses the familiar graphical notation of UML class diagrams (classes, associations, attributes, generalisations) but adds specific stereotypes and constraints to represent the ontological distinctions provided by UFO.

- **Syntax:** OntoUML borrows the visual syntax (boxes for classes, lines for associations) from UML, making it relatively easy to learn for those already familiar with UML.

- **Semantics:** While standard UML is often semantically ambiguous (a 'Class' could represent many different kinds of things), OntoUML assigns precise meanings to its stereotyped classes and relationships based on UFO. For example, a class stereotyped as *«kind»* has a very different meaning and set of constraints than one stereotyped as *«role»*.

UFO provides the theory (ontology), UML provides the notation, and OntoUML integrates them into a practical modeling language [GBF+22].

2.6.3 OntoUML structure

Stereotyped Classes:

Classes are categorized based on UFO principles, indicated by stereotypes:
Sortal Substantials: Provide identity and persistence.

- *«kind»*: Represents essential, rigid types (e.g., Person, Organization). Every object must instantiate exactly one Kind throughout its lifetime.
- *«subkind»*: Represents rigid specializations of Kinds or other Subkinds (e.g., Man as a subkind of Person).
- *«quantity»*: Represents rigid types that are uncountable. (e.g Sand, Water)
- *«collective»*: Represents rigid types that have members. (e.g. Family, Band). All members must be equally perceived by all members.
- *«role»*: Represents types played contingently in a relational context (e.g., Student, Husband, Employee). Requires a relationship to exist. Anti-rigid.
- *«phase»*: Represents types defined by intrinsic properties that contingently apply during a certain period (e.g., Living Person, Pupil, Adult). Anti-rigid.

Figure 2.1 shows the kind "vehicle" that has three subkinds.

Non-Sortal Substantials (Mixins): Group common properties but don't provide identity on their own. They are abstract and represent properties that can be shared across different Sortals.

- *«category»*: Rigid and applies to different Kinds (e.g., Physical Object, Agent).
- *«roleMixin»*: Groups common properties of roles (e.g., Customer, applicable to both Persons and Organizations). Anti-rigid.
- *«mixin»*: A general non-sortal type, neither necessarily rigid nor anti-rigid on its own. This allows it to be a common property of Rigid and Anti-rigid types.

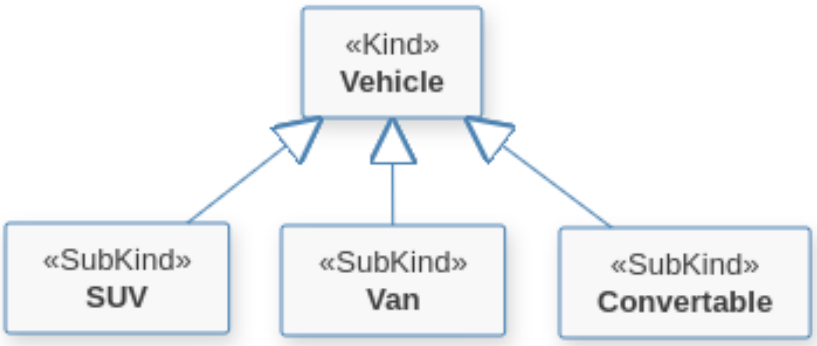

Figure 2.1: Example with kind

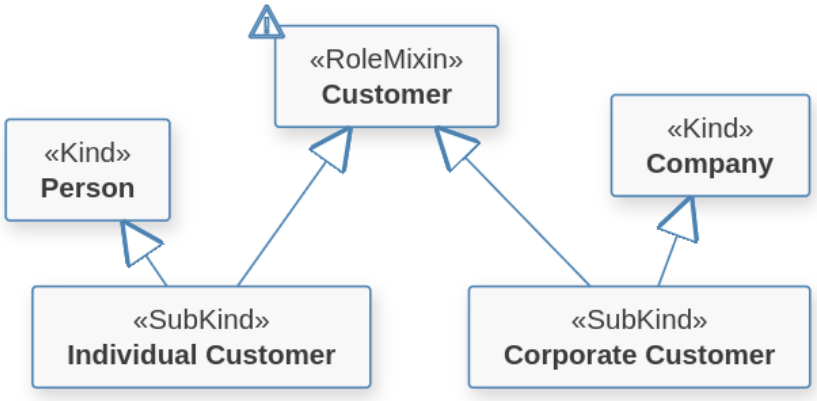

Figure 2.2: Example with role mixin

Figure 2.2 shows the roleMixin customer, which has individual customer and company customer as possible values.

Moments: Existentially dependent entities.

- *«mode»*: Intrinsic dependent entities (e.g., a person's Skill, a car's Power). An example of a mode can be seen in figure 2.3.
- *«relator»*: Represents truth-makers for material relations, embodying the connection between entities (e.g., Marriage, Enrollment, Employment).

Figure 2.4 shows the foundational model behind OntoUML.

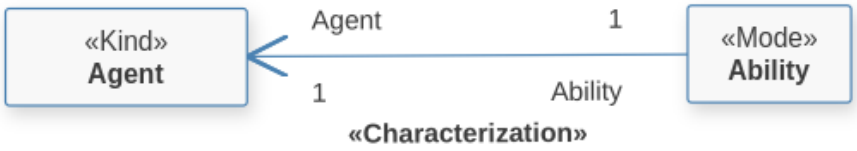

Figure 2.3: Example with mode

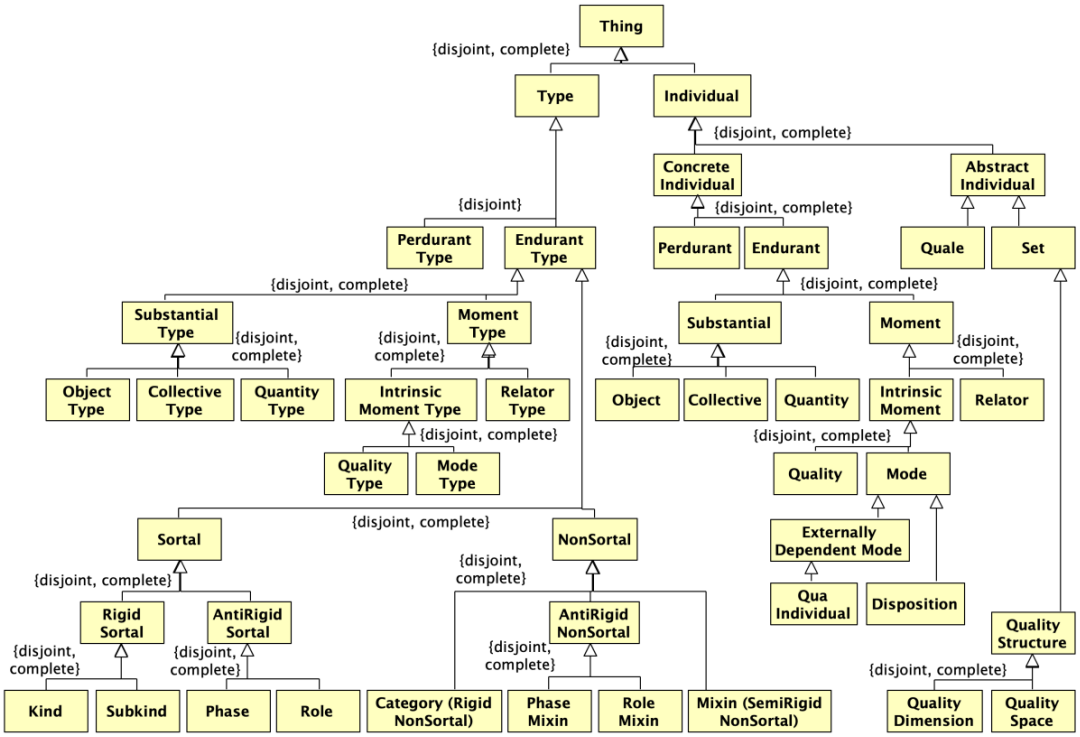

Figure 2.4: OntoUML Foundation [GBF+22]

