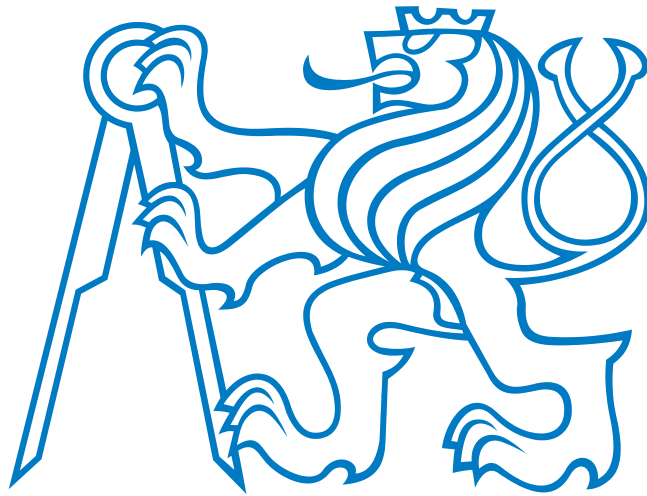


CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF ELECTRICAL ENGINEERING

DIPLOMA THESIS



Bc. Vladislav Vávra

System for analysis of damages of cultural heritage objects

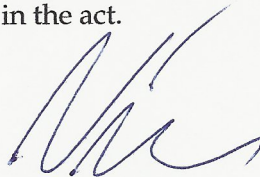
Department of Computer Graphics and Interaction
Supervisor: **Doc. Ing. Zdeněk Kouba, CSc.**

Prague, 2014

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used. I have no objection to usage of this work in compliance with the act §60 Zákona č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 5, 2014



.....
signature

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Vladislav Vávra**

Study programme: Open Informatics
Specialisation: Software Engineering

Title of Diploma Thesis: **System for analysis of damages of cultural heritage objects**

Guidelines:

- 1) Get familiar with: (i) OWL and SPARQL-DL languages, (ii) Monument damage ontology.
- 2) Describe the use case of Mondis users on analysis of substantial relationships in the above mentioned ontology.
- 3) Design the system for above mentioned use case. The API of the components shall be generic to allow for further use cases.
- 4) Implement the designed system.
- 5) Design test cases and test the implemented system. The thesis shall contain the project report.
- 6) The thesis write in English.

Bibliography/Sources:

Baader F. at all: The Description Logic Handbook Theory, Implementation and Applications, Cambridge University Press, 2007, ISBN 978-0-521-87625-4

Blaško M., Cacciotti R., Křemen P., Kouba Z.: Monument Damage Ontology, Proc. of the 4th International Conference on Cultural Heritage, Lemesos, Cyprus, 221-230, LNCS 7616, Springer-Verlag, Heidelberg, 2012, ISBN 978-3-642-34233-2

Clark & Parsia Inc.: Pellet: OWL 2 Reasoner for Java, <http://clarkparsia.com/pellet/>, citováno 20.12.2013

Diploma Thesis Supervisor: doc.Ing. Zdeněk Kouba, CSc.

Valid until the end of the summer semester of academic year 2014/2015


prof. Ing. Jiří Žára, CSc.
Head of Department




prof. Ing. Pavel Ripka, CSc.
Dean

Prague, February 24, 2014

Abstrakt

Diplovomá práce se zabývá návrhem a implementací softwarového systému pro analýzu poškození kulturních objektů. Tato úloha je motivována projektem MONDIS, který je zaměřen na vývoj informačního systému určeného pro analýzu poškození kulturních památek a vztahů mezi poškozeními a jejich příčinami. Vyvinutý systém je určen pro pomoc neexpertních uživatelů v analýze poškození památek, například zobrazení, která intervence odstraňuje nějaké projevy poškození. Systém je postavený na MONDIS ontologii, což je ontologická reprezentace zpracované domény. Systém je schopný reprezentovat více případů použití a jeden konkrétní případ užití, Intervenční, byl implementován. Tento případ použití je určen k popsání projevů poškození, jejich příčin a intervencí jim zabraňujícím. Je formálně popsán Deskripční Logikou a později je tento popis vyjádřen v jazyce OWL2. V práci je představena navržená architektura systému a jako důkaz její správnosti je prezentována její implementace. Tato implementace byla otestována jednotkovými testy s pokrytím řádek kódu přes 75 % v průměru. Uživatelské rozhraní bylo implementováno jako webová aplikace a bylo otestováno automatizovanými Selenium testy.

Abstract

The main goal of this thesis is to design and implement a software system for the analysis of damage to cultural heritage objects. This task is motivated by the MONDIS project that is focused on the development of an information system aimed at analyzing damage to cultural heritage objects and the relations between them. The system developed in this thesis is designed to help non expert users to see relations in the domain, for example which intervention is stopping which manifestation of damage. The system is built on top of the MONDIS ontology, which is an ontological model of the represented domain. The system is able to represent more use cases and one such a use case - the Intervention use case - is implemented. This use case aims to describe the causes of manifestation of damages and interventions preventing them. It is formally described using description logic and then this description is refined into OWL2 language. The architecture of the proposed system is presented and as a proof of concept an implementation of this architecture is given. Implemented solution is tested by unit tests with average test line coverage over 75 %. The user interface is implemented as a web application and it was tested by Selenium automated tests.

I would like to thank my supervisor, Mr. Kouba, for his great help and patience with this thesis. I would also like to thank Mr. Blasko for his help with the design of the implemented component and for his advice regarding ontologies. And I would also like to give thanks to Zuzka, my girlfriend, for her patience with me over last few months.

Contents

1	Introduction	1
1.1	Intervention Use Case	2
1.2	Design and Implementation	2
2	Knowledge Representation	4
2.1	Intuition	4
2.2	Description Logic	5
2.2.1	Syntax	6
2.2.2	Semantics	7
2.2.3	Examples of usage	8
2.2.4	Queries	9
2.2.5	Intervention Use Case - Formal definition	10
2.3	RDF(S), OWL2 And Semantic Web	12
2.3.1	RDF	12
2.3.2	OWL2	14
2.3.3	SPARQL	16
2.3.4	Intervention Case One - Semantic web representation	18
3	Architecture	25
3.1	Typographical conventions	25
3.2	Overall architecture	26
3.3	Knowledge-Matrix-Server	27
3.3.1	Data Transfer Objects	28
3.3.2	Query Objects	30
3.3.3	Service Objects	34
3.3.4	Class relationships	36
3.4	Knowledge-Matrix-Component	39
3.4.1	Tree-Matrix-Component	39
3.4.2	Info-Card	43
3.4.3	Auto-Complete-Tree	43

4	Implementation And Testing	50
4.1	Knowledge-Matrix-Server	50
4.1.1	Knowledge Framework	50
4.1.2	Testing	51
4.2	Knowledge-Matrix-Component	52
4.2.1	Web Framework	54
4.2.2	Testing of knowledge-matrix-component and auto-complete-tree . .	55
4.3	Knowledge-Matrix-User-Interface	56
5	Conclusion	59
	Attachment A: Content of CD	64

List of Figures

1.1	Example of a tree of manifestations of damage.	2
2.1	Visualization of <i>Concepts</i> of Intervention Use case and <i>Roles</i> between them. .	10
2.2	Example of an rdf triplet capturing relation hasMaterial between bridge and oak.	13
2.3	Example of an rdf graph representing Material and Component classes and their instances.	14
2.4	Example of Material, Component and Manifestation Of Damage representation.	18
3.1	Deployment diagram.	27
3.2	knowledge-matrix-component class diagram.	29
3.3	Data loading sequence diagram.	37
3.4	Getting axis and their values	38
3.5	Example of getting content nodes	40
3.6	Example of possible state of tree-matrix-component.	41
3.7	Class diagram of TreeMatrixComponent.	44
3.8	Example of visualization of info-card.	45
3.9	Example of possible state of auto-complete-tree.	45
4.1	knowledge-matrix-component deployed to Liferay portlet	58

Chapter 1

Introduction

This thesis deals with the design and implementation of a software component that aims to help users analyze damage to historical monuments. The component is a part of the MONDIS project [3], which is an information system developed to help restorers and students of cultural heritage with their professional work. There are already some existing information systems that deal with cultural heritage, but they are mainly concerned with cultural and historical information. The MONDIS project aims to describe the technical state of monuments with respect to their damage and to help the users with an analysis of the interventions that would prevent or repair the manifestation of damage.

There are already some implemented software artifacts in the MONDIS project. There is an ontology that describes the domain of historical monuments and the manifestation of their damage. This ontology is mainly concerned with technical knowledge about the domain. It captures the relations between materials, constructions, their causes of damage and preventative and reparatory interventions. This ontology is the main data source for all other applications that are developed for the MONDIS project.

As expected, users of the MONDIS project are not experts about ontologies, a tool that would help the users to update and modify the ontology was developed within the MONDIS project. This tool is named OntoMind [25] and it is based on the idea of mind mapping. Mind mapping is an intuitive approach used to sort ideas and thoughts but it lacks formal semantics. OntoMind combines mind mapping with ontological semantics which makes it usable by non technical users and yet keeps data semantically correct.

Another part of the MONDIS project that was already developed is a mobile application. This application is named Mondis Mobile [4] and it is a multi-platform application designed for tablets and other mobile devices to support gathering and processing data related to on-site surveys. The goal of this application is to help restorers during their on-site work. It can work online - connected to the central server, or offline working only with a local data model.

The MONDIS project still lacks an application that would show general relations between the components of monuments, their materials, their manifestations of damage and the mechanisms causing them. This thesis aims to design and develop such an application. The goal is to help non expert users and students to see and understand the relations that are contained in the MONDIS ontology. This component is termed Knowledge Matrix and as its name suggest, it presents the knowledge in the form of a matrix. Each column and row can represent one element of the ontology and the cells represent relations between those elements. An example of such a matrix is rows representing ma-

materials and columns representing components of monuments and each cell represents a relation between an associated material and a component.

The Knowledge Matrix is designed to be a generic component that is configurable by given use cases. A use case defines the meaning of columns and rows and specifies the relations between them. The provided implementation is realizing one such use case. This use case termed the Intervention use case and is informally described in the next section.

1.1 Intervention Use Case

The purpose of this use case is to present the relations between the manifestations of damage of some material or component and associated interventions. The manifestations of damage are hierarchically organized in a tree structure where the children of a node represent more specific concepts and the parents represent more general concepts. An example of such a tree is in figure 1.1. Each manifestation of damage is associated with some material and/or some component. Materials and components are also organized in tree structures.

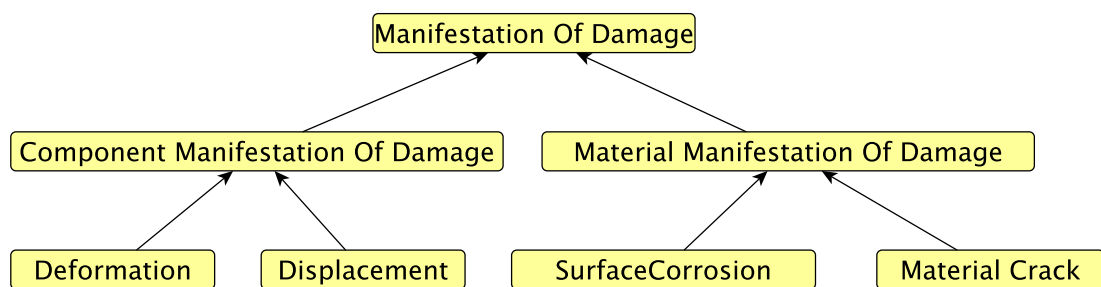


Figure 1.1: Example of a tree of manifestations of damage.

Interventions are represented by three groups. The first group contains all interventions that are directly stopping some manifestation of damage, but are not stopping the cause of this manifestation. An example of such an intervention is water cleaning. The second group represents all such interventions that are stopping some mechanism that is causing some manifestation of damage. An example of this relation is salt crystallization, which belongs to mechanisms, and cleaning, which belongs to interventions. The last, third, group is concerned with an analysis of the relation between agents, mechanisms and manifestations of damage. A manifestation of damage is caused by some mechanism and this mechanism is triggered by some agent. An example of this relation is moisture (as an agent) and condensation (as a mechanism) causing erosion (as a manifestation of damage). The agent can be prevented by some intervention and when the agent is prevented, the associated mechanism is mitigated and manifestation of damage is then eliminated.

1.2 Design and Implementation

The implementation of the component shall be as generic as possible in order to implement more use cases and not only the Intervention use case. For this reason, the architecture of this component is given in this thesis and rationale explaining the applied decisions is presented. As a proof of concept, an implementation that adheres to the proposed

architecture is given. This implementation was tested by unit tests and by automated tools testing user interfaces.

The rest of this thesis is organized as follows: The second chapter describes the knowledge representation used to store data and presents an introduction to description logic and two technologies related to it: RDF and OWL. Description logic is the formalism that is used in ontologies, and RDF and OWL are technologies that are used to represent ontologies in standard form. The third chapter describes the proposed architecture and the decisions that are behind it. The fourth chapter describes the implementation, testing and the technologies used in the implemented component. The last chapter contains the conclusion and presents results of this thesis.

Chapter 2

Knowledge Representation

The requirements for this thesis are to implement use cases specified by the MONDIS [7] project and yet this solution should be as generic as possible for the future reuse of the implemented code. This task is not straightforward and it requires a more systematic approach to solving given problems. This chapter aims to describe how the knowledge of this project is organized and stored. The first section of this chapter informally explains the big picture: how data is represented, how data mutually corresponds, and how knowledge can be extracted from it. After this section the reader should have a basic intuitive idea about the data representation. The second section refines the intuition into a more mathematical form using description logic [2]. Description logic is a well elaborated subset of first order predicate logic and therefore there are many standard ways of representing knowledge within it. The intervention use case is expressed in description logic in this section. When the formal description has been completed a technology that is backed by description logic is introduced. This technology is based on the ideas of Semantic Web and Linked Data. The utilized technology is a standardized solution so there are many independent projects and applications based on it. The technology is explained at the level of detail required to understand the implementation of this thesis.

2.1 Intuition

This section is aimed at creating reader's intuitive understanding of the basic idea that stands behind the rest of this thesis. The idea of the proposed approach is based on discussions with Miroslav Blasko [6]. The explanation of this approach in this section is not meant to be exact but rather to give the first idea of how the system is organized. More formal explanation is given in the next section.

The whole knowledge organized by the proposed solution can be seen as a hypercube in a coordinate system consisting of several axes. The dimensions of the hypercube is defined by a number of axes. The axes represent concepts, like materials or components. Those axes will from now on be termed TopCoordinates. A TopCoordinate is the root of a tree that represents a hierarchy of concepts where the children of a node are specializations of their parent. An example of the specialization relation can be wood as a parent and oak and spruce as children. The nodes of the tree are called Coordinates and one Coordinate represents one concept, like wood or bridge. Each Coordinate can have a set of children Coordinates that are specializations of the Coordinate.

The hypercube cell is a tuple of the form $\langle coord_1, coord_2, \dots, coord_n \rangle$ when n is a number less or equal to the dimension of the hypercube and each $coord_i$ (for $i \in \langle 1, n \rangle$) represents one Coordinate. The cell content is a mapping of a cell to a set of all elements that are associated together: $\langle coord_1, coord_2, \dots, coord_n \rangle \rightarrow \{e_1, e_2, \dots\}$. Elements represent the output of a query specified by the Coordinates. The association is defined by the intersection of $coord_1, coord_2, \dots, coord_n$. An example of such a cell can be the intersection of component and material that can result in "roof made of wood" or "bridge made of stone". Any combination of TopCoordinates or Coordinates can be used in intersection operation. An example of such a combination can be component and metal which can result in "Bell made of bronze".

An intersection relation intuitively represents connectors or adjectives in sentences like "bell made of bronze" or "rotting wood". The relation between any two axis has to be defined either explicitly or implicitly. This definition can be intuitively done for example by relation "made of" as was already suggested by the example "bell made of bronze". Another example of the relation between three axes (component, material and manifestation of damage) is "rotting wooden roof".

2.2 Description Logic

The formalism that is used for the description of use cases that are implemented in this thesis is description logic. At the time this thesis was written only one such use case was defined - the Intervention use case. This use case is formally described in this section and the relation to intuition from the previous section is given.

Description logic is a (typically) decidable subset of first order predicate logic. It has historically evolved from semantic networks and frames [2]. Description logic is the formalism that stands behind some Semantic Web technologies [24] and for this reason this section presents short introduction to it. Description logic is based on the following ideas (taken from [2]):

- The basic syntactic building blocks are atomic concepts (unary predicates), atomic roles (binary predicates), and individuals (constants).
- The expressive power of the language is restricted in that it uses a rather small set of constructors for building complex concepts and roles.
- Implicit knowledge about concepts and individuals can be inferred automatically with the help of inference procedures.

Concept can be understood as a set of somehow related objects, e.g. *Car*. *Individual* is one element of a concept, e.g. *car_9H9_7903*. One individual can belong to more concepts at the same time. *Role* is a set of binary tuples that relate two concepts together. Example of a such role can be *hasColor* that establishes relation between *Car* and *Color*. The organization of *Concepts*, *Roles* and *Individuals* is contained in a so called knowledge base that is composed of three mutually disjoint sets: The TBox that contains used terminology (e.g. *Cars*), the ABox that stores assertions made about individuals (e.g. *car_9H9_7903* is a *Car*) and the RBox that stores relations between roles (e.g. *hasColor* is a subproperty of *hasProperty*).

One important feature of description logic is called open world assumption. This assumption says that if some piece of information is not known, it does not mean that it is automatically false. For example let *Parent* be a concept of all individuals that have at least one child and *Mother* be a concept of all individuals that are females with at least one child. Then let state that Amanda is a mother ($Mother(Amanda)$), but no more information about Amanda is provided. But because Amanda is a mother and all mothers have at least of child, also Amanda has to have at least on child, although unknown. And because Amanda has at least one child, she must also be a parent since parents are all individuals with at least on child.

There are many dialects of description logic that differ in concept and relation constructors. An example of such a constructor for concepts is logical AND, denoted by \sqcap , ($Car \sqcap Red$ is a new concept representing all red cars). The dialect used in this thesis is $SR\mathcal{OIQ}^{(D)}$. The reason for using this dialect is that OWL2 - a popular technology used in Semantic Web (explained in the following section) is backed by this dialect [15].

An explanation of description logic is given precisely in [2]. The explanation given in this thesis is meant as a quick introduction and serves to establish vocabulary used in this thesis. The semantics of $SR\mathcal{OIQ}$ is explained by equivalent formulas in set theory. Note that $SR\mathcal{OIQ}^{(D)}$ is a language based on $SR\mathcal{OIQ}$ with an additional feature - data properties (D). The reason for presenting the simplified version is to make explanation shorter and easier to understand.

