FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

# Assignment of master's thesis

| | |
|---|---|
| **Title:** | Searching Inside (Onto)UML Structural Conceptual Models |
| **Student:** | Bc. Richard Husár |
| **Supervisor:** | doc. Ing. Robert Pergl, Ph.D. |
| **Study program:** | Informatics |
| **Branch / specialization:** | Knowledge Engineering |
| **Department:** | Department of Applied Mathematics |
| **Validity:** | until the end of summer semester 2021/2022 |

## Instructions

The topic is focused on analysis and implementation of searching inside conceptual models.

1. Acquaint yourself with UML and OntoUML languages and the XMI format for their serialization.
2. Design, implement and test XMI parser for Enterprise Architect/OpenPonk and possibly Visual Paradigm models and storing the models in a suitable knowledge database for their searching and querying.
3. Implement the search/query functionality as a module for the Repocribro model repository.
4. Document and discuss your solution.

–

https://openponk.github.io
https://en.wikipedia.org/wiki/XML_Metadata_Interchange
https://pypi.org/project/repocribro

*Electronically approved by Ing. Karel Klouda, Ph.D. on 5 January 2021 in Prague.*

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

# Searching Inside (Onto)UML Structural Conceptual Models

***Bc. Richard Husár***

Faculty of Information Technology
Supervisor: doc. Ing. Robert Pergl, Ph.D.

May 6, 2021

# Declaration

In Prague on May 6, 2021 …………………

## Citation of this thesis

Husár, Richard. *Searching Inside (Onto)UML Structural Conceptual Models.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

# Abstrakt

Práca sa zaoberá reprezentáciou konceptuálnych modelov v XMI formáte a spôsobom ako v nich jednoducho vyhľadávať potrebné informácie. Súčasťou riešenia tohto problému je správne zvolenie prostriedkov, ako výber dátového uložiska, ktoré bude vyhovovať vyhľadávaniu informácií z XMI formátu. Ďalším kritériom je transformácia XMI modelov do dátového úložiska a to tak, aby boli informácie čo najčitateľnejšie pre užívateľa. Najlepším riešením sa zdá byť použitie grafových databáz, konkrétne Neo4j, pretože reprezentácia konceptuálnych modelov je ľahko transformovateľná do grafov. Na transformovanie modelov z XMI formátu je použitý Python parser, ktorý je spolu s grafovou databázou zakomponovaný do aplikácie Repocribro. Tá slúži ako platforma pre spomínané vyhľadávanie a reprezentovanie výsledkov pre užívateľov.

**Klíčová slova** XMI, Neo4j, konceptuálny model, python parser, Neovis

# Abstract

The thesis discusses ways of representing conceptual models in XMI format for easy searching and querying. The elaboration of the solution to the problem includes choosing an adequate storage medium that can easily represent XMI models. Another requirement is a way of transforming raw XMI models into a more human-readable form inside the storage tool. The most suitable version for storage seems to be the graph database which utilizes its properties such as a representation of data in nodes and relations, which highly resembles conceptual models. Mapping of given models in XMI format is done in a Python parser which is designed to be easily extendable. Everything is integrated into the GitHub repository management tool - Repocribro, which in conjunction with embedded tools to display graph results, serves as a platform for user experience.

**Keywords** XMI, Neo4j, conceptual model, python parser, Neovis

# Contents

# List of Figures

# List of Tables

# Introduction

The concept of sharing knowledge and avoidance of the “reinventing the wheel” method when designing software products is a powerful way to speed up development and deployment. Utility applications that provide this functionality have become quickly integrated inside the developers’ community and it seems that without them, the breakthrough of new ready to use applications would be much slower. The rise of applications such as GitHub or Stackoverflow means that programmers all over the world can come together and each one can contribute to the community with much less effort and more quickly than ever before. Although these applications help with the workflow of programmers and software engineers, there is currently a lack of solutions regarding issues with conceptual models and model designing. This master thesis addresses this issue by creating a software solution that helps designers of contextual models to easily share work with the community, to enable them to collaborate easily and provide tools necessary for searching, querying and simply retrieving the information already done by the modelling community with much bigger comfort than before. The fact that there are multiple modelling tools on the market, each with its caveat and way of representing models makes this task non-trivial. In this thesis, we would like to use the already defined standards for conceptual models to help with designing a scalable and maintainable solution. The thesis aims to implement a parser which in conjunction with a graphical interface, is able to process conceptual models from different kinds of modelling tools. This processed data will be presented to the user and the graphical interface will enable a reasonably easy way of querying the results and retrieving information about the models, mainly information where the user can find the searched model.

In the first chapter, we discuss the current state-of-art of the issue, namely what standards do exist, their structure and what information can be harnessed to the final solution. Also, this chapter introduces the current state of technology, which can be used to produce such systems, for example, graphical databases. To be specific Neo4j. Furthermore, this chapter discusses the integration tools and already working solution, which can be leveraged to simplify the user experience and enact the avoidance of “reinventing the wheel” policy.

The second chapter addresses the various issues and provides a deeper analysis of chosen technologies such as the programming language used or data storage method. Besides that, the chapter debates the differences when it comes to the loosely defined standards regarding the storage of conceptual models in the XMI format.

The last chapter describes the implementation process and goes through the stages of development, from the early mock-up solution to the final solution which includes the integration of the developed parser to the existing system of Repocribro. Additionally, the chapter includes a section that implements testing of the whole project and a section where demonstration what a real use case might look like for the actual user.

Chapter 1

# State-of-the-art

## 1.1 XML and XMI specification

XMI specification describes the representation of objects as the XML elements and attributes. Moreover, it specifies a way of linking objects within the same file or across different files. It is an Object Management Group (OMG) standard designed to exchange metadata through XML format. It is fully described inside the official specification [5]. In this section, some relevant passages for this assignment are presented.

### 1.1.1 Model class representation

As mentioned in the official documentation in section Model class representation: *"Every model class is represented in the schema by an XML element whose name is the class name, as well as a complexType whose name is the class name. The declaration of the type lists the properties of the class. By default, the content models of XML elements corresponding to model classes do not impose an order on the properties. By default, XMI allows you to serialize features using either XML elements or XML attributes; however, XMI allows you to specify how to serialize them if you wish. Composite and multi-valued properties are always serialized using XML elements".* The model class representation is not strictly given and is based on a specific implementation.

## 1.2 XMI model

The three diagrams from XMI specification 1.1 describe XMI model. In the figure 1.1, we can see that XMI class consists of documentation, differences and extensions. The documentation class consists of many fields which help describe the document for non-computational purposes. The extension class contains the metadata for external information. Usually, this is the class where additional information about rendering the graph such as the position of nodes and other parameters are. That information is specific to the model tool in which they were created. The model also contains primitive type DateTime.

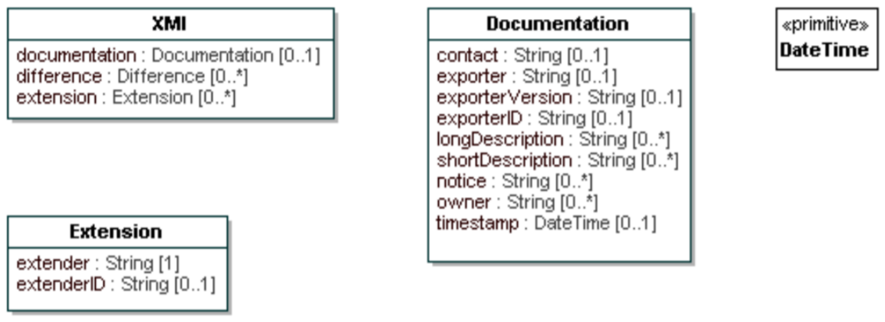

XMI
documentation : Documentation [0..1]
difference : Difference [0..*]
extension : Extension [0..*]
Documentation
contact : String [0..1]
exporter : String [0..1]
exporterVersion : String [0..1]
exporterID : String [0..1]
longDescription : String [0..*]
shortDescription : String [0..*]
notice : String [0..*]
owner : String [0..*]
timestamp : DateTime [0..1]
«primitive»
DateTime
Extension
extender : String [1]
extenderID : String [0..1]

Figure 1.1: XMI model composition

## 1.3 Meta object facility specification (MOF)

*The MetaObject Facility Specification™ (MOF™)is the foundation of OMG's industry-standard environment where models can be exported from one application, imported into another, transported across a network, stored in a repository and then retrieved, rendered into different formats (including XMI™, OMG's XML-based standard format for model transmission and storage), transformed, and used to generate application code. These functions are not restricted to structural models, or even to models defined in UML - behavioral models and data models also participate in this environment, and non-UML modeling languages can partake also, as long as they are MOF-based.*[6]

## 1.4 UML

UML, short for Unified Modeling Language, is a standardized modelling language. The standard consists of an integrated set of diagrams developed to help system and software developers to specify, visualize, construct, and document the artefacts of software systems. Business models and non-software systems can also find value in using this modelling language. The UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997. The UML represents a collection of best engineering practices that have proven successful in the modelling of large and complex systems. The UML proved to be a very important part of developing object-oriented software. The UML consists of mostly graphical notations which express the structure of software projects. Using UML greatly helps with communication, presentation of ideas and quicker development of software and non-software oriented projects.[7][8]

### 1.4.1 UML diagrams

There are several UML diagram types 1.4, which can be generalized in two main categories; structural diagrams and behavioral diagrams [5]. As category names suggest, structural diagrams show basic structure or objects inside the system. On the other hand, behavioural diagrams describe how objects interact with each other to create a functioning system.

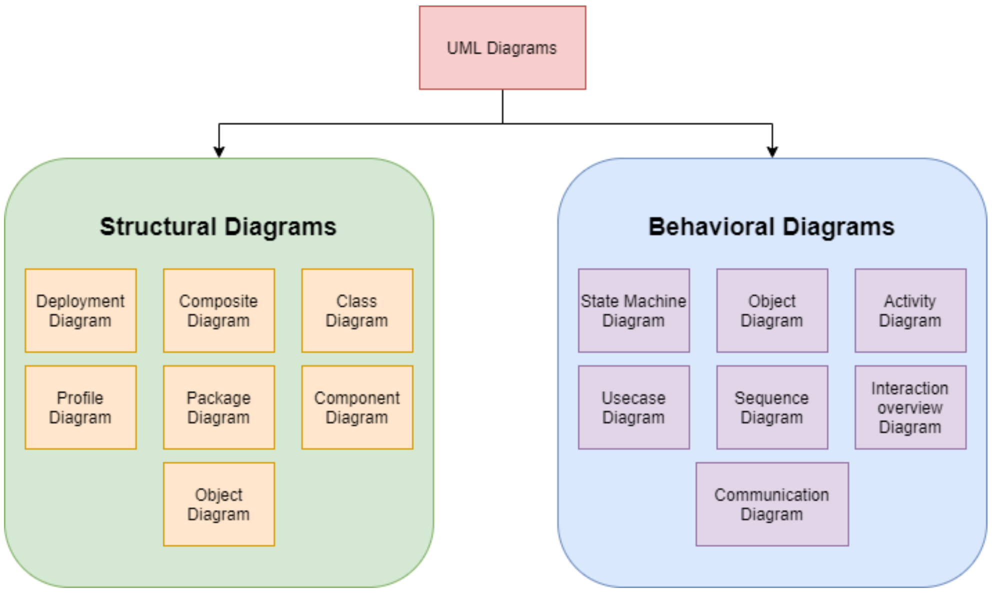


Figure 1.2: UML diagram types

### 1.4.2 Class diagrams

The class diagram is a kind of structural UML diagram and it represents the static view of the application. The main purpose of this diagram is not only to describe, visualizing and documenting the application but also it provides some sort of blueprint for the development process. More specifically, it can be used to construct executable code, because the class diagram contains objects (classes) and their respective attributes and operations. It can also reflect constraints imposed on the system [9].

### 1.4.3 Class representation

Every class in the UML diagram is represented as a rectangle, inside of which is the name of the class. The class can also represent owned attributes and operations. Attributes are typically shown in the section below the name of the class. Attributes are represented at least with a name but also the type of the attribute can be included (aligned to the right). The same rules apply for the operations of the class with the addition of return type.

### 1.4.4 Relations

The UML allows the definition of several relations between classes in the class diagram. Relations join objects which have some interactions between them. Based on the type of interaction, we define different relations.

#### 1.4.4.1 Association

According to the Visual Paradigm documentation [10] associations are defined as: *"If two classes in a model need to communicate with each other, there must be a link between them, and that can be represented by an association (connector). Association can be represented by a line between these classes with an arrow indicating the navigation direction. In case an arrow is on both sides, the association is known as a bidirectional association. We can indicate the multiplicity of an association by adding multiplicity adornments to the line denoting the association."* Another source [11] states that: *"Name of the association can be shown somewhere near the middle of the association line but not too close to any of the ends of the line. Each end of the line could be decorated with the name of the association end."*

**1.4.4.1.1 Binary association** *The Binary association relates two typed instances. It is normally rendered as a solid line connecting two classifiers, or a solid line connecting a single classifier to itself (the two ends are distinct). The line may consist of one or more connected segments.* [11]

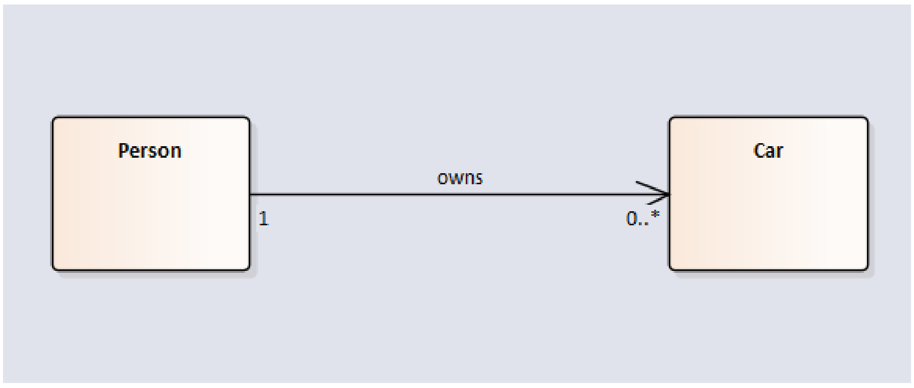


Figure 1.3: Association example

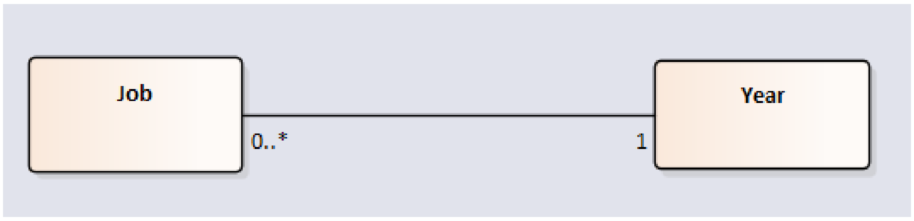


Figure 1.4: Binary association example

#### 1.4.4.1.2 N-ary association

*Any association may be drawn as a diamond (larger than a terminator on a line) with a solid line for each association end connecting the diamond to the classifier that is the end's type. N-ary association with more than two ends can only be drawn this way.* [11]

### 1.4.4.2 Aggregation

As is stated in IBM's documentation [12], aggregation can be defined as follows:

*"In UML models, an aggregation relationship shows a classifier as a part of or subordinate to another classifier. An aggregation is a special type of association in which objects are assembled or configured together to create a more complex object. An aggregation describes a group of objects and how you interact with them. Aggregation protects the integrity of an assembly of objects by defining a single point of control, called the aggregate, in the object that represents the assembly. Aggregation also uses the control object to decide how the assembled objects respond to changes or instructions that might affect the collection. Data flows from the whole classifier, or aggregate, to the part. A part classifier can belong to more than one aggregate classifier and it can exist independently of the aggregate. For example, a Department class can have an aggregation relationship with a Company class, which indicates*