Stereotyped Associations:

While OntoUML uses the basic visual syntax of UML associations (lines connecting classes), it imbues these connections with specific ontological meanings derived from UFO, often indicated by stereotypes or specific modeling patterns. The goal is to represent the nature of the relationship accurately, going beyond simple linkage.
Parthood (Memero): UFO has a detailed theory of part-whole relationships. OntoUML uses specific stereotypes to distinguish kinds of parthood:

- *«componentOf»*: Relates a functional part to a whole (e.g., an Engine is a componentOf a Car). Usually mandatory for the part.
- *«memberOf»*: Relates a member to a collection (e.g., a Tree is a memberOf a Forest). Often non-essential for the member.
- *«subCollectionOf»*: Relates a collection to a larger collection (e.g., a specific Fleet is a subCollectionOf a larger group of Fleets).
- *«subQuantityOf»*: Relates a portion of matter to a larger quantity (e.g., a specific volume of Water is a subQuantityOf the Water in a Lake). Usually involves non-discrete entities.

Dependency Relationship:

- *«mediation»*: Relates a *«Realtor»* to the connected objects. A realtor must be connected to two different individuals (e.g. connects a person to an university trough a realtor enrollment)
- *«characterisation»*: Relates a *«Mode»* to the owning Bearer. (e.g. connects an agent to an ability, seen in figure 2.3)
- *«derivation»*: Is applied to relations to indicate that the existence of this relationship is derived from other elements in the model.

Other Relationships:

- *«formal»*: Is applied to relations that can be reduced to comparisons (e.g. younger than, heavier than).
- *«material»*: Is applied to relations that have material structures on their own (e.g.employment, marriage). They are connected to a *«Relator»*.

CHAPTER 3

Related Work

There already exist a few tools to facilitate the creation of OntoUML models. This chapter takes a look at the features of those tools and how they are implemented. The features will be evaluated on the following criteria:

- **Programming language:** This criteria evaluates what technologies were used to create this tool.
- **Live validation (LV):** Some tools validate the OntoUML model on every model change. Other tools only validate on request by the user.
- **Pattern based modeling (PB):** Pattern based modeling allows for faster modeling and more well founded models. [GdGG11] This criteria looks if there are defined patterns that can be put into the model, without having to manually create all classes and associations.
- **Anti pattern detection (AP):** To support the creation of well founded models, some tools utilize automatic detection of OntoUML anti patterns [SG15].
- **Model transformations (MT):** For models to be useful, they often need to be transformed into other models such as OWL or Alloy.
- **Available platforms:** This criteria evaluates which platforms are supported by a modeling tool.

3.1 OntoUML for Visual Paradigm

Probably the most advanced and popular tool for OntoUML is the extension for Visual Paradigm. Visual Paradigm is a commercial modeling tool based on Java. It has a free version for basic UML editing as well as paid versions for advanced UML editing

or BPML modeling. The OntoUML extension is part of the OntoUML Server project [Ontc].

3.1.1 OntoUML Server

This project provides model intelligence as a service [FSV+21]. It encapsulates these services behind a web service. The services it provides are the following:

- **verification:** checking the syntactical correctness of models and returning warnings and errors for inconsistencies and underspecification.
- **OntoUML2gufo:** Transforms OnotUml Models to OWL models.
- **OntoUML2db:** Generates table definitions for a relational database, based on the knowledge contained in the OntoUML model.
- **OntoUML2alloy:** generates files to be able to simulate a model with the Alloy analyzer tool.
- **complexity management:** provides tools to refactor complex models like modularization and model abstraction.

3.1.2 Visual Paradigm Plugin

The OntoUML-VP plugin extends Visual Paradigm to interact with OntoUML servers. Integrating with OaaS, it adds OntoUML constructs (stereotypes), enables model (de)serialization via OntoUML-schema, and provides access to the OntoUML-server. As Visual Paradigm is written in Java, the plugin is also written in Java. Due to the portability of Java, Visual Paradigm is available for all Desktop environments such as Windows, Linux and Mac. However it is not available on the web.

3.2 OpenPonk

Based on the Pharo development platform, OpenPonk is a modeling framework that facilitates the creation of extensions for multiple languages(e.g. BPMN, UML, Petri nets) [UP16]. It implements modeling with OntoUML concepts [Bě21], model validation and anti-pattern recognition. It was created by the Technical University in Prague and is still under active development. A large part of development is focused on anti-pattern detection. Due to being built on the Pharo platform, it is developed in SmallTalk and runs on common desktop environments like Linux, Windows, and Mac. It supports model transformations into relational databases [Jab24].

3.3 A Modeling Infrastructure for OntoUML

This paper relies [Car10] on the Eclipse Modeling Framework (EMF) to separate the model editor in two parts. For the user interface, it utilizes the UML editor of the Ecore platform. The OntoUML specific stereotypes are provided by an UML2 profile. For validation and model intelligence, it creates a separate model in Ecore. To use validations, it transforms between the two models. OntoUML constraints are implemented into the Ecore model. Unfortunately, the Ecore platform has incompatibilities between versions, so it is hard to run this editor on modern versions of the Eclipse modeling tools.

3.4 Tonto

A tool supports textual representations, Tonto is a Visual Studio Code plugin that allows to create and edit OntoUML models in a textual form. [CASG24] Due to its JSON export capabilities, it is compatible with the OntoUML Server project. Additionally, it allows for syntax highlighting, auto complete and export int OWL. Since it is built on the VS Code platform, it runs Linux, Windows, Mac as well as web based VS Code implementations.

3.5 OntoGrapher

OntoGrapher is a purely web based tool for creating ontological models. While it supports, basic validations there is no support for advanced features like pattern based modeling or anti pattern detection. It uses OWL as its primary data format, thereby eliminating the need to transform models into this language. However, there is no option to transform to other models [BK21].

3.6 OLED

OLED is an OntoUML editor, that was maintained by the NEMO research group. However, maintenance was discontinued in favor of the OntoUML Server project. It is written in Java, and therefore available for Linux, Windows, and Mac. It has advanced support for anti-pattern detection and pattern based modeling [GSGA15].

3.7 Summary

There exists a multitude of tools, built for creating OntoUML models. Table 3.1 shows an overview of the discussed tools and a selection of features. With the rise of web technologies it becomes increasingly more important, that modeling tools are also available through a web browser. Here Tonto and OntoGrapher already provide first attempts to allow this. However, both tools are severely limited. Tonto only allows textual input, while OntoGrapher doesn't support model intelligence.