The following subsection explains the used syntax and the next subsection defines its meaning. The subsection 2.2.3 presents two simple examples of applied description logic, the subsection 2.2.4 is about querying knowledge stored in a knowledge base and the last subsection 2.2.5 formally defines the Intervention use case. Note that semantics is explained by set theory and this explanation does not detail any precise method regarding how to reason with this data. For information about reasoning algorithms please see [2].

2.2.1 Syntax

This section presents the syntax commonly used in description logic. The meaning of the symbols used is given in the section 2.2.2. *Atomic Concept* is a *Concept* that is defined when a domain that is described is defined. *Atomic Concept* is denoted by the letter A . *Atomic Role* is a *Role* that is defined when the described domain is defined. Let C, D be *Concepts*, R, R_1, \dots, R_m, L *Roles*, a, b *Individuals* and n, m natural numbers.

Then TBox can be defined by the following relations:

$$C \sqsubseteq D \mid C = D$$

RBox can be defined by the following relations:

$$\begin{aligned} R \sqsubseteq L \mid R_1 \circ R_2 \circ \dots \circ R_n \sqsubseteq L \mid Func(R) \\ Sym(R) \mid Asy(R) \mid Tra(R) \mid \\ Ref(R) \mid Irr(R) \mid Dis(R, L) \end{aligned}$$

ABox can be defined by the following relations:

$$C(a) \mid R(a, b)$$

A new *Concept* E can be defined by using the already defined *Concepts* by the following rules:

2.2. DESCRIPTION LOGIC

$$\begin{aligned}
 E := & A \mid \perp \mid \top \mid \\
 & \neg C \mid C \sqcap D \mid C \sqcup D \mid \\
 & \forall R.C \mid \exists R.C \mid \{a\} \mid \\
 & \exists C.\text{self} \mid \geq nR.C \mid \leq nR.C
 \end{aligned}$$

2.2.2 Semantics

The semantics of the *SR_{OIQ}* dialect of description logic is explained by the set theory. The set theory was used because it is well known and the explanation is straightforward. Interpretation I is a tuple of Δ^I and interpretation function \cdot^I . Δ^I is called the domain of interpretation and it represents objects from "real world". The interpretation function \cdot^I maps *Individuals*, *Concepts* and *Roles* to this set and thus it gives meaning to them. It formally maps *Individual* a to $a^I \in \Delta^I$, *Concept* A to $A^I \subseteq \Delta^I$ and *Role* (a, b) to $(a, b)^I = (a^I, b^I) \in \Delta^I \times \Delta^I$.

Table 2.1 shows interpretation of relations in TBox, RBox and ABox in the set theory and table 2.2 shows semantics of concept and relation constructors. The table is inspired by [24].

TBox Relation	Interpretation \cdot^I	Description
$A \sqsubseteq B$	$A^I \subseteq B^I$	Subset of
$A = B$	$A^I = B^I$	Equality
RBox Relation	Interpretation \cdot^I	Description
$R \sqsubseteq L$	$R^I \subseteq L^I$	Subset of
$R_1 \circ R_2 \circ \dots \circ R_n \sqsubseteq L$	$R_1^I \circ R_2^I \circ \dots \circ R_n^I \subseteq L^I$	Role hierarchy
$Func(R)$	$(a, b) \in R^I \wedge (a, c) \in R^I \rightarrow a = c$	Functionality
$Sym(R)$	$(a, b) \in R^I \rightarrow (b, a) \in R^I$	Role symmetry
$Asy(R)$	$(a, b) \in R^I \rightarrow (b, a) \notin R^I$	Role asymmetry
$Tra(R)$	$(a, b) \in R^I \wedge (b, c) \in R^I \rightarrow (a, c) \in R^I$	Transitivity
$Ref(R)$	$\forall a : (a, a) \in R^I$	Reflexivity
$Irr(R)$	$\forall a : (a, a) \notin R^I$	Irreflexivity
$Dis(R, L)$	$R^I \cap L^I = \emptyset$	Disjointness
ABox Relation	Interpretation \cdot^I	Description
$C(a)$	$a^I \in C^I$	Individual assertion
$R(a, b)$	$(a^I, b^I) \in R^I$	Role assertion
$\{a\} \approx \{b\}$	$a^I = b^I$	Individuals are same
$\{a\} \not\approx \{b\}$	$a^I \neq b^I$	Individual difference

Table 2.1: Semantic of TBox, RBox and ABox.

Concept C	Interpretation C^I	Description
\top	Δ^I	Universal Concept
\perp	\emptyset	Unsatisfiable Concept
$\neg A$	$\Delta^I \setminus A^I$	Complement
$A \sqcap B$	$A^I \cap B^I$	Conjunction
$A \sqcup B$	$A^I \cup B^I$	Disjunction
$\forall R.C$	$\{a \in \Delta^I \mid \forall b : (a, b) \in R^I \rightarrow b \in C^I\}$	Universal qualification
$\exists R.C$	$\{a \in \Delta^I \mid \exists b : (a, b) \in R^I \wedge b \in C^I\}$	Existential qualification
$\{a\}$	a^I	Nominals
$\exists R.self$	$\{a \in \Delta^I \mid \exists a : (a, a) \in R^I\}$	Reflection
$\geq nR.C$	$\{a \in \Delta^I \mid \exists b : (a, b) \in R^I \wedge b \in C^I \geq n\}$	Qualified number restriction
$\leq nR.C$	$\{a \in \Delta^I \mid \exists b : (a, b) \in R^I \wedge b \in C^I \leq n\}$	Qualified number restriction
Role R	Interpretation R^I	Description
\top^R	$\Delta^I \times \Delta^I$	Universal role
R^-	$\{(a, b) \mid (b, a) \in R\}$	Inverse role

Table 2.2: Semantics of Concept and Role constructors.

2.2.3 Examples of usage

The following text presents an example of a knowledge base and its interpretation. Note that *Individuals* start with prefix $a_$ and they represent some particular objects from the described domain.

Let

- Δ^I be a domain of interpretation defined as:
 $\Delta^I = \{a_roof, a_bridge, a_church, an_oak, a_granite\}$.
- Component, StoneMaterial, WoodenMaterial be *Atomic Concepts* and hasMaterial a *Role*.
- $Component^I = \{a_roof, a_bridge, a_church\}$.
- $WoodenMaterial^I = \{an_oak\}$.
- $StoneMaterial^I = \{a_granite\}$.
- $Material = WoodenMaterial \sqcup StoneMaterial$.
- $hasMaterial^I = \{(a_roof, an_oak), (a_bridge, an_oak), (a_bridge, a_granite)\}$.

Notice that a_church has no associated material. This information can be understood that the material of a_church was not known during the definition of the interpretation domain. The statement $\exists hasMaterial.Material$ is interpreted as a set that is composed of elements that are in binary relation with **Material**. Such elements are $\{a_roof, a_bridge\}$ because

$$(\exists hasMaterial.Material)^I =$$

$$= \{a \in \Delta^I \mid \exists b : (a, b) \in hasMaterial^I \wedge b \in Material^I\} =$$

2.2. DESCRIPTION LOGIC

$$\begin{aligned}
&= \{a \in \Delta^I \mid \exists b : (a, b) \in \text{hasMaterial}^I \wedge b \in \text{WoodenMaterial}^I \cup \text{StoneMaterial}^I\} = \\
&= \{a \in \Delta^I \mid \exists b : (a, b) \in \{(a_roof, an_oak), (a_bridge, an_oak), (a_bridge, a_granite)\} \wedge \\
&\quad b \in \{an_oak\} \cup \{a_granite\}\} = \\
&= \{a \in \Delta^I \mid \exists b : (a, b) \in \{(a_roof, an_oak), (a_bridge, an_oak), (a_bridge, a_granite)\} \wedge \\
&\quad b \in \{an_oak, a_granite\}\} \\
&= \{a_roof, a_bridge\}
\end{aligned}$$

On the other hand, the statement $\forall \text{hasMaterial.WoodenMaterial}$ results in $\{a_roof, a_church\}$. This might seem to be strange that a_church , that has no material, is a part of the result of the evaluation of the given expression. The following equation gives the explanation:
 $\forall \text{hasMaterial.WoodenMaterial} =$

$$\begin{aligned}
&= \{a \in \Delta^I \mid \forall b : (a, b) \in \text{hasMaterial}^I \rightarrow b \in \text{WoodenMaterial}^I\} = \\
&= \{a \in \Delta^I \mid \forall b : (a, b) \in \{(a_roof, an_oak), (a_bridge, an_oak), (a_bridge, a_granite)\} \rightarrow \\
&\quad b \in \{an_oak\}\} = \\
&= \{a \in \{a_roof, a_bridge, a_church, an_oak, a_granite\} \mid \\
&\quad \forall b : (a, b) \in \{(a_roof, an_oak), (a_bridge, an_oak), (a_bridge, a_granite)\} \rightarrow b \in \{an_oak\}\} = \\
&= \{a_roof, a_church\}
\end{aligned}$$

The reason why the a_church is a part of the result is because it belongs to Δ^I and it does not have any associated *Role* in hasMaterial^I at the same time. Interpretation of $\forall R.C$ is defined by implication and because a_church has no associated *Role* in hasMaterial^I it is evaluated as part of the answer. This is an example of the open world assumption.

2.2.4 Queries

Any knowledge representation system would be of little use if there was no way of extracting data out of it. A description logic knowledge base contains explicit data, the one that the designer created, and implicit data, one that can be inferred by a reasoner from explicit data. There are many types of queries but only three of them are introduced here and this introduction is very brief. For more information please see [2].

For all inferences introduced here let T be a TBox, $.^I$ an interpretation function and C, D Concepts.

The first type of inference is a satisfiability check. As the name suggests, this type of query checks whether the given knowledge base is satisfiable, i.e. if every *Concept* contained in TBox has a non empty interpretation set. This type of query is very useful during the design of the knowledge base and is also used in other types of queries that can be transformed into a satisfiability check. This type of query can be formally expressed as $\exists.^I : \forall C \in T : C^I \neq \emptyset$.

The second type of inference is called a subsumption problem. This inference checks whether the all interpretations of one *Concept* (C) are subset of the all interpretations of other *Concept* (D), formally if $C^I \sqsubseteq D^I$ for all possible interpretation functions $.^I$. This way it is possible to sort *Concepts* by their interpretation sets and it can be used for example in the optimization of execution of queries. This inference can be transformed into satisfiability check by checking if the following TBox is not satisfiable: $T' = T \sqcup (\neg D \sqcap C)$

The third type of query introduced here is called an instance check. This query answers true or false based on whether an *Individual* a belongs to certain *Concept* C . Formally whether the following statement is satisfiable: $T' = T \sqcup C(a)$.

2.2.5 Intervention Use Case - Formal definition

This section defines the Intervention use case specified in the chapter 1 using description logic. The definition by description logic is useful because it is a formal definition and is independent of any particular technology. The following text slightly abuses notation by mutually exchanging *Individuals* and *Concepts* in plain text. For example when the text describes a relation between material and component, it describes it as "Component that has associated Material" although it should be "*Individuals* that are instances of *Component* are associated with *Individuals* that are instances of *Material*". The later case would make the text too long and hard to read so this shortcut is used. Also, when the text is presenting examples of some *Concepts*, like the example of Wood is oak, those examples are particular individuals that are instances of the concept.

The first step in the process of definition is to identify *Atomic Concepts* and *Atomic Roles*. The interpretation of those *Concepts* is defined by the designer of represented data. Such *Atomic Concepts* and *Atomic Roles* are then used for building more complex *Concepts*. The relations between *Atomic Concepts* and *Atomic Roles* are visualized on figure 2.1, where rectangles represent *Atomic Concepts* and edges represent *Atomic Roles*.

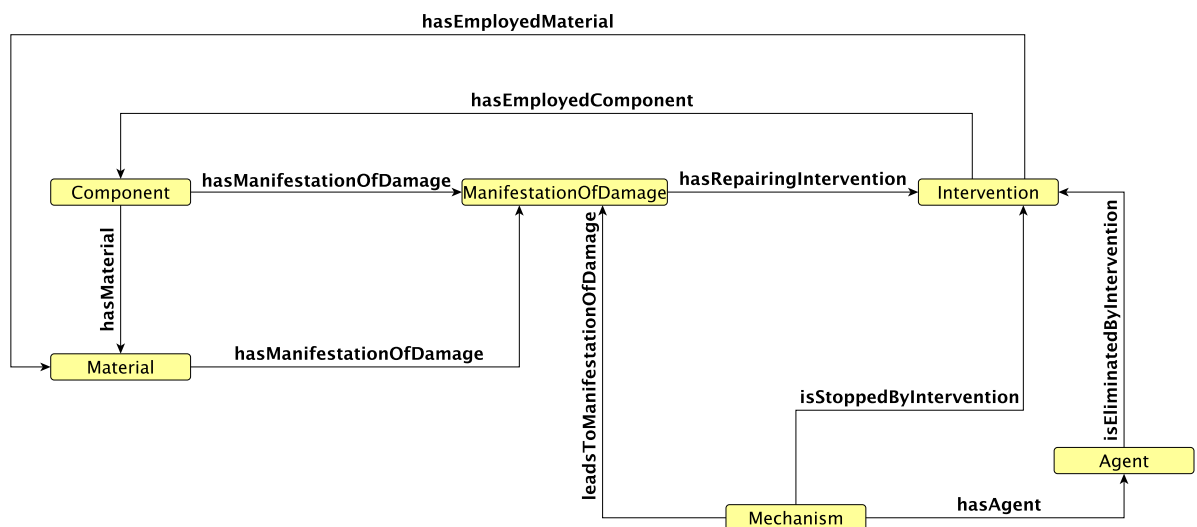


Figure 2.1: Visualization of *Concepts* of Intervention Use case and *Roles* between them.

The *Component* is a *Concept* that represents a set of all cultural components, like churches and bridges. *Material* is a *Concept* that defines a set of all materials, like wood or stone. The relation between *Component* and their *Material* is defined by a *Role* *hasMaterial*. Not all individuals that belong to *Component* have to have an associated *Material* as this knowledge might be unknown at the time the knowledge base is specified. Those subclasses are not listed here but they can be found in the [7] ontology (explained later).

ManifestationOfDamage defines a *Concept* of all visible or somehow detectable manifestations of damage to some *Components* and *Materials*. Examples of individuals that belong to this *Concept* can be crack and lichen.

2.2. DESCRIPTION LOGIC

Intervention defines a *Concept* of all repairing interventions that either (i) directly stop some manifestation of damage, or (ii) stop mechanism causing some manifestation of damage, or (iii) stop an agent causing a mechanism. Examples of such a repairing interventions can be cleaning or desalination. A relation between *ManifestationOfDamage* and its repairing *Intervention* is represented by *hasRepairingIntervention Role*. Relations between *Intervention* and *Mechanism* or *Agent* are explained in the next paragraph. It is possible that some *Intervention* requires additional *Material* or *Component* to be applied. For example cleaning would require water to be applied. Such relations are expressed by *hasEmployedMaterial* and *hasEmployedComponent Roles*.

Mechanism Concept defines a set of all mechanisms that lead to certain *ManifestationOfDamage*. Example of such a mechanisms are condensation or oxidation. The relation between *Mechanism* and *ManifestationOfDamage* is determined by *Role leadsToManifestationOfDamage*. *Mechanism* can be prevented by certain *Intervention*. The relation that defines this prevention is *Role isStoppedByIntervention*.

The last *Concept* that is left to define is *Agent*. *Agent* represents a set of all carrying factors (agents) that have some associated mechanism (leading to some manifestation of damage) that they causes. Example of such an agent is moisture that causes oxidation (that leads to erosion). The association between *Mechanism* and *Agent* is represented by *hasAgent Role*. *Agent* can be eliminated by *Intervention* and this information is encoded in *isEliminatedByIntervention Role*. Example of such an elimination is jacketing that is eliminating moisture.

All *Concepts* defined in this paragraph can have appropriate subsumed concepts, for example *Component* can have a *Construction* \sqsubseteq *Component* and *Material* can have a *Wood* \sqsubseteq *Material*.

With the *Concepts* and *Roles* above it is possible to define new *Concepts* that simplifies definitions of the *Intervention* use case:

- $\text{ComponentWithMaterial} = \text{Component} \sqcap \exists \text{hasMaterial}.\text{Material}$
- $\text{MaterialWithComponent} = \text{Material} \sqcap \exists \text{hasMaterial}^{-1}.\text{Component}$
- $\text{RelevantMoD} = \exists \text{hasManifestationOfDamage}^{-1}.\text{ComponentWithMaterial} \sqcup \exists \text{hasManifestationOfDamage}^{-1}.\text{MaterialWithComponent}$

With the definition above it is already possible to define main *Concepts* that contain the elements needed for the *Intervention* use case. The first *Concept* to define is *RepairingIntervention*. This *Concept* contains all reparative interventions that are associated with a manifestation of damage that is associated with some component or material. *RepairingIntervention Concept* can be seen as an intersection of *Material*, *Component*, *ManifestationOfDamage* and *Intervention*. This notion of intersection was introduced in the section 2.1. The *Concept* *RepairingIntervention* can be defined as follows:

- $\text{RepairingIntervention} = \text{Intervention} \sqcap \exists \text{hasRepairingIntervention}.\text{RelevantMoD}$

The second *Concept* to define is *MechanismIntervention*. This *Concept* represents a set of all individuals that prevent some mechanism that leads to *RelevantMoD*. *MechanismIntervention Concept* can be seen as an intersection of *Material*, *Component*, *ManifestationOfDamage*, *Intervention* and *Mechanism* axes. To simplify the definition of *MechanismIntervention* a new *Concept*, *RelevantMechanism*, that represents a set of all mechanism

that leads to RelevantMoD, is defined. MechanismIntervention *Concept* is defined as follows:

- RelevantMechanism = Mechanism \sqcap \exists leadsToManifestationOfDamage.RelevantMoD
- MechanismIntervention = Intervention \sqcap \exists isStoppedByIntervention.RelevantMechanism

The last *Concept* that is left to define for the Intervention use case is AgentIntervention. This *Concept* represents a set of all interventions that are stopping some agent. A stopped agent has to be associated with some RelevantMechanism by hasAgent *Role* . The formal definition of AgentIntervention follows:

- MechanismAgent = \exists hasAgent⁻.RelevantMechanism
- AgentIntervention = Intervention \sqcap \exists isEliminatedByIntervention.MechanismAgent

2.3 RDF(S), OWL2 And Semantic Web

This section introduces two technologies related to Semantic Web. Those technologies are used in the provided implementation of the thesis and for this reason they are introduced briefly here. The first technology to be mentioned is called RDF(S). RDF(S) is a framework designed to describe resources and their relations (RDF(S) means Resource Description Framework (Schema)). The expressivity of RDF(S) is similar to entity-relationship or class diagrams in the sense that it establishes relations between resources. The semantics of RDF(S) is simple and is not able to perform much of inferences such as relation transitivity.