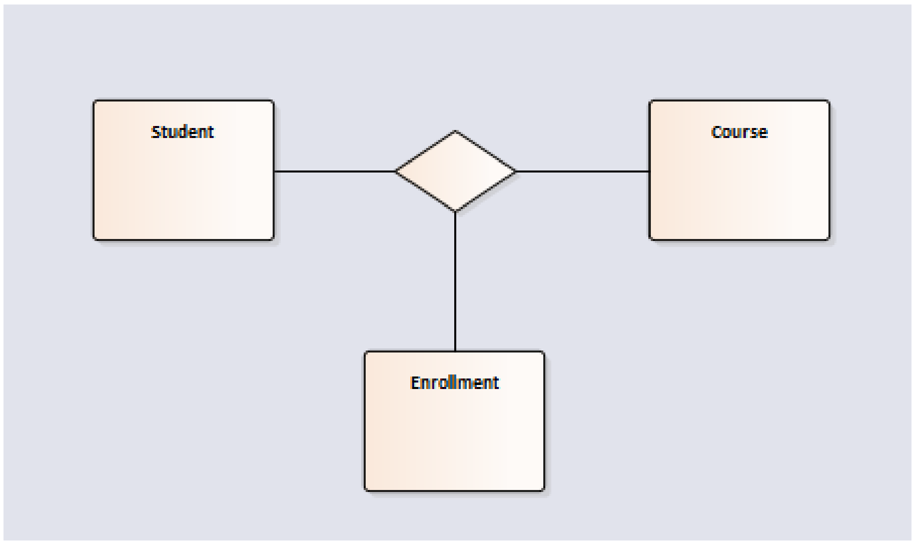


Figure 1.5: N-ary association example

*that the department is part of the company. Aggregations are closely related to compositions. As the following figure 1.6 illustrates, an aggregation association appears as a solid line with an unfilled diamond at the association end, which is connected to the classifier that represents the aggregate. Aggregation relationships do not have to be unidirectional."*

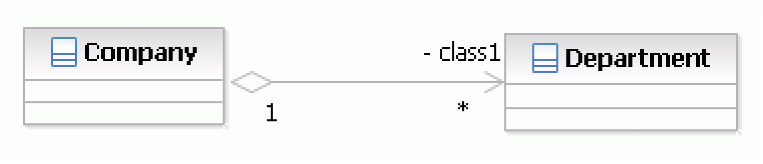


Figure 1.6: Aggregation example [1]

#### 1.4.4.3 Composition

*A composition association relationship represents a whole–part relationship and is a form of aggregation. A composition association relationship specifies that the lifetime of the part classifier is dependent on the lifetime of the whole classifier. In a composition association relationship, data usually flows in only one direction (that is, from the whole classifier to the part classifier). For example, a composition association relationship connects a Student class with a Schedule class, which means that if you remove the student, the schedule is also removed. You can name any association to describe the nature of* *the relationship between the two classifiers; however, names are unnecessary if you use association end names. As the following figure 1.7 illustrates, a composition association relationship appears as a solid line with a filled diamond at the association end, which is connected to the whole, or composite, classifier.*[12]

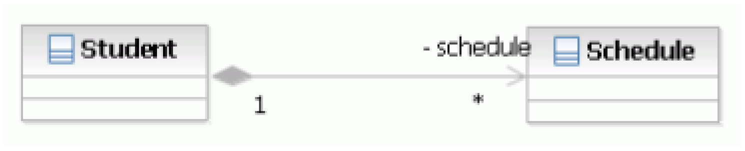


Figure 1.7: Composition example

*The composition and aggregation are two subsets of association. In both of the cases, the object of one class is owned by the object of another class; the only difference is that in composition, the child does not exist independently of its parent, whereas in aggregation, the child is not dependent on its parent i.e., standalone. An aggregation is a special form of association, and composition is the special form of aggregation.*[13]

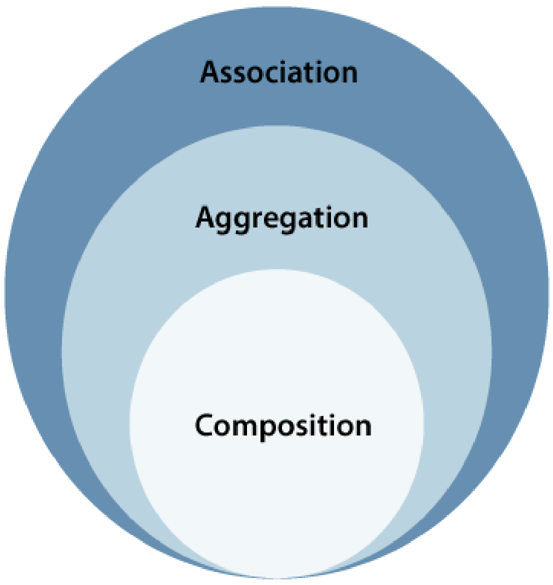


Figure 1.8: UML relations visualization [2]

#### 1.4.4.4 Inheritance / Generalization

*A generalization is a binary taxonomic (i.e. related to classification) directed relationship between a more general classifier (superclass) and a more specific classifier (subclass). Each instance of the specific classifier is also an indirect instance of the general classifier, so that we can say "Patient is a Person", "Savings account is an Account", etc. Because of this, generalization relationship is also informally called "Is A" relationship. Generalization is owned by the specific classifier. A generalization is shown as a line with a hollow triangle as an arrowhead between the symbols representing the involved classifiers. The arrowhead points to the symbol representing the general classifier. This notation is referred to as the "separate target style". The parent model element can have as many children, and also, the child can have one or more parents. But most commonly, it can be seen that there is one parent model element and multiple child model elements. The generalization relationship does not consist of names. The generalization relationship is represented by a solid line with a hollow arrowhead pointing towards the parent model element from the child model element.*[9]

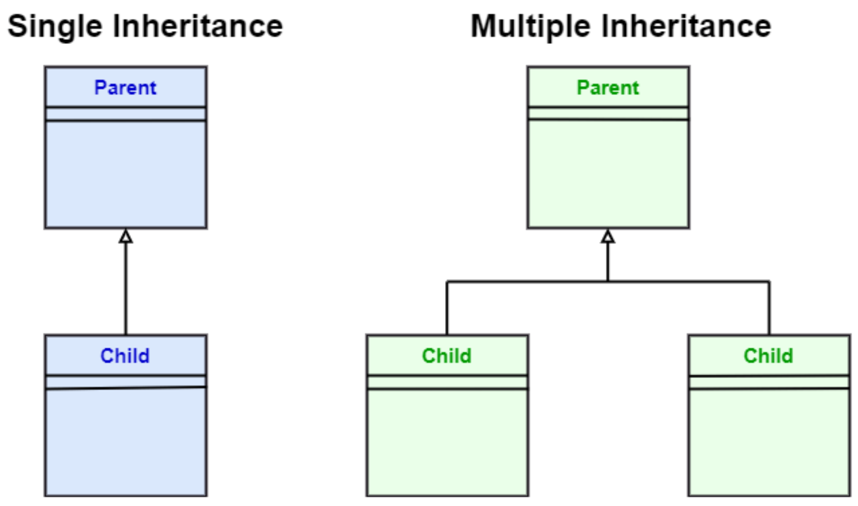


Figure 1.9: UML inheritance diagram [3]

### 1.4.5 Association class

An association may be refined to have its own set of features; that is, features that do not belong to any of the connected classifiers but rather to the association itself. Such an association is called an association class. It is both an association, connecting a set of classifiers and a class, and as such could have features and might be included in other associations.

An association class can be seen as an association that also has class properties, or as a class that also has association properties.

An association class is shown as a class symbol attached to the association path by a dashed line. The association path and the association class symbol represent the same underlying model element, which has a single name. The association name may be placed on the path, in the class symbol, or on both, but they must be the same name.[9]

### 1.4.6 Generalization sets

*Generalization set is a packageable element that allows us to define classification hierarchies by combining some generalizations of a particular general classifier into (sub)sets. Each generalization set may be also associated with a classifier called its powertype.*[9] Each generalization set has two properties - isCovering (complete or incomplete constraint) and isDisjoint (disjoint or overlapping constraint), to clarify what kind of set it is.

*The* isCovering *property of generalization set specifies whether the set of specific classifiers in that generalization set is complete. For the covering (complete) generalization set, every instance of the general classifier is also an instance of (at least) one of the specific classifiers. If the set is not covering (incomplete), there could be some instances of the general classifier that could not be classified as any of the specific classifiers from the generalization set.*[5]

*The* isDisjoint *property specifies whether the specific classifiers of the generalization set may overlap. Generalization set constrained as disjoint has no instance of any specific classifier may also be an instance of another specific classifier (i.e there is no overlapping of classifiers). If generalization set is overlapping, some or all of its specific classifiers could share common instances. By default, in UML 2.0 to UML 2.4.1 generalization set is incomplete, disjoint, while in UML 2.5 default was changed to {incomplete, overlapping}.*[5]

## 1.5 Neo4j

Neo4j is an open-source, NoSQL, native graph database that provides an ACID-compliant transactional backend for applications. The source code, written in Java and Scala is publicly available. Neo4j is referred to as a native graph database because it efficiently implements the property graph model down to the storage level. This means that the data is stored exactly as a model written on the whiteboard, and the database uses pointers to navigate and traverse the graph. In contrast to graph processing or in-memory libraries, Neo4j also provides full database characteristics, including ACID transaction compliance, cluster support, and runtime failover - making it suitable to use graphs for data in production scenarios.[14]

Neo4j is a graph database, thus it stores not only data but also connections (relationships) between them. This fact makes graph databases an ideal candidate for this task because conceptual models consist of objects and relationships between them. We can use this resemblance further, to represent conceptual models as nodes and relationships in a graph database.

#### Features and advantages

**Cypher** is a declarative query language similar to SQL, but optimized for graphs. This feature will be further discussed in section 1.6.3.

**Constant time traversals** in big graphs for both depth and breadth due to efficient representation of nodes and relationships. Enables scale-up to billions of nodes on moderate hardware.

**Flexible** property graph schema that can adapt over time, making it possible to materialize and add new relationships later to shortcut and speed up the domain data when the business needs change.

**Drivers** for popular programming languages, including Java, JavaScript, .NET, Python, and many more. Python driver will be discussed further in the section.

## 1.6 Structure

### 1.6.1 Nodes

Nodes are the entities in the graph. They can hold any number of attributes (key-value pairs) called properties. Nodes can be tagged with labels, representing their different roles in the domain. Node labels may also serve to attach metadata (such as index or constraint information) to certain nodes.

### 1.6.2 Relationships

Relationships provide directed, named, semantically relevant connections between two node entities (e.g. Employee WORKS_FOR Company). A relationship always has a direction, a type, a start node, and an end node. Like nodes, relationships can also have properties. In most cases, relationships have quantitative properties, such as weights, costs, distances, ratings, time intervals, or strengths. Due to the efficient way relationships are stored, two nodes can share any number or type of relationships without sacrificing performance. Although they are stored in a specific direction, relationships can always be navigated efficiently in either direction.

### 1.6.3 Cypher

Cypher is Neo4j's graph query language that allows users to store and retrieve data from the graph database. The syntax is based on ASCII art, making it easy to learn and understand. Cypher provides many features just like other query languages, for example, basic CRUD operations, filtering results, built-in aggregation functions, support for database native types such as DATE or DATETIME, use of subqueries and many others.[15]

#### Custom functions and procedures

A special feature that is important for this task is the utilization and creation of custom functions and procedures. They will allow developers to prepare custom functions and procedures needed for easy manipulation and information retrieval from graph database which will be storing data and metadata about conceptual models. Custom functions and procedures can be added to the Neo4j database as plugins. Plugins are written in Java and stored as .jar files.

### 1.6.4 Neo4j from python

Neo4j offers its official python driver to easily connect and manage Neo4j databases from within python code. It makes use of binary protocol and is very light and simple. There are lots of extensions built on top of this driver. For example, when working with python 2, neo4j offers py2neo which is a small intuitive tool kit for working with neo4j. Another viable option is Neomodel. The Neomodel was the product of choice for this task because it offers easy API for communication with Neo4j itself, as well as other features further discussed in the section below.

#### 1.6.4.1 Neomodel

The Neomodel is a kind of Object Graph Mapper, which is built on top of the mentioned Neo4j python driver. It provides node definitions and a powerful query API. It is completely thread safe and has full transaction support.[16]

### Structure

Nodes of the model are defined in the same way as classes are in Python, with the only difference that data members of those classes, that are planned to be stored to the database, must be defined as Neomodel property objects.

### Property objects and types

The Neomodel includes various property objects which can represent nodes. The most elementary and the most basic is StructuredNode class. It represents a basic node that can be enriched with properties such as StringProperty, to represent string value, IntegerProperty to represent integer value or more complex properties such as JSONProperty to include JSON structured data in the node attribute. Another object which can represent a node is called *SemiStructuredNode.* This type of object allows storing properties on the node that are not specified in its definition. Property addition can be realized dynamically right when inserting and saving to the database. Possible conflicts in property names are signaled by special *DeflateConflict* exception.

### Property types

As mentioned above, the Neomodel facilitates a wide range of property types. From simple types used in the programming world such as string, integer boolean, array and many others, to more complex ones satisfying needs of simply and efficiently representing real-life properties such as geolocation with PointProperty.

### Default values

Default values can be defined for any property. Value can be represented as a callable or function.

```
my_id = StringProperty(unique_index=True, default=uuid4)
```

### Mandatory / Optional properties

In the definition of a property, one can specify if the given property is mandatory or optional. This behaviour is achieved with the parameter required. Setting this property to true, means the property is mandatory and cannot have a default value.

### 1.6.5 Embeddable tools with built-in Neo4j connections

Embedding the visualization within the application allows the developer to create applications that include the visualization as part of the user interface. This also means that the developer can write other components and customize the application experience and other components involved in the application to the exact business requirements. [17]

The disadvantage with embedding is, that the libraries implementing them, do not always support complex and heavy queries. Also, these visualizations are usually connected directly to the database which might not be the desired architecture.

#### 1.6.5.1 Neovis.js

The paper on visualizations via Neo4j [18] refers to the neovis.js as follows: *"Neovis.js and is used for creating JavaScript based graph visualizations that are embedded in a web app. It uses the JavaScript Neo4j driver to connect to and fetch data from Neo4j and a JavaScript library for visualization called vis.js for rendering graph visualizations. Neovis.js can also leverage the results of graph algorithms like PageRank and community detection for styling the visualization by binding property values to visual components."*

## 1.7 Integration tools

### 1.7.1 Repocribro

Repocribro is a web application allowing users to register their GitHub repository so they can be managed, searched, browsed, tested, etc. (depends on used extensions) with the site. The main idea is to provide a simple but powerful modular tool for building groups of GitHub repositories that are developed by different users and organizations. [19]

### 1.7.2 Docker

Docker is an open platform for developing, shipping, and running applications. Docker enables us to separate our applications from the infrastructure so we can deliver software quickly. With Docker, we can manage your infrastructure in the same ways we manage our applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, we can significantly reduce the delay between writing code and running it in production [20].

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow us to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so we do not need to rely on what is currently installed on the host. We can easily share containers while you work and be sure that everyone we share with, gets the same container that works in the same way.

### 1.7.3 Celery

Celery is a simple, flexible, and reliable distributed system to process vast amounts of messages while providing operations with the tools required to maintain such a system.

It's a task queue with a focus on real-time processing, while also supporting task scheduling.[21]

Celery provides an easy way of extending the Repocribro app without interrupting the user experience by waiting while all models are parsed into the storage.