	LV	PB	AP	MT	Programming Language	Available Platforms
OntoUML for VP			y	y	Java, JavaScript	Desktop
OpenPonk		y	y	y	Pharo, SmallTalk	Desktop
Modeling Infrastructure	y				Java	Desktop
Tonto	y			y	TypeScript	Desktop, Web
OntoGrapher					TypeScript	Web
OLED		y	y	y	Java	Desktop

Table 3.1: Comparison of OntoUML tools

In order to provide more flexibility in building the input platform, the model logic needs to be separated from the user interface. The OntoUML server project is already a first step in this direction. With this approach, the client however still needs a lot of logic related to the model. This is something that can be improved by relying on the GLSP. [BLO23] This is the basis for new editor, bigOntoUML that incorperates these principles.

CHAPTER 4

Implementation

This chapter describes how the bigOntoUML frontend and backend is implemented.

4.1 Requirements

The goal of this project is to extend the existing BigUml software to allow it to handle OntoUML models. This can be broken down into three parts.

- Model visualization
- Model validation
- Import / export model

4.1.1 Model visualization

A core component of OntoUML are the UML stereotypes. Therefore, the BigUml application needs to be extended to support stereotypes for classes and associations. Additionally, in OntoUML, associations are directed, meaning there is a source and a target class. To display this to the user, associations are displayed as arrows instead of lines.

4.1.2 Model validation

A big part of OntoUML are the validations that constrain the uses of stereotypes on classes and associations. This is crucial for productive modeling, as well as for learning the OntoUML modeling language. bigOntoUML has to evaluate these constraints to provide real time feedback to the modeler.

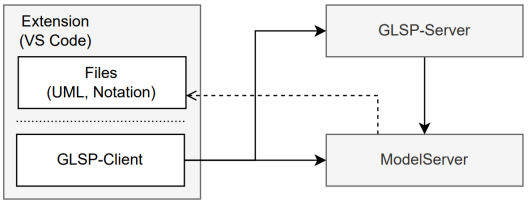

Figure 4.1: BigUml architecture [MB23a]

4.1.3 Import / export model

In order to interface with the rest of the OntoUML community, the functionality to import and export OntoUML models in the OntoUML schema [Ontd] needs to be implemented.

From a technical perspective the client-server architecture has to be kept as defined in [MB23a]. This architecture can be seen in figure 4.1.

4.2 OntoUML Metamodel

The base of the OntoUML model is formed by the Eclipse UML 2 library. It is an EMF based implementation of the UML standard. Using the primitives defined in UML, a profile is created, containing stereotypes for classes and associations. The stereotypes are built as a tree with abstract parents, to facilitate model validations later on (e.g. every sortal needs to have a substance sortal as ancestor). The UML 2 profile was developed by [Car10]. This metamodel is referenced to the UML2 types to which it can be applied. This model is serialized to a ".uml" file and distributed together with the OntoUML plugin. Figure 4.2 shows the class stereotypes and figure 4.3 shows the association stereotypes.

When creating a new model, the OntoUML meta model gets loaded from the resource folder with the file name `"\OntoUML.uml"`. It contains the OntoUML profile. The profile then needs to be added to the model and applied, so UML2 actually recognizes the profile. This can be seen in listing 4.2. The URI provided to the input stream can be anything, as it doesn't actually have one, even if it is required by the library.

```
1 var profile = loadOntoUMLProfile();
2 model.getPackagedElements().add(profile);
3 model.applyProfile(profile);
4
5 private Profile loadOntoUMLProfile() {
6    var resourceFile = UMLSourceModelStorage.class.getResourceAsStream
       ("/OntoUML.uml");
7    var resourceSet = this.getOrCreateEditingDomain().getResourceSet()
       ;
8
9    // Create a URI for the in-memory model (temporary, as InputStream
        has no URI)
```

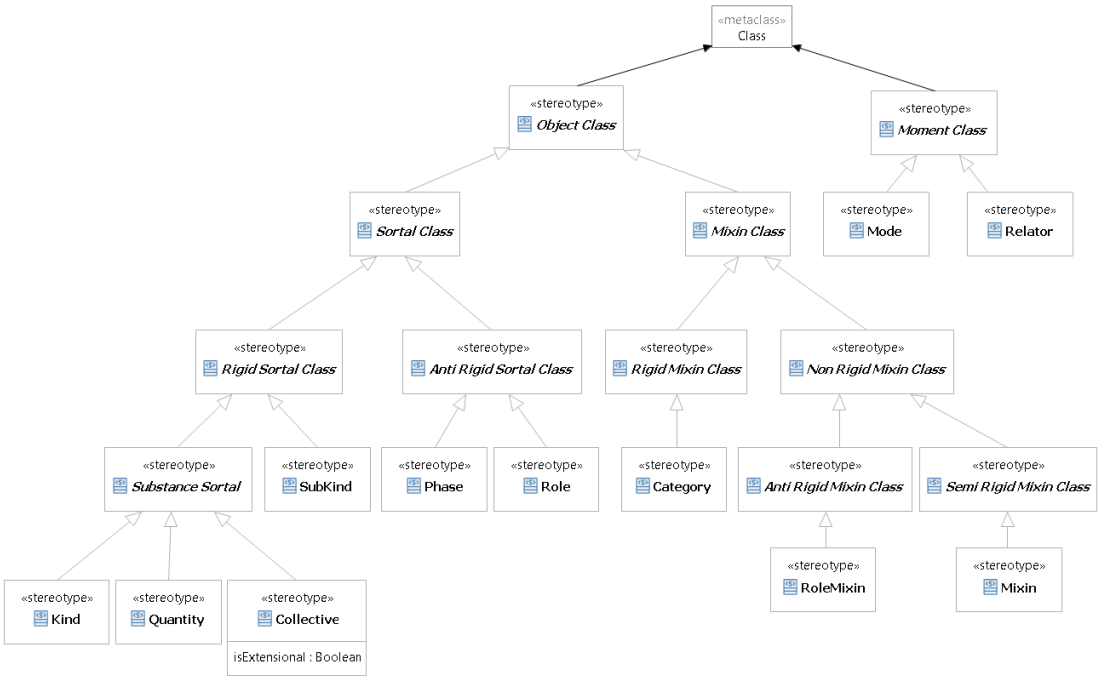

Figure 4.2: Class stereotype inheritance [Car10]