OWL2 addresses this problem. This technology is backed by $SR\mathcal{OIQ}^{(D)}$ Description Logic, which makes it able to do all inferences as in $SR\mathcal{OIQ}^{(D)}$. OWL2 is build on the top of RDF(S) so anything that is valid in RDF(S) in also valid in OWL2 (with slightly modified syntax). RDF(S) is also used for the serialization of OWL2 ontologies.

2.3.1 RDF

RDF(S) is combination of two technologies: RDF and RDFS. While RDF is aimed at establishing a simple framework for resource definition, it does not provide any way how to perform metamodeling, that is, how to state that something is a class and something is a property. This problem is addressed by RDFS which establishes simple constructs for specifying classes and their relations. The formal specifications of both technologies are defined in [9] and [11]. This section presents an intuitive explanation of RDF(S) that is needed to understand the implementation of this thesis. Not all features of RDF(S) are explained here as it is not part of this thesis. For detailed information about RDF(S) the reader is asked to consult [9], [30] and [11].

An RDF(S) document can be intuitively seen as a graph, where nodes are either (i) resources identified by their IRI; (ii) literals; or (iii) blank nodes. A resource is anything that can be identified by it's IRI. IRI (internationalized resource identifier) is a string literal similar to URI except that it allows for Unicode characters. An example of an IRI is

2.3. RDF(S), OWL2 AND SEMANTIC WEB

`http://kbss.felk.cvut.cz/ontologies/2011/monument-damage-core.owl#hasEmployedComponent.`

Literals are used to represent concrete values, like strings or numbers. Their form is "literal"^^"type" where literal is Unicode representation of a value and type references to the type of the value. An example of such a literal is "128"^^xsd:integer representing integer value of 128.

Blank nodes are special types of nodes that can represent any other resource that is unknown in the time the document is being created. They are useful, for example, in order to state that something exists but is not known yet or when it is needed express a relation that is of arity greater than two.

An RDF(S) document is composed of triples of the form subject-predicate-object. Subject and object represent two nodes of the graph that are connected by an edge labeled with a predicate. The subject can be either a resource or a blank node and the object can be either a resource, a literal or a blank node. An example visualization of such a triplet is on figure 2.2. Notice that IRI prefix `http://kbss.felk.cvut.cz/ontologies/2011/monument-damage-core.owl#` is abbreviated on the figure to `mondis:`, which is a standard way how to make IRIs shorter in RDF(S).



Figure 2.2: Example of an rdf triplet capturing relation hasMaterial between bridge and oak.

There are several syntaxes regarding how an RDF(S) document can be stored. The most simple one is called N3 [36] and has form:

```
subject1 predicate1 object1.
subject2 predicate2 object2.
...
subjectn predicaten objectn.
```

Where one line represents one statement and n is a number of statements.

RDF(S) is able to provide a simple language for metamodeling. The dictionary for such modeling is composed of the objects and predicates listed in table 2.3 with an included explanation given by description logic. Note that C , $C1$, I , R , $R1$ are symbols representing RDF(S) resources. The first column represents RDF(S) symbols. The second column declares if the symbol in the left column is an object or an predicate. The third column shows example of the usage of the symbol and the last column shows an equivalent example in description logic. The description logic explanation is meant only as an intuitive explanation but is not exact because RDF(S) allows resources to be a class and an instance of a class at the same time, which is not directly possible in description logic.

An example of the usage of RDF(S) metamodeling language is on figure 2.3. The middle column of resources represents classes and a property as is defined by the right column. The left column specifies concrete instances of the classes. An instance of a class is a resource that has an edge labeled with `rdf:type` starting in the resource and leading to a resource representing the particular class.

RDF(S) has several methods of how to extract data from a graph. Those ways are called RDF(S) entailment regimens and there are four types of them: (i) Simple entailment; (ii)

Predicate	Type	Usage	DL equivalent
<code>rdfs:Class</code>	Object	$C \text{ rdfs:type rdfs:Class.}$	C is a <i>Concept</i>
<code>rdf:type</code>	Predicate	$I \text{ rdf:type C.}$	$C(i)$
<code>rdfs:subClassOf</code>	Predicate	$C1 \text{ rdfs:subClassOf C.}$	$C1 \sqsubseteq C$
<code>rdf:Property</code>	Object	$R \text{ rdf:type rdf:Property.}$	R is a <i>Role</i>
<code>rdfs:subPropertyOf</code>	Predicate	$R1 \text{ rdfs:subPropertyOf R.}$	$R1 \sqsubseteq R$
<code>rdfs:range</code>	Predicate	$R \text{ rdfs:range C.}$	$\exists R.T \sqsubseteq C$
<code>rdfs:domain</code>	Predicate	$R \text{ rdfs:domain C.}$	$\exists R^{-}.T \sqsubseteq C$

Table 2.3: Table of RDF(S) metamodeling predicates and objects.

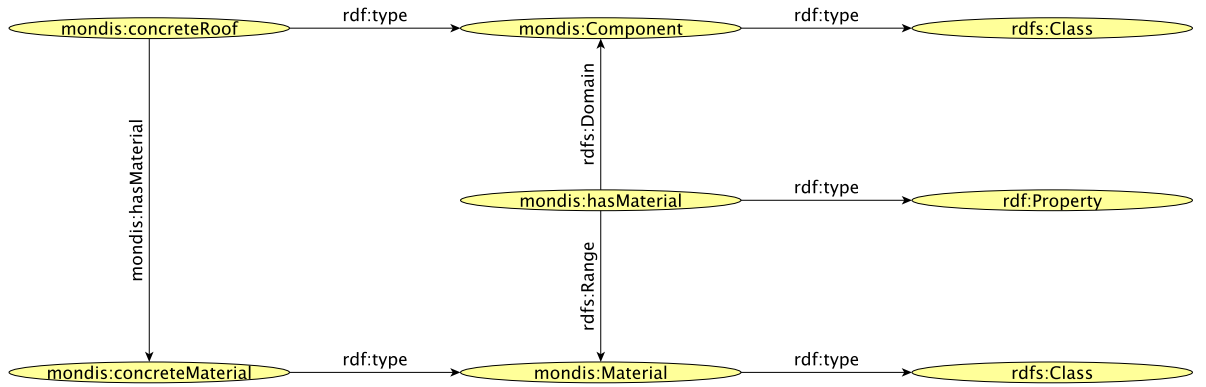


Figure 2.3: Example of an rdf graph representing Material and Component classes and their instances.

RDF entailment; (iii) RDF(S) entailment; and (iv) D entailment. Each entailment regime is explained in [30], but they are not directly used in this thesis and for this reason they are not explained here. If the reader is interested, detailed information can be found in [30].

2.3.2 OWL2

RDF(S) is able to express classes and simple relations between them, but is not able to express more complex relations like transitivity or qualified number restrictions. Such relations are possible to be expressed in the OWL2 language family [14], which is built on top of RDF(S). There are several dialects of OWL2 language differing in expressivity and therefore in reasoning complexity and decidability. OWL2 dialects, ordered by expressivity, are OWL2 (Full), OWL2 DL, OWL2 Lite. OWL2 (Full) is not a decidable language. The one that is used in this thesis is OWL2 DL, which is backed by $SR\mathcal{OIQ}^{(D)}$ description logic, which makes it decidable in exponential time. The last dialect is less expressive and therefore are not suitable for this thesis.

Because OWL2 DL is based on $SR\mathcal{OIQ}^{(D)}$ description logic, there is direct mapping between OWL2 DL constructs and description logic terms. Note that this thesis does not explain data properties, for the same reason as was in the section 2.2. OWL2 syntax used in this chapter is called Manchester syntax [26], but it is not the only possible syntax for OWL2 DL, see [14] for more information. OWL2 DL is based from the following basic building blocks.

Class

OWL2 class is an equivalent to description logic *Concept*. It is a basic building block for describing a domain. OWL2 class is not semantically equivalent to RDF(S) class in a way that RDF(S) allows a resource to be a class and individual at the same time. This is not directly possible in OWL2. OWL2 supports a mechanism that can create an individual from a class whose name is same as the class name, but such an individual is not semantically related to the class.

Let $C1$ and $C2$ be Description Logic *Concepts*. OWL2 class can be defined as a subclass of another class. This is equivalent to the $C1 \sqsubseteq C2$ in description logic. Two classes can be declared as disjoint, which means that $C1 \sqsubseteq \neg C2$. A class can also be directly stated to be equivalent to another class, which means that $C1 = C2$. An example definition of a class in Manchester syntax is:

```
Class : :Roof
  SubClassOf: :Component
  EquivalentTo: :Strecha
  DisjointClasses: :Floor
```

This class definition can be expressed in Description Logic in a TBox as:

Term in Description Logic	Description
$\text{Roof} \sqsubseteq \text{Component}$	Roof is a subclass of Component
$\text{Roof} = \text{Strecha}$	Roof and Strecha are the same class
$\text{Roof} \sqsubseteq \neg \text{Floor}$	Roof is disjoint with Floor

Individual

OWL2 individual is an equivalent to description logic *Individual*. An individual can also be called an instance of a class and can belong to more classes at the same time. An individual can be related to some other individual by some object property. This is equivalent to *Roles* in description logic. It is possible to state that some individual is different from other individual or is the same as another one. Individuals can also have properties. Properties are explained in the following paragraph. An example of definition of an individual is:

```
Individual : :PragueCastleRoof
  Types: :Roof
  DifferentFrom: :CharlesBridge
  SameAs: :StrechaPrazskehoHradu
  Facts: hasMaterial :PragueWood
```

This class definition can be expressed in Description Logic in a ABox as:

Term in Description Logic	Description
Roof(PragueCastleRoof)	PragueCastleRoof is an instance of Roof
PragueCastleRoof \neq CharlesBridge	PragueCastleRoof is different individual than CharlesBridge
PragueCastleRoof \approx StrechaPrazskehoHradu	PragueCastleRoof is the same individual as StrechaPrazskehoHradu
hasMaterial(PragueCastleRoof, PragueWood)	PragueCastleRoof is in the relation with PragueWood named hasMaterial

Object property

Object properties in OWL2 are generally statements made about two individuals describing their relation. Their equivalent in description logic is *Role*. Unlike RDF(S), OWL2 properties are much richer constructs and are able to express more information about the described domain. Basically everything that $\mathcal{SROIQ}^{(D)}$ is able to express about a relation OWL2 is able to express too. It is possible to describe characteristics of properties like transitivity, functionality and all others defined in description logic section. Like RDF(S) properties, OWL2 properties can have a domain and a range. Properties can be expressed in a hierarchy where one property is a subproperty of another. This is equivalent to *Roles* subsets in description logic. Two properties can be declared either equal or disjoint.

```

DataProperty : :hasMaterial
  Characteristics: Asymmetric, Irreflexive
  Domain: :Component
  Range: :Material
  InverseOf: :isContainedIn
  DisjointWith: :hasManifestationOfDamage
  EquivalentTo: inverse :isContainedIn
  
```

This property definition can be expressed in Description Logic in a RBox as:

Term in Description Logic	Description
<i>Asy</i> (hasMaterial)	hasMaterial is an asymmetrical relation
<i>Irr</i> (hasMaterial)	hasMaterial is an irreflexive relation
\forall hasMaterial. $\top \sqsubseteq$ Component	Domain of hasMaterial is Component
\forall hasMaterial ⁻ . $\top \sqsubseteq$ Material	Range of hasMaterial is Material
isContainedIn = hasMaterial ⁻	isContainedIn is an inverse of hasMaterial
hasMaterial = isContainedIn ⁻	hasMaterial is the same as inverse of isContainedIn

2.3.3 SPARQL

SPARQL [34] is a language similar to SQL in that it is used for retrieving data. SPARQL is aimed at retrieving data from RDF(S) graphs. Since OWL2 is usually serialized into RDF(S), it is also possible to query OWL2 ontologies. In that case SPARQL-DL is used as

```
1 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2 PREFIX mod-core: <http://kbss.felk.cvut.cz/ontologies/2011/
   monument-damage-core.owl#>

4 SELECT DISTINCT ?COMPONENT ?MATERIAL
5 WHERE {
6     ?MATERIAL rdfs:subClassOf mod-core:Material.
7     ?MATERIAL_INSTANCE a ?MATERIAL.

9     ?COMPONENT rdfs:subClassOf mod-core:Component.
10    ?COMPONENT_INSTANCE a ?COMPONENT.

12    ?COMPONENT_INSTANCE mod-core:hasMaterial
     ?MATERIAL_INSTANCE.
13 }
```

Listing 2.1: Component and Material query

a name for this language. A SPARQL query consists of triple patterns and conjunctions and junctions between them. SPARQL queries are executed against an SPARQL endpoint. A SPARQL endpoint is a service that accepts a SPARQL query string, executes it against associated RDF(S) graphs and returns the appropriate result.

The syntax and semantics of SPARQL is not fully covered in this thesis, but only relevant parts of the specification are presented here. The explanation given here is based on examples, for an exact description of SPARQL 1.1. please see [34].

The first example 2.1 selects all such components that have associated material. This query represents the intersection of material and component *TopCoordinates* as was explained in the Intuition section 2.1. The first two lines are a declaration of prefixes of used namespaces. Those namespaces are used in the associated RDF(S) graph. The fourth line declares the output variables of the query. Each variable name starts with a question mark followed by the variable's name. The **DISTINCT** keywords states that all results should be unique among the answer. The next line opens a **WHERE** clause. This clause specifies the triples that should be present in the RDF(S) graph. One triple is of the form subject-predicate-object. Variable names are replaced by actual values of matched triplets of the RDF(S) graph. Line six states that `?MATERIAL` should be a subclass of `mod-core:Material` and line seven states that `?MATERIAL_INSTANCE` should be an instance of this class. Lines nine and ten say the same thing only about `?COMPONENT` and the line twelve says that the instance of `COMPONENT` should have associated `MATERIAL` by relation `mod-core:hasMaterial`.

The next query 2.2 shows an example of a selection of components with associated material that have associated manifestations of damage. This query represents the intersection of material, component and manifestation of damage *TopCoordinates* as was explained in the Intuition section 2.1. The lines from 1 to 3 are a declaration of used namespaces as was explained in the previous example. The line 5 declares that output of this query should be values of variables `?COMPONENT`, `?MATERIAL` and `?MOD` (Manifestation Of Damage). The new syntax construct, **VALUES** keyword, is used on lines 8 and 13. Those lines say that the values of variable `?COMPONENT_RESTRICTION` are one of the

values specified in the brackets, in this case `mod-taxonomy: Tower`. The same holds for line 13 except that the restricted variable is `MATERIAL_RESTRICTION` and the value is `mod-core:Material`. The lines from 20 to 24 represents the union of two subqueries. It states that manifestation of damage can be associated with either component or material. The line 26 says that the instance of manifestation of damage belongs to class `?MOD`.

2.3.4 Intervention Case One - Semantic web representation

This subsection presents a simplified version of the [7] ontology in OWL2 language. The full ontology is part of an appendix to this thesis. This ontology defines all data needed for the Intervention use case. Data for this use case is extracted from Mondis-Ontology by SPARQL-DL queries. Those queries are also presented in this subsection.

The definition of main OWL2 classes are in the listing 2.3. Their names directly correspond to the part of the monument domain they represent. For each class there are individuals that belong to it. A visualized example of such individuals is contained in figure 2.4. Note that individuals are omitted from listing 2.3 in order to make the listing shorter and more readable. Classes and their relations correspond to the ones that are depicted on figure 2.1. Each class can be considered as a `TopCoordinate` from Intuition section. The intersection of `TopCoordinates` and their `Coordinates` are realized by SPARQL-DL queries that are presented later in this subsection.

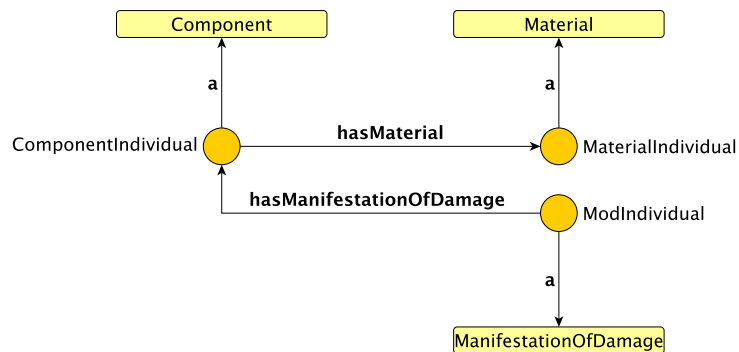


Figure 2.4: Example of Material, Component and Manifestation Of Damage representation.

```

1 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2 PREFIX mod-core: <http://kbss.felk.cvut.cz/ontologies/2011/
   monument-damage-core.owl#>
3 PREFIX mod-taxonomy:
   <http://kbss.felk.cvut.cz/ontologies/2011/
   monument-damage-taxonomies.owl#>

5 SELECT DISTINCT ?COMPONENT ?MATERIAL ?MOD
6 WHERE {

8     VALUES ?COMPONENT_RESTRICTION
9         { mod-taxonomy: Tower }
10    ?COMPONENT rdfs:subClassOf ?COMPONENT_RESTRICTION.
11    ?COMPONENT_INSTANCE a ?COMPONENT.

13    VALUES ?MATERIAL_RESTRICTION
14        { mod-core: Material }
15    ?MATERIAL rdfs:subClassOf ?MATERIAL_RESTRICTION.
16    ?MATERIAL_INSTANCE a ?MATERIAL.

18    ?COMPONENT_INSTANCE mod-core:hasMaterial
   ?MATERIAL_INSTANCE.

20    {
21        ?COMPONENT_INSTANCE
   mod-core:hasManifestationOfDamage ?MOD_INSTANCE.
22    } UNION {
23        ?MATERIAL_INSTANCE mod-core:hasManifestationOfDamage
   ?MOD_INSTANCE.
24    }

26    ?MOD_INSTANCE a ?MOD.

28 }

```

Listing 2.2: Component Material and ManifestationOfDamage query

2.3. RDF(S), OWL2 AND SEMANTIC WEB

```
Prefix: : <http://kbss.felk.cvut.cz/ontologies/2011/monument-damage-simplified.owl#>

Ontology: <http://kbss.felk.cvut.cz/ontologies/2011/monument-damage-simplified.owl>

Class : :Component

Class : :Material

ObjectProperty: :hasMaterial
  Characteristics: Asymmetric, Irreflexive
  Domain: :Component
  Range: :Material
  InverseOf: :isContainedIn

Class: :ManifestationOfDamage

ObjectProperty: hasManifestationOfDamage
  Range: :ManifestationOfDamage

ObjectProperty: hasComponentManifestationOfDamage
  SubPropertyOf: :hasManifestationOfDamage
  Domain: :Component
  Range: :ManifestationOfDamage

ObjectProperty: hasMaterialManifestationOfDamage
  SubPropertyOf: :hasManifestationOfDamage
  Domain: :Material
  Range: :ManifestationOfDamage

Class : :Intervention

ObjectProperty: hasRepairingIntervention
  Domain: ManifestationOfDamage
  Range: Intervention

Class : :Mechanism

ObjectProperty: leadsToManifestationOfDamage
  Domain: :Mechanism
  Range: :ManifestationOfDamage

ObjectProperty: isStoppedByIntervention
  Domain: Mechanism
  Range: Intervention

Class: :Agent
```

```
ObjectProperty: hasAgent
  Domain: Mechanism
  Range: Agent

ObjectProperty: isEliminatedByIntervention
  Domain: Agent
  Range: Intervention
```

Listing 2.3: Simplified MONDIS ontology

With simplified MONDIS ontology it is already possible to define SPARQL-DL queries that return such individuals that correspond to the definition of Intervention use case. The following paragraphs slightly abuse notation by mutually exchanging Individuals and Classes in plain text. The reasons are the same as were in the definition of the Intervention use case by description logic.

