Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student:	Bc. Jakub D u n d á l e k
Study programme:	Open Informatics
Specialisation:	Artificial Intelligence
Title of Diploma Thesis:	Flexible User Interface for Ontology Applications

Guidelines:

1. Become familiar with methodologies and techniques for creating efficient and flexible user interfaces with respect to ontology-based applications.
2. Become familiar with relevant semantic web technologies, in particular with OWL ontology language and OWL integrity constraints.
3. Design and implement a flexible user interface allowing efficient authoring and validation of ontological data, including integrity constraints checking.
4. Design a set of functional and integration tests and verify system functionality on a suitable ontology.

Bibliography/Sources: Will be provided by the supervisor.

Diploma Thesis Supervisor: Ing. Petr Křemen, Ph.D.

Valid until: the end of the winter semester of academic year 2014/2015

L.S.

prof. Ing. Vladimír Mařík, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, May 31, 2013

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	Bc. Jakub D u n d á l e k
Studijní program:	Otevřená informatika (magisterský)
Obor:	Umělá inteligence
Název tématu:	Flexibilní uživatelské rozhraní pro ontologické aplikace

Pokyny pro vypracování:

1. Seznamte se s problematikou tvorby efektivních a flexibilních uživatelských rozhraní s ohledem na ontologické aplikace.
2. Seznamte se s jazyky relevantními sémantickými technologiemi, zejména s jazykem OWL a integritními omezeními v něm vyjádřenými.
3. Navrhněte a implementujte flexibilní uživatelské rozhraní umožňující efektivní editaci a validaci ontologických dat, včetně validace integritních omezení.
4. Vytvořte sadu funkčních a integračních testů a funkcionalitu výsledného systému ověřte na vhodné ontologii.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí diplomové práce: Ing. Petr Křemen, Ph.D.

Platnost zadání: do konce zimního semestru 2014/2015

L.S.

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 31. 5. 2013

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Master's Thesis

Flexible user interface for ontology applications

Jakub Dundálek

Supervisor: Ing. Petr Křemen, Ph.D.

Study Programme: Open Informatics

Field of Study: Artificial Intelligence

May 13, 2014

Poděkování

Tímto bych chtěl poděkovat vedoucímu Petru Křemenovi za poskytnutí zajímavého tématu a vedení práce.

Dále ochotným a vždy příjemným studijním referentkám paní Býmové a paní Zichové.

V neposlední řadě patří velký dík mé rodině za neustálou podporu po celé délce studia.

Prohlášení autora práce

Abstract

Ontologies are becoming a viable option for knowledge storage and compete with traditional relational databases. This can be utilized to create ontology-based informations systems that offer a greater flexibility and expressiveness of the data model than traditional systems. Ontologies allow evolving of the data model to satisfy changing needs required from the software. However, it may be difficult to keep the user interface up to date. The goal of this thesis is to explore methods for automatic generation of user interfaces for ontology-based information systems to address this problem. This work explores possibilities for ontology-based user interfaces and implementation of a prototype interface.

Abstrakt

Ontologie se stávají použitelnou možností pro ukládání znalostí a konkurují klasickým relačním databázím. Toho lze využít k vytvoření ontologických informačním systémů, které oproti klasickým systémům poskytují větší flexibilitu a pokročilejší vyjadřovací prostředky. Ontologie dovolují vývoj datového modelu pro uspokojení měnících se potřeb pro daný software. Avšak udržování uživatelského rozhraní aktuálním je obtížné. Cílem této diplomové práce je prozkoumat metody pro automatické generování uživatelských rozhraní pro ontologický aplikace ve snaze řešit tento problém. Tato práce prozkoumává možnosti pro ontologické uživatelské rozhraní a implementaci prototypu rozhraní.

Contents

Chapter 1

Introduction

We rely heavily on information systems to store and organize our information. Information systems help us making better decisions. However, developing an information system is a complex task, which is both time and money consuming.