#### Task queues

Task queues are used as a mechanism to distribute work across threads or machines. A task queue's input is a unit of work called a task. Dedicated worker processes constantly monitor task queues for new work to perform. Celery communicates via messages, usually using a broker to mediate between clients and workers. To initiate a task the client adds a message to the queue, the broker then delivers that message to a worker. A Celery system can consist of multiple workers and brokers, giving way to high availability and horizontal scaling. Celery is written in Python, but the protocol can be implemented in any language. In addition to Python, there's node-celery and node-celery-ts for Node.js, and a PHP client. Language interoperability can also be achieved by exposing an HTTP endpoint and having a task that requests it (webhooks).[22]

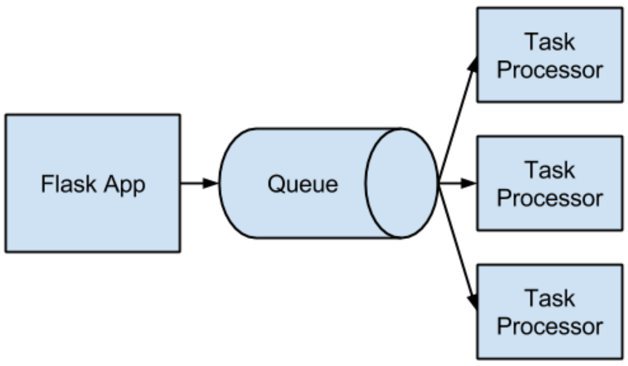


Figure 1.10: Celery task queue example

Chapter 2

# Analysis and design

The main goal of this thesis is to explore the possibility to effectively search and retrieve information from multiple conceptual models. The conceptual models could come from various sources and can be of different types, such as activity diagram, class diagram or others. The main concepts thus will be; standardization of models, parsing, storage and searching.

## 2.1 Standardization of models

The models are usually developed and designed in graphical tools such as Enterprise Architect, Visual Paradigm or StarUML. These tools have options to export models in XMI format. As mentioned in the review section, XMI consists of three parts; documentation, extensions and differences. To create an XMI file complaint with OMG XMI specification, different tool distributors can use different approaches. This means that XMI files representing the same model, but modelled in different modelling tools, would have slightly different XMI files as output. Sometimes the differences are small and cause no trouble with the actual parsing of the model because only the section where modelling tool-specific metadata differ. However, usually, the differences are also present in the section where the model itself is stored. This means that just the reliance on the OMG XMI specification does not guarantee unified rules when parsing XMI files from different distributors. Parsing of XMI files thus means, that there must be some kind of mechanism identifying format specific for a given modelling tool program.

### Specific XMI model representation differences

Example of interpretation of the same class diagram in Openponk and Enterprise architect

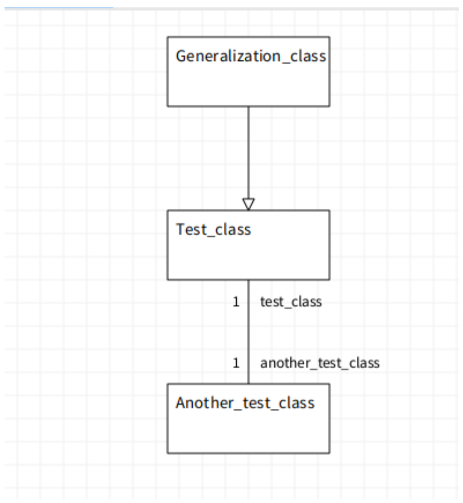


Figure 2.1: Openponk model example

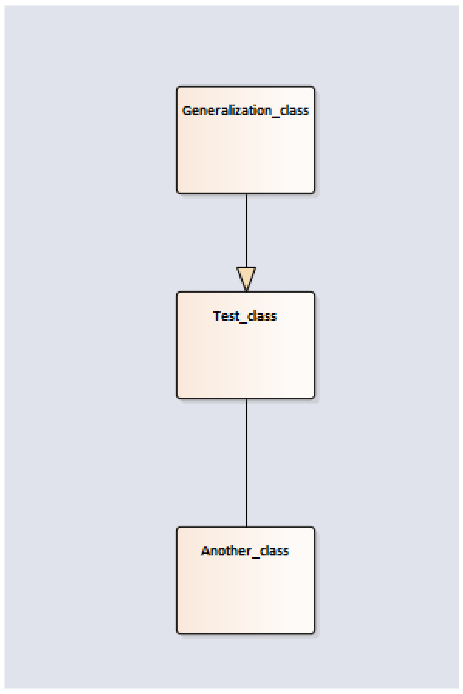


Figure 2.2: Enterprise Architect model example

Listing 2.1: Openponk XMI example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmlns:uml="UML" xmlns:xmi="XMI">
 <uml:Model xmi:type="uml:Model" xmi:id="1b1ce627..." name="New Project">
  <packagedElement xmi:type="uml:Class" xmi:id="6310e627..."
    name="Test_class"/>
  <packagedElement xmi:type="uml:Class" xmi:id="6310e627..."
    name="Another_test_class"/>
  <packagedElement xmi:type="uml:Association" xmi:id="6310e627..."
   memberEnd="a32fe627... 8b33e627...">
        <ownedEnd xmi:type="uml:Property" xmi:id="a32fe627..."
        association="6310e627..." name="test_class" type="6310e627..."/>
        <ownedEnd xmi:type="uml:Property" xmi:id="8b33e627..."
        association="6310e627..."
            name="another_test_class" type="6310e627..."/>
   </packagedElement>
   <packagedElement xmi:type="uml:Class" xmi:id="6310e627..."
   name="Generalization_class">
        <generalization xmi:type="uml:Generalization" xmi:id="7b0ce627..."
        general="6310e627..."/>
 </packagedElement>
</uml:Model>
</xmi:XMI>
```

Listing 2.2: Enterprise Architect XMI example

```
<?xml version="1.0" encoding="windows-1252"?>
<xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.1"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
 <xmi:Documentation exporter="Enterprise Architect".../>
 <uml:Model xmi:type="uml:Model" name="EA_Model" visibility="public">
  <packagedElement xmi:type="uml:Package" xmi:id="EAPK..."
  name="Package1"
  visibility="public">
   <packagedElement xmi:type="uml:Class" xmi:id="EAID..."
   name="Another_class" visibility="public"/>
   <packagedElement xmi:type="uml:Association" xmi:id="EAID..."
   visibility="public">
     <memberEnd xmi:idref="EAID_dst..."/>
     <memberEnd xmi:idref="EAID_src..."/>
       <ownedEnd xmi:type="uml:Property" xmi:id="EAID_src..."
       visibility="public"
       association="EAID..." isStatic="false" isReadOnly="false"
       isDerived="false"
       isOrdered="false" isUnique="true" isDerivedUnion="false"...>
        <type xmi:idref="EAID..."/>
       </ownedEnd>
       <ownedEnd xmi:type="uml:Property" xmi:id="EAID_dst..." ..."
        association="EAID..." isStatic="false" isReadOnly="false"
        ...
        isDerivedUnion="false" aggregation="none">
        <type xmi:idref="EAID..."/>
       </ownedEnd>
   </packagedElement>
   <packagedElement xmi:type="uml:Class"
   xmi:id="EAID..." name="Generalization_class"
   visibility="public">
     <generalization xmi:type="uml:Generalization" xmi:id="EAID..."
     general="EAID..."/>
   </packagedElement>
   <packagedElement xmi:type="uml:Class" xmi:id="EAID..."
     name="Test_class" visibility="public"/>
   </packagedElement>
  </uml:Model>
  <xmi:Extension extender="Enterprise Architect" extenderID="6.5">
    ...
  </xmi:Extension>
</xmi:XMI>
```

As seen from figures and listings 2.1, 2.1, 2.2 and 2.2, ignoring the extensions section (Openponk format does not have one), both formats describe the same model, but are different in the actual XML representation of the model. In this small example, only a few differences are seen, such as usage of memberEnds and ownEnds. In Openponk format, memberEnds and ownEnds - references to other objects involved in relation - are included in the given relation as attributes of XML element (uml:Association in this case). On the other hand, Enterprise architect uses memberEnds and ownEnds as separate XML elements inside the parent element of relation.

## 2.2 Storage

The greatest challenge of this thesis is to accurately and efficiently store metamodel provided in the form of XMI files which could be from different programs. The solution should bring easy retrieval and possibly provide some sort of querying for stored models. Different approaches come to mind in regards to this problem. Usage of traditional relational databases, or usage of more recent technology in the context of storage, like No-SQL solutions; document-based or graph-based.

### 2.2.1 Relational databases

Relational databases would provide well-known structures such as tables and relations between them. Provided with SQL, querying would be simple enough, that no other simplification would be needed. However, the expression of the query could be tricky even with simple basic use cases, because models can contain transitional relations and querying would have to be complex and use various levels of recursion. The model itself can be represented with a table containing nodes or objects and a table containing relations between them.

### 2.2.2 Document based databases

Document-based databases such as Elasticsearch or MongoDB provide another way of storing conceptual models. Currently, there are lots of document-oriented databases on the market, and they use different types of formats in which documents are stored. Elasticsearch or MongoDB use some variation of JSON which would mean that in addition to parsing the conceptual model from XMI, we would have to create an additional structure to encode it into JSON, which would prove ineffective. Another format for storing is of course XML, where main database providers such as Oracle, Microsoft SQL Server or PostgreSQL each have their XML support.

According to Matthias [23]: *Native XML databases are especially tailored for working with XML data. As managing XML as large strings would be inefficient, and due to the hierarchical nature of XML, custom optimized data structures are used for storage and querying. This usually increases performance both in terms of read-only queries and updates.*

### 2.2.3 Elasticsearch as storage

Elasticsearch is a distributed, RESTful search and analytics engine. It is suited for working with and searching in large data files, for example, logs. As the heart of the Elastic Stack, it centrally stores data for fast search, fine-tuned relevancy, and powerful analytics that can be scaled easily. [24]

If Elasticsearch was used as a base component for the storage of metamodels, we would have to create mapping which would translate metamodels into JSON documents. These documents would be indexed and stored in Elasticsearch. The advantage of indexed data is very quick retrieval when the data is queried. On the other hand, data is stored denormalized. This means that pieces of information are stored in the system redundantly, therefore it creates more requirements on disc space. Also, problems could emerge when updating data, which is present in the system multiple times.

Overall Elasticsearch is best suited for consuming a large amount of data in real-time and also to quickly search and retrieve relevant documents. This deems Elasticsearch to be a powerful tool but does not quite meet the requirements for this task, as it has different demands, such as easy querying and mapping of the XMI files.

### 2.2.4 Graph database as storage

Graph databases provide a convenient way to store data that resemble graphs in the real world. Metamodels can be easily transformed to graph notation of nodes and their relationships. Every object in a metamodel can be interpreted as a node in a graph and every attribute or characteristics can be directly stored inside this node. Similarly, the relationships between objects also can be labelled and can store additional information about the ongoing relationship between given objects. Currently, there are few popular graph-oriented databases in the database ecosystem. Cassandra, Titan, Dgraph or Neo4j, every one of them work on a similar principle, but each one of them has its pluses and minuses.

#### 2.2.4.1 Neo4j

Neo4j offers several advantages when it comes to storing conceptual models. Easy to use and relatively well documented APIs and drivers in regards to python connection to the parsing part of the conceptual models. Another plus is graphical UI and web browser which will provide a solution to the visualization part of this problem. A key aspect of choosing Neo4j over the other solutions is the usage of the query language Cypher. Cypher will in conjunction with the earlier mentioned web browser greatly reduce the need to design and implement a new way of visualizing and presenting results to the end-users. Cypher was designed to resemble SQL and thus is very easy to learn even for non-tech users. There is also the possibility of creating stored procedures where we can prepare the most useful commands and queries.

This use case requires the usage of a powerful tool, which can process queries with the possibility of hundreds, thousands or even millions of nodes and relationships with them. Graph databases and especially Neo4j is smart choice to use for this task. For illustration, comparison between relational database (MySQL) and graph database (Neo4j) was conducted on 1,000,000 users [25]. The objective was to find connections between friends in the social network. Query times for both databases are presented in the table 2.1.

| Depth | Execution Time – MySQL | Execution Time –Neo4j |
|---|---|---|
| 2 | 0.016 | 0.010 |
| 3 | 30.267 | 0.168 |
| 4 | 1,543.505 | 1.359 |
| 5 | Not Finished in 1 Hour | 2.132 |

Table 2.1: Table with execution time comparison betweeen Noe4j and MySQL

For the simple friends of friends query, Neo4j is 60% faster than MySQL. For friends of friends of friends, Neo is 180 times faster. And for the depth four queries, Neo4j is 1,135 times faster. And MySQL just chokes on the depth 5 queries. The results are dramatic.

Another factor when choosing neo4j was licencing. Neo4j project is fully open-source and licensed and distributed under GPL v3. Neo4j offers many commercial licensing options, outlined above: both paid and free, including free licenses for development, startup, and academic-educational uses and of course evaluation.[26]

Neo4j is overall considered as the number one graph database based on the ranking from db-engines.com. However, it has its drawbacks. Neo4j does not support multi-graphs as data structures, which can be a limiting factor when encoding conceptual models into it. But overall Neo4j is the well-rounded solution when it comes to the storing part of parsed conceptual models.

## 2.3 Model parsing

The conceptual models can come in various forms and although they all will be compliant with XMI international standard, the actual form and representation of XML file structure will slightly differ. This section will be covering the analysis of possible solutions in regards to parsing conceptual models.

When it comes to parsing files, there are several programming languages for selection. Among many, Java, Python, C# or Perl. My choice for this task was Python because it offers great results in comparison with how easy it is. According to a paper from the Computer Science department, Zakho University, Kurdistan, Iraq [4] where a comparison between major programming languages when parsing XML files, was done, with regards to speed, memory usage, CPU consumption or lines of code needed.

Python came with very good results. “It was the second faster one to parse the XML file after C# on both operating systems. Moreover, Python was the third better one in terms of less memory usage by consuming 4.6 gigabytes on Linux and 6 gigabytes on Windows. As other languages, Python CPU time consumption was very close to other languages. It also did not need a lot of lines to do the task and the lines number was the same of Perl lines number.“

| | **Time** | | **Memory MB** | | **CPU** | | **Line number** | |
|---|---|---|---|---|---|---|---|---|
| **Pro.Lang** | **Linux** | **Windows** | **Linux** | **Windows** | **Linux** | **Windows** | **Linux** | **Windows** |
| **Perl** | 3.42min | 4.25min | 13.9 | 12 | 25% | 29% | 46 | 46 |
| **Python** | 26 sec | 34 sec | 4.6 | 6 | 25% | 29% | 46 | 46 |
| **PHP** | 1.35 min | 0.8min | 4.3 | 5.9 | 20% | 29% | 66 | 66 |
| **Java** | 40 sec | 12.5 | 57 | 11.2 | 44% | 29% | 108 | 108 |
| **C#** | 19 sec | 7 sec | 15.2 | 2.8 | 25% | 24% | 56 | 56 |
| **C++** | 38 58C | 57 sec | 0.37 | 1 | 25% | 29% | 109 | 109 |

Table 2.2: Comparison table [4]

### 2.3.1 XMI python solutions

There are several tools for parsing XML files in python, such as The ElementTree XML API or lxml - XML and HTML with Python. However, to the author’s knowledge, there are currently no good modules for parsing files in XMI. The xmiparser 1.5 exists, but it was last updated in 2010. Also, there is little to no documentation to this module and if it would have been used, almost all of the functionality would be considered as a black box with a very hard way of customization or modification of behaviour. There are also other modules such as PyXMI, but every module found, solves only one specific implementation of XMI from a specific modelling tool. PyXMI for example parses files from a modelling tool called Poseidon. This leaves no other option, but to develop an own parser that would parse XMI files as needed, and connect it to the Neo4j storage through the Neo4j Python driver. Advantages

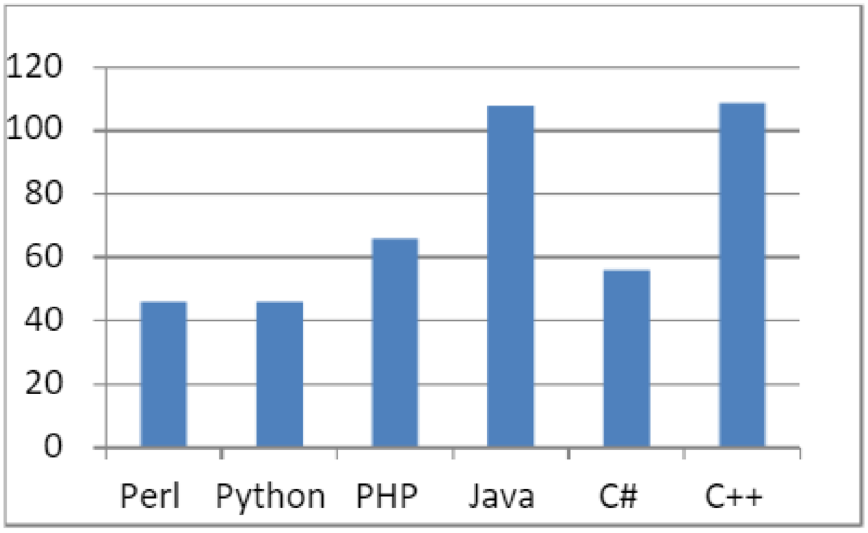


Figure 2.3: The number of lines needed to parse XML file

of developing own parsers are that internal parts of parsing will be well understood and can be easily changed, modified or extended. Another advantage is that it will be much easier to connect the storage and also other programs using this parser which are also written in python, for example, Repocribro. The disadvantage would be of course the time consumption of development itself.

Chapter 3

# Realisation

The basic outlook on the solution is to create a system of applications - python parser, Neo4j and visualization apps using Neovis or default neo4j browser. The whole system would be contained in docker and individual components would communicate with each other and with the rest of the world. On the input, there will be an XMI file that will be parsed and stored into Neo4j. Users can examine the results in one of the possible visualization tools which will be implemented using the Neovis component.

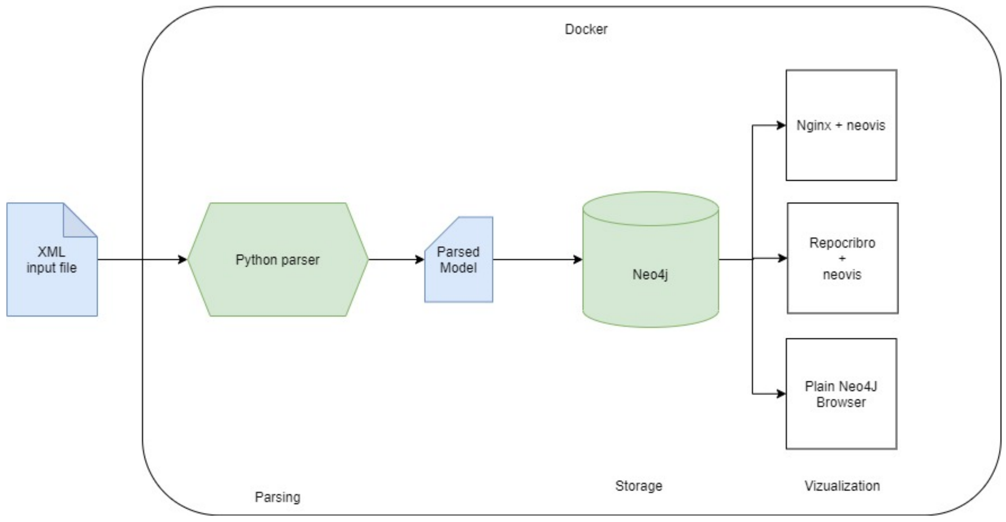


Figure 3.1: High level solution overview

## 3.1 Python parser

As a quick proof of concept, that everything done in the analysis part can be practically done using just the python and neo4j without bigger problems, I designed a simple parser and connected it to the Neo4j storage. The parser contains a model class, which when instantiated, takes an XMI file as input and loads all of its components into data structures. This parser counts only with class diagrams on the input. The model class needs to store all semantically important information. This information is stored in dictionaries and arrays. Which information is semantically important, is determined from the XMI itself. Initial temporary simplification to limit input only on class diagrams allowed me to prepare a list of objects (python classes) that will be parsed from the input XMI file.

- classes - represent objects in XMI file which are labeled as *xmi:type="uml:Class"*. The python class representing the UML class object contains information that needs to be persisted. Id, to easily track and query graphs and also to link with other objects using relations (association, generalization...). Class attributes and methods are not parsed as it is unnecessary for the proof of concept.

- associations - represent relations between classes. From the XMI file they are recognized as *xmi:type="uml:Association"*. As they can be bidirectional or unidirectional, there is a need to look up which node or class is on which side of the relation. Neo4j does not support unidirectional relations and all relations are treated as bidirectional. Therefore I created custom properties for nodes; *src_node* and *dest_node*. Src_node represents the source of the association and dest_node represents the destination of the association. When the association is bidirectional or the direction is not specified, the parser does not recognize this information. This means that source and destination are taken randomly, as it does not affect the structure of the model in a significant way, because for searching purposes the Neo4j treats connections as bidirectional anyway. Associations contain additional information about the multiplicity, label or ID which are also stored.

- generalizations - represent another type of relation and store similar information as associations. Generalizations can only be unidirectional and thus src_node and dest_node is implemented here, similarly as with associations. They are parsed based on the *"uml:Generalization"* type from the XMI file.

- generalization sets - represents objects in XMI file which are labeled as *uml:GeneralizationSet*. This type of object is not connected to any other node in the model as it is some kind of additional information regarding generalizations. The generalization set contains two boolean attributes isCovering and isDisjoint.

- enumerations - represent objects which store a set of values that can given type acquire. They are labeled as a *uml:Enumeration* XMI type. Parsing of this type of object is straight forward and only requires the creation of the Enumeration object and a list of values to be stored.
- class attributes - are embodied by a node and not as an internal property of a class node. This decision comes from the fact that attributes can be more easily queried, filtered and visualized when they are represented by a standalone node. They are identified as a *uml:Property* type inside of a class in the XMI file.
- association nodes - represent nodes associations have more than one connection or join more than two classifiers. They are not specified in the XMI file. The creation of an association node is determined programmatically by simply counting the connections of the association. The section 3.5 specifically discuses the association nodes in detail.
- class types and association types - denote the stereotypes which can classes and associations acquire. They are not included in the XMI specification and parsing is highly dependent on the source designing tool of the exported model. In case of the Openponk for instance, the stereotypes are present on their own as a `<OntoUml:type>` elements inside the parent `<uml:Model>` element

### Model class

The model class is in this case responsible for parsing. The solution in this case of the early stage counts only with class diagrams and two XMI formats (Enterprise architect and Openponk), which differ in a few small instances. Basic graphs were created inside model designing tools and then they were parsed and stored to Neo4j.The graphs contained objects from which the class diagram consists to test the functionality and to design generalized parsing methods.

The model class aims to restructure the XMI file given as an input to the graph database notation; nodes and relationships, no matter what kind of conceptual model is inside.

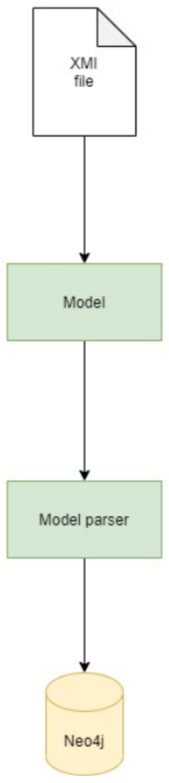


Figure 3.2: Initial design of python parser

### Node

Node class represents a general object inside a graph database which is called a node. To abstract from specific objects inside various conceptual models - classes, packages, activities and many more, I had to come up with a universal definition, which would be easy and simple to implement and also able to extend the other types of conceptual models. This means that whether inserting a class object or package object or another object into the Node class, one will be able to store it with everything needed later inside the graph database. For this purpose several attributes are present; ID, name, type, and dictionary of custom attributes. The first three are straightforward as they specify the identity of the object, and the last one, attributes are implemented as a dictionary. This means that the developer working on the extension can fill the values for the specific object which is being stored, independently. All of the contents of the attributes dictionary will be propagated to the graph database and querying based on its contents will also be enabled.

One of the main use cases of this project is to query conceptual models and retrieve information about them. The most important information to retrieve is the origin of the conceptual model and/or a path where to find the original model. For this reason, I introduced *Base node* which is a base class from which every Node is derived. This class contains metadata about the model such as the name of the model, URL path to the original model XMI or other relevant information. This information is passed through the parser while parsing the file and can be fully customized to include other relevant metadata about the model.

### Relation

Relation class represents a one-way relation between two nodes. Relation does not place constraints on the types of objects which can be connected. Similarly, as with the Node class, the Relation class also needs to be generalized for all possible connections. Attributes required to create relation objects are ID, name, src_node, dest_node and relation_type. ID is required for identification inside neo4j. Name is used as a label displayed inside Neo4j. Src_node and dest_node stand for source node and destination node respectively. The source node is the node from which the relation flows to the destination node. In Neo4j all relations are bidirectional, which means if a connection exists, querying is possible from both ends of the connection. But on the other hand, UML models also specify some relations which are not bidirectional (generalization, associations...) and thus the need to specify direction was required. However, this source-destination connection can only be visible through these attributes inside Neo4j. Querying based on connection directions thus must consider these attributes when designing queries.

## 3.2 Class parsing

The class parsing consists of calling two methods. First parse_classes() 3.1 which does not take any input parameter. This class finds all classes inside the XMI file using the python module *lxml*. XML elements that represent classes are returned and stored in the list. This list is subsequently iterated and every individual class is then parsed through another method called parse_class(). This method takes an XML element representing class as an input parameter and automatically stores the parsed class into the list of classes.

Listing 3.1: Class parsing