The first query 2.4 is used for retrieving all such interventions that have associated manifestations of damage that have associated material or component. An additional ontology is used in this query: `mod-taxonomy`. This ontology contains additional subclasses of components, materials and manifestations of damage. `Tower` is a subclass of `Component`, `Brick of Material` and `Leaning and Erosion` are subclasses of `Manifestation-OfDamage`. This ontology was not listed here as is it simple to understand and is part of the attachment to this thesis.

The second query 2.5 is used to retrieve all interventions that are stopping some mechanisms causing manifestation of damages. The manifestation of damage has to have an associated material or component. For simplicity, all restrictions are set to `UNDEF` which is a keyword that states that the restriction matches everything. The `UNDEF` can be replaced by any valid restriction.

The last query 2.6 that is left to define is retrieving such interventions that are eliminating some agents. An agent is cause of a mechanism that is in turn the cause of a manifestation of damage. Eliminating such an agent eliminates the associated manifestation of damage.

```

1  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2  PREFIX mod-core: <http://kbss.felk.cvut.cz/ontologies/2011/
   monument-damage-simplified.owl>
3  PREFIX mod-taxonomy:
   <http://kbss.felk.cvut.cz/ontologies/2011/
   monument-damage-taxonomies.owl#>

5  SELECT DISTINCT ?COMPONENT ?MATERIAL ?MOD ?INTERVENTION
6  WHERE {

8     VALUES ?COMPONENT_RESTRICTION { mod-taxonomy: Tower }
9     ?COMPONENT rdfs:subClassOf mod-core:Component.
10    ?COMPONENT rdfs:subClassOf ?COMPONENT_RESTRICTION.
11    ?COMPONENT_INSTANCE a ?COMPONENT.

13    VALUES ?MATERIAL_RESTRICTION { mod-taxonomy: Brick }
14    ?MATERIAL rdfs:subClassOf mod-core:Material.
15    ?MATERIAL rdfs:subClassOf ?MATERIAL_RESTRICTION.
16    ?MATERIAL_INSTANCE a ?MATERIAL.

18    ?COMPONENT_INSTANCE mod-core:hasMaterial
   ?MATERIAL_INSTANCE.

20    {
21        ?COMPONENT_INSTANCE
   mod-core:hasManifestationOfDamage ?MOD_INSTANCE.
22    } UNION {
23        ?MATERIAL_INSTANCE
   mod-core:hasManifestationOfDamage ?MOD_INSTANCE.
24    }
25    VALUES ?MOD_RESTRICTION {
26        mod-taxonomy: Leaning
27        mod-taxonomy: Erosion
28    }
29    ?MOD rdfs:subClassOf mod-core:ManifestationOfDamage.
30    ?MOD rdfs:subClassOf ?MOD_RESTRICTION.
31    ?MOD_INSTANCE a ?MOD.

33    VALUES ?INTERVENTION_RESTRICTION { UNDEF }
34    ?INTERVENTION rdfs:subClassOf ?INTERVENTION_RESTRICTION.
35    ?INTERVENTION_INSTANCE a ?INTERVENTION.

37    ?MOD_INSTANCE mod-core:hasRepairingIntervention
   ?INTERVENTION_INSTANCE.
38 }

```

Listing 2.4: Intervention query

```

1 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2 PREFIX mod-core: <http://kbss.felk.cvut.cz/ontologies/2011/
   monument-damage-simplified.owl>
3 PREFIX mod-taxonomy:
   <http://kbss.felk.cvut.cz/ontologies/2011/
   monument-damage-taxonomies.owl#>

5 SELECT DISTINCT ?COMPONENT ?MATERIAL ?MOD ?INTERVENTION
   ?MECHANISM
6 WHERE {
7   VALUES ?COMPONENT_RESTRICTION { UNDEF }
8   ?COMPONENT rdfs:subClassOf mod-core:Component.
9   ?COMPONENT rdfs:subClassOf ?COMPONENT_RESTRICTION.
10  ?COMPONENT_INSTANCE a ?COMPONENT.

12  VALUES ?MATERIAL_RESTRICTION { UNDEF }
13  ?MATERIAL rdfs:subClassOf mod-core:Material.
14  ?MATERIAL rdfs:subClassOf ?MATERIAL_RESTRICTION.
15  ?MATERIAL_INSTANCE a ?MATERIAL.

17  ?COMPONENT_INSTANCE mod-core:hasMaterial
   ?MATERIAL_INSTANCE.

19  {
20    ?COMPONENT_INSTANCE
   mod-core:hasManifestationOfDamage ?MOD_INSTANCE.
21  } UNION {
22    ?MATERIAL_INSTANCE
   mod-core:hasManifestationOfDamage ?MOD_INSTANCE.
23  }
24  VALUES ?MOD_RESTRICTION { UNDEF }
25  ?MOD rdfs:subClassOf mod-core:ManifestationOfDamage.
26  ?MOD rdfs:subClassOf ?MOD_RESTRICTION.
27  ?MOD_INSTANCE a ?MOD.

29  VALUES ?MECHANISM_RESTRICTION { UNDEF }
30  ?MECHANISM rdfs:subClassOf mod-core:Mechanism.
31  ?MECHANISM rdfs:subClassOf ?MECHANISM_RESTRICTION.
32  ?MECHANISM_INSTANCE a ?MECHANISM.

34  ?MECHANISM_INSTANCE
   mod-core:leadsToManifestationOfDamage ?MOD_INSTANCE.
35  ?MECHANISM_INSTANCE mod-core:isStoppedByIntervention
   ?INTERVENTION_INSTANCE.
36 }

```

Listing 2.5: Agent query


```

1  \thisfloatpagestyle{empty}
2  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3  PREFIX mod-core: <http://kbss.felk.cvut.cz/ontologies/2011/
   monument-damage-simplified.owl>
4  PREFIX mod-taxonomy:
   <http://kbss.felk.cvut.cz/ontologies/2011/
   monument-damage-taxonomies.owl#>

6  SELECT DISTINCT ?COMPONENT ?MATERIAL ?MOD ?INTERVENTION
   ?MECHANISM ?AGENT
7  WHERE {
8     VALUES ?COMPONENT_RESTRICTION { UNDEF }
9     ?COMPONENT rdfs:subClassOf mod-core:Component.
10    ?COMPONENT rdfs:subClassOf ?COMPONENT_RESTRICTION.
11    ?COMPONENT_INSTANCE a ?COMPONENT.

13    VALUES ?MATERIAL_RESTRICTION { UNDEF }
14    ?MATERIAL rdfs:subClassOf mod-core:Material.
15    ?MATERIAL rdfs:subClassOf ?MATERIAL_RESTRICTION.
16    ?MATERIAL_INSTANCE a ?MATERIAL.

18    ?COMPONENT_INSTANCE mod-core:hasMaterial
   ?MATERIAL_INSTANCE.

20    {
21        ?COMPONENT_INSTANCE
   mod-core:hasManifestationOfDamage ?MOD_INSTANCE.
22    } UNION {
23        ?MATERIAL_INSTANCE
   mod-core:hasManifestationOfDamage ?MOD_INSTANCE.
24    }
25    VALUES ?MOD_RESTRICTION { UNDEF }
26    ?MOD rdfs:subClassOf mod-core:ManifestationOfDamage.
27    ?MOD rdfs:subClassOf ?MOD_RESTRICTION.
28    ?MOD_INSTANCE a ?MOD.

30    VALUES ?AGENT_RESTRICTION { UNDEF }
31    ?AGENT rdfs:subClassOf
   mod-core:Agent.
32    ?AGENT rdfs:subClassOf
   ?AGENT_RESTRICTION.
33    ?AGENT_INSTANCE a ?AGENT.

35    ?MECHANISM_INSTANCE
   mod-core:leadsToManifestationOfDamage ?MOD_INSTANCE.
36    ?MECHANISM_INSTANCE mod-core:hasAgent
   ?AGENT_INSTANCE.
37    ?AGENT_INSTANCE
   mod-core:isEliminatedByIntervention
   ?INTERVENTION_INSTANCE.
38 }

```

Listing 2.6: Agent query

Chapter 3

Architecture

This chapter informally explains proposed architecture that solves the problems given in the previous chapters. The architecture is a set of principle design decisions [31]. Those decisions are explained in this chapter and there is a rationale given as to why this approach was used and what the advantages of it are. This chapter is not meant to describe any implementation but rather propose the basic principles of the architecture. A description of the implementation provided in this thesis is given in chapter 4.

The text in this chapter is divided into several sections and subsections. The first section gives a high-level overview about the architecture and describes its basic principles. The following sections describe each part of the architecture in greater detail and explains the relations between components of the architecture. UML diagrams are also provided to help understand the explained architecture.

3.1 Typographical conventions

The following text uses several typographical conventions to simplify reading. All **names of components** of the architecture are written in verbatim text, lower case letters (if they are not starting a statement) and words are separated by hyphens. Example of such a component is `knowledge-matrix-server`.

All the names of **classes and interfaces** are written in verbatim text, starting with upper case letter and words are separated by camel case convention. An example of such an interface is `TopCoordinate`. When the text uses a name of a class or interface in a statement as a subject or object, this subject (or object) represents an instance of that class or interface. If this notation is not clear from the text, it is explicitly stated what is a class and what is an object of the class.

Methods of an interface or class are written in verbatim text, starting with lower case letter and words are separated by camel case convention. When the name of the associated class is expressed, it precedes the name of the method and their names are separated by a dot. An example of such a method is `Coordinate.getLabel()`.

3.2 Overall architecture

The proposed architecture is based on the client-server architecture pattern. This approach allows for splitting responsibilities and therefore keeping the code clean and easily manageable. Interfaces are designed to be independent of the underlying knowledge framework. The reason for this is not to limit any implementation by a knowledge framework and let the implementers to choose any framework that would satisfy their needs. Although the interfaces are able to use any knowledge framework, there are still some limits placed on them:

1. The knowledge framework has to be able to execute queries that are represented as plain text.
2. The knowledge framework has to be able to represent and work with hierarchies of concepts.

The whole system is composed of three basic components. The first one is called `knowledge-matrix-server`. This server is responsible for managing data processing queries. The server provides read-only access to data so it can not change any data in an underlying knowledge framework which greatly simplifies implementation and allows for better scalability. The server can support more use cases that can be specified in some other code than the server's.

The second component is named `knowledge-matrix-component`. This is a set of user interfaces for `knowledge-matrix-server`. It is generally use case specific. This component is meant to be pluggable into other user interfaces in order to be able to compose the final application from several `knowledge-matrix-components`. It can be statefull but not necessarily.

The last part of the architecture is `knowledge-matrix-user-interface`. This is a component that the end user will interact with. It is composed of possibly more `knowledge-matrix-components` and it is generally statefull from the perspective of the user. Because this component is not limited by any principal design decision it is not described in this chapter, but is explained in the chapter about implementation 4.

The deployment of this architecture can be seen on figure 3.1. The `knowledge-matrix-server` can be deployed on a separate server or possibly more servers in order to increase throughput. `Knowledge-matrix-component` should be deployed on the same server as `knowledge-matrix-user-interface` and those components have to have a connection to the `knowledge-matrix-server`. Those components can be also deployed on more servers but care has to be taken as they might keep users sessions and the requests from one user must always go the the same server in that case.

Most of the interfaces and classes explained in the rest of this chapter are realized by static polymorphism (in Java it is called generics). The template parameter should be instantiated with appropriate class taken from the underlying knowledge framework. But for simplicity the template parameter is omitted in the explanation. If the reader is interested, they can find the details in the provided source codes where it is used.

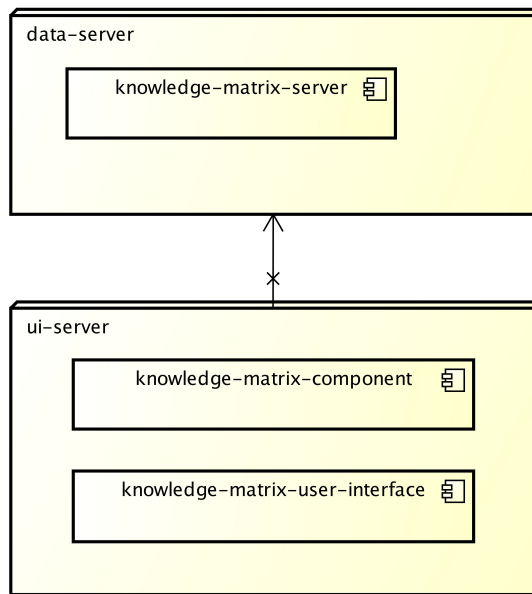


Figure 3.1: Deployment diagram.

3.3 Knowledge-Matrix-Server

This component is the main data source of this project. Its overall structure can be seen on class diagram 3.2 which depicts used classes and their relations. The server accepts users queries and returns appropriate answers. It is stateless from the perspective of the user so it can accept any query from any user. `Knowledge-matrix-component` must not be aware of any `knowledge-matrix-components` that use it. The reasons for this are scalability and independence between components of the whole system.

`Knowledge-matrix-component` can be statefull from a data layer perspective. If it is statefull or not depends on the underlying knowledge framework. At start up `knowledge-matrix-server` loads data from given data sources and preprocesses it. The preprocessed data is then kept in the memory for the whole lifetime of the server. When data is preprocessed, `Scenarios` that the `knowledge-matrix-server` will provide are instantiated. One `Scenario` represents a list of all axes that are available for queries. Each `Scenario` can access preprocessed data during its instantiation.

After data is loaded and preprocessing is done the `knowledge-matrix-server` is ready to accept queries. Communication with `knowledge-matrix-server` is done by sending requests and replying by appropriate responses. Objects that are used in `knowledge-matrix-server` are explained in the following subsections. Those objects can be split into three groups. The first is called data transfer objects (dto). Objects from this group are used to transfer data from `knowledge-matrix-server` to its client and they are independent of the underlying knowledge framework. The second group is called the query group. This group contains all classes that are related to querying the encapsulated knowledge framework. Objects from this groups are used for preparing and processing the queries. The last group is called the service group. Objects from this group are used to prepare and execute queries and create answers by transforming objects from the underlying knowledge framework into appropriate dto objects. They represent the

interface that is used by clients.

The rest of this section is organized as follows: The first subsection describes data transfer objects and their methods. The second subsection explains classes that are responsible for communication with the `knowledge-matrix-server-service` classes. The last subsection shows relations between data transfer objects and service classes. UML sequence diagrams are used to help the reader understand relationships.

3.3.1 Data Transfer Objects

Main purpose of data transfer objects is to shield users of `knowledge-matrix-server` from implementation details, especially from the underlying knowledge framework. Other reason is to make a uniform and simple interface that separates a data source and a date usage. This is done by encapsulating data retrieved from the underlying knowledge framework in objects that are independent on used knowledge framework. This is useful for example when objects from the underlying knowledge framework use transactions and it is not effective to keep the transactions open on the client side.

Coordinate, TopCoordinate

Those classes represent Coordinates and TopCoordinates as were explained in the section 2.1. The class `Coordinate` is composed of a URI, which is a unique identification of the represented object, a label, which is a string object used for the visualization of the `Coordinate`, and a reference to a `TopCoordinate` that represents main axis of the `Coordinate`. `Coordinate` class is immutable which allows it to be cached and reused later.

`TopCoordinate` class is a specialization of `Coordinate` class. `TopCoordinate` represents a main axis while `Coordinate` represents some element of a main axis. An example of usage can be the following: `TopCoordinate` object represents Material and `Coordinate` object represents some element of that Material, e.g. Wood. Wood itself can have some child `Coordinates`, like `OakWood`.

`TopCoordinate` class is a specialization of `Coordinate` class that is designed to represent a main axis. It contains one additional method, `getTopCoordinateName()`, which returns name of the `TopCoordinate` object that is used in the underlying knowledge framework query. This name has to be unique among one `Scenario` (that is explained later), but this requirement does not need to be checked by `knowledge-matrix-server` so the implementator has to take care about it. `TopCoordinate` objects can be instantiated only by a `Scenario` object and each `TopCoordinate` can be instantiated only once during the `Scenario`'s instance lifetime. This requirement ensures that only valid `TopCoordinates` are used and any possible changes in their configuration can be easily done in their associated `Scenario`.

Both classes are use case independent but instances of those classes are use case specific as they represent some concrete part of the use case. Their methods can be seen on 3.1.