Many of the requirements are unknown or change after the software is released and deployed. Information systems need to be adapted to fit the new needs. This require additional costs. Conventional information systems use relational database systems for storing data. A change in the data model usually requires non-trivial changes in multiple places of a source code.

By choosing alternative approach we can reduce the development complexity which leads to reduced costs. Ontology-base storage systems are very flexible for storing knowledge. Using ontology-based information systems (OIS) provides easier way to modify and evolve functionality of information systems. Such a system was proposed and developed by [14]. It proposes a formal contract between an ontology and object model of an information system. The consistency of data is guaranteed by utilizing expressive reasoning and transactional support.

The storage solution is only a one of many parts that make useful information system. The other part is a graphical interface that provides access to data for users. An inappropriate interface can undermine value and effectivity of an system. Therefore a great amount of effort needs to be put on a development of an information system. A survey in [16] shows that up to 50% of the total efforts in application development is spent on the development of the user interface . Therefore a reuse of existing user interface components yields significant benefit.

The goal of this work is to explore possibilities of reducing the amount of work needed to implement information systems. By using an ontology-based storage we can reduce development costs. The main contribution is to develop a graphical component that can be integrated in an ontology-based system and thus reducing the costs and time even further. This component will provide an adaptable input form where users enter the data they want to store in the information system.

Implementation of data input interface is a part of every information systems. A flexible component that provides common functionality can be easily integrated into the application.

In the result programmers save time and can spend the time implementing application specific business logic. This helps to reduce costs and ensure that system is developed on time and budget.

Furthermore the flexibility of this approach allows for effective implementation of future changes in a data model. After a change in ontology the graphical interface is automatically re-generated and does not require any manual change. This ensures fast implementation and reduces software bugs. Every manual interaction in a codebase can potentially introduce a bug so by automating the changes we can reduce risks.

1.1 Thesis outline

First we discussed motivation for ontology-based information systems and flexible user interface. In chapter 1 we also briefly introduce basic technologies behind the ontology-based systems.

Next we list related work and state of the art in OIS chapter 2.

The design of the software is described in chapter 3. We discuss software requirements and decisions of used architecture.

Implementation details shown are discussed in chapter 4. We will go through specifics about used technologies and how to integrate all the components together.

At the end we evaluate reached goals and discuss possibilities of future improvements of the software.

1.2 Technologies and Concepts

The base of ontological applications consists of many technologies and concepts working together in a harmony. In this chapter we briefly introduce them to a reader. An ontology is an concept how to express our knowledge. We will introduce a formal definition. RDF describes the format how information is stored. OWL introduces a framework for describing additional concepts and relations on top of RDF. When we have our knowledge collected and stored, we use SPARQL language ask questions and queries about our data. Lets get started.

1.2.1 Ontologies

The dominating definition of an ontology is based on [17]:

An ontology is a formal explicit specification of a shared conceptualization of a domain of interest.

Several characteristics captured in this definition are explained in [11]:

Formality – An ontology is expressed in a knowledge representation language that is based on the grounds of formal semantics and principles of logic. This ensures that the specification of domain knowledge in an ontology is machine-processable and is being interpreted in a well-defined way.

Explicitness – An ontology states knowledge explicitly to make it accessible for machines. Notions that are not explicitly included in the ontology are not part of the machine-interpretable conceptualization it captures, although humans might take them for granted by common sense.

Consensus – An ontology reflects an agreement on a domain conceptualization among people in a community. The larger the community, the more difficult it is to come to an agreement on sharing the same conceptualization. In this sense, the construction of an ontology is associated with a social process of reaching consensus.

Conceptuality – An ontology specifies knowledge in a conceptual way in terms of conceptual symbols that can be intuitively grasped by humans, as they correspond to the elements in their mental models. Moreover, an ontology describes a conceptualization in general terms and does not only capture a particular state of affairs. Instead of making statements about a specific situation involving particular individuals, an ontology tries to cover as many situations as possible that can potentially occur.