```
def parse_classes(self):
    classes = self.model.findall(
    './/packagedElement[@xmi:type="uml:Class"]', self.namespaces)
    for c in classes:
        self.parse_class(c)
```

The parse class method firstly parses all attributes which are contained inside. Then parses id and creates an object representing class as a node inside Neo4j. This object is then appended to the general list of nodes which will later be inserted into Neo4j.

There are instances where a class contains references to another class or generalization. References to another class happen when class is a descendant in the inheritance hierarchy or specification of the parent class. There can be a chain of nested classes with each one will be more specific than the parent. The parsing of this structure highly resembles the parsing of an n-nary tree and recursion is an easy way to iterate over it. If the currently parsed node contains nested classes, the function iterates over them and parses them by calling itself (parse_class) on those classes again. This ensures that no matter how long the chain of specification and inheritance in the model is, the parser correctly parses all of them.

Lastly function checks on the generalizations contained inside the class. Process of parsing those is described in section explaining the function *parse_generalization*. However the main principle is the same as with parsing class attributes.

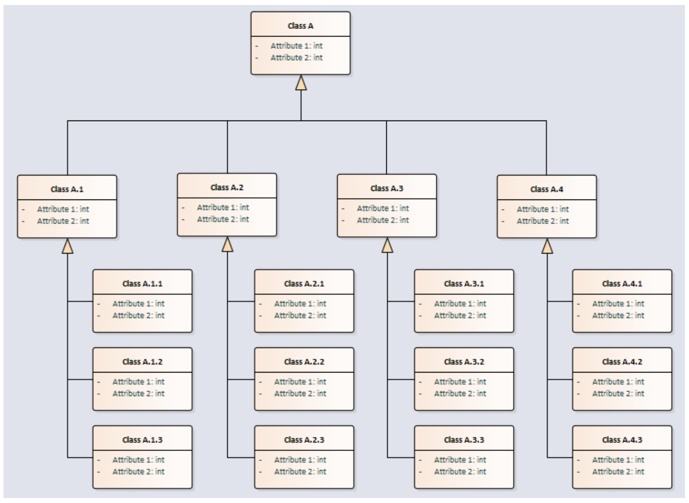


Figure 3.3: Example of a hierarchical class parsing - EA model

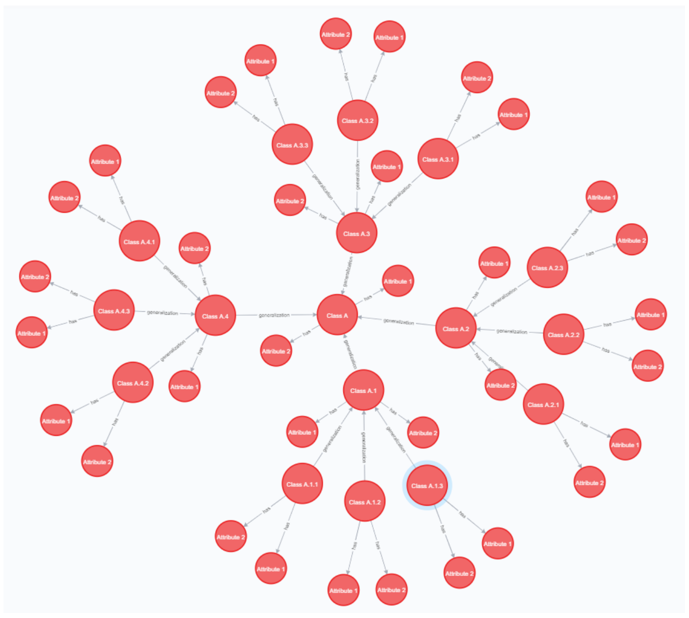


Figure 3.4: Example of a hierarchical class parsing - Parsed representation in Neo4j

Listing 3.2: Class parsing 2