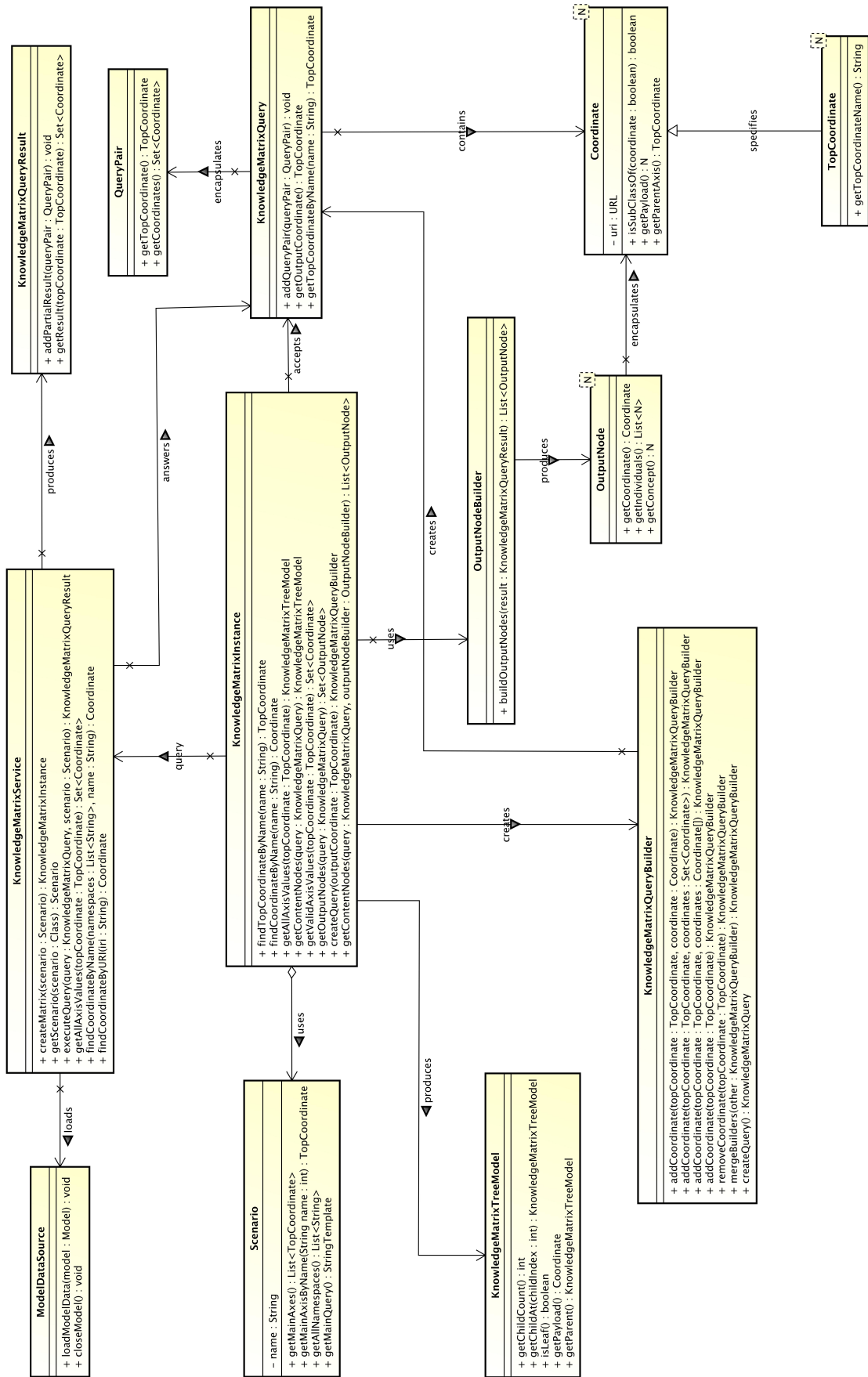


Figure 3.2: knowledge-matrix-component class diagram.

Coordinate Methods

- `boolean isSubClassOf(Coordinate coordinate)`
This method determines if the given coordinate is a subclass of this coordinate. The subclass does not need to be direct but can be represented by a chain of `Coordinates`: If `a.isSubClassOf(b)` is true and `b.isSubclass(c)` is true then `a.isSubClassOf(c)` is also true, where `a`, `b`, `c` are instances of the `Coordinate`.
- `URL getURI()`
Returns an URI that uniquely identifies an instance of `Coordinate` in the represented domain.
- `N getPayload()`
Returns underlying knowledge framework object that is encapsulated by an instance of `Coordinate`. This object should be used only in situations where there is no other way how to obtain required information about the `Coordinate` because it makes the client's code knowledge framework specific.
- `TopCoordinate getParentAxis()`
Returns main axis to which the `Coordinate` object belongs. `TopCoordinate` class is explained in the following paragraph.
- `String getLabel()`
Returns a string that should be used for rendering this `Coordinate`.

Method List 3.1: Coordinate Methods

OutputNode

The `OutputNode` class represents part of a result of a single query. This class encapsulates an associated `Coordinate` object and gives few more methods for processing the result. Those methods return additional information about the answer. Methods of `OutputNode` can be seen on 3.2.

KnowledgeMatrixTreeModel

The `Coordinate` class does not store any information about relations between instances of `Coordinates`. It can only support determination if one instance of `Coordinate` is a subclass of another. `KnowledgeMatrixTreeModel` is supposed to represent this relation. It represents this relation as a tree for which the following invariant holds: For each child (`child`), either direct or indirect, of any node (`parent`) in the tree `parent.isSubclassOf(child)` true. This class has methods listed in 3.3.

3.3.2 Query Objects

Objects from this group are used to build queries, hold results of those queries and prepare `OutputNodes`. They are similar to data transfer objects, but they differ in the di-

OutputNode Methods

- `Coordinate getCoordinate()`
Returns a `Coordinate` that represents this `OutputNode` and that was returned by the executed query.
- `List<N> getIndividuals()`
Returns a list of all individuals that belong to this `OutputNode`. Returned objects are instances of some class of the underlying knowledge framework that represents objects from the represented domain.
- `N getConcept()`
Returns an object of the underlying knowledge framework that represents this `OutputNode`. This object should be used only in situations where there is no other way how to obtain the required information about the `Coordinate` because it makes the client's code knowledge framework specific. Use `getProperty()` method with custom `OutputNodeBuilder` (explained later) instead.
- `GraphQueryResult<N> getGraph()`
Returns a graph that represents additional information about this `OutputNode`. This information can represent for example text description of the `OutputNode`, author of that description and so on.

Method List 3.2: OutputNode Methods

rection of data they pass. Data transfer objects are used to transfer data from `knowledge-matrix-server` to the client, query objects are used to transfer query parameters from the client to `knowledge-matrix-server`.

QueryPair

This class implements a tuple of `TopCoordinate` and a set of `Coordinates`. This tuple represents a main axis and a set of coordinates that belong to this axis. `QueryPair` has methods listed on 3.4.

KnowledgeMatrixQuery

This class represents a single query for the `KnowledgeMatrixService`. It is composed of a set of `QueryPairs` which specify which axes are going to be used for intersection. Any main axis in the query can be also limited by a set of `Coordinates` which specifies limits of possible values of the associated `TopCoordinate`. Those `Coordinates` are encapsulated in the associated `QueryPair`. If more `Coordinates` are provided in the query logical disjunction between those `Coordinates` is performed. Objects of `KnowledgeMatrixService` class should not be created directly, but `KnowledgeMatrixQueryBuilder` should be used for creating instances. `KnowledgeMatrixQuery`'s methods can be see in 3.5.

KnowledgeMatrixTreeModel Methods

- `int getChildCount()`
Returns a number of children of this particular node.
- `KnowledgeMatrixTreeModel getChildAt(int index)`
Returns a child at the given index. If an invalid index is provided (e.g. `index < 0` or `index >= getChildCount()`) an exception is thrown.
- `KnowledgeMatrixTreeModel getParent()`
Returns parent of this tree. For any `KnowledgeMatrixTreeModel` obtained by `KnowledgeMatrixTreeModel child = anyTree.getChildAt(i)` must hold `child.parent == anyTree` where `anyTree` and `child` are instances of `KnowledgeMatrixTreeModel` and `i` is a valid index.
- `boolean isLeaf()`
Returns true if and only if `anyTree.getChildCount() == 0` where `anyTree` is an instance of `KnowledgeMatrixTreeModel`.
- `Coordinate getPayload()`
Returns the encapsulated `Coordinate` object.

Method List 3.3: KnowledgeMatrixTreeModel Methods

QueryPair Methods

- `TopCoordinate getTopCoordinate()`
Returns encapsulated `TopCoordinate`.
- `Set<Coordinate> getCoordinates()`
Returns a set of encapsulated `Coordinates` that belong to the encapsulated `TopCoordinate`.

Method List 3.4: QueryPair Methods

KnowledgeMatrixQueryBuilder

`KnowledgeMatrixQueryBuilder` should be used for creating objects of `KnowledgeMatrixQuery`. This class is an implementation of the builder design pattern. This design pattern is a convenient way how to create a `KnowledgeMatrixQuery` instance and separate the implementation and the interface. The instances of this builder shall be created and returned by `KnowledgeMatrixInstance`. `KnowledgeMatrixQueryBuilder` has methods listed on 3.6.

KnowledgeMatrixQueryResult

This interface represents a result of executed `KnowledgeMatrixQuery`. It encapsulates all rows that are returned by the underlying knowledge framework. One row is

KnowledgeMatrixQuery Methods

- `void addQueryPair(QueryPair pair)`
Adds one `QueryPair` to the `KnowledgeMatrixQuery` instance. If more `QueryPairs` with the same `TopCoordinate` are added their `Coordinate` sets are unioned.
- `TopCoordinate getOutputCoordinate()`
Returns the `TopCoordinate` that represents result of this query. This method may be unused if an `OutputNodeBuilder` is used (explained later).
- `TopCoordinate getTopCoordinateByName(String name)`
Returns an `TopCoordinate` that was added to the query by `addQueryPair()` method that has the given name.

Method List 3.5: KnowledgeMatrixQuery Methods

KnowledgeMatrixQueryBuilder Methods

- `KnowledgeMatrixQueryBuilder addCoordinate(TopCoordinate topCoordinate, Coordinate coordinate)`
Adds the given `TopCoordinate` that is limited by the given `Coordinate` to the query.
- `KnowledgeMatrixQueryBuilder addCoordinate(TopCoordinate topCoordinate, Set<Coordinate> coordinates)`
Adds the given `TopCoordinate` that is limited by the given set of `Coordinates` to the query.
- `KnowledgeMatrixQueryBuilder removeCoordinate(TopCoordinate topCoordinate)`
Removes given `TopCoordinate` and all its associated `Coordinates` from the query.
- `KnowledgeMatrixQueryBuilder mergeBuilders(KnowledgeMatrixQueryBuilder other)`
Merges given `KnowledgeMatrixQueryBuilder` to this `KnowledgeMatrixQueryBuilder`. This operation is equivalent to adding all `TopCoordinates` and `Coordinates` from the other `KnowledgeMatrixQueryBuilder` to this.

Method List 3.6: KnowledgeMatrixQueryBuilder Methods

presented as a set of `Coordinates`. Rows are sorted the in same order as they were returned by the underlying knowledge framework. This interface is mainly meant to be used by `OutputNodeBuilder` that creates `OutputNodes` from it. `KnowledgeMatrixQueryResult` has methods listed in 3.7.

KnowledgeMatrixQueryResult Methods

- `void addRow (Set<Coordinate> row)`
This method adds one row from executed query to the instance of this object. It is supposed to be executed by KnowledgeMatrixService when the the underlying knowledge framework is queried.
- `List<Coordinate> getSpecificResult (TopCoordinate topCoordinate)`
Returns a list of Coordinates that are associated with given TopCoordinate.
- `Iterator<Set<Coordinate> > iterator ()`
Returns an Iterator that gives sequential access to retrieved rows. One row is represented as a set of Coordinates.

Method List 3.7: KnowledgeMatrixQueryResult Methods

OutputNodeBuilder

The OutputNodeBuilder is used to transform KnowledgeMatrixQueryResult into a list of OutputNodes. This transformation can process all rows of KnowledgeMatrixQueryResult and perform additional operations on it including additional queries. An example of such a usage is building OutputNodes for Interventions, where an Intervention is associated with some Material that is required to perform that intervention, and the OutputNodeBuilder can be used for the additional query. Methods of OutputNodeBuilder are listed in 3.8.

OutputNodeBuilder Methods

- `List<OutputNode> buildOutputNodes (KnowledgeMatrixQueryResult result, KnowledgeMatrixService service)`
This method accepts an instance of KnowledgeMatrixQueryResult and KnowledgeMatrixService and produces list of OutputNodes.
- `TopCoordinate getMainOutputCoordinate ()`
Returns a TopCoordinate that should be encapsulated by all OutputNodes produced by buildOutputNodes () method.

Method List 3.8: OutputNodeBuilder Methods

3.3.3 Service Objects

Service objects are objects that directly interacts with the underlying knowledge framework. They also represent an interface between the user and the underlying knowledge framework and they create instances of data transfer objects. Generally said, they contain all the data related logic of this whole project.

KnowledgeMatrixService

The `KnowledgeMatrixService` class is a component encapsulating the underlying knowledge framework. It is able to construct `KnowledgeMatrixQueryResult` by querying the underlying knowledge framework. This request is represented by a `KnowledgeMatrixQuery` object and a `Scenario` object. The request is then transformed to the query string for the underlying knowledge framework. The transformation is done by setting appropriate attributes of the `StringTemplate` obtained from the `Scenario` object. `KnowledgeMatrixService` is parametrized by a list of `ModelDataSources` (objects that represent sources of data) and a list of `Scenarios` that implement supported use cases.

All string queries to the underlying knowledge framework must go only from this class, i.e. is it not allowed to perform any string based call to the knowledge framework outside this class. This restriction reduces dependency on the underlying knowledge framework as it is called from only one place. Reduced dependency simplifies modifications of the code of whole `knowledge-matrix-server` and possible bug fixing. `KnowledgeMatrixService` has methods listed in 3.9.

KnowledgeMatrixInstance

`KnowledgeMatrixInstance` is an interface that is used to communicate with the `KnowledgeMatrixService`. An instance of this interface is user specific because it holds the user's `Scenario` as well as a reference to the `KnowledgeMatrixService` instance. The object of this class should be stored on the client side as it keeps state (reference to the used `Scenario` and `KnowledgeMatrixService`). It is also responsible for closing connections between data transfer objects and the underlying knowledge framework. This is done in order to prevent any unpredictable interactions between the user and the underlying knowledge framework. `KnowledgeMatrixInstance` has methods listed in 3.10.

Scenario

An object of this interface represents one use case, e.i. is use case specific. The use case is defined by a list of main axes which are used to make queries. The main axis is defined by its associated `TopCoordinate`. Any supported main axis can be found by its `TopCoordinate`'s name.

The class itself may or may not be use case specific. This decision is left for an implementation. A `Scenario` object is represented by its name, the list of main axis, a list of all used namespaces and a `StringTemplate` containing a query for the underlying knowledge framework. Name of the `Scenario` is used for identification of some concrete `Scenario`. Any supported scenario can be acquired by its name or by its `Class` object.

The `StringTemplate` contained by the `Scenario` represents a resource for creating queries for the underlying knowledge framework. The `StringTemplate` must be customizable by any given set of `TopCoordinates` defined by the associated `Scenario`. This set defines the intersection of the main axis as was explained in the chapter 2, section Intuition 2.1. The methods of `Scenario` are listed in 3.11.

ModelDataSources

This interface represents a single data source for KnowledgeMatrixService. Its methods are executed by KnowledgeMatrixService when KnowledgeMatrixService.loadData() method is called. ModelDataSource is responsible for adding data to KnowledgeMatrixService. The interface has two methods listed in 3.12.

3.3.4 Class relationships

This subsection shows and describes some examples of possible scenarios of the communication of objects from the previous subsections. Those examples are aimed to help the reader to understand relationships between classes, but not to give a comprehensive list of all possible interactions. The examples are expressed as UML sequence diagrams with brief textual description. The first example is about knowledge-matrix-server initialization. The second example describes getting axes and their values and the last example shows process of placing a query and retrieving OutputNodes.

Knowledge-Matrix-Server initialization

Knowledge-matrix-server initialization is shown on figure 3.3. Data loading is done in inverse of control fashion where the ModelDataSources are responsible for setting the knowledge-matrix-server data model. ModelDataSource accepts the data model of knowledge-matrix-server and adds data to it independently on knowledge-matrix-server. Kms-user represents a user of knowledge-matrix-server that performs initialization. A list of ModelDataSources is passed from the kms-user to the instance of KnowledgeMatrixService. KnowledgeMatrixService closes previous data sources (if such sources exists) and then it creates a new data model. Then it calls loadModelData() method on each of passed ModelDataSource and this method performs data loading to the newly created model. The general way how to add data from ModelDataSource to KnowledgeMatrixService is to create a local data model in the ModelDataSource and then add the data to this local data model. When the local data model is ready it is added to the KnowledgeMatrixService's data model. The creation of the local data model may be skipped if the data source allows it and then data is added directly to KnowledgeMatrixService's data model. When the data from all ModelDataSources is initialized KnowledgeMatrixService is ready to create passed Scenarios and to create KnowledgeMatrixInstance from a particular Scenario. This process is labeled by messages numbered 2 and 3 in figure 3.3.

Getting axes and their values

Getting axes and their values is shown in figure 3.4. Return values and method parameters are omitted in the figure in order to make the figure more readable. Message number one is used for getting the TopCoordinate of some main axis. This example shows the variant where the TopCoordinate is retrieved by its name. Message number two shows how the obtained TopCoordinate is used for getting all Coordinates that are associated with the retrieved TopCoordinate. Message number two is composed of one more call to the KnowledgeMatrixService which performs a query to the underlying knowledge framework.

3.3. KNOWLEDGE-MATRIX-SERVER

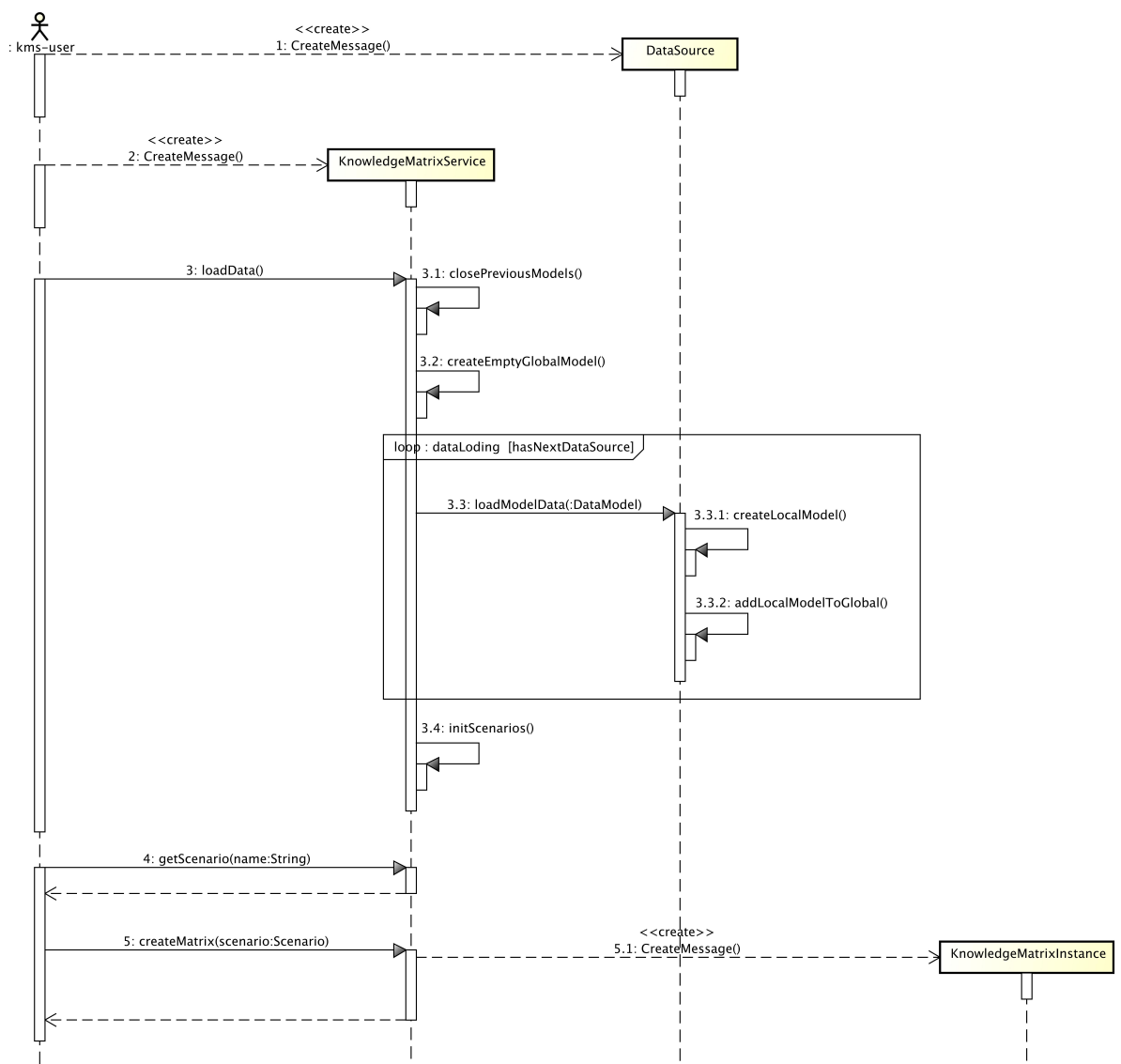


Figure 3.3: Data loading sequence diagram.