Domain Specificity – The specifications in an ontology are limited to knowledge about a particular domain of interest. The narrower the scope of the domain for the ontology, the more an ontology engineer can focus on capturing the details in this domain rather than covering a broad range of related topics.

In summary, an ontology used in an information system is a conceptual yet executable model of an application domain. It is made machine-interpretable by means of knowledge representation techniques and can therefore be used by applications to base decisions on reasoning about domain knowledge [11].

1.2.2 RDF

The Resource Description Framework (RDF) is a framework for representing information in the Web [7]. The RDF data model is based on sets of triples that describe relationships among resources. Each triple consists of a subject, a predicate and an object. The resources are uniquely identified by Internationalized Resource Identifier (IRI). We can represent RDF triplets as a graph where resources are nodes and predicates are edges.

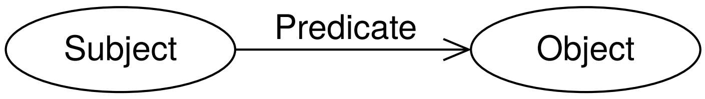


Figure 1.1: Ilustration of an RDF graph [7]

We can describe physical things, people or any abstract entity. These things are called `resources`. A resource can have IRI which stands for Internationalized Resource Identifier. Therefore we can also combine information from multiple sources. Different information about a particular entity can be stored on different places on the Internet. By combining the sources together based on resource's IRI we can get additional valuable information. However this feature is not relevant to goals of this work so we won't describe any more details.

When we want to store RDF graphs, we need to encode them into DRF documents. We can use many different formats that represent the same meaning, i.e. XML, Turtle, RDFa, JSON-LD [7].

A main difference when compared to traditional data storage is that RDF does not have a fixed schema and can represent any kind of information. Traditional relational databases have a fixed schema that describes the structure of our data. New kind of data cannot be stored unless schema is changed and migrations are executed to ensure the old data fit into the new format. RDF allows for greater flexibility.

1.2.3 RDFS

RDF Schema is a semantic extension of RDF. It provides mechanisms for describing groups of related resources and the relationships between these resources [8]. RDFS is written using RDF. RDFS allows us to work with a more structured data model. This is achieved by introducing concepts of Classes, Properties, Domains and Ranges.

Property name	comment	domain	range
rdf:type	The subject is an instance of a class.	rdfs:Resource	rdfs:Class
rdfs:subClassOf	The subject is a subclass of a class.	rdfs:Class	rdfs:Class
rdfs:subPropertyOf	The subject is a subproperty of a property.	rdf:Property	rdf:Property
rdfs:domain	A domain of the subject property.	rdf:Property	rdfs:Class
rdfs:range	A range of the subject property.	rdf:Property	rdfs:Class

Table 1.1: List of selected RDFS properties

The RDFS class system is similar to type systems of object-oriented programming (OOP) languages. Most of the OOP languages define a class in terms of the properties its instances may have. RDFS differs from this by describing properties in terms of the classes of resource to which they apply. The concepts of domains and ranges of properties is used for this. A benefit of this is that we can define additional properties without a need to re-defined the original description of a class.

1.2.4 OWL

The Web Ontology Language (OWL) is language for defining web ontologies [1]. It is used to describe entities in the world and how they are related. OWL is a vocabulary extension of RDF and adds additional semantics on top of RDFS like relations between classes, cardinality, equality, richer typing of properties, characteristic of properties, and enumerated classes [2].

The main concepts of OWL are classes, properties and their instances.

1.2.4.1 Classes

OWL Class is defined using `owl:Class`. To create class hierarchy we use `rdfs:subClassOf` to define a subclass.

Every individual in the OWL world is a member of the class `owl:Thing`.

When we have multiple ontologies and we want to indicate that a particular class in one ontology is equivalent to a class in a second ontology, we can use `owl:equivalentClass` property.