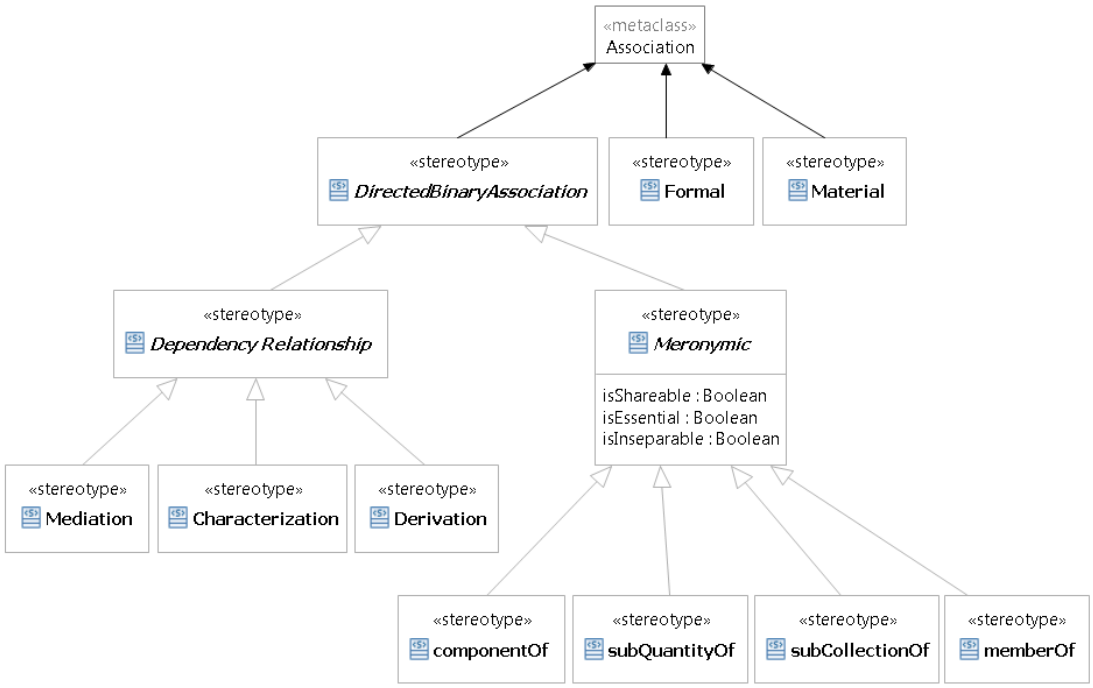

Figure 4.3: Association stereotype inheritance [Car10]

```
10      URI uri = URI.createURI("https://www.vorstieg.eu/OntoUML.uml");
11
12      // Create a resource and load it from the InputStream
13      Resource resource = resourceSet.createResource(uri);
14      try {
15          resource.load(resourceFile, Collections.emptyMap());
16      } catch (IOException e) {
17          throw new RuntimeException(e);
18      }
19      return (Profile) resource.getContents().get(0);
20  }
```

For the application to be able to use the UML 2 libraries outside an Ecore project, the libraries have to be loaded in the Gradle file. The required dependencies can be seen in listing 4.2.

```
21  p2deps {
22      into('implementation') {
23          p2repo('https://download.eclipse.org/releases
                /2023-09/202309131000/')
24          install('org.eclipse.uml2.uml')
25          install('org.eclipse.uml2.uml.profile.standard')
26          install('org.eclipse.uml2.uml.resources')
27      }
28  }
```

4.3 User Interface

BigUml already provides a user interface for most UML components that are required to create OntoUML models. However, a way to add stereotypes to a class needs to be created. Therefore, the property palate is extended to allow stereotype applications. The method getApplicableStereotypes() provided by UML 2 is used to only display relevant stereotypes. When changing the selected stereotype, the old stereotype is removed and the new one is applied. This is necessary, since in standard UML, a class can have multiple stereotypes. However, in OntoUML this is not possible. Listing 4.3 shows the code that is used to select stereotypes. Figure 4.4 shows the interface to select stereotypes in bigOntoUML.

```
29  @Override
30  public List<ElementPropertyItem> doProvide(final Classifier element)
        {
31      var options = element.getApplicableStereotypes().stream()
32              .map(stereotype -> ElementChoicePropertyItem.Choice.
                    builder().value(stereotype.getQualifiedName()).label(
                    stereotype.getName()).build())
33              .toList();
34
```

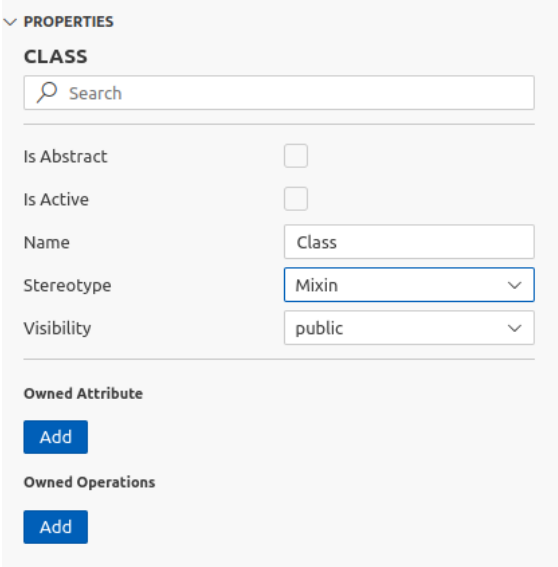

Figure 4.4: Stereotype selection user interface

```
35      var elementId = providerContext.idGenerator().getOrCreateId(
            element);
36      var builder = new ElementPropertyBuilder(elementId)
37              .choice(STEREOTYPE, "Stereotype", options, element.
                    getAppliedStereotypes().stream().findFirst().map(
                    Stereotype::getQualifiedName).orElse(""));
38      return builder.items();
39  }
40
41  @Override
42  public Command doHandle(final BGUpdateElementPropertyAction action,
        final Classifier element) {
43      var value = action.getValue();
44      var argument = UMLUpdateElementCommand.Argument
45              .<Element>updateElementArgumentBuilder()
46              .consumer(e -> {
47                  if (action.getPropertyId().equals(STEREOTYPE)) {
48                      e.getAppliedStereotypes().forEach(e::
                            unapplyStereotype);
49                      e.applyStereotype(e.getApplicableStereotype(value)
                            );
50                  }
51              })
52              .build();
53
54
55      return new UMLUpdateElementCommand<>(context, modelState.
            getSemanticModel(), element, argument);
```

```
56 }
```

Most relevant parts of UML class diagrams are already implemented in BigUML. To add stereotypes to class diagrams, the GLSP code needs to be extended with stereotypes. To achive this, the GClassBuilder and GAssociationBuilder get extended as shown in listing 4.3.

```
58 @Override
59 protected List<GCProvider> createComponentChildren(final GNode
       modelRoot, final GCModelList<?, ?> componentRoot) {
60    return List.of(createHeader(componentRoot), createClassBody(
          componentRoot));
61 }
62
63 protected GCProvider createHeader(final GCModelList<?, ?> root) {
64    var namedElementOptions = GCNamedElement.Options.builder()
65            .container(root);
66
67    origin.getAppliedStereotypes().forEach(stereotype ->
68            namedElementOptions.prefix(BGQuotationMark.
                quoteDoubleAngle(stereotype.getName())));
69
70    if (origin.isAbstract()) {
71        namedElementOptions.nameCss(BGCoreCSS.FONT_ITALIC);
72    }
73
74    return new GCNamedElement<>(context, origin, namedElementOptions.
          build());
75 }
```

4.4 Model Validation

The validation is split into two groups. In the first group the model nodes (classes, generalizations) get validated. In the second group the edges (associations) are validated. The model validations implemented in BigOntoUML are based on [Car10]. If a constraint violation is found, the addMarker() method is called, it creates a warning sign in the user interface, together with a human readable description.

4.4.1 Classes

The first and simplest validation is to make sure that every class has a stereotype attached to it. If this condition is not fulfilled, a warning gets displayed, and all other validations are skipped for the element.