```
def parse_class(self, c):
    parsed_attributes = self.parse_attributes(c)
    #new node
    node_id = c.attrib["{" + self.namespaces['xmi'] + "}" + "id"]
    n = Node(c.attrib["name"], node_id, "uml:Class",
    parsed_attributes)
    self.nodes.append(n)
    #parse sub classes, if present
    nestedClassifiers = c.findall('nestedClassifier',
    namespaces=self.namespaces)
    for nc in nestedClassifiers:
        if self.type_attrib in nc.attrib
        and nc.attrib[self.type_attrib] == "uml:Class":
            self.parse_class(nc)
    #parse generalization, if present
    generalization = c.find('generalization',
    namespaces=self.namespaces)
    if generalization is not None:
        self.parse_generalization(generalization, node_id)
```

## 3.3 Associations parsing

Function parse_associations finds all `packedElements` with type attribute set to `uml:Association` and then iterates over them and parses them one by one.

The function responsible for parsing associations `parse_association` takes one argument, association to be parsed. As mentioned in earlier passages, association always has a source and destination and finding those is the primary task of this function. Every modelling tool has its specific way of defining references to the source and destination of the association and implementation can vary even though the model is fully compliant with UML and XMI specification. This first proof-of-concept-like solution only considers format from Enterprise architect. This specific format has ownedEnds and memberEnds elements for further identification and references to the source and destination nodes.

**MemberEnd** MemberEnd as cited in UML superstructure specification [5]; *"Each end represents participation of instances of the classifier connected to the end in links of the association"*. This element points to the elements which are part of the association, inside ownedEnd.

**OwnedEnd** OwnedEnds are ends owned by the association itself. This element specifies the reference id to the classifier which is connected by the association [5].

#### 3.3.0.1 Parsing different association types

In figure 3.5 are four possible association connections with regards to direction. The association connecting `Class1` and `Class2` does not have a specified direction and it is seen that no arrow points to any of the joining class. `Class3` is joined to the `Class4` by a bi-directional connection. In UML it represents the fact that both classes "know of each other" or have reference to one another. The last two cases are variations of unidirectional connection where only one class has reference to the other class. For parsing XMI and representing given connections in Neo4j, there is a need to look at the structure of exported XMI files generated by the Enterprise Architect.

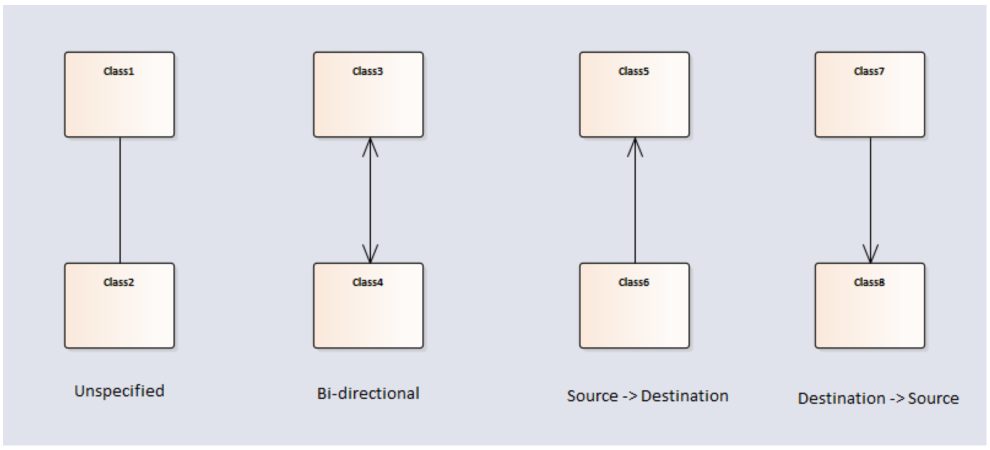
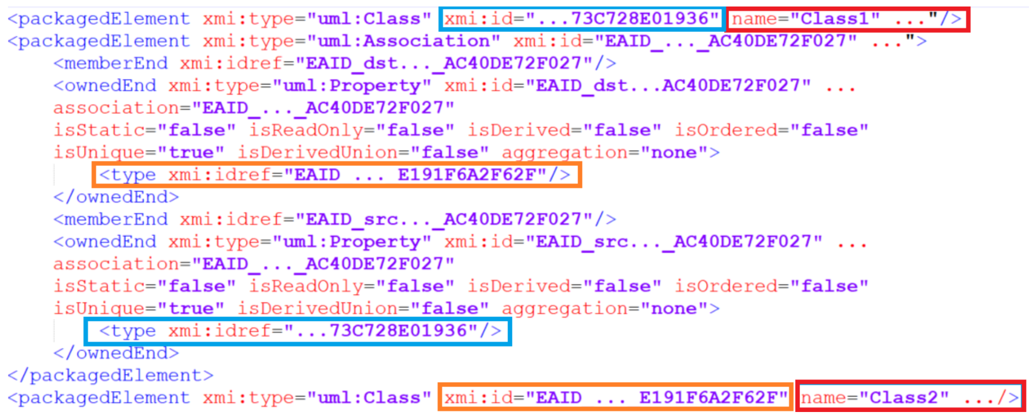


Figure 3.5: Association examples in XMI and EA

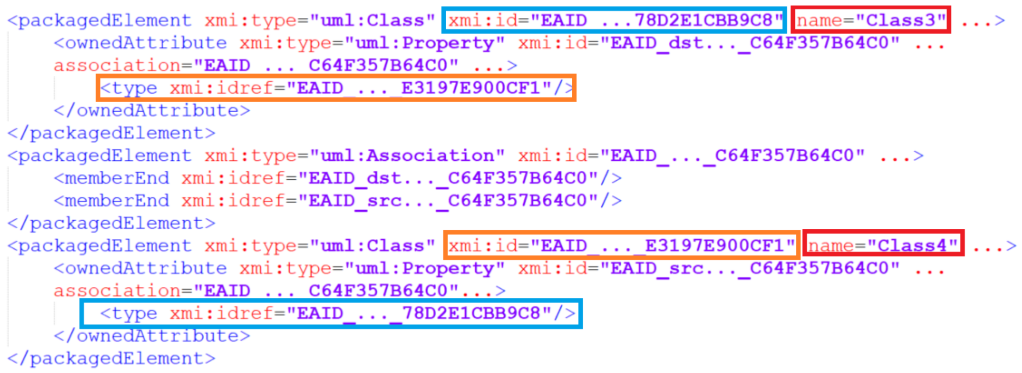
**Case 1 - Unspecified** As it is seen in the figure 3.6, association with unspecified direction is a basic type, where neither class does not have a reference to one another. Only association references both joining classes through element `<type>` inside element `<ownedEnd>`.



```
<packagedElement xmi:type="uml:Class" xmi:id="...73C728E01936" name="Class1" ..."/>
<packagedElement xmi:type="uml:Association" xmi:id="EAID_..._AC40DE72F027" ...">
    <memberEnd xmi:idref="EAID_dst..._AC40DE72F027"/>
    <ownedEnd xmi:type="uml:Property" xmi:id="EAID_dst...AC40DE72F027" ...
    association="EAID_..._AC40DE72F027"
    isStatic="false" isReadOnly="false" isDerived="false" isOrdered="false"
    isUnique="true" isDerivedUnion="false" aggregation="none">
        <type xmi:idref="EAID ... E191F6A2F62F"/>
    </ownedEnd>
    <memberEnd xmi:idref="EAID_src..._AC40DE72F027"/>
    <ownedEnd xmi:type="uml:Property" xmi:id="EAID_src..._AC40DE72F027" ...
    association="EAID_..._AC40DE72F027"
    isStatic="false" isReadOnly="false" isDerived="false" isOrdered="false"
    isUnique="true" isDerivedUnion="false" aggregation="none">
        <type xmi:idref="...73C728E01936"/>
    </ownedEnd>
</packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="EAID ... E191F6A2F62F" name="Class2" .../>
```

Figure 3.6: Example of XMI representation of association with unspecified direction

**Case 2 - Bi-directional** In the case of using bi-directional associations (**??**), both classes have each other stored inside an attribute as a reference. The difference between unspecified and bi-directional is that association has only the `memberEnd` elements inside. The actual elements pointing to the class, stored inside `ownedEnd` is in this case inside the class itself.

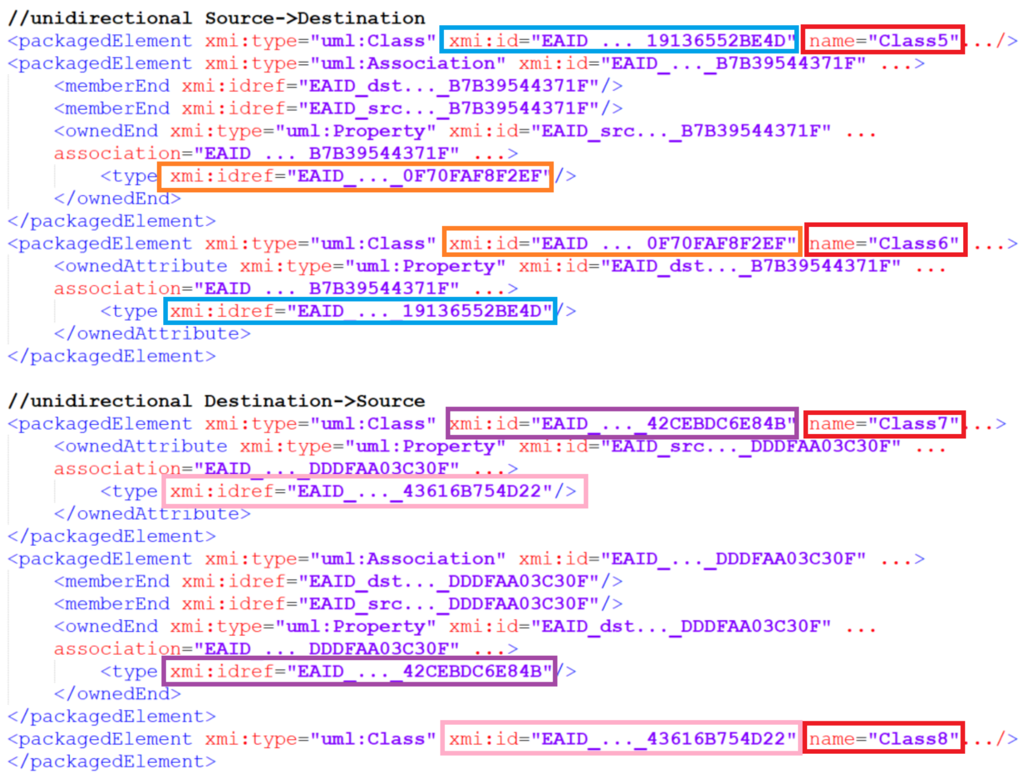


```
<packagedElement xmi:type="uml:Class" xmi:id="EAID ...78D2E1CBB9C8" name="Class3" ...>
    <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_dst..._C64F357B64C0" ...
    association="EAID ... C64F357B64C0" ...>
        <type xmi:idref="EAID_... E3197E900CF1"/>
    </ownedAttribute>
</packagedElement>
<packagedElement xmi:type="uml:Association" xmi:id="EAID_..._C64F357B64C0" ...>
    <memberEnd xmi:idref="EAID_dst..._C64F357B64C0"/>
    <memberEnd xmi:idref="EAID_src..._C64F357B64C0"/>
</packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="EAID_... E3197E900CF1" name="Class4" ...>
    <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_src..._C64F357B64C0" ...
    association="EAID ... C64F357B64C0"...>
        <type xmi:idref="EAID_..._78D2E1CBB9C8"/>
    </ownedAttribute>
</packagedElement>
```

Figure 3.7: Example of XMI representation of bi-directional association

**Case 3 and 4 - Source - Destination and vice versa** Similarly, in the case of unidirectional association 3.8, the reference to the other class is present, where the tip of the arrow is. In the generated XMI file, the examples of unidirectional associations look very similar to each other. In both cases, only one class has the reference to the other class in its attribute inside the `ownedEnd` element. The other reference stays inside the association itself. The difference between the cases of the direction of the arrow from source to destination and vice versa is that in the case of the source source ->destination, the destination class has a source node as reference. It also satisfies the statement that the class at which the arrow tip is, contains the reference of the other.

Enterprise architect marks its associations with shortcuts `src` and `dest` inside the reference ID. (e.g `EAID_dst..._B7B39544371F`). This marking does not correlate to the cases mentioned here, as the unspecified case has them as well as bi and unidirectional cases. It marks only the order of how the association was dragged out inside the designing tool by the user.



```
//unidirectional Source->Destination
<packagedElement xmi:type="uml:Class" xmi:id="EAID ... 19136552BE4D" name="Class5"../>
<packagedElement xmi:type="uml:Association" xmi:id="EAID_..._B7B39544371F" ...>
    <memberEnd xmi:idref="EAID_dst..._B7B39544371F"/>
    <memberEnd xmi:idref="EAID_src..._B7B39544371F"/>
    <ownedEnd xmi:type="uml:Property" xmi:id="EAID_src..._B7B39544371F" ...
    association="EAID ... B7B39544371F" ...>
        <type xmi:idref="EAID_..._0F70FAF8F2EF"/>
    </ownedEnd>
</packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="EAID ... 0F70FAF8F2EF" name="Class6" ...>
    <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_dst..._B7B39544371F" ...
    association="EAID ... B7B39544371F" ...>
        <type xmi:idref="EAID ... 19136552BE4D"/>
    </ownedAttribute>
</packagedElement>