We can define complex classes using set operators like `intersectionOf`, `unionOf`, `complementOf`. We can also specify a class via a direct enumeration of its members using `oneOf` construct.

1.2.4.2 Individuals

We describe an individual as a member of a class. We use `rdf:type` to tie an individual to a class of which it is a member.

Similarly to classes we can declare two individuals to be identical using `sameAs`. For opposite effect we can use `owl:differentFrom` and `owl:AllDifferent`. To specify that one individual is distinct to other individuals we use `owl:differentFrom`. To conveniently define a set of mutually distinct individuals we use `owl:AllDifferent`.

1.2.4.3 Properties

A property is a binary relation. Properties let us describe facts about class members and individuals. There are two types of properties:

- *datatype properties*
  these are relations between instances of classes and simple values like text, numbers and dates. These can be RDF literals and XML Schema datatypes. They are defined using `owl:DataProperty`,
- *object properties*
  relations between instances of two classes. They are defined using `owl:ObjectProperty`.

Similarly to classes we can subclass properties to create hierarchy of properties using `rdfs:subPropertyOf`. To indicate equivalence of two properties we can use `owl:equivalentProperty`.

We can define characteristics of properties to provide powerful mechanism for reasoning about properties. *Property characteristics* among others include `owl:TransitiveProperty`, `owl:SymmetricProperty`, `owl:inverseOf`.

To further constrain the range of a property we can use *property resctrictions*. The `owl:allValuesFrom` restriction requires that all values of a property must be members of a given class. The `someValuesFrom` restriction requires that at least one value of a property must be a member of a given class.

It is possible to restrict properties even further using exact *cardinality*. The `owl:cardinality` is used to specify exact cardinality, `owl:minCardinality` is used for specification of a lower bound and `owl:maxCardinality` for upper bound.

1.2.4.4 Sublanguages

When we create an ontology using arbitrary OWL constructs is not guaranteed that all conclusions are computable (completeness) and that all computations will finish in finite time (decidability). Therefore OWL provides three sublanguages with increasing expressiveness: *OWL Lite*, *OWL DL*, *OWL Full* [2].

OWL Lite supports a classification hierarchy and simple constraints. For example it only supports values of 0 or 1 for cardinality constraints.

OWL DL supports maximum expressiveness while retaining computational completeness and decidability. The name of OWL DL corresponds to *description logics*. It includes all OWL constructs but there are restrictions on how they can be used. For example a class cannot be an instance of another class and cardinality constraints cannot be placed on transitive properties [3].

OWL Full allows maximum expressiveness with no computational guarantees. OWL Full can be viewed as an extension of RDF, while OWL Lite and OWL DL can be viewed as extensions of a restricted view of RDF. Every OWL (Lite, DL, Full) document is an RDF document, and every RDF document is an OWL Full document, but only some RDF documents will be a legal OWL Lite or OWL DL document [2].

Each of these sublanguages is an extension of its simpler predecessor. Following statements hold: [2]

- Every legal OWL Lite ontology is a legal OWL DL ontology.
- Every legal OWL DL ontology is a legal OWL Full ontology.
- Every valid OWL Lite conclusion is a valid OWL DL conclusion.
- Every valid OWL DL conclusion is a valid OWL Full conclusion.

1.2.5 OWL 2

The OWL 2 Web Ontology Language (OWL 2) is a new version of OWL. It has very similar structure to OWL 1 and adds some new functionality. It also adds three new tractable profiles *OWL 2 EL*, *OWL 2 QL*, *OWL 2 RL*. They differ by their restrictions and guarantee different complexity of algorithms for reasoning. OWL 2 keeps backwards compatibility with OWL 1: all OWL 1 Ontologies remain valid OWL 2 Ontologies, with identical inferences in all practical cases [5].