Object class

Object classes are only allowed to have one substance sortal ancestor. This is achieved by counting all ancestors of the class that have a substance sortal ancestor. If the count is greater than one, a validation error is displayed. The implementation can be seen in listing 4.4.1.

```
77 private void evaluateObjectClass(GNode element, Class _class,
       ArrayList<Marker> markers, Stereotype stereotype) {
78     if (hasParentStereotype(stereotype, OBJECT_CLASS)) {
79         if (countAncestorsWithStereotype(_class, SUBSTANCE_SORTAL) >
               1) {
80             addMarker(element, markers, formatClassMessage("Every
                   Object Class must not have more than one Substance
                   Sortal ancestor", _class));
81         }
82     }
83 }
```

Sortal class

Sortal classes need to have a substance sortal ancestor or be themselves a substance sortal. This is achieved by checking the stereotypes of all ancestors and making sure that one is a substance sortal as seen in listing 4.4.1.

```
84 private void evaluateSortalClass(GNode element, Class _class,
       ArrayList<Marker> markers, Stereotype stereotype) {
85     if (hasParentStereotype(stereotype, SORTAL_CLASS) && !
           hasParentStereotype(stereotype, SUBSTANCE_SORTAL)) {
86         if (countAncestorsWithStereotype(_class, SUBSTANCE_SORTAL) ==
               0) {
87             addMarker(element, markers, formatClassMessage("Every non-
                   abstract Sortal must have a Substance Sortal ancestor (
                   or be a Substance Sortal)", _class));
88         }
89     }
90 }
```

Rigid Sortal Class

If a class is a rigid sortal, it can not have an anti-rigid ancestor. However since the meta structure does not have an abstract concept for anti rigit stereotypes, role mixin have to be checked in addition to anti rigid sortal classes (listing 4.4.1).

```
91 private void evaluateRigidSortalClass(GNode element, Class _class,
       ArrayList<Marker> markers, Stereotype stereotype, ) {
92     if (hasParentStereotype(stereotype, RIGID_SORTAL_CLASS)) {
```

```
93          if (countParentsWithStereotype(_class, "Anti Rigid Sortal
                Class") != 0 || countAncestorsWithStereotype(_class, "Role
                Mixin") != 0) {
94              addMarker(element, markers, formatClassMessage("A Rigid
                    Sortal cannot have an Anti-Rigid parent (Role, Phase
                    and RoleMixin)", _class));
95          }
96      }
97  }
```

Forbidden ancestry

A mixin class can not have a sortal class ancestor, a category can not have a role mixin parent, a substance sortals is not allowed to have a rigid sortal ancestor and a mixin can not have a role mixin parent. To make sure this is the case, all ancestors with the corresponding stereotype are counted as seen in the mixin example in listing 4.4.1.

```
98  private void evaluateMixinClass(GNode element, Class _class,
        ArrayList<Marker> markers, Stereotype stereotype) {
99      if (hasParentStereotype(stereotype, MIXIN_CLASS)) {
100         if (countAncestorsWithStereotype(_class, SORTAL_CLASS) != 0) {
101             addMarker(element, markers, "A Mixin Class (Category,
                    Mixin, RoleMixin) cannot have a Sortal parent (Kind,
                    Quantity, Collective, ");
102         }
103     }
104 }
```

Relator Class

A realtor must directly or through one of its generalizations be connected to an association with the mediation stereotype. If this is the case, the cardinality of member ends must be greater than two (listing 4.4.1).

```
105 private void evaluateRelatorMediation(GNode element, Class _class,
        ArrayList<Marker> markers, Stereotype stereotype) {
106     if (hasParentStereotype(stereotype, RELATOR)) {
107         if (!hasAssociationWithStereotype(_class, QUALIFIED_MEDIATION)
                ) {
108             addMarker(element, markers, formatClassMessage("A Relator
                    must be connected (directly or indirectly) to a
                    Mediation", _class));
109         } else if (countLowerAssociationWithStereotype(_class,
                QUALIFIED_MEDIATION) < 2) {
110             addMarker(element, markers, formatClassMessage("The sum of
                     the minimum cardinalities of the mediated ends must be
                     greater or equal to 2", _class));
111         }
```

```
112     }
113 }
```

Required Association

A role or a role mixin must be directly through one of its generalizations be connected to an association with the mediation stereotype.

A role or a role mixin must be directly through one of its generalizations be connected to an association with the characterization stereotype.

To verify this, the hasAssociationWithStereotype() method recursively checks all ancestors of the class in question, and iterates over all connected associations, as described in figure 4.4.1.

```
114 private void evaluateRoleMediation(GNode element, Class _class,
        ArrayList<Marker> markers, Stereotype stereotype) {
115     if (hasParentStereotype(stereotype, ROLE)) {
116         if (!hasAssociationWithStereotype(_class, QUALIFIED_MEDIATION)
                ) {
117             addMarker(element, markers, formatClassMessage("A Role
                    must be connected (directly or indirectly) to a
                    Mediation", _class));
118         }
119     }
120 }
121
122 private boolean hasAssociationWithStereotype(Classifier _class,
        String qualifiedName) {
123     var hasStereotype = _class.getAssociations().stream().filter(
            association -> association.getAppliedStereotype(qualifiedName)
             != null).findFirst();
124     if (hasStereotype.isPresent())
125         return true;
126     return _class.getGeneralizations().stream().anyMatch(
            generalization -> {
127         var general = generalization.getGeneral();
128         return hasAssociationWithStereotype(general, qualifiedName);
129     });
130 }
```

Material Class

Every material association must be connected to exactly one derivation (listing 4.4.1).

```
131 private void evaluateMaterialAssociationDerivation(GNode element,
        Class _class, ArrayList<Marker> markers, Stereotype stereotype) {
132     if (hasParentStereotype(stereotype, MATERIAL_ASSOCIATION)) {
```

```
133         if (countAssociationWithStereotype(_class, DERIVATION) != 1) {
134             addMarker(element, markers, formatClassMessage("Every
                    Material Association must be connected to exactly one
                    Derivation", _class));
135         }
136     }
137 }
```

4.4.2 Generalisation

Object classes are only allowed to have other object classes as ancestors. So if there is a generalization with an object class participating, the other class has to also be an object class. Validation for this can be seen in figure 4.4.2

```
138 private void validateGeneralisation(GNode element, Generalization
        generalization, ArrayList<Marker> markers) {
139     if ((hasParentStereotype(generalization.getGeneral(), OBJECT_CLASS
            ) &&
140             !hasParentStereotype(generalization.getSpecific(),
                    OBJECT_CLASS)) ||
141             (!hasParentStereotype(generalization.getGeneral(),
                    OBJECT_CLASS) &&
142                     hasParentStereotype(generalization.getSpecific(),
                            OBJECT_CLASS))) {
143         addMarker(element, markers, "An Object Class only participates
                in a Generalization with another Object Class");
144     }
145 }
```

4.4.3 Associations

Just as classes, associations need a stereotype, otherwise there will be a validation error, and the rest of the validation rules will be skipped.