//unidirectional Destination->Source
<packagedElement xmi:type="uml:Class" xmi:id="EAID_..._42CEBDC6E84B" name="Class7"...>
    <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_src..._DDDFAA03C30F" ...
    association="EAID_..._DDDFAA03C30F" ...>
        <type xmi:idref="EAID_..._43616B754D22"/>
    </ownedAttribute>
</packagedElement>
<packagedElement xmi:type="uml:Association" xmi:id="EAID_..._DDDFAA03C30F" ...>
    <memberEnd xmi:idref="EAID_dst..._DDDFAA03C30F"/>
    <memberEnd xmi:idref="EAID_src..._DDDFAA03C30F"/>
    <ownedEnd xmi:type="uml:Property" xmi:id="EAID_dst..._DDDFAA03C30F" ...
    association="EAID ... DDDFAA03C30F" ...>
        <type xmi:idref="EAID_..._42CEBDC6E84B"/>
    </ownedEnd>
</packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="EAID_..._43616B754D22" name="Class8"../>
</packagedElement>
```

Figure 3.8: Example of XMI representation of unidirectional association

## 3.4 Association class parsing

Parsing association class and mapping it to the graph notation using only nodes and edges can be challenging because, in the graph-based syntax of Neo4j, one cannot connect edge to another edge. Association class is represented as a regular class, which is connected to the association itself by the dashed line. This means that when the association class is present in the model, there are two semi-distinct parts that need to be parsed. Firstly, the basic association between two nodes (in the example above `Class9` and `Class10`) and secondly Association class itself. The first case is covered in the section discussing association parsing.

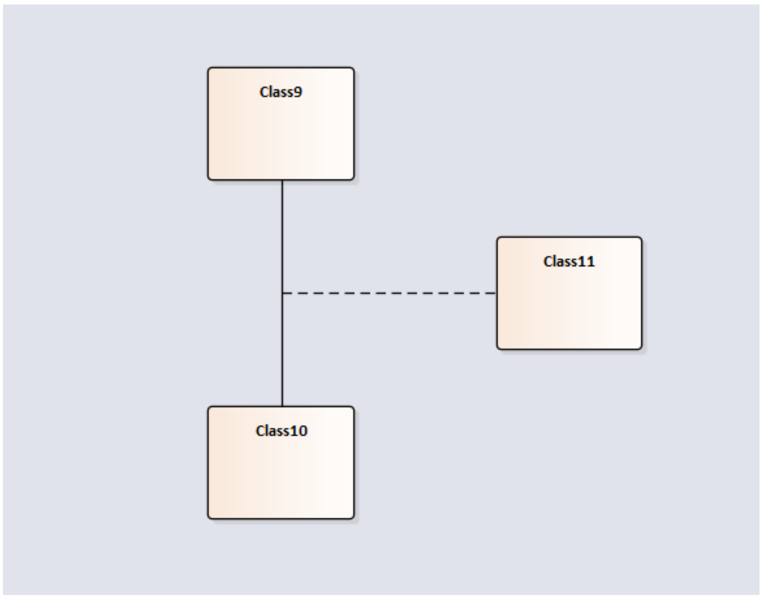


Figure 3.9: Example of association class

For the second part - parsing the class connected to the association class ( `Class11`), a new kind of node was created called `AssociationClass`. This node represents the fact that a connection is made with the association class and not with the basic association. The new node is needed as in Neo4j, there is no such thing as connecting relation to another relation. Although relations can contain values and properties on their own. Reference to the association class (`Class11`) could be stored inside. However the use case of this model searching and retrieval will be mainly used to lookup models and relations and based on this, the user can then open the specific XMI file inside the favourite designing tool. So making sure, that a clear visual distinction between this case of connection and basic association is needed. Additionally, user can filter out these nodes (`AssociationClasses`) and basic association (between `Class9` and `Class10`) would remain.

Connections between the `AssociationClass` node (`AssociationClass1` 3.10) and its classes (two classes similarly as in basic association; `Class9` and `Class10` and one representing the association itself - `Class11`) are not other association connections but rather some informative meta connection. (dash lined connection in the picture) This connection is not present in the UML specification. In the parser, it is called `association_class_connection` and it always joins the `AssociationClass` node to the `ClassNode`.The `ClassNode` does not have the reference for this connection as it is not needed.

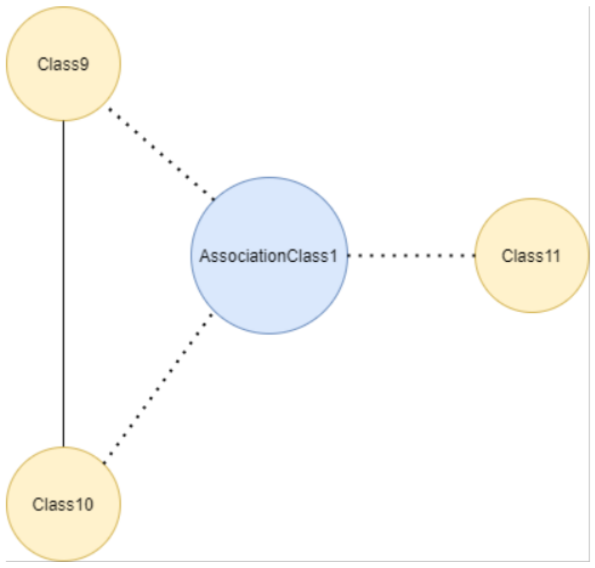


Figure 3.10: Example of association class represented in Neo4j

## 3.5 Association node

Association does not need to connect only two classes. There are plenty of examples where association connects multiple classes. This is usually represented through diamond-shaped object inside model design tools. There is no restriction on how many associations can be connected through this diamond object.

Parser identifies case when an association has more than two `memberEnds`. As mentioned in section 3.3 `memberEnds` are always present inside an association, even though ownedEnds (actual references to the classes) can be elsewhere in the XMI file. This means that `memberEnds` can be used to determine that more associations are connected. The parser then needs to create a node, which represents the diamond-shaped object. This node is called `AssociationNode`. Node does not have any additional parameters other than a unique id as it does not need any.

After `AssociationNode` creation is done, connections need to be set correctly. Each class participating inside this association needs to point to the newly created `AssociationNode`. This is simply done by reusing the parsing of basic 2 class association and instead of pointing `src` and `dest` to the classes themselves, now the parser points `src` to the `AssociationNode` and `dest` to the class. This gives us an equivalent representation of association connection using diamond shape inside Neo4j graph.

### 3.5.1 Improved proof-of-concept solution

A slightly improved design of the parser consists of 3 main parts. Parser dispatcher, parser dictionary and Model base. On the input of the parser is the XMI file, which can be from any of the modelling tools (Visual paradigm, Enterprise Architect, Openponk) which are supported by the parser. As mentioned in the analysis section, XMI files can differ due to the different implementation of the XMI standard by the design tools. This is addressed in the parser dispatcher. A parser dispatcher is a dictionary that recognizes XMI file format and chooses an appropriate parser for it. As this parser only parses class diagrams, `ClassDiagramParser` instances are returned.

This solution does not account for expansion in terms of different diagram types. And thus it is very limited in the parsing as it can only work with class diagrams. On the other hand, the development of other format-specific models is fairly straightforward, as it only has to inherit from the model base. The extension for different formats is also very simple; the addition of the new format specific model to the parser dictionary is the only requirement. The flaw in this design is a bad composition of the whole parsing process. Specifically, the parsing process should be detached from the model itself, but in this design, parsing tasks are done in Model classes. The proof-of-concept showed a basic outline of the solution, as well as some mistakes which could be worked on, and improved.

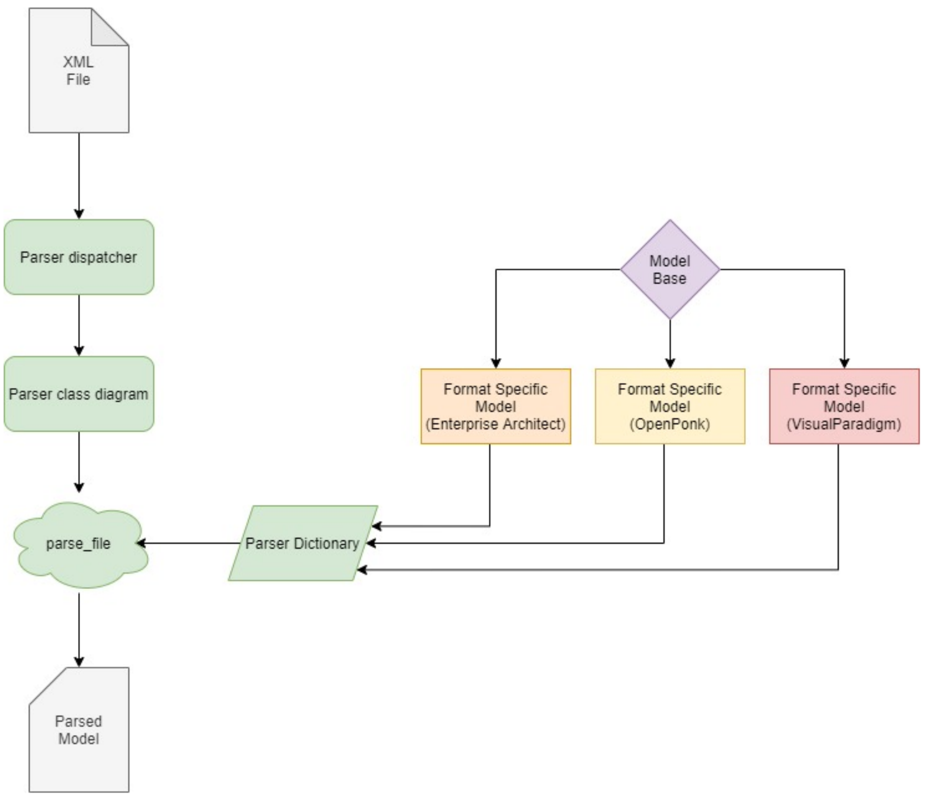

XML File
Parser dispatcher
Parser class diagram
parse_file
Parsed Model
Parser Dictionary
Model Base
Format Specific Model (Enterprise Architect)
Format Specific Model (OpenPonk)
Format Specific Model (VisualParadigm)

Figure 3.11: Overview of second proposed realisation

## 3.6 Final stage of implementation

Initial proof of concept and an additional more advanced solution did not satisfy the overall need to parse a variety of different conceptual models encoded in multiple XMI formats from multiple designing tools. The biggest weakness of those solutions was the limited extensibility of the application as it was primarily coded for class diagrams and nothing else. A new approach had to be taken into mind when designing the final solution. The structure of the code had to be changed to introduce more abstraction. The previous solutions lacked the ability to easily parse multiple XMI files produced by various design models containing several conceptual models. Mainly two parts had to be addressed; firstly different formats of XMI files and secondly different kinds of conceptual models. (previously only class diagram) The idea is to identify diagram types inside the model and also specify from which designing tool the XMI file came. Then this information is passed to the `ParserFactory`, which will be responsible for choosing the right parser based on that information. The chosen parser then takes the XMI file and tries to parse it. The parsed file will have a unified structure and it does not matter which conceptual model type was parsed. This abstraction is possible by encoding models to the notation of only nodes and relations which are then passed to the component responsible for taking care of the Neo4j database. This thesis is focused on parsing only class diagrams, but a foundation for other types of contextual models is present as well.

### 3.6.1 XMI file

`XMIFile` class represents the input XMI file. This class tries to identify the format of the models contained inside as well as the design tool format. For this tasks, it uses functions `get_format` and `get_diagrams`.

**get_format** The purpose of this function is to correctly identify the format in which the file was modelled. This function is contained inside the `XMIFile` class and does not take any additional parameters other than self. Currently, there are two formats identified by it. The Enterprise Architect is detected by the presence of the attribute exporter inside the `uml:Documentation` part of the XMI file. Openponk is set as a default format when there is no indication that the format is Enterprise architect. The problem of this format determinations is, no real unified way of telling from which format the model came. Determination gets even harder when users can omit the metadata section when exporting to the XMI. Then default Openponk format will be chosen and the process of parsing can end with failure. The possible but partial solution to this problem can be a brute-force approach; trying all of the available formats accepting the result where parsing completed without error and maximum of the nodes & relations were recognized.

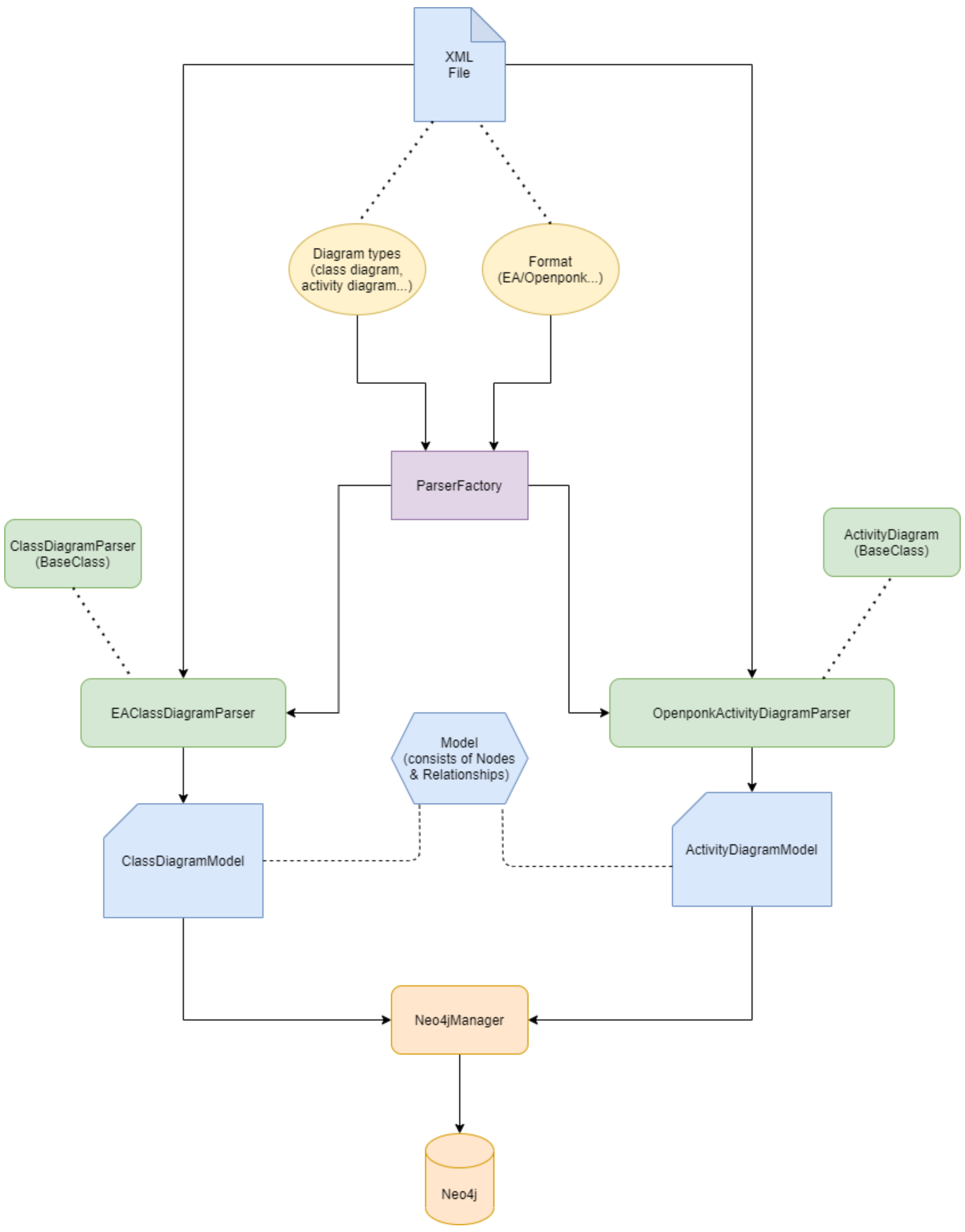


Figure 3.12: Overview of implementation with regards to extensibility

**get_diagram** This function is responsible for diagram recognition. The challenge with this task is the fact that there can be several diagrams inside one `<uml:Model>` element. The parsers should ideally recognise all of them correctly and based on the diagram, choose the right parser. The problem is, that there is no universal way of determining the diagram type. Enterprise architect for instance includes some information about diagram types in the `xmi:extensions` section, but this section can be omitted when exporting the XMI. A similar brute-force solution could be applied to this problem as with `get_format` function. However, when iterating through all of the possible diagrams, there is a possibility that multiple of them end without error. This is because currently, parsers work on the principle, that they parse only sections of XMI file that are familiar - there is a function that supports this kind of element. For example, if a class diagram will be parsed by a parser intended for package diagrams, the package parser will recognise the initial package where all of the classes are stored and then return with success. No classes were parsed, just the initial package. This case can be solved by deciding that only parser with the maximum nodes and relations parsed is chosen to be stored into the Neo4j.

Another solution to the problem could be input provided by the user. However, this would create problems on its own with the possibility of multiple diagrams, for which the would user had to fill the correct type. As this solution could be tedious for the user, the automatic identification of the diagram is preferable.

### 3.6.2 ParserFactory

A class `ParserFactory` is created as a solution to the problem of having multiple types of parsers. Various parsers need to be created and each one will be responsible for parsing a specific diagram in a specific format. The application thus needs to select the right one for the job. Using common design pattern in programming - Factory method, `ParserFactory` class facilitates this behaviour and parser creation without the need to expose internal logic to the outside world.

### 3.6.3 Class_diagram_parser

This class represents the interface or required behaviour of parsers, which are to be registered as class diagram parsers. This is the base class from which all format-specific parsers inherit their functionality and further extend or override it. The class contains various methods and functions to provide simple parsing of class diagrams. Every parser derived from this class should implement methods that are split up into two main groups. Utility functions, which provide basic logic when parsing models. They are not related to the class models in general. In the other group are functions, which are specifically created to parse objects and relations found inside the class diagram. Other parsers which would be derived from a different base class, for example, the base class for parsing action diagrams, would have different functions inside this second, model-specific group. However, the utility group would be very similar.