Getting valid axis values is shown by message number three. The first step in retrieving valid axis values is creating an appropriate `KnowledgeMatrixQuery`. This `KnowledgeMatrixQuery` is then passed to the `KnowledgeMatrixService` instance. `KnowledgeMatrixService` then prepares the query string from the given `KnowledgeMatrixQuery` by filling the `StringTemplate` obtained from the associated `Scenario`. When the query string is ready it is passed to the underlying knowledge framework and the `KnowledgeMatrixQueryResult` is created. When the instance of `KnowledgeMatrixQueryResult` is passed back to the `KnowledgeMatrixInstance` a `KnowledgeMatrix-TreeModel` instance is created and returned to the originator of the whole query.

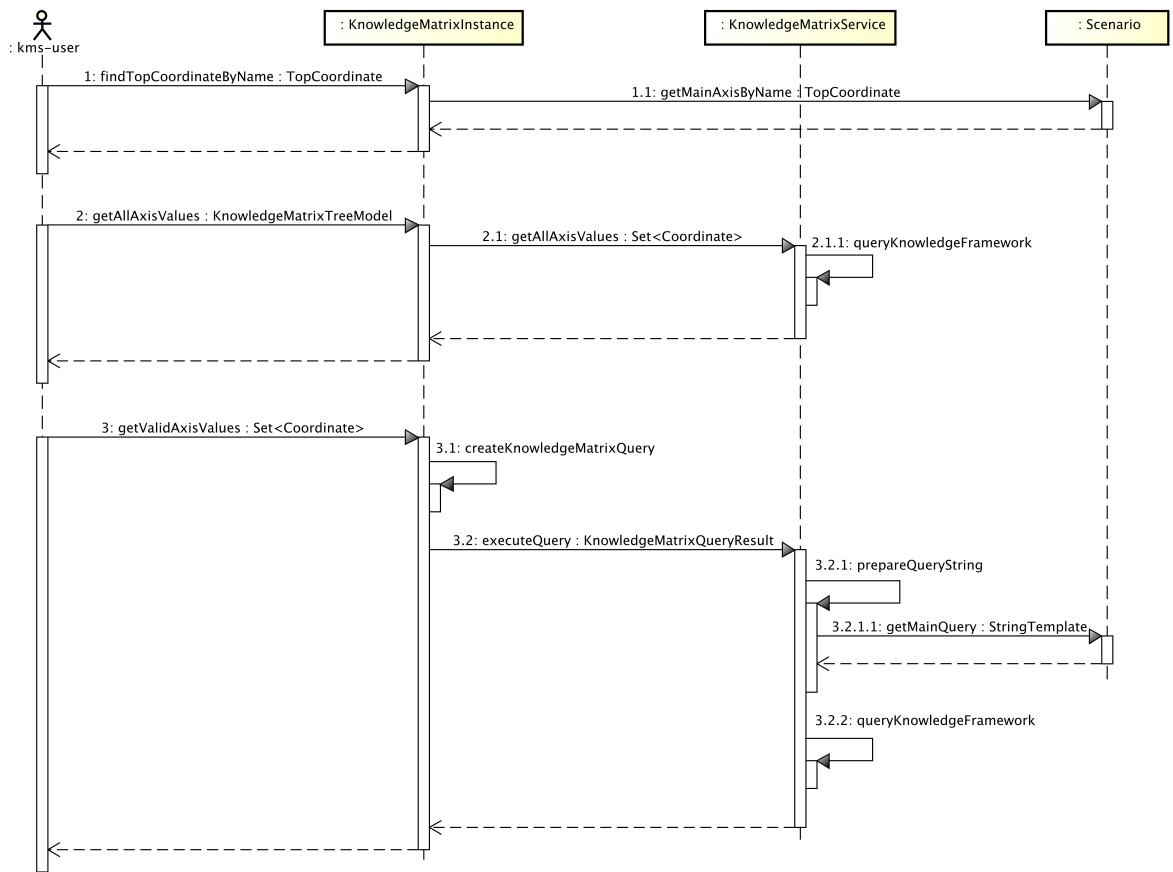


Figure 3.4: Getting axis and their values

Retrieving output nodes

This diagram shows a situation when the user has already initialized `knowledge-matrix-server`, has already retrieved `KnowledgeMatrixInstance` and is ready to make queries. An example of retrieving output nodes is shown in figure 3.5. Return values and method parameters are omitted in the figure in order to make the figure more readable. The first step is to obtain an instance of `KnowledgeMatrixQueryBuilder`. This instance is created by the `KnowledgeMatrixInstance.createQueryBuilder()` as shown in the figure by message number one. When the user has the instance they can specify the `TopCoordinates` and `Coordinates` which is they interested in. An example of this process is represented by messages number two and three. When the specification is done the user creates a `KnowledgeMatrixQuery` object by the method `KnowledgeMatrixQueryBuilder.createQuery()`, message number four. The query object then can be passed back to the `KnowledgeMatrixInstance` by the `getOutputNodes()` method. This call starts the process of retrieving data from `knowledge-matrix-server` and an example of it is represented by message number five. The process is composed of two subroutines. The first subroutine prepares and executes the query string for the underlying knowledge framework and the second transforms the retrieved `KnowledgeMatrixQueryResult` into more specific `OutputNodes`. The first subroutine, message number 5.1, is composed of two other subroutines. Message number 5.1.1 prepares the query string from the template stored in the given `Scenario` and fills it with values that are

specified in the `KnowledgeMatrixQuery`. When this step is done and the query string is ready, the underlying knowledge framework is queried and the answer is stored in an instance of `KnowledgeMatrixQueryResult`, as is shown by message number 5.1.2. This object is then passed back to the `KnowledgeMatrixInstance` where it is transformed into a list of `OutputNodes` (message number 5.2). This transformation is done in order to shield the user from the complicated `KnowledgeMatrixQueryResult` and prepare the `OutputNodes` as it is required by the use case in question.

3.4 Knowledge-Matrix-Component

`Knowledge-matrix-component` represents a set of pluggable user interface components for the `knowledge-matrix-server`. This approach is used in order to make the final application easily configurable for required use cases. Components may or may not be use case specific. This decision is left to the designers and implementors of a particular component. Since there is no general description of the `knowledge-matrix-component` this section introduces an example of the three components that are implemented as a part of this thesis

The first component represents the knowledge matrix visualization and is named `TreeMatrixComponent`. This visualization is done as a matrix of `OutputNodes` which represents the intersection of two or more axes. The second component is used for the visualization of a particular `OutputNode` as was specified in section 2.2.5 and is termed `infocard`. This visualization represents a particular `OutputNode` in greater detail and represents particular information about the given `OutputNode`. The third component is called `auto-complete-tree`. This component was adapted from [25]. The purpose of this component is to show a tree and let the user select any subset of nodes of this tree and help the user to filter those nodes.

For an easier explanation of all the implemented components, several figures are used. Those figures are not meant to describe precisely the look and feel of the explained features, but rather to help the reader to understand explained behavior and purpose. The figures are taken from the implementation that was done as a part of this thesis.

3.4.1 Tree-Matrix-Component

This component implements the visualization of a matrix that represents the intersection of two particular axes. Other axes can be specified as well, but their intersection can not be directly shown by this component. An example of how this component could look can be seen in figure 3.6. Axes are represented by instances of `GeneralAxis` that are composed of one or more instances of `HeaderObjects`. `HeaderObject` is a class that encapsulates `TopCoordinate`. The axes have to be defined during the instantiation of the component. This component is use case independent and therefore can be reused in more use cases.

Axes are divided into two groups: One group is called `drillable` and is designed to traverse `KnowledgeMatrixTreeModel`. `KnowledgeMatrixTreeModel` is represented by a list of nodes and a parent of those nodes. The nodes themselves are represented by instances of `KnowledgeMatrixTreeModel`. When the user clicks on a shown node this node becomes the new parent and children of this node are shown as the new list (if the clicked node has any children). This action is called `drilling` and because of it this

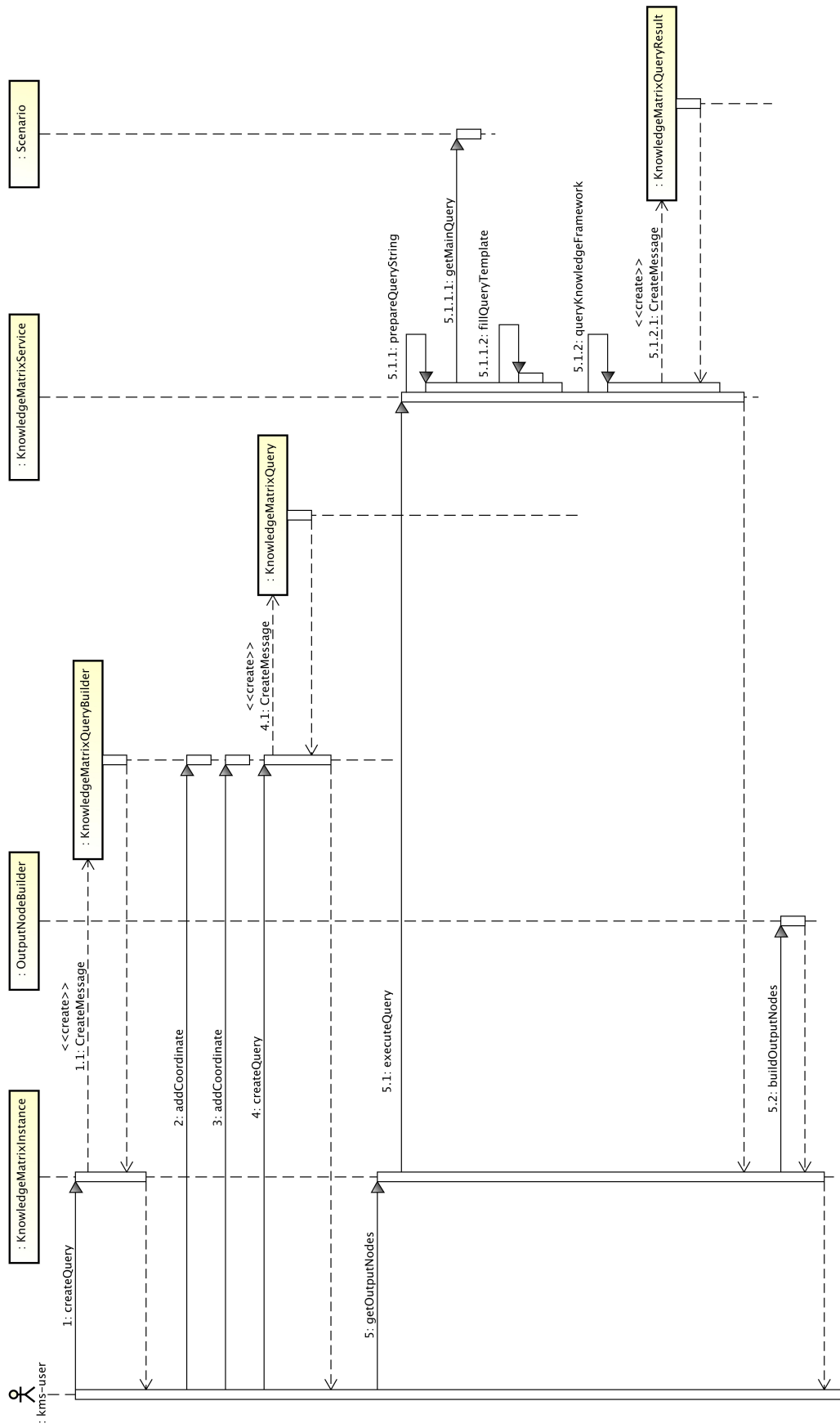


Figure 3.5: Example of getting content nodes

3.4. KNOWLEDGE-MATRIX-COMPONENT

Manifestation of damage /	Manifestation of damage	<u>Intervention</u>	<u>Mechanism</u>	<u>Agent</u>
<u>Component Manifestation Of Damage</u>		<ul style="list-style-type: none"> • <u>Repointing</u> • <u>scuci-cuci</u> 	<ul style="list-style-type: none"> • <u>Stitching</u> 	<ul style="list-style-type: none"> • <u>functional change intervention</u>
<u>Material Manifestation Of Damage</u>		<ul style="list-style-type: none"> • <u>water cleaning</u> 		<ul style="list-style-type: none"> • <u>installation of water proofing course</u> • <u>installation of drainage system</u> • <u>poulticing</u> • <u>application of sacrificial layer</u>

Figure 3.6: Example of possible state of tree-matrix-component.

group is called drillable. Besides this behavior, the actual path from the root node to the actual parent is also shown. This is placed next to the table to help the user with orientation. This path is also clickable so the user can return to any `KnowledgeMatrixTreeModel` that she already went through. When the user clicks on a node in the path the clicked `KnowledgeMatrixTreeModel` becomes the new root of the actual path. The size of the path can be limited to a given length (in the sense of visualized nodes). When the number of visualized nodes is greater than this limit (named n), only the first and last n nodes are visible and other nodes are replaced by dots. The drillable axis is the left one on the figure 3.6.

The second type of axis is called filterable. This type of axis is designed to filter values of certain `TopCoordinates` where each `TopCoordinate` represents one main axis. Filtering is done by selecting a subset of `Coordinates` that are valid for a certain axis. This selection is done by `auto-complete-tree` that is described in section 3.4.3. When the set is selected `knowledge-matrix-component` shows only those `OutputNodes` that are returned by method `KnowledgeMatrixInstance.getOutputNodes()`. The method call is parametrized by the selected set. The filterable axis is the top one in figure 3.6.

The general UML diagram of `knowledge-matrix-component` can be seen in figure 3.7. The design approach used for this component is model-view-controller. This approach allows for the splitting of the data model, data controller and view renderer. This decomposition greatly simplifies the development and testing of the component. The data model is represented by `knowledge-matrix-server`. The controller is implemented by `TreeMatrixModel` and the view can be implemented by any graphical user interface technology that satisfies the user's needs.

HeaderObject, DrillableHeaderObject, FilterableHeaderObject

The class of type `HeaderObject` represents the header of the matrix. The purpose of the header is to represent elements of the horizontal and vertical axes. Axes are specified by `TopCoordinates` that are encapsulated by `HeaderObjects`. Valid values of associated `TopCoordinates` are also encapsulated. The abstract class `HeaderObject` contains declaration of the methods listed in 3.13.

`DrillableHeaderObject` is used for representing objects of drillable axes. This object encapsulates instance of `KnowledgeMatrixTreeModel` that stores information about the tree that is represented by `DrillableHeaderObject`. It has one additional method `KnowledgeMatrixTreeModel.getNodeToDrillDown()` which returns `KnowledgeMatrixTreeModel` which is the root of the tree that is encapsulated by this class.

`FilterableHeaderObject` is used for representing objects of filterable axes. Each instance of `FilterableHeaderObject` represents one main axis and contains all valid values for this axis. The user can select any subset of those values and those values are used for filtering the content of the intersection with this main axis. For a simpler selection of the values, this class stores an `auto-complete-tree` model that helps the user with the selection of required nodes (see 3.4.3). This class also stores an adapter class between `KnowledgeMatrixTreeModel` and a class that is used for representing trees in some graphical user interface framework that is used. The adapter allows the easy rendering of the encapsulated `KnowledgeMatrixTreeModel`.

GeneralAxis, DrillableAxis, FilterableAxis

These classes are used to represent axes (in graphical sense) in the `knowledge-matrix-component`. The most general form is called `GeneralAxis` and it defines only one method: `List<HeaderObject> getHeaderElements()`. This method returns a list of encapsulated `HeaderObjects` which represents this axis. The returned list should be implemented as unmodifiable.

`DrillableAxis` stores beside the list of `HeaderObjects` the root of the current tree represented by the `KnowledgeMatrixTreeModel` object and it also stores the path from the root of main axis to the the actual root. The label of the actual root of `DrillableAxis` is shown in the first column of the first row of the matrix. The `DrillableAxis` is composed of the methods listed in 3.14.

`FilterableAxis` implements `GeneralAxis` and it contains a list of `DrillableHeaderObjects` where each `DrillableHeaderObjects` represents one row (or column) of the matrix.

TreeMatrixModel

`TreeMatrixModel` controls whole interaction between the user and `knowledge-matrix-server`. This interaction is done either by direct communication with `TreeMatrixModel` or by horizontal or vertical axes that are stored in objects of `GeneralAxis`. Direct communication can set any restriction on any axis that is supported by the used `knowledge-matrix-server`. This is useful because `TreeMatrixModel` can hold only two axes and by this way it is possible to perform the intersection with an arbitrary number of axes. The interaction with the model through the use of horizontal or vertical

axes is used to set limits of the visualized axes. `TreeMatrixModel` class has the methods listed in 3.15.

3.4.2 Info-Card

This is a very simple component that is composed of only one screen. The purpose of this component is to present detailed information about a particular `OutputNode`. The `OutputNode` object is passed as an argument to the constructor and relevant data fields are read from it and are visualized. The figure of visualization of such a component can be seen in figure 3.8.