1.2.6 SPARQL

SPARQL is a query language for querying RDF graphs. It can be used to query across multiple data sources.

1.2.6.1 Query forms

SPARQL has four query forms. These query forms use the solutions from pattern matching to form result sets or RDF graphs. The query forms are: [4]

- SELECT
  Returns all, or a subset of, the variables bound in a query pattern match.
- CONSTRUCT
  Returns an RDF graph constructed by substituting variables in a set of triple templates.
- ASK
  Returns a boolean indicating whether a query pattern matches or not.
- DESCRIBE
  Returns an RDF graph that describes the resources found.

This is an example of a SPARQL query [4]:

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT ?nameX ?nameY ?nickY
WHERE
  { ?x foaf:knows ?y ;
       foaf:name ?nameX .
    ?y foaf:name ?nameY .
    OPTIONAL { ?y foaf:nick ?nickY }
  }
```

1.2.7 OWA

There are two ways how to handle unknown information. The Closed World Assumption (CWA) accepted by traditional databases assumes that any unknown statement is considered false. If an information is missing the constraint violation is reported. This is important for ensuring data quality but limits flexibility. Even small change in data model requires significant amount of work to update application model and business logic [14].

OWL in contrast adopts Open World Assumption (OWA). When a knowledge is missing its existence is inferred. This approach can discover new knowledge within our data. It is an important to consider the difference between OWA and CWA when designing an information system.

Chapter 2

Related work

In this chapter we introduce related work. We first start by listing approaches for OWL access and ontology storage. Then we will go through existing graphical editors for ontologies. These will serve as a inspiration for designing a graphical interface for OIS.

2.1 OWL Access

A description and classification of programmatic OWL access approaches is presented in [14]. The approaches are divided into Type 1 and Type 2 APIs.

Type 1 APIs

These are low-level APIs for OWL access. They are useful for developing generic tools like ontology editors or semantic web search engines. They cannot make any assumption about a particular domain, thus their use for development of a domain specific applications is generally time consuming and error-prone [14]. Examples of Type 1 APIs are OWLAPI or Jena.

Type 2 APIs

Most of Type 2 approaches use ad-hoc mappings between ontologies and object models. There is also a more robust model-driven architecture (MDA) based approach. In summary these methods are not capable of using expressiveness of OWL or generated models are too complex. They also do not consider potential ontology evolution during the life of application. Example of Type 2 APIs are Sommer, Elmo, Jastor, RDFReactor, JAOB, or Owl2Java.

2.2 Java Ontology Persistence API

As a way to address shortcomings of the previous approaches a new approach is introduced in [14]. Its reference implementation is called JOPA.

Java Ontology Persistence API (JOPA) is a framework for creating ontology-backed information systems developed by researchers at the Department of Cybernetics at the Faculty of Electrical Engineering of the Czech Technical University in Prague.

The framework of building ontology-based information systems is proposed in [14]. Its core idea is to create a contract between ontology and application. This contract is formalized with the ontology language and an model in the target object-oriented language represents that contract. The framework satisfies following requirements [14]:

- contract stability – the contract has to be static or slowly evolving comparing to the ontology; the interface shall survive most ontology refinements,
- contract maintainability – the contract between an ontology and the respective object model has to be easy to establish and maintain,
- non-restrictive – the framework has to provide full access to the ontological knowledge, including entailment checking and expressive query answering,
- validation – the framework has to ensure that modification of the ontology by the application violates neither the consistency of the ontology, nor the contract between the application and the ontology.

2.2.1 Integrity constraints

To ensure the stability of the contract between the application and the ontology the notion of integrity constraints is introduced [14]. They are written using OWL syntax and can be authored and maintained using standard ontology editors. Their well-defined semantics allow them to be checked using OWL2 DL reasoners.

2.2.2 Persistence layer

JOPA API tries to follow JPA 2.0 standard which makes is easy to use by Java developer. The persistence is achieved by using a transactional persistence manager.