### 3.6.4 Utility functions

**parse_model** The function takes two input parameters. Model - section of the XMI file, represented by the element `xmi:Model`. A second parameter is a dictionary with namespaces used. The purpose of the class is to parse each significant object inside the class diagram and return the model in a special format. As it was mentioned earlier, to assure that application will be extensible, all parsers have to return the result in format, which is universal and agreed upon in advance. It has to be ready to be stored in the graph database easily which means, nodes and relations are the best representations. Function aggregates all syntactically significant objects and creates `ClsDiagramModel`, which is a class representing the class model from the input XMI file. This class is derived from the base class Model. Currently, there are 6 objects available for parsing from the class diagram other than id (classes, associations, association_classes, generalizations, and class and association types) but additional objects can be added freely in the future.

**parse_file** This function is responsible for parsing XMI file. It takes the name of the file as an input parameter. The function identifies namespaces present inside the XMI file, extracts the model part of the XMI file with another utility function `get_model`, and then calls a function to parse the model. The function returns the result of function `parse_model`.

**add_model_from_file** The function tries to parse the model from the path of the file provided as an input argument. The file should be in XMI or XML format. The most suitable parser is chosen for parsing. If parsing is successful, the model is added into the Neo4j by the `Neo4jManager` instance. Before the addition, the function checks whether a model with the same ID is not already present inside the Neo4j. If yes, `Neo4jManager` deletes the old model. A new model is inserted and the function returns with "True" value. If model addition is not successful, the function returns "False".

**add_model_from_github** The function tries to download all XMI or XML files inside the repository provided as an input argument. The function also expects the name of the owner of a repository provided. If the repository is not public, a token for private connection is also required. When XMI/XML files are recognized, they are iterated and in cycle, they are downloaded. If the function successfully downloaded file, the file is passed to the `add_model_from_file` which tries to store it inside the Neo4j. The function returns a tuple with a number of successfully recognized XMI files and a number of added models.

### 3.6.5 Parser

Parsers are created specifically for the type of the model (e.g. activity diagram) and format of the XMI file ( e.g. Enterprise Architect)

**Enterprise architect** The class diagrams produced in Enterprise architect are parsed inside `EA_class_diagram_parser`. This class inherits from the base class `ClassDiagramParser`. Here are defined and implemented all of the functions which specifically relate to the class diagram produced inside Enterprise architect. The implementation was to the great extent recycled from the previous proposals for a solution.

### 3.6.6 Model

**ClassDiagram model** The `ClassDiagramModel` class inherits from the Model class, which acts as an interface that requires the derived classes to implement method `get_neo4j_model`. This class should be implemented in such a way, that it returns a list consisting of nodes and a list consisting of relationships between them. This abstraction to represent the model as a tuple of two lists ensures that the `Neo4jManager` class then does not have a problem with adding the model to the Neo4j.

**Neo4jManager** `Neo4jManager` class is responsible for establishing connection and configuration with the Neo4j database as well as providing basic API to manage the database. The class is able to connect to the Neo4j and provides simple static functions for deleting all models from the database, inserting/updating the model to the database, or deleting a specific model to the database. The `Neo4jManager` class can insert any kind of model, as long as it implements the Model interface. This is achieved by using abstraction of models, more specific usage of universal encoding, that objects are represented strictly as nodes (Node classes) and relations between objects are represented by relations (Relation classes). `Neo4jManager` can then use this fact, to save all nodes into the database and subsequently iterate over every relation and connect the right nodes included in the relation.

## 3.7 Integration to repocribro

Repocribro serves as an easy overview and sharing platform for GitHub repositories. Those repositories can be viewed by other people inside different organisations. Repositories listed inside the Repocribro can contain various projects and different contents, but one common use case is when a team of people are working on the modelling and designing conceptual models. As mentioned many times before, those models can be represented as actual pictures of models or exports in various formats such as XMI. All of those formats are not useful when it comes to quick searching for a specific model with specific content. For example, when designing a new library system, the designer could want to see, in which models, the node with connection from Person to Book is present. Integration of parser created for this purpose, with Repocribro app, could enable this feature and provide valuable tools for designers/teachers/students who work with, design and model conceptual models. Repocribro provides great extensibility features and I tried to take advantage of it, to incorporate a parser with its Neo4j graph database storage inside this app.

Repocribro is available as a dockerized set of containers, consisting of the Repocribro application itself and a simple SQL database that provides storage for saved data about users and repositories. The idea of integrating parser consists of modifying user interface in one part and connecting separate part of docker images needed to provide structure to run parser behind the scene in the background. The extensions in Repocribro are simple to implement. All that is needed is to write an own class that extends the Extension class and returns itself. In this class, two main features are introduced. Tab in the UI which provides control and the management of repositories that will be parsed, and the interface to actively search and query the models inside the Neo4j without the need of redirection to the default Neo4j browser.

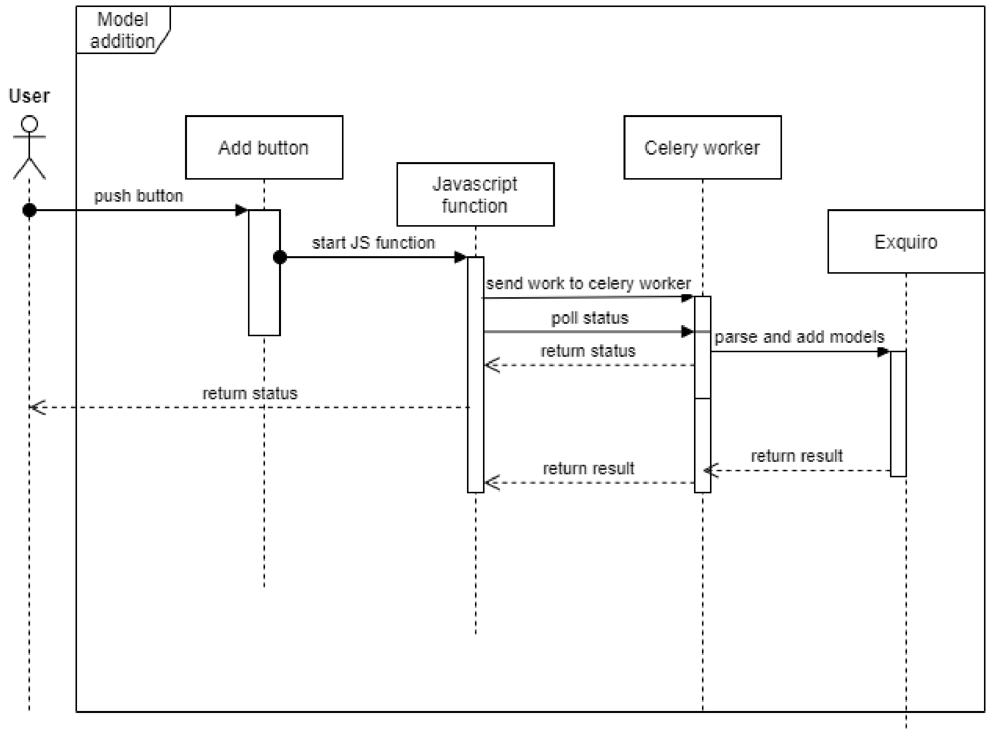


Figure 3.13: Model addition sequence diagram

### 3.7.1 Neo4j management tab (Exquiro)

The tab which is responsible for adding, updating and deleting models from/to Neo4j is called Exquiro. It lists all of the user-owned repositories and provides two buttons, one to add a model to the Neo4j and one to delete models from the Neo4j. Tab also informs the user, how many models in the repository have been recognized, which means how simply how many files inside the repository have the extension .xmi or .xml, and also informs about how many models were successfully added/deleted to/from the database. The add button acts as an upgrade as well, because all the models which are present inside the repository are on addition deleted first, when they are present in the system, and just after that, they are added for the second time. The model recognition inside the system works on an ID basis. The ID of the model is provided by the `get_id` function from the python parser (ref to section).

#### 3.7.1.1 Model addition/deletion

The background processe which facilitate the actual parsing and addition/deletion are initiated by the button click. The button runs the Javascript function which calls background job and sends it to the task queue. If a celery worker listening to the task queue is available, it accepts a new task and starts to parse the repository. Worker uses designed parser and calls function *add_models_from_github(repository, owner)*. This function then parses all models present in the repository as it is described in the section 3.6.4. Meanwhile, the task is being processed, the Javascript is polling the status of the task and periodically updates the page about the progress of the task. When the task is completed or failed, results are displayed on the page with the notification of success or failure. Also, there is information about the number of recognized XMI/XML files inside the repository and the number of successfully added are updated.

This process requires some modifications and additions to the Repocribro code, namely new controller - exquiro, which handles every request dealing with the parsing of repository models. With the controller, obviously, the view tab template was needed to display information. And lastly, some modifications to the SQL model of the Repository class, as there was a need to store information about the state of the parsing, a number of recognized repositories and a number of already added repositories. The state of the parsing is initially set to the default value of ”Not added”, and other values are set to zero. This modification then guarantees that when the Repocribro is for some reason restarted, these values are preserved and loaded from the persistent SQL storage.

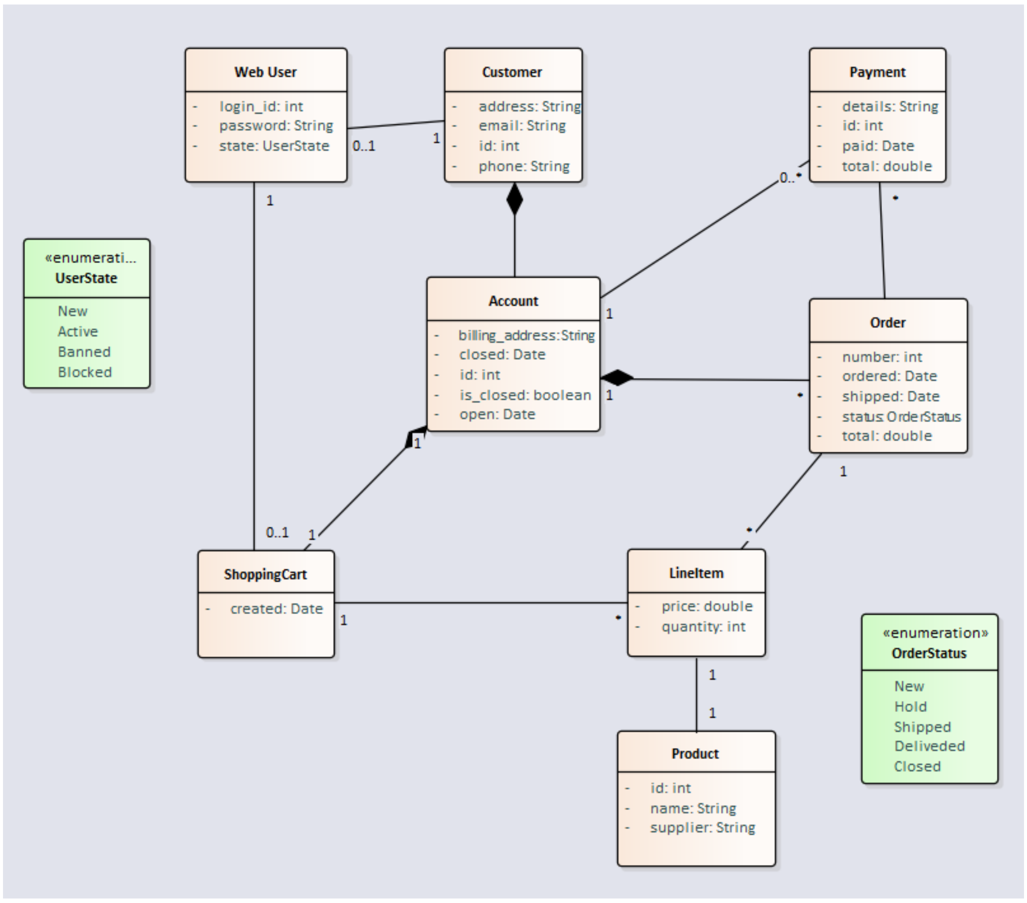


Figure 3.14: Example model from Enterprise architect

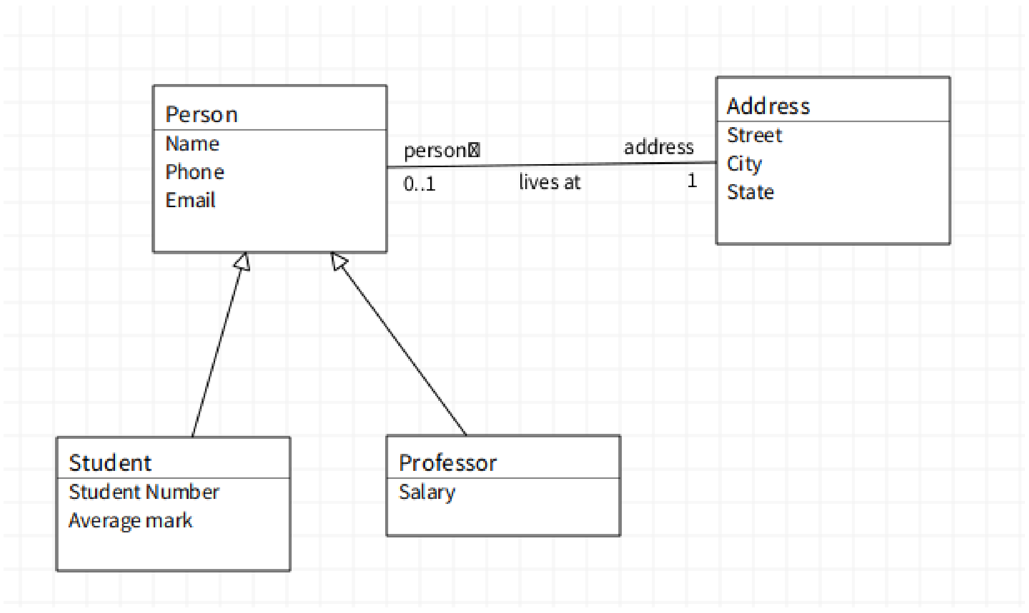


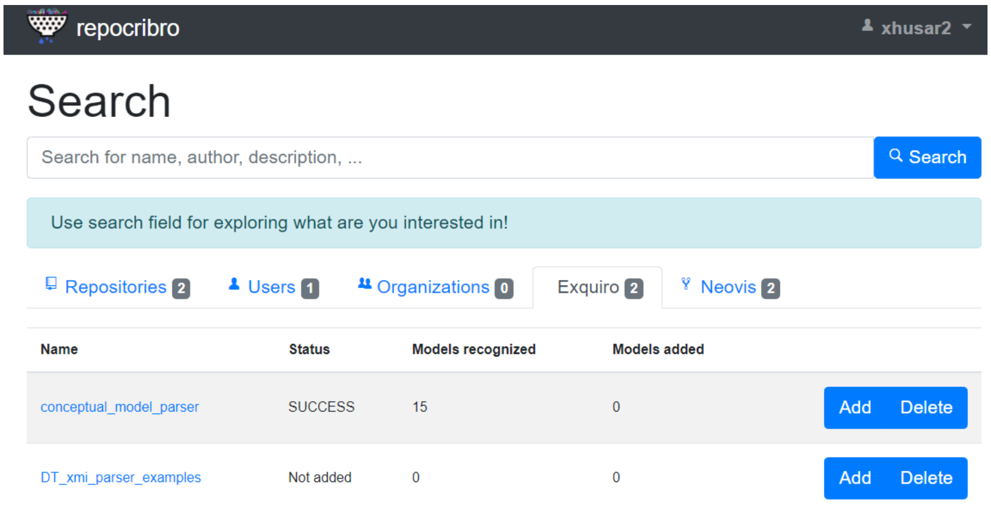
Figure 3.15: Example model from openponk

## 3.8 Example use case scenario

This section will demonstrate the real-life use case of the proposed solution. Taking it from the process of designing a conceptual model, through registering a repository with the exported model to the actual searching of the models inside the Neo4j. Firstly I created an example model in Enterprise architect seen in figure 3.14 which resembles a basic E-shop class diagram. It contains enumeration, classes, attributes, associations with multiplicity and aggregations. The second example shown in figure 3.15 comes from the OpenPonk software and shows a simple class diagram with few classes, associations and generalizations. These examples have been exported to the XMI format, each in its design tool of origin. The models in image format and also in newly generated XMI format were then uploaded to the example repository, which will be added to the Repocribro. The repository was set up to be publicly available.

On the figure 3.16 we can see the Repocribro search page, with two newly added tabs - Exquiro and Neovis. The selected tab Exquiro shows the user information about the repositories which are selected in Repocribro. Specifically, only those repositories are shown, which are present also inside the Repositories tab. The figure then shows the two repositories, of which one was previously added with success, 15 models were recognized and currently, zero models are added to the Neo4j. The second repository *DT_xmi_parser_examples* contains the two examples mentioned above, one from Enterprise architect and one from the OpenPonk. The status informs that the repository was not added yet and thus no models were recognized nor added.

After the user clicks on the add button, the application starts to process the repository in the background as was mentioned in the figure 3.13. When the parser is done, a notification will appear on the screen with the result status. After a refresh of the page, the user can see how many models have been recognized inside the repository and how many have been successfully added by the parser. At the moment there is no way to tell which models have been recognized, added or failed to parse as it would require a separate managing page and due to the time restrictions, this additional feature was not implemented.



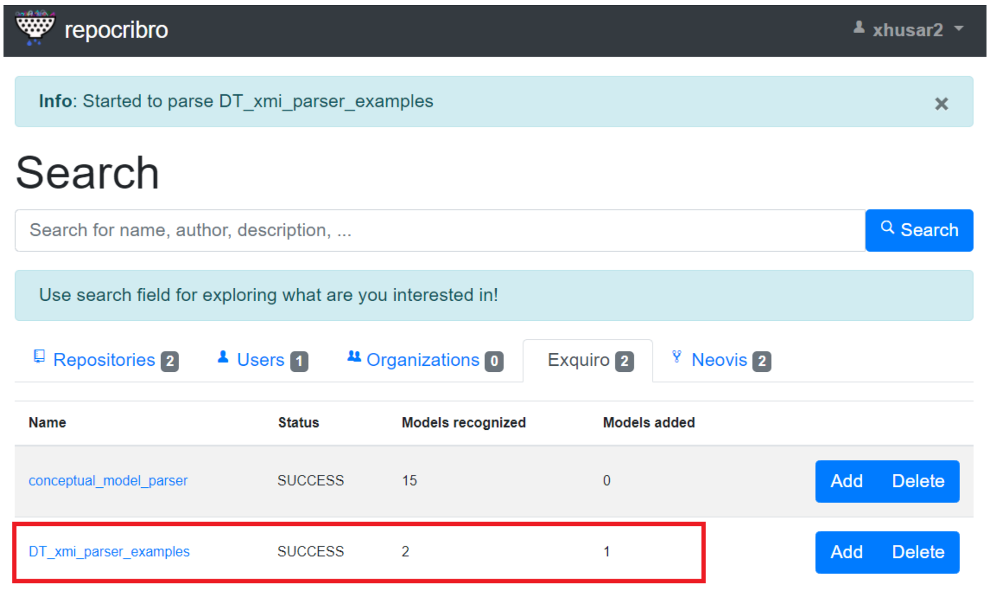
Use search field for exploring what are you interested in!

| Name | Status | Models recognized | Models added | |
|---|---|---|---|---|
| conceptual_model_parser | SUCCESS | 15 | 0 | Add Delete |
| DT_xmi_parser_examples | Not added | 0 | 0 | Add Delete |

Figure 3.16: Initial user interface

When the user is satisfied with the addition of repositories to the Neo4j, he/she can click on another tab called Neovis, to see all of the models successfully added, including the models of other users who did the same. As shown in the figure 3.18 the Neovis tab contains an embedded window with a connection to the Neo4j database and it conveniently shows the contents of the Neo4j inside the small embedded window. The tab also has a text field for writing queries in the Cypher language. Implicitly the Neovis window shows all of the models inside the database.

The main purpose of this Repocribro extension is to enable users who are interested in the particular kind of models to easily find them. At the time of writing this thesis, there is only one way to query these models, and that is by Cypher queries. Those queries can be directly applied inside of the Neo4j browser or more conveniently inside the Neovis extension tab. In this example case, the query could display some section of a model whose attribute is equal to the value "phone". As both of the models contain this kind of node, sections connected to that node, from both models are displayed in the window.


repocribro | xhusar2

**Info**: Started to parse DT_xmi_parser_examples ×

Search

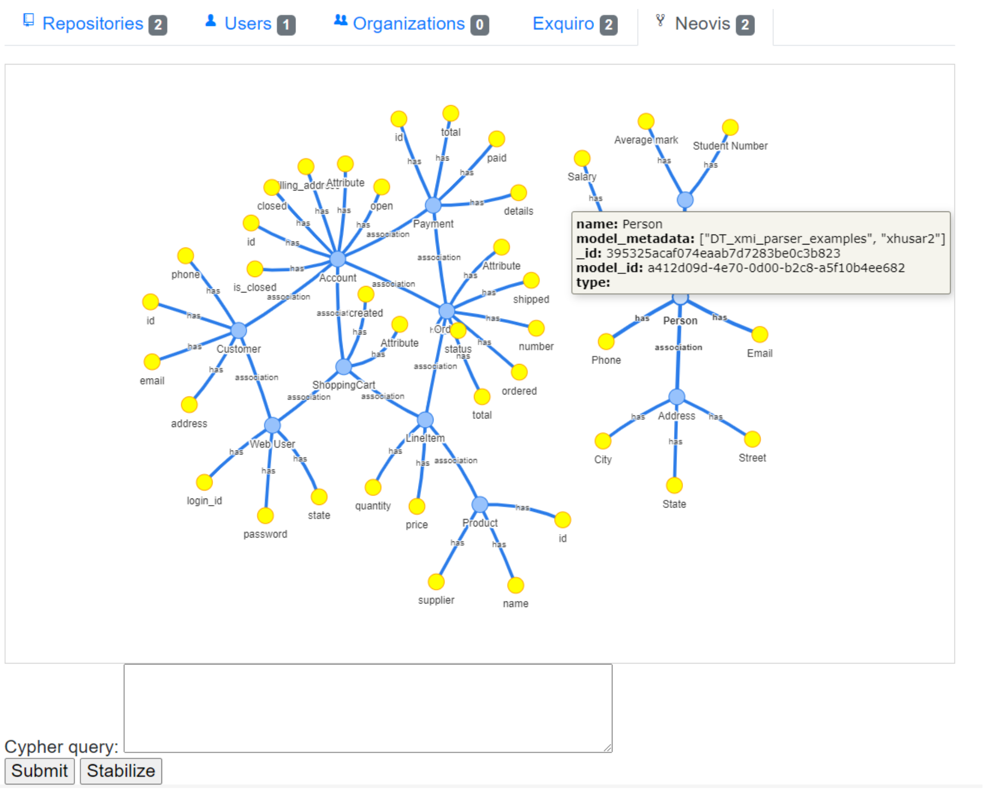
Search for name, author, description, ... | Search

Use search field for exploring what are you interested in!

Repositories 2 | Users 1 | Organizations 0 | Exquiro 2 | Neovis 2

| Name | Status | Models recognized | Models added | |
|---|---|---|---|---|
| conceptual_model_parser | SUCCESS | 15 | 0 | Add Delete |
| DT_xmi_parser_examples | SUCCESS | 2 | 1 | Add Delete |

Figure 3.17: Exquiro tab after model addition


Repositories 2 | Users 1 | Organizations 0 | Exquiro 2 | Neovis 2

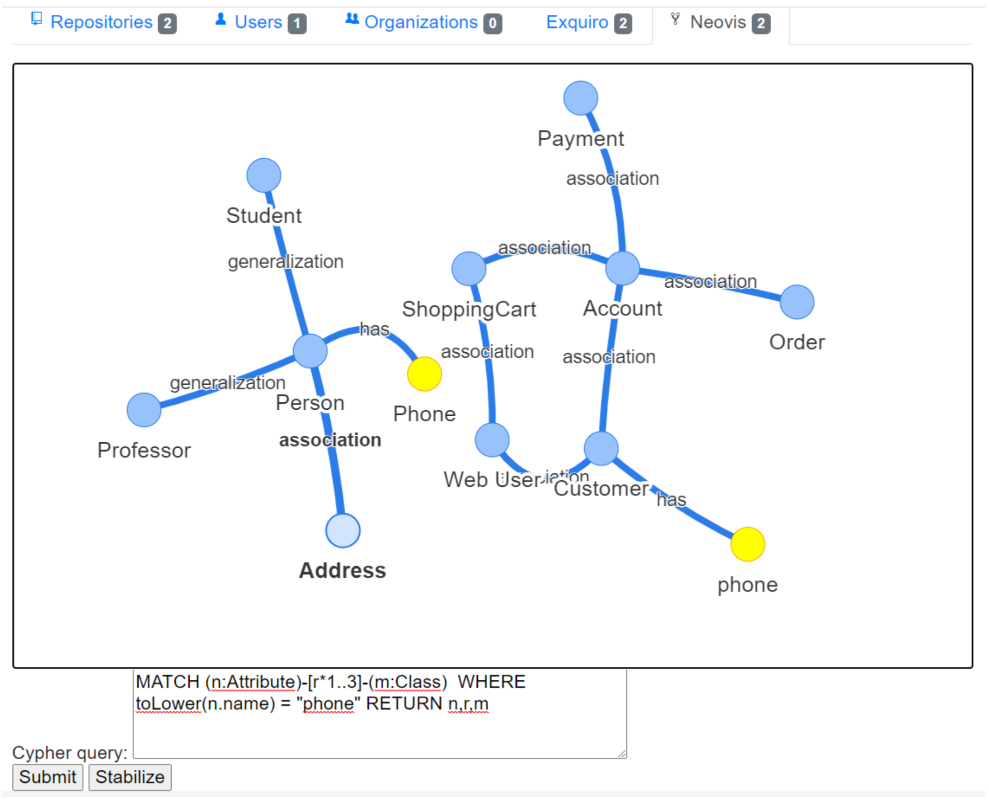
**name:** Person
**model_metadata:** ["DT_xmi_parser_examples", "xhusar2"]
**_id:** 395325acaf074eaab7d7283be0c3b823
**model_id:** a412d09d-4e70-0d00-b2c8-a5f10b4ee682
**type:**

Cypher query:

Submit | Stabilize

Figure 3.18: Neovis tab with both models added

Users are free to explore the possibilities which are provided by the Cypher language in conjunction with the graph database. For example, users can be interested in models, where a specific connection is present, or a node that starts with "birth" is contained in the model and that node must be connected to the node "person". Also, users can choose to show only just a fraction of a model or whole model which satisfies all of the conditions. When the results are shown, the user can easily retrieve information about the model, by hovering over any of the nodes to find metadata about the origin repository of the model 3.19. This fact enables users to easily search for the information they require and the application will point them to the source where the desired models are present. From there, users can download XMI files and collaborate end extend the models on their own in any of the model designing tools that enables XMI import. This way users do not have to begin on the project on "the green field", but have some foundation in the works of the community stored inside the application.