3.4.3 Auto-Complete-Tree

`Auto-complete-tree` is designed to help the user with the selection of nodes of a given `KnowledgeMatrixTreeModel`. This feature is useful if there is a huge number of nodes and user can be easily confused. Nodes may for example represent valid values of some axis (`TopCoordinate`). An example of a usage of `auto-complete-tree` can be seen in figure 3.9. `Auto-complete-tree` is able to filter nodes of encapsulated `KnowledgeMatrixTreeModel` by a string that is typed by the user (See string "Se" at the top of the figure 3.9). All nodes that do not start with the typed string or are not parent of a node that starts with this string are removed from the tree. This feature greatly simplifies the selection of desired nodes by the user when the `KnowledgeMatrixTreeModel` contains a lot of nodes. Many users are familiar with this kind of behavior from well known websites such as Google search. The architecture of `auto-complete-tree` component can be found in [25].

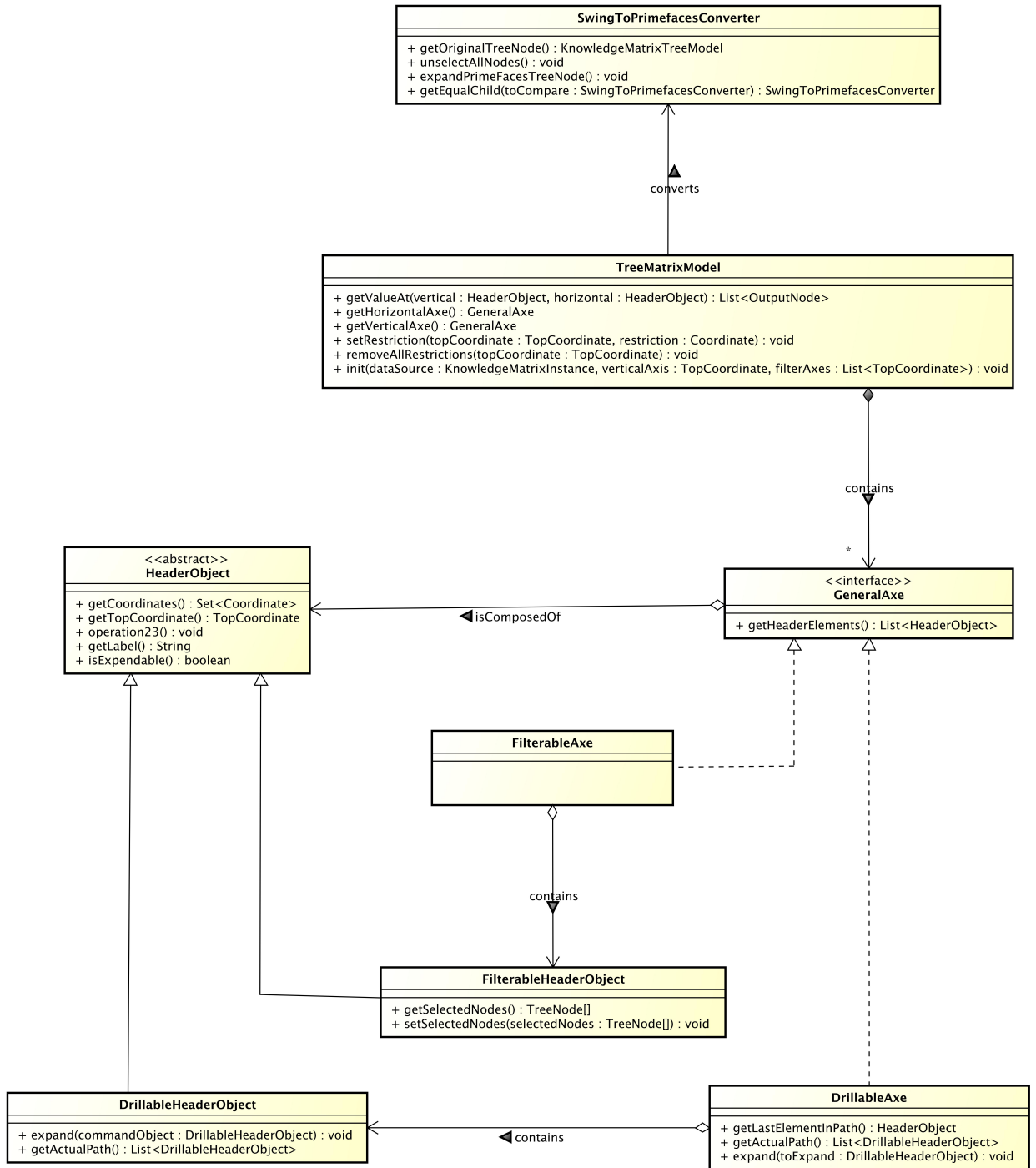


Figure 3.7: Class diagram of TreeMatrixComponent.

Intervention Name: Repointing	
Intervention Definition: Repairing the mortar joints and cracks in stone masonry is referred to as re-pointing, or tuck-pointing.	
Material: Malta	Component:
Procedure	
Advantages	
Limitations	
References	
Close	

Figure 3.8: Example of visualization of info-card.

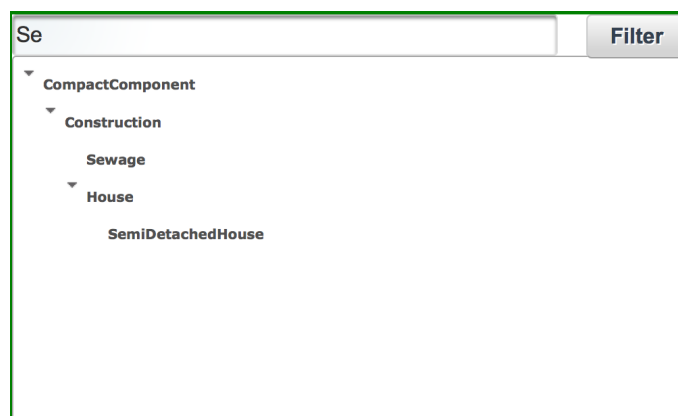


Figure 3.9: Example of possible state of auto-complete-tree.

KnowledgeMatrixService Methods

- `void loadData(List<ModelDataSource> dataSources), void loadData()`
This method loads data from the underlying knowledge framework. A list of `ModelDataSources` objects can be passed as an argument which adds data to the main model. If no argument is passed, data from the previous sources are reloaded. This method must be executed before the first query.
- `Scenario getScenario(String name), Scenario getScenario(Class<Scenario> clazz), Scenario getScenarios()`
This method returns the scenario with the given name (or Class object). If no scenario with the given name is found `ScenarioNotFoundException` is thrown. This method returns the list of all supported Scenarios if no argument is passed.
- `KnowledgeMatrixInstance createMatrix(Scenario scenario)`
This method takes a `Scenario` object that was returned by `loadData()` method. The outcome of this method is an instance of `KnowledgeMatrixInstance` which is the main interface for querying the `KnowledgeMatrixService` by the user.
- `Set<Coordinate> getAllAxisValues(TopCoordinate topCoordinate)`
This method returns the set of all `Coordinates` that are subclasses of the given `TopCoordinate` by means defined in 3.3.1. Returned set is unmodifiable.
- `KnowledgeMatrixQueryResult executeQuery(KnowledgeMatrixQuery knowledgeMatrixQuery, Scenario scenario)`
This is the main point for placing queries. It takes a `KnowledgeMatrixQuery` instance and a `Scenario` instance. The set of `TopCoordinates` is extracted from the `KnowledgeMatrixQuery` along with sets of `Coordinates` for each `TopCoordinate` limiting their possible values. Then the query for knowledge framework is composed and is executed. The algorithm for creating the query is implementation specific. The instance of `KnowledgeMatrixQueryResult` that satisfies the given query is returned.
- `GraphQueryResult executeStringQuery(String basicIRI, String queryString)`
The purpose of this method is to execute arbitrary query string and return the answer in the form of a graph. The root of this graph is determined by the `basicIRI`. The `queryString` should be written so that the underlying knowledge framework is able to construct a graph from its result.
- `Coordinate findCoordinateByName(List<String> namespaces, String name), Coordinate findCoordinateByName(String namespace, String name)`
These methods find the `Coordinate` object by the given name. If a list of namespaces is provided, those namespaces are used when locating the required `Coordinate`. If only a single namespace is provided, behavior is the same as the previous method with a list of size one.

Method List 3.9: KnowledgeMatrixService Methods

KnowledgeMatrixInstance Methods

- Scenario `getScenario()`
Returns the Scenario object that is encapsulated by the instance.
- KnowledgeMatrixTreeModel `getAllAxisValues(TopCoordinate topMatrixNode)`
Returns all possible values of given TopCoordinate that are stored in the underlying knowledge framework model of KnowledgeMatrixService. Those values may be invalid within the associated Scenario as they are not filtered.
- KnowledgeMatrixTreeModel `getValidAxisValues(TopCoordinate topMatrixNode)`
This method is similar to `getAllAxisValues()` except that returned values are always valid within the associated Scenario.
- List<OutputNode> `getOutputNodes(KnowledgeMatrixQuery query)`
Executes the given query in the context of associated Scenario and return the answer represented by the list of OutputNodes.
- KnowledgeMatrixQueryBuilder `createQuery(TopCoordinate topCoordinate)`
Creates and returns KnowledgeMatrixQueryBuilder associated with the given TopCoordinate.
- Coordinate `findCoordinateByName(String name)`, TopCoordinate `findTopCoordinateByName(String name)`
Returns Coordinate respectively TopCoordinate for the given name. If there is no such Coordinate respectively TopCoordinate then null is returned.

Method List 3.10: KnowledgeMatrixInstance Methods**Scenario Methods**

- String `getName()`
Returns name of the represented scenario.
- List<TopCoordinate> `getMainAxes()`
Returns the list of TopCoordinates that represents main axis of the Scenario.
- List<String> `getAllNamespaces()`
Returns names of all used namespaces by represented Scenario.
- StringTemplate `getMainQuery()`
Returns the object of StringTemplate that holds template for creating queries for the underlying knowledge framework.

Method List 3.11: Scenario Methods

ModelDataSources Methods

- `loadModelData (Model model)`
This method loads data from any source to the given model. This model is data source for underlying knowledge framework.
- `closeModel ()`
When data from the loaded model are no longer required this method is executed and is responsible for terminating this data source. This is useful method for example when a remote data source is used and network connection should be closed.

Method List 3.12: ModelDataSources Methods**HeaderObject Methods**

- `boolean isExpandable ()`
This method determines if the associated `HeaderObject` is able to perform any action (e.g. expansion, explained later).
- `TopCoordinate getTopCoordinate ()`
Returns the associated `TopCoordinate`.
- `Set<Coordinate> getCoordinates ()`
Returns set of all valid values for encapsulated `TopCoordinate`. For each retrieved `Coordinate` object must hold that `coordinate.getParent () = getTopCoordinate ()`.

Method List 3.13: HeaderObject Methods**DrillableAxis Methods**

- `List<DrillableHeaderObject> getActualPath ()`
This method returns the path from the main axis root to the actual root. This list should be unmodifiable.
- `HeaderObject getLastElementInPath ()`
This method returns the last element from `getActualPath ()`. It is useful in the view for easier implementation of visualizing the last element of the path.

Method List 3.14: DrillableAxis Methods

TreeMatrixModel Methods

- `List<OutputNode> getValueAt(HeaderObject vertical, HeaderObject horizontal)`
 This method is used for getting data from the the model as a list of `OutputNodes`. As its parameters suggest, it takes one element from the vertical axis and one from the horizontal and performs their intersection. Restrictions that are set by `setRestriction()` method are taken into account.
- `GeneralAxe getHorizontalAxe(), GeneralAxe getVerticalAxe()`
 These two methods are used for retrieving horizontal respectively vertical axis from the `TreeMatrixModel`.
- `void setRestriction(TopCoordinate topCoordinate),`
 `void setRestriction(TopCoordinate topCoordinate, Coordinate restriction),`
 `void setRestrictions(TopCoordinate topCoordinate, Set<Coordinate> restrictions)`
 These methods set additional restrictions that are used for the intersection. The `TopCoordinate` specifies the main axis that is used and the second argument specifies restrictions for this axis. Set restrictions are kept in the component until they are removed by `removeAllRestrictions()` method.
- `void removeAllRestrictions(TopCoordinate topCoordinate)`
 This method is used for removing restrictions that were previously set by `setRestriction()`. The given `TopCoordinate` is removed as well as any restrictions that were specified for this `TopCoordinate`. If the given `TopCoordinate` was not set before by `setRestriction()`, nothing happens.
- `void init(KnowledgeMatrixInstance dataSource, TopCoordinate verticalAxis, List<TopCoordinate> filterAxes)`
 This method forces `TreeMatrixModel` to reload data from given `KnowledgeMatrixInstance` and to reload new data to the visualized axes.

Method List 3.15: TreeMatrixModel Methods

Chapter 4

Implementation And Testing

The chapter 3 explained detailed information regarding the principles and design of the system. This chapter aims to describe the implementation of this architecture and presents results of it. Each implemented component is presented in a separate section where the utilized technologies are introduced. A small overview of similar technologies is presented and rationale is given on why the selected technologies were used. At the end of each section, there is a subsection detailing the testing of the described component, explaining the testing methods used and the results of those tests. Sections in this chapter have the same order as in chapter 3.

In general, the provided implementation is based mainly on the Java language and environment. Java was chosen because it is a statically typed language, it supports generic types, is portable among different environments and there is great number of open-source projects that support development in this language. Project management is performed by Maven [13]. This tool allows to split the source codes and libraries that this project depends on which makes it ideal for a source code management tool like svn or git. Maven also has support for many plugins like JUnit or Cobertura, which help with project development. Those tools are explained in the following text. The implementation is not limited by any dependency injection framework (like Spring) but can be easily incorporated into one that is used by a project that would use the implemented components.

4.1 Knowledge-Matrix-Server

The implementation of the `Knowledge-matrix-server` component is based on RDF(S) and OWL2. The reason for this decision has already been given in chapter 2. The implementation of the `knowledge-matrix-server` raised several questions. The first question was the selection of an appropriate knowledge framework that would satisfy the requirements of this project and that would also make the development as simple as possible. The second question was about the testing methods of the implemented project, the tools that were used for these tasks, and how to measure the results of those tests.

4.1.1 Knowledge Framework

There are several frameworks that support cooperation between Java programs and OWL2 ontologies. These frameworks can be generally split into two groups: direct and

indirect.

The direct group aims at direct mapping of Java classes to OWL2 classes and vice versa. This approach is very similar to the object relation mapping that is used for relation databases, e.g. JPA [29]. An advantage of this solution is the simple access to ontology data through java beans that represent business objects. An example of such a framework is Jastor [5]. This framework generates POJOs from a given ontology that represent OWL classes. It is only a one way conversion, so the changes made to POJOs are not propagated back to the ontology. Classes are generated during the development time from the ontology and therefore the classes have to be known before implementation. This limitation represents a disadvantage for this thesis because generated classes are usable only for its original ontology and therefore are not general. Other example of such an approach are Owl2java project [37] and Kazuki project [1]. The Jenabean project [10] is based on a different approach. It converts annotated Java classes and generates an appropriate RDF/OWL representation. This approach is not suitable for this thesis because the used ontology is already created and there is no need for creating a new one.

The indirect group takes a different approach regarding how to deal with ontologies. It does not directly map Java classes to an ontology, but rather provides a universal interface for interaction with it. This approach is similar to the one that is used in JDBC API [28]. The advantages of this approach are its universal access to all ontologies and its ability to use associated technologies like SPARQL [17]. These features fit more to the requirements for `knowledge-matrix-server` and therefore a knowledge framework based on this approach is used. OwlApi [27] is an project that implements this approach. It is an open source project designed to work with OWL2 ontologies. It supports the creation, manipulation and serialization of OWL2 ontologies but it can not handle SPARQL queries.

Another framework for indirect work with ontologies is Apache Jena [12]. It is mainly designed to work with RDF(S) graphs but OWL2 ontologies can be handled as well. Several associated technologies are supported by this framework including SPARQL queries, which are realized by the ARQ library. ARQ can handle SPARQL queries up to version 1.1 which is the most recent version of SPARQL at the time this thesis was written. Because the architecture proposed in chapter 3 requires a framework that supports textual queries and SPARQL is textual based language, the Jena framework was chosen as the most appropriate underlying knowledge framework.

Jena can itself handle basic reasoning on RDF(S) graphs but it can not handle OWL2 DL. The lack of this feature can be overcome by an external reasoner. For the purpose of this thesis, Pellet reasoner [8] is used, but it is possible to make some small changes to the code and use any other reasoner that is able to process OWL2 DL. The project in this thesis also requires access to a remote triple store, namely Sesame [33]. This requirement was realized by JenaSesameConnector provided by KBSS [23] which manages remote connections.

4.1.2 Testing

The `knowledge-matrix-server` was tested from two perspectives. The first perspective is about the testing of the proposed architecture of the `knowledge-matrix-server`. The testing of the architecture should answer whether the proposed architecture satisfies the requirements of this project and is flexible enough for some small changes

that are inevitable in every project. This testing was done by specifying several use cases and then modeling them as UML sequence diagrams. Modeled use cases included (i) an initialization of `knowledge-matrix-server`, (ii) getting axes and their values and (iii) querying `knowledge-matrix-server` and retrieving relevant results. Sequence diagrams of those use cases are shown in the chapter 3 as well as short descriptions of the modeled processes. The testing was done as a discussion over the designed diagrams between the author of the architecture and the supervisor of this project. This testing improved the design process of the architecture and made the proposed interfaces simpler to understand and yet generic enough for the future extension. Sequence diagrams show that the proposed architecture is suitable for the requirements of this thesis.

The second perspective is concerned with testing the implementation that was provided as a proof of the concept of the proposed architecture. `Knowledge-matrix-server` implementation was tested mainly by manual testing and unit tests. Unit tests allow the separate testing of each implemented class and to test the interactions of those classes together. Unit tests were written in JUnit 4.9 testing framework [32]. This framework allows repeatable test execution and there is number of plugins and tools for visualizing the results of executed tests. This approach not only tests the code that is under development but also helps with regression testing. Regression tests help to detect bugs that are created during the refactoring process and therefore allow for the speeding up of the project development. For mockup generation `mockito` [35] framework is used. This simple framework supports the simple creation of mockups and allows for the encoding of simple conditions for the generated mockups. For coverage measuring `Cobertura` tool is used. This tool is designed to execute the implemented unit tests and determine which lines and branches of developed code were tested and which not. After unit tests are completed `Cobertura` generates a HTML page or a XML document that presents the achieved results.

All unit test that were implemented passed and the code coverage can be seen in table 4.1. The table is vertically divided into groups where each group represents one package of source code. The first column shows name of tested class, the second column shows line rate coverage in percent, the third column shows branch coverage in percent and the last column show the McCabe cyclomatic code complexity of the tested class. Line coverage describes how many lines were executed during the tests. Branch coverage describes how many branches were passed during the tests out of total number of possible branches. Cyclomatic code complexity measures how many decision points are contained in a tested class. The higher this number is the more complicated the tested class is.

Unfortunately unit tests can not prove the absence of a bug but they can only prove the presence of a bug. This means that although all tests passed and test coverage is good this component may not be bug free. This proof can be only given by mathematical verification but it would be too complex with respect to this thesis.