Directed Binary Relationship

For directed binary relationships, source cardinality needs to be greater than one. If not, an error is raised. In OntoUML Relationships are often directed, while in UML this is not the case. So the UML 2 library doesn't provide a good distinction between source and target. The only way to differentiate them, is by the index. So the source class has index zero and the target one. Once the member ends are established, the cardinality can be checked by the getLower() method as seen in figure 4.4.3.

```
146 private void validateDirectedBinaryAssociation(GEdge element,
        Association association, ArrayList<Marker> markers, Stereotype
        stereotype) {
```

```
147     if (hasParentStereotype(stereotype, DIRECTED_BINARY_ASSOCIATION))
            {
148         if (association.getMemberEnds().get(0).getLower() < 1) {
149             addMarker(element, markers, "The source end minimum
                    cardinality must be greater of equal to 1.");
150         }
151     }
152 }
```

Derivation

For a derivation, the source must be a material association and the target must be a realtor. The cardinality of a derivation must be exactly one. Otherwise an error is raised. (figure 4.4.3)

```
153 private void validateDerivation(GEdge element, Association
        association, ArrayList<Marker> markers, Stereotype stereotype) {
154     if (hasParentStereotype(stereotype, DERIVATION)) {
155         var derivationSourceStereotype = association.getMemberEnds().
                get(1).getType().getAppliedStereotypes().stream().findFirst
                ();
156         var derivationTargetStereotype = association.getMemberEnds().
                get(0).getType().getAppliedStereotypes().stream().findFirst
                ();
157         if (derivationSourceStereotype.isEmpty() ||
                hasParentStereotype(derivationSourceStereotype.get(),
                MATERIAL_ASSOCIATION)) {
158             addMarker(element, markers, "(Derivation) The source must
                    be a Material Association");
159         }
160         if (derivationTargetStereotype.isEmpty() ||
                hasParentStereotype(derivationTargetStereotype.get(),
                RELATOR)) {
161             addMarker(element, markers, "(Derivation) The target must
                    be a Relator");
162         }
163         if (association.getMemberEnds().get(0).getLower() != 1 ||
                association.getMemberEnds().get(0).getUpper() != 1) {
164             addMarker(element, markers, "The Relator end cardinality
                    is exactly one");
165         }
166     }
167 }
```

Mediation

The source of a mediation must be a relator and the cardinality must be exactly one. Otherwise, an error is raised (listing 4.4.3).

```
168 private void validateMediation(GEdge element, Association association
        , ArrayList<Marker> markers, Stereotype stereotype) {
169     if (hasParentStereotype(stereotype, MEDIATION)) {
170         var mediationEndStereotype = association.getMemberEnds().get
                (1).getType().getAppliedStereotypes().stream().findFirst();
171         if (mediationEndStereotype.isEmpty() || !hasParentStereotype(
                mediationEndStereotype.get(), RELATOR)) {
172             addMarker(element, markers, "(Mediation) The source must
                    be a Relator");
173         }
174         if (association.getMemberEnds().get(1).getLower() < 1) {
175             addMarker(element, markers, "The Mediated end minimum
                    cardinality must be greater or equal to 1");
176         }
177     }
178 }
```

Characterization

For a characterization, the source must be a mode and the cardinality needs to be exactly one. Otherwise, an error is raised (listing 4.4.3).

```
179 private void validateCharacterization(GEdge element, Association
        association, ArrayList<Marker> markers, Stereotype stereotype) {
180     if (hasParentStereotype(stereotype, CHARACTERIZATION)) {
181         var modeEndStereotype = association.getMemberEnds().get(1).
                getType().getAppliedStereotypes().stream().findFirst();
182         if (modeEndStereotype.isEmpty() || !hasParentStereotype(
                modeEndStereotype.get(), MODE)) {
183             addMarker(element, markers, "(Characterization) The source
                    must be a Mode");
184         }
185         if (association.getMemberEnds().get(0).getLower() != 1 ||
                association.getMemberEnds().get(0).getUpper() != 1) {
186             addMarker(element, markers, "The Characterized end
                    cardinality is exactly one.");
187         }
188     }
189 }
```

4.5 OntoUML Patterns

By codifying modeling choices into patterns of well understood robust modeling solutions, the creation of well founded conceptual models can be improved. This reduces the mental load on the modeler, as they can rely on existing solutions to common problems. Relying on patterns also improves communication between modelers, since the modelers can

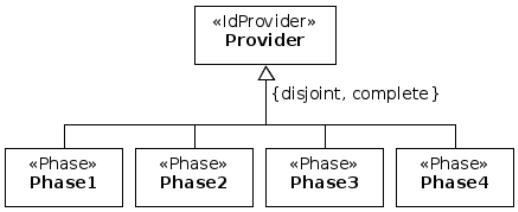

Figure 4.5: Phase partition pattern structure [Ontb]

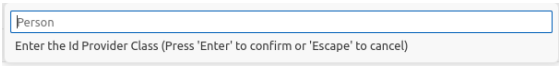

Figure 4.6: Id provider class name

discuss models using a shared vocabulary of already existing solutions. Codifying these design patterns into modeling tools allows for faster modeling and learning since they make them more accessible. Since VS Code allows for commands with complex dialogs, bigOntoUML creates the possibility of applying design patterns to a model.

4.6 Phase Partition Pattern

The Phase Partition (PP) pattern in modeling dictates that every Phase is inherently part of a partition structure. This structure is characterized by a single, unique root supertype which must be a Sortal. Figure 4.5 shows a generic phase partition pattern.

To facilitate the creation of a phase partition, bigOntoUML creates the command "OntoUML Patterns: Phase Partition". The phase partition command has three parameters:

- **Id Provider:** The name of the id provider class
- **Phase amount:** The amount of phases that need to be connected to the id provider
- **Phase names:** The class names of the phases.

The dialog shown to the user can be seen in figure 4.6.

After the user inputs these parameters, an OntoUML structure gets created as shown in figure 4.7

4.7 Import Model

In order to facilitate working together with modelers that don't work with bigOntoUML, an importer was created to import models from the OntoUML schema [Ontd]. The schema is supported by the OntoUML plugin for Visual Paradigm. The OntoUML

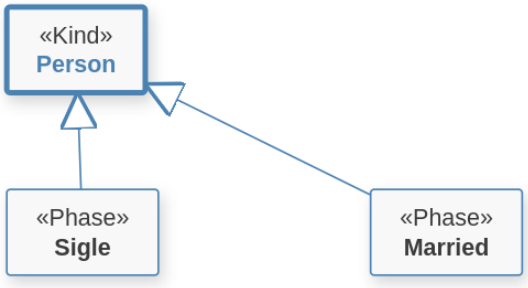

Figure 4.7: Phase pattern result

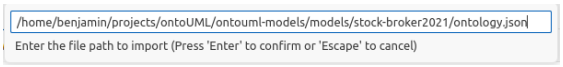

Figure 4.8: Import model: specify file path

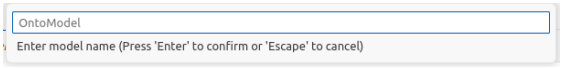

Figure 4.9: Import model: specify model name

catalog [Onta] provides a curated list of existing OntoUML models, for learning purposes, that are available in the OntoUML schema.

The importer works by de-serializing the JSON structure and mapping it to the UML2 data structure. When the user calls the import model command, they are prompted to specify the file path seen in figure 4.8. The user has to specify a model name figure 4.9. In figure 4.10 the result of the importer can be seen, importing the stock broker model [Ontf].

4.8 Export Model

The model exporter allows exporting a model to the OntoUML Schema [Ontd]. It creates a JSON file in the working directory, that can be imported into other programs like Visual Paradigm.

4.9 Evaluation

To evaluate the model editor, the first 100 OntoUML models from the OntoUML catalog [Onta] sorted by name were imported.