```
MATCH (n:Attribute)-[r*1..3]-(m:Class)  WHERE
toLower(n.name) = "phone" RETURN n,r,m
```

Figure 3.19: Neovis tab after querying the "phone" nodes with the 3 closest neighbour nodes

## 3.9 Testing

Testing of applications correct behaviour rests in the unit testing. A few dozen simple unit tests for each parser were written to ensure the correctness of the parsing. Each test was specified on one task; e.g class parsing or whole model parsing. For each testing cases, simple models were provided. Each one contained various elements. This provides an easy and comfortable way of confirming, that new additions to the models, parsers or other classes do not alter the behaviour of the application.

# Conclusion

The thesis elaborates on the analysis and implementation of searching inside the conceptual models. When working on the solution, several steps had to be taken, analysed and solved. Firstly, implementing the parser which processes XMI representations of conceptual models, enabled to store of various kinds of conceptual models (in the thesis, class diagrams were used to demonstrate the solution) inside the best-suited data storage medium. The data storage possibilities have been analysed and the best-suited solution was provided by the use of graph databases. After this stage, final integration to the Repocribro was designed and implemented, to provide a convenient way for end-user to search and query conceptual models. Several embedded solutions were considered and one which provided the best result was worked into the solution - Neovis. The final solution provides users with a simple way of integrating this parser to Repocribro, and use it to further multiply the usefulness by providing not only an easy repository filtering application but also an application that provides a way of convenient search and retrieval of data and metadata about the conceptual models inside the repositories. The parser was designed to be extensible for other types of conceptual models and also for other types of model designing tools.

The work provides value in trying to solve the need for reusability and recycling of the already developed conceptual models without using too much effort on the users' side. By creating the application, knowledge from the conceptual models included in the system can be shared between the community much easier and quicker.

There are several points where the future of the work could be heading. The issue with the need for the user to know query language Cypher can be avoided by designing and implementing a system of queries where the user could provide only the relevant terms. The application will automatically create a relevant query in the background and retrieve the information from the database. Another way to improve the design is to include more ways for inputting the source XMI models other than Github, such as google drive or other cloud storage. Furthermore, the security of the application was not covered by this work. Much more work can be done to improve the security of the Repocribro extension; preventing users from accessing Neo4j directly or using Neovis connection more securely without exposing Neo4j credentials. Also, other diagrams can be included in the application as well as extending the reach of current diagrams, by implementing other, less frequent constructs from UML. I believe my solution will simplify the act of sharing information from the already created models, by the users to the rest of the community.

# Bibliography

[1] Aggregation relationships. 2021, [Online; accessed April 25, 2021]. Available from: https://www.ibm.com/docs/en/SSCLKU_7.5.5/com.ibm.xtools.modeler.doc/images/2aggassoc.gif

[2] UML Association vs. Aggregation vs. Composition. 2018, [Online; accessed April 25, 2021]. Available from: https://static.javatpoint.com/tutorial/uml/images/uml-association-vs-aggregation-vs-composition3.png

[3] UML Generalization. 2018, [Online; accessed April 25, 2021]. Available from: https://static.javatpoint.com/tutorial/uml/images/uml-generalization.png

[4] Abdo, A.; Alali, S. Comparing Common Programming Languages to Parse Big XML File in Terms of Executing Time, Memory Usage, CPU Consumption and Line Number on Two Platforms. *European Scientific Journal*, volume 12, 09 2016, doi:10.19044/esj.2016.v12n27p325.

[5] Object Management group. *Unified Modeling Language 2.5.1 Specification [online]*. [publication date December 2017]. Available from: https://www.omg.org/spec/UML/2.5.1/PDF

[6] Meta-object facility. 2020. Available from: https://www.omg.org/mof/

[7] What is Unified Modeling Language (UML)? 2020. Available from: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/

[8] Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. USA: Addison-Wesley Longman Publishing Co., Inc., third edition, 2003, ISBN 0321193687.

[9] Fakhroutdinov, K. UML Class and Object Diagrams Overview. 2020. Available from: `https://www.uml-diagrams.org/class-diagrams-overview.html`

[10] UML Association vs Aggregation vs Composition. 2020. Available from: `https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition`

[11] Aggregation relationships. 2020. Available from: `https://www.ibm.com/docs/en/rsm/7.5.0?topic=diagrams-aggregation-relationships`

[12] Fakhroutdinov, K. UML Association vs. Aggregation vs. Composition. 2020. Available from: `https://www.javatpoint.com/uml-association-vs-aggregation-vs-composition`

[13] UML Class and Object Diagrams Overview. 2020. Available from: `https://www.uml-diagrams.org/association.html??context=class-diagrams`

[14] Van Bruggen, R. *Learning Neo4j*. Birmingham: Packt Publishing, 2014, ISBN 978-1-84951-716-4.

[15] Panzarino, O. *Learning Cypher*. Packt Publishing, 2014, ISBN 1783287756.

[16] Edwards, R. Neomodel documentation. 2019. Available from: `https://neomodel.readthedocs.io/en/latest/index.html`

[17] Graph Visualization Tools. 2021. Available from: `https://neo4j.com/developer/tools-graph-visualization/#embed-graph-vis`

[18] Lyon, W. Graph Visualization With Neo4j Using Neovis.js. 2020. Available from: `https://medium.com/neo4j/graph-visualization-with-neo4j-using-neovis-js-a2ecaaa7c379`

[19] Suchánek, M. Repocribro documentation. 2017. Available from: `https://repocribro.readthedocs.io/en/latest/`

[20] Docker Engine overview. 2020. Available from: `https://docs.docker.com/engine/`

[21] Introduction to Celery. 2018. Available from: `https://docs.celeryproject.org/en/stable/`

[22] Celery - Distributed Task Queue. 2018. Available from: `https://docs.celeryproject.org/en/stable/`

[23] Nicola, M.; John, J. XML parsing: A threat to database performance. 01 2003, pp. 175–178, doi:10.1145/956863.956898.

[24] Gormley, C.; Tong, Z. *Elasticsearch: The Definitive Guide*. O'Reilly Media, Inc., first edition, 2015, ISBN 1449358543.

[25] How much faster is a graph database really. 2021. Available from: `https://neo4j.com/news/how-much-faster-is-a-graph-database-really`

[26] Licencing. 2021. Available from: `https://neo4j.com/licensing/`

Appendix A

# Acronyms

| | |
|---|---|
| **GUI** | Graphical user interface |
| **XML** | Extensible markup language |
| **XMI** | XML metadata interchange |
| **OMG** | Object management group |
| **UML** | Unified modeling language |

Appendix **B**

# Contents of enclosed CD

```
 readme.txt ....................... the file with CD contents description
 docker_images ......................... the directory with executables
 src .................................... the directory of source codes
   repocribro_integration .. implementation sources of repocribro with
     extension
   exquiro ................... implementation sources of python parser
 text ...................................... the thesis text directory
   thesis ............. the directory of LaTeX source codes of the thesis
   thesis.pdf ......................... the thesis text in PDF format
```