4.2 Knowledge-Matrix-Component

The `knowledge-matrix-component` is composed of three separate components. The first component is `knowledge-matrix-component` which represents the visualization of the intersection of two or more axes. The second component is `infocard` which shows detail information about a particular `OutputNode`. And the last component is `auto-complete-tree` that helps the user with the selection of values from a given tree.

name	line-rate[%]	branch-rate[%]	complexity
Package cz.cvut.kbss.kmserver.impl			
KnowledgeMatrixInstanceImpl	89.19	100.00	1.750
KnowledgeMatrixServiceImpl	75.89	68.97	3.579
QueryUtils	85.71	100.00	2.000
Package cz.cvut.kbss.kmserver.impl.configuration			
FileSystemDataSource	100.00	100.00	1.000
JenaSesameDataSource	100.00	100.00	1.000
Package cz.cvut.kbss.kmserver.impl.dto			
CoordinateImpl	96.43	100.00	1.636
KnowledgeMatrixTreeModelImpl	84.72	80.00	2.688
OutputNodeImpl	70.83	32.14	2.600
TopCoordinateImpl	100.00	80.00	3.000
Package cz.cvut.kbss.kmserver.impl.query			
GraphQueryResultImpl	83.33	73.33	4.000
KnowledgeMatrixQueryBuilderImpl	84.09	81.25	2.200
KnowledgeMatrixQueryImpl	93.94	75.00	3.364
KnowledgeMatrixQueryResultImpl	100.00	100.00	1.750
SimpleQueryPair	93.55	69.23	3.125
Package cz.cvut.kbss.kmserver.impl.usecases			
InterventionOutputNodeBuilder	93.10	100.00	2.000
InterventionScenario	82.98	90.00	2.200

Table 4.1: Table of unit test coverage of knowledge-matrix-server

As in the previous section the first subsection is concerned with selection of appropriate technology and the second subsection deals with testing the implemented solution and presents the results of the tests.

4.2.1 Web Framework

One of the general requirements of this project was the request that the user interface shall be implemented as a web application. There are many web technologies for Java and this chapter shortly introduces a few of them and gives rationale as to why the selected technology was utilized and what its advantages for this project are.

The first solution mentioned in this chapter are Servlets [19]. They were firstly introduced by Sun Microsystems in 1997 and are considered as a low-level approach for web applications. The servlet is a special class that is used to extend server capabilities through communication with external clients by some protocol. Servlets are mainly implemented using HTTP protocol but it is possible to use them for other protocols as well. They act as an interface which shields the developer of the application from the implementation details of the underlying protocol. When the user initiates an HTTP request, the servlet wraps data from this request and presents it to the application as an instance of `HttpServletRequest`. The application then processes this request and writes an answer to an instance of `HttpServletResponse`. How the answer should be composed is completely left to the application which gives a lot of variability to the developers but also makes development very tedious. This technology is usually used by other web technologies that are built on top of it.

The first technology built on top of servlets introduced in this thesis is Java Server Pages (JSP) [20]. This technology takes the template approach to generate web pages. The web application developer specifies the page in a language that is a combination of HTML and special JSP tags. Those special tags represent Java code snippets that are evaluated during request and are replaced by their evaluated values. This way the developer can easily combine static portion of pages like headers and styles with dynamically generated content. Compared to servlets this technology allows much faster development. Implementation of this technology is done by compiling JSP into equivalent servlets when the user firstly request the page.

While JSP allows to easily create dynamic web content it does not solve the way the user enters data to the page and generally the way how user interacts with it. This problem is addressed by Java Server Faces technology (JSF) [21]. JSF is a Java specification for building component based user interfaces in the form of web pages. The developer writes pages using special JSF tags and those pages are translated by a JSF servlet into the resulting web pages. The JSF servlet takes care of data validation and passes validated data from the user's input to the application, which makes application development faster. The disadvantage when compared to JSF is performance, which is worse because of the additional overhead.

Comparing the three mentioned web technologies, the most fitting one for this project is JSF because it is capable of composing the final web page from partial solutions, it has a simple approach to processing the user's input and there are many already implemented components based on JSF that can be reused in this project. Version of JSF specification used is 2.2 [21] and used implementation of JSF servlet in this project is Mojarra 2.2.6 [16].

4.2. KNOWLEDGE-MATRIX-COMPONENT

name	line-rate[%]	branch-rate[%]	complexity
Package cz.cvut.kbss.autocompletetree			
TreeSelector	96.00	100.00	1.800
Package cz.cvut.kbss.autocompletetree.matcher			
CaseInsensitivePrefixMatcher	100.00	50.00	3.000
TreeAutocompleteMatcherIface	100.00	100.00	1.000
Package cz.cvut.kbss.autocompletetree.model			
AutocompleteTreeModel	86.96	73.08	2.000
Package cz.cvut.kbss.tree_matrix.model			
TreeMatrixModel	83.33	50.00	1.300
Package cz.cvut.kbss.tree_matrix.model.axes			
DrillableAxis	80.56	64.29	2.167
FilterableAxis	100.00	100.00	1.500
GeneralAxis	100.00	100.00	1.000
SwingToPrimefacesConverter	90.91	88.89	2.444
Package cz.cvut.kbss.tree_matrix.model.axes.headers			
DrillableHeaderObject	86.67	50.00	1.857
FilterableHeaderObject	100.00	100.00	1.375
HeaderObject	100.00	100.00	1.000
HeaderObject\$Action	100.00	100.00	1.000

Table 4.2: Table of unit test coverage of `knowledge-matrix-component`

4.2.2 Testing of knowledge-matrix-component and auto-complete-tree

These two components are tested together because `knowledge-matrix-component` uses `auto-complete-tree` very intensively and thus it makes sense to test them together. The testing of implemented components was done partially by unit tests and partially by automated web user interface testing tools. The purpose of unit testing was mainly concerned with data models and the interactions between axes and their associated header objects. Automated web user interface testing was done because some JSF features are easier to test this way than by unit tests.

Unit tests used the same technologies as were used in the testing of `knowledge-matrix-server`. Line coverage, branch coverage and cyclomatic complexity can be seen in table 4.2.

For automated web user interface testing Selenium tool was used. This tool simulates user when performing testing. It follows predefined test scenarios and performs certain assertions during the test run. This way the tests can be executed automatically and can be used as regression tests during project evolution. For all test cases the following pre-conditions must be met:

1. The utilized `knowledge-matrix-server` must be filled with `ModelDataSources` that

represent data from the given testing set that is placed in the same folder as test cases.

2. The `DrillableAxis` root must be set to the associated `TopCoordinate`.
3. No `Coordinates` must be selected for all used `FilterableAxes`.
4. No restriction on any used axis may be used.

The table 4.3 shows predefined test cases, describes what parts of `knowledge-matrix-component` are tested and how tests are performed. The last column is very brief and does not cover all steps in detail, but rather gives basic ideas as to how the tests should be performed. For detailed information the reader can look at the Selenium test implementation provided in this thesis. All tests were executed using Selenium 2.5.0 and Firefox 27.0.1 on Mac OSX 10.9.2.

4.3 Knowledge-Matrix-User-Interface

This component has integration character as it combines all previously described components and makes the final application from them. The first question that has to be answered is how and where the resulting application will be deployed. Because all components of `knowledge-matrix-component` are implemented using JSF, a server container that can work with this technology is required. All implemented components in this thesis are mainly developed for the MONDIS project and this project already has several applications developed so there is a need to integrate them all together. One of possible solution how to do this task is to use Java Portlet technology.

"Portlets are web components, like servlets, specifically designed to be aggregated in the context of a composite page. Usually, many portlets are invoked to in the single request of a portal page. Each portlet produces a fragment of markup that is combined with the markup of other portlets, all within the portal page markup." (from the Portlet Specification [22], JSR 168)

Portlet technology is able to easily combine more independent applications and provide the same look and feel for them. Examples of such advantages are a common place for logging in to the application, same ways as to how to configure common options like size and borders of an window or same way how to add more applications to one particular page. The features of Portlet technology satisfy the conditions placed on the whole MONDIS project and therefore Portlet technology is used.

Because this is a standardized solution there are independent implementations of portlet servers which means that this implementation is not limited to any particular solution. The implementation used in this thesis is deployed to the Liferay portlet implementation [18]. This implementation is able to execute and render JSF pages and because of this all of components developed for this thesis are deployed to Liferay portlet container, version 6.1.2. An example of deployed `knowledge-matrix-component` into the Liferay portlet can be seen in figure 4.1

Scenario Name	What is tested	How is it tested
Drillable Axis Test	<ul style="list-style-type: none"> • Drilling action of <code>DrillableAxis</code> • Presence of correct elements in the actual path visualization • Correct length of the actual path visualization • Correct navigation by using actual path visualization 	<p>Click on predefined <code>DrillableHeaderObjects</code> and observe if correct data loading is being performed. When the loading is done, check if the actual path shows the correct nodes. Do this operation several times. Click on predefined element of the actual path and check if correct data loading is being performed. Check if the new actual path and the new root of <code>DrillableAxis</code> are correct. Check that visualized path is rendered correctly if the path is longer than n.</p>
Filterable Axis Test	<ul style="list-style-type: none"> • Selection of <code>Coordinates</code> for <code>FilterableAxis</code> • Filtering of appropriate <code>OutputNodes</code> 	<p>Click on selected filterable axis and wait for the selection window to open. Select one node and then click on the Filter button. Wait for a reload of the current web page and then assert that only selected values are listed in the appropriate cell. Repeat this procedure more times.</p>
External Axis Test	<ul style="list-style-type: none"> • Selection of <code>Coordinates</code> for axis that is not present in the matrix header (e.g. material) • Filtering of appropriate <code>OutputNodes</code> 	<p>Click on Material link and wait for the selection window to open. Select some material node and then click on Filter button. Wait for reload of current web page and then assert that only selected values are listed in appropriate cell. Repeat this procedure more times with more different nodes.</p>
AutoCompleteTree	<ul style="list-style-type: none"> • Filtering of visualized nodes • Ajax updates of visualized nodes 	<p>Click on Component link and wait for selection window to open. Type "Wall" into text field and assert that only Components whose name start with "Wall" are listed. Remove "all" from "Wall" and assert that only Components whose name start with "W" are listed. Filtering should be case insensitive.</p>

Table 4.3: Table of predefined Selenium test cases

Welcome to Liferay Portal

What We Do

knowledge-matrix-portlet

Select Component Select Material Reload data

Material	Intervention	Mechanism	Agent
Manifestation Of Damage Discoloration And Deposit	<ul style="list-style-type: none"> water cleaning 		<ul style="list-style-type: none"> installation of water proofing course installation of drainage system poulticing application of sacrificial layer

Manifestation of damage / Material Manifestation Of Damage /

Figure 4.1: knowledge-matrix-component deployed to Liferay portlet

Chapter 5

Conclusion

The main goal of this thesis was to design and implement a software system for the analysis of damage to cultural heritage objects. This software system should be able to perform given use cases and should be generic so that a new use case can be easily incorporated into the implemented solution. At the time this thesis was written only one such a use case was known: the Intervention use case. This use case analyzed manifestations of damages, their causes, and interventions preventing them.

In order to satisfy these requirements, it was necessary to propose a general approach to representing knowledge about the domain. The proposed solution is based on the idea of intersection of arbitrary axes representing part of the described domain. The implementation of this solution is backed by MONDIS ontology and therefore this ontology was studied. The ontology is defined in OWL2 language and this language is based on Description Logic. For this reason chapter 2 contains an introduction to this formalism and to OWL2 language and to associated technologies. The intervention use case is described using description logic and then this description is encoded into OWL2 and SPARQL-DL languages.

Based on the knowledge gained from the previous steps a software architecture of the system was designed and rationale for this design was given. The designed system is generic so it can represent more use cases than the Intervention one. The system was implemented according to this design. The user interface was implemented using web technologies, namely JSF and Liferay Portlet. Those technologies allow end users to use this system simply through a web browser without a need to install anything.

This implementation was tested by unit tests and by Selenium, which is an automated tool for testing user interfaces. More than 100 unit tests were written and code coverage was over 75 %. Selenium tests were designed and a short intuitive description of those tests was given in this thesis.

The reason why the test coverage was not 100 % was due to time limitations and because of the changes to the data representation in the late phase of the project. During the project development the way how input data were represented was changed and this change led to the necessity of modifying some parts of the implementation. The first implementation represented individuals that belonged to some class by the name of the associated class. This approach combined with OWL2 punning made SPARQL queries simple and readable. But this approach was not comfortable for the end users who asked for using the OntoMind tool (also developed in the MONDIS project). To accept this requirement data representation was modified. Each class is now represented by one indi-

vidual that is the instance of this and only this class. So what the punning method did implicitly in the previous data representation is now done explicitly. This change to the code proved that the proposed architecture is flexible enough to incorporate such changes as only one file was modified.

Bibliography

- [1] ? The kazuki project. <http://projects.semwebcentral.org/projects/kazuki>. Year: 2007.
- [2] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, Cambridge, 2003.
- [3] Knowledge Base and Software systems group on CTU Prague. <http://www.mondis.cz>. Accessed: 2014-03-19.
- [4] Knowledge Base and Software systems group on CTU Prague. <http://www.mondis.cz/en/web/portal/mobile>. Accessed: 2014-03-19.
- [5] Joe Betz Ben Szekely. Typesafe, ontology driven rdf access from java. <http://jastor.sourceforge.net>. Accessed: 2014-03-13.
- [6] Miroslav Blasko. Personal communication, 2014. A discussion of a desing and implementation.
- [7] Miroslav Blasko, Riccardo Cacciotti, Petr Kremen, and Zdenek Kouba. Monument damage ontology. volume 7616 of *Lecture Notes in Computer Science*, pages 221–230. Springer, 2012.
- [8] LLC Clark & Parsia. <http://clarkparsia.com/pellet/>, 2005–2013. Accessed: 2014-03-13.
- [9] R.V. Guha Dan Brickley. Rdf schema 1.1. <http://www.w3.org/TR/2014/REC-rdf-schema-20140225/>, February 2014. Accessed: 2014-04-10.
- [10] drdonohue@gmail.com. A library for persisting java beans to rdf. <https://code.google.com/p/jenabean/>. Accessed: 2014-03-13.
- [11] Guus Schreiber Fabien Gandon. Rdf 1.1 xml syntax. <http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/>, February 2014. Accessed: 2014-04-10.
- [12] The Apache Software Foundation. A free and open source java framework for building semantic web and linked data applications. <https://jena.apache.org>. Accessed: 2014-03-13.
- [13] The Apache Software Foundation. <http://maven.apache.org>, 2002-2014. Accessed: 2014-03-19.

BIBLIOGRAPHY

- [14] W3C OWL Working Group. Owl 2 web ontology language. <http://www.w3.org/TR/owl2-overview/>, December 2012. Accessed: 2014-04-10.
- [15] Pascal Hitzler, Markus Krötzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. CRC Press, Boca Raton, FL, 2009.
- [16] <https://www.java.net>. <https://javaserverfaces.java.net>. Accessed: 2014-03-19.
- [17] <http://www.w3.org>. Sparql query language for rdf. <http://www.w3.org/TR/rdf-sparql-query/>. Accessed: 2014-03-13.
- [18] LIFERAY INC. <http://www.liferay.com>, 2014. Accessed: 2014-03-19.
- [19] Inc. Java Community Process, Oracle America. <https://jcp.org/aboutJava/communityprocess/final/jsr315/>. Accessed: 2014-04-24.
- [20] Inc. Java Community Process, Oracle America. <https://jcp.org/aboutJava/communityprocess/final/jsr245/>. Accessed: 2014-04-24.
- [21] Inc. Java Community Process, Oracle America. <https://www.jcp.org/aboutJava/communityprocess/final/jsr344/index.html>. Accessed: 2014-03-19.
- [22] Inc. Java Community Process, Oracle America. <https://www.jcp.org/en/jsr/detail?id=168>. Accessed: 2014-03-19.
- [23] KBSS. <http://krizik.felk.cvut.cz/m2repo>, 2014. mvn artifactId: backend-jena-remote-sesame, mvn groupId: cz.cvut.kbss.ontomind, version: 1.1-SNAPSHOT.
- [24] Petr Kremen. *Building Ontology-Based Information Systems*. PhD thesis, Czech Technical University in Prague Faculty of Electrical Engineering Department of Cybernetics, february 2012.
- [25] Petr Křemen, Pavel Mička, Marek Šmíd, and Miroslav Blažko. Ontology-driven mindmapping. *I-SEMANTICS 2012, 7th Int. Conf. on Semantic Systems*, 2012.
- [26] Peter F. Patel-Schneider Matthew Horridge. Owl 2 web ontology language - manchester syntax (second edition). <http://www.w3.org/TR/owl2-manchester-syntax/>, December 2012. Accessed: 2014-04-10.
- [27] Sean Bechhofer Matthew Horridge. The owl api: A java api for owl ontologies. *Semantic Web Journal 2(1), Special Issue on Semantic Web Tools and Systems*, pages 11–21, 2011.
- [28] Oracle. Java database connector. <http://www.oracle.com/technetwork/java/javase/jdbc/index.html>. Accessed: 2014-03-13.
- [29] Oracle. Java persistence api. <http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html>. Accessed: 2014-04-24.
- [30] Peter F. Patel-Schneider Patrick J. Hayes. Rdf 1.1 semantics. <http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/>, February 2014. Accessed: 2014-04-10.
- [31] Eric M. Dashofy Richard N. Taylor, Nenad Medvidovic. *Software Architecture Foundations, Theory, and Practice*. Wiley, 2010.

BIBLIOGRAPHY

- [32] David Saff, Kevin Cooney, Kent Beck, and Marc Philipp. Junit, a programmer-oriented testing framework for java. <http://junit.org>. Accessed: 2014-04-24.
- [33] Aduna Software. <http://www.openrdf.org>. Accessed: 2014-03-13.
- [34] Andy Seaborne Steve Harris, Garlik. Sparql 1.1 query language. <http://www.w3.org/TR/2013/REC-sparql11-query-20130321/>, March 2013. Accessed: 2014-04-17.
- [35] szczepiq@gmail.com. <https://code.google.com/p/mockito/>, 2010–2013. Accessed: 2014-03-19.
- [36] Dan Connolly Tim Berners-Lee. Notation3 (n3): A readable rdf syntax. <http://www.w3.org/TeamSubmission/n3/>, March 2011. Accessed: 2014-04-10.
- [37] Michael Zimmermann. A java code generator for owl. <http://www.incunabulum.de/projects/it/owl2java>. Accessed: 2014-03-13.

Content of CD

Attached CD contains

Directorr	Description
implementation	implementation of the project
thesis	source codes of the thesis
tests	Selenium tests
thesis.pdf	text of the thesis

Table 1: Directory structure of thesis