4.9.1 Limitations

While the importer is able to import simple models correctly, some limitations were found. The result of import errors can be found in table 4.1.

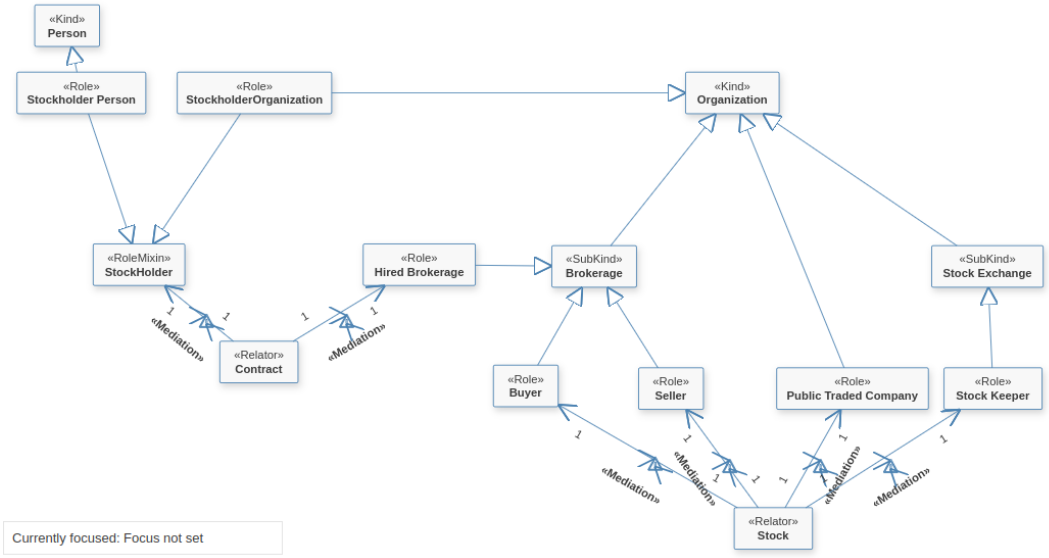

Figure 4.10: Import mode: imported stock broker model

- The Json standard [Bra14] mandates UTF encoding. Therefore Jackson, the JSON mapper used by bigOntoUML also requires encoding in UTF. However many models from the OntoUML catalog are encoded in ISO 8859-1, making it impossible to load these models into bigOntoUML.
- It is currently not possible to create an association that is connected to another association. Therefore models that contain this element will not be imported correctly.
- The *«event»* [AFG19] stereotype has been introduced to OntoUML in recent years. However it is still not part of the OntoUML standard and therefore not included in this work. Models that contain the *«event»* stereotype are therefore not correctly loaded.
- Some models in the OntoUml catalog contain multiple diagrams linked to one model. When importing these models, the whole model gets imported. Diagram information only gets loaded for one diagram. So model elements from other diagrams are loaded in one spot, instead of loading the correct location. This can be seen in figure 4.11.
- A small number of models in the catalog did not use standard UML stereotypes on associations. Instead they relied on custom stereotypes. These models lost those stereotypes on import.

Even when the import works correctly, there are sometimes validation errors, as some models don't follow the constraints defined in [Car10].

Encoding	Association	Event	Multiple diagrams	No stereotypes	Correct import
13	8	5	40	8	26

Table 4.1: Model import errors

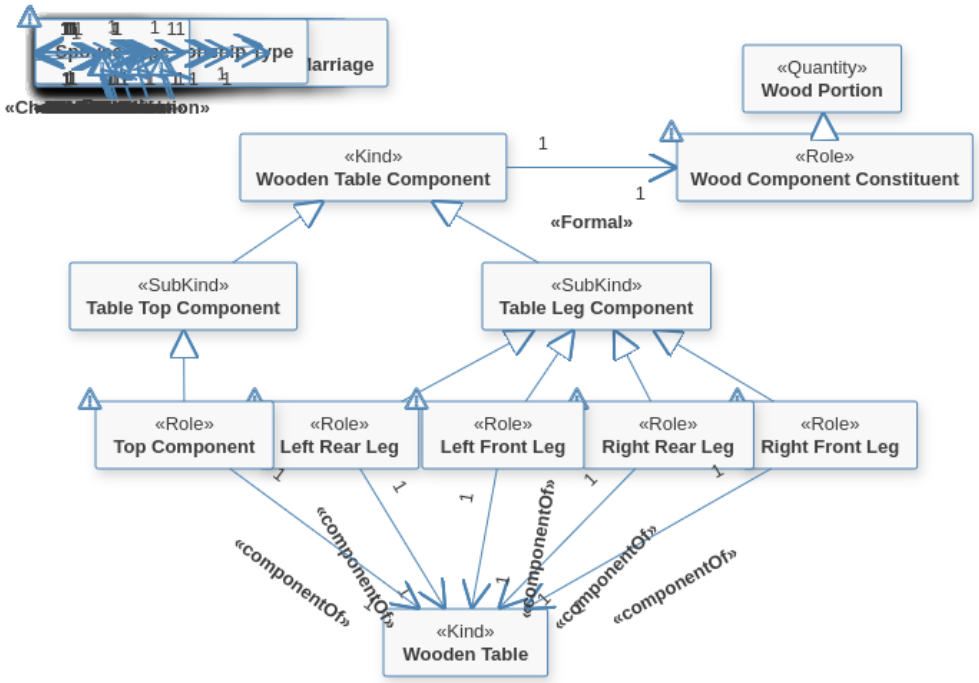

Figure 4.11: Model with multiple diagrams

CHAPTER 5

Future work

This paper only covers the fundamentals of implementing OntoUML in a VS code editor. There are many additional features that can be implemented to facilitate better modeling. This chapter describes additional features that can be implemented in bigOntoUML

5.1 Integrate OntoUML server

Instead of relying on UML2 to handle model serialization and basic model editing, the bigOntoUML language server could be implemented using the OntoUML server [Onte] that is used in the Visual Paradigm plugin for OntoUML [Ontc]. Since this already has many model intelligence services implemented, this could improve the functionality of bigOntoUML greatly, without having to do everything from scratch. Since the OntoUML server is implemented in JavaScript, this architecture would allow bigOntoUML to remove all Java dependencies from the project and therefore allow bigOntoUML to fully run in a web browser.

5.2 Anti pattern validation

There is a big body of research into OntoUML anti-patterns [SG15]. Adding automatic detection of these anti-patterns could allow modelers to create more robust models and allow students to learn OntoUML more quickly, by getting direct feedback on models.

5.3 Multible diagram support

The most frequent issue identified while importing models from the OntoUML catalog [**?**] was the lack of support for multiple diagrams for one model. This is also an area of improvement.

Figure 5.1: model actions on elements

5.4 Model actions on specific elements

Currently OntoUML only supports actions that operate on the whole model. To improve the modeling experience GLSP also allows actions to be executed on a single modeling node. This can be seen in figure 5.1. Model refactoring or pattern application can profit from this kind of user interface.

CHAPTER 6

Conclusion

This bachelor thesis presents the development of bigOntoUML, a platform-independent modeling tool for OntoUML, leveraging the Graphical Language Server Protocol (GLSP). The motivation behind this work stemmed from the limitations of existing OntoUML editors being tied to specific platforms, hindering flexibility and collaboration.

bigOntoUML addresses this challenge by adopting a client-server architecture. The core modeling logic and validation rules for OntoUML are encapsulated within a Language Server. This separation of concerns allows for the potential development of various client applications capable of interacting with the same underlying OntoUML model and validation engine.

The thesis detailed the implementation of key features of bigOntoUML, including:

- **Support for the OntoUML metamodel:** Enabling the creation and manipulation of core OntoUML elements with their specific stereotypes and properties.
- **A user-friendly graphical interface:** Providing an intuitive environment within Visual Studio Code for users to create and edit OntoUML diagrams.
- **Model validation based on OntoUML constraints:** Ensuring the semantic correctness of the models by identifying and reporting violations of the defined ontological rules.
- **Implementation of OntoUML patterns:** Demonstrating the tool's capability to support and potentially guide users in applying established ontological patterns, specifically focusing on the Phase Partition Pattern.
- **Model import and export functionalities:** Facilitating interoperability with existing modeling tools and standard model formats.

In conclusion, this thesis contributes to the field of conceptual modeling by presenting a novel, platform-independent approach to OntoUML modeling. By utilizing the GLSP framework, bigOntoUML paves the way for greater accessibility, interoperability, and collaboration in the creation and validation of ontologically well-founded conceptual models. The groundwork laid in this thesis offers promising avenues for future work, aiming to further enhance the capabilities and broaden the applicability of bigOntoUML within the conceptual modeling landscape.

Overview of Generative AI Tools Used

In this work, Google Gemini was used to improve clarity and readability. It was also used to translate the abstract into the Kurzfassung. In the implementation phase, ChatGPT and Google Gemini where used in bug fixing. The initial skeleton code for model import and export was created using ChatGPT. However, this was later heavily adapted.

List of Figures

2.1 Example with kind 7
2.2 Example with role mixin 7
2.3 Example with mode 8
2.4 OntoUML Foundation [GBF+22] 8

4.1 BigUml architecture [MB23a] 16
4.2 Class stereotype inheritance [Car10] 17
4.3 Association stereotype inheritance [Car10] 17
4.4 Stereotype selection user interface 19
4.5 Phase partition pattern structure [Ontb] 27
4.6 Id provider class name 27
4.7 Phase pattern result 28
4.8 Import model: specify file path 28
4.9 Import model: specify model name 28
4.10 Import mode: imported stock broker model 29
4.11 Model with multiple diagrams 30

5.1 model actions on elements 32

List of Tables

3.1 Comparison of OntoUML tools 14

4.1 Model import errors 30

List of Algorithms

Bibliography

[AFG19] João Paulo A Almeida, Ricardo A Falbo, and Giancarlo Guizzardi. Events as entities in ontology-driven conceptual modeling. In *Conceptual Modeling: 38th International Conference, ER 2019, Salvador, Brazil, November 4–7, 2019, Proceedings 38*, pages 469–483. Springer, 2019.

[BK21] Alice Binder and Petr Kremen. Ontographer: a web-based tool for ontological conceptual modeling. *en. In:()*, page 15, 2021.

[BL23] Dominik Bork and Philip Langer. Language server protocol: An introduction to the protocol, its use, and adoption for web modeling tools. *Enterp. Model. Inf. Syst. Archit. Int. J. Concept. Model.*, 18:9:1–16, 2023.

[BLO23] Dominik Bork, Philip Langer, and Tobias Ortmayr. A vision for flexible glsp-based web modeling tools. In João Paulo A. Almeida, Monika Kaczmarek-Heß, Agnes Koschmider, and Henderik A. Proper, editors, *The Practice of Enterprise Modeling - 16th IFIP Working Conference, PoEM 2023, Vienna, Austria, November 28 - December 1, 2023, Proceedings*, volume 497 of *Lecture Notes in Business Information Processing*, pages 109–124. Springer, 2023.

[Bra14] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, March 2014.

[Bě21] Bc. Marek Bělohoubek. Extending ontouml modelling capabilities on the openponk platform. Master's thesis, faculty of information technology ctu in prague, 2021.

[Car10] Roberto Carraretto. A modeling infrastructure for ontouml. *Graduation Thesis, Federal University of Espírito Santo*, 2010.

[CASG24] {Matheus L.} Coutinho, {João Paulo A.} Almeida, {Tiago Prince} Sales, and Giancarlo Guizzardi. A textual syntax and toolset for well-founded ontologies. In Cassia Trojahn, Daniele Porello, {Pedro Paulo Favato} Barcelos, and {Pedro Paulo Favato} Barcelos, editors, *Formal Ontology in Information Systems - Proceedings of the 14th International Conference, FOIS 2024*, Frontiers in Artificial Intelligence and Applications, pages 208–222, Netherlands, 2024. IOS.

[Foua] Eclipse Foundation. Eclipse mdt uml2.

[Foub] Eclipse Foundation. Glsp.

[FSV+21] Claudenir M Fonseca, Tiago Prince Sales, Victor Viola, Lucas BR Da Fonseca, Giancarlo Guizzardi, and João Paulo A Almeida. Ontology-driven conceptual modeling as a service. In *CEUR workshop proceedings*, volume 2969. Rheinisch Westfälische Technische Hochschule, 2021.

[GBF+22] Giancarlo Guizzardi, Alessander Benevides, Claudenir Fonseca, Daniele Porello, João Almeida, and Tiago Prince Sales. Ufo: Unified foundational ontology. *Applied Ontology*, 01 2022.

[GdGG11] Giancarlo Guizzardi, Alex Pinheiro das Graças, and Renata S. S. Guizzardi. Design patterns and inductive modeling rules to support the construction of ontologically well-founded conceptual models in ontouml. In Camille Salinesi and Oscar Pastor, editors, *Advanced Information Systems Engineering Workshops*, pages 402–413, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[GSGA15] John Guerson, Tiago Prince Sales, Giancarlo Guizzardi, and João Paulo A. Almeida. Ontouml lightweight editor: A model-based environment to build, evaluate and implement reference ontologies. In *2015 IEEE 19th International Enterprise Distributed Object Computing Workshop*, pages 144–147, 2015.

[Jab24] Jakub Jabůrek. Implementation of the transformation of an ontouml model in openponk into its realization in a relational database. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2024.

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

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

[Onta] OntoUML. Ontouml catalog.

[Ontb] OntoUML. Ontouml documentation phase pattern.

[Ontc] OntoUML. Ontouml plugin for visual paradigm.

[Ontd] OntoUML. Ontouml schema.

[Onte] OntoUML. Ontouml server.

[Ontf] OntoUML. Ontouml stock broker model.

[RLR14] Gianna Reggio, Maurizio Leotta, and Filippo Ricca. Who knows/uses what of the uml: A personal opinion survey. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, pages 149–165, Cham, 2014. Springer International Publishing.

[SG15] Tiago Prince Sales and Giancarlo Guizzardi. Ontological anti-patterns: empirically uncovered error-prone structures in ontology-driven conceptual models. *Data Knowledge Engineering*, 99:72–104, 2015. Selected Papers from the 33rd International Conference on Conceptual Modeling (ER 2014).

[Tha11] Bernhard Thalheim. *The Theory of Conceptual Models, the Theory of Conceptual Modelling and Foundations of Conceptual Modelling*, pages 543–577. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[UP16] Peter Uhnák and Robert Pergl. The openponk modeling platform. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*, pages 1–11, 2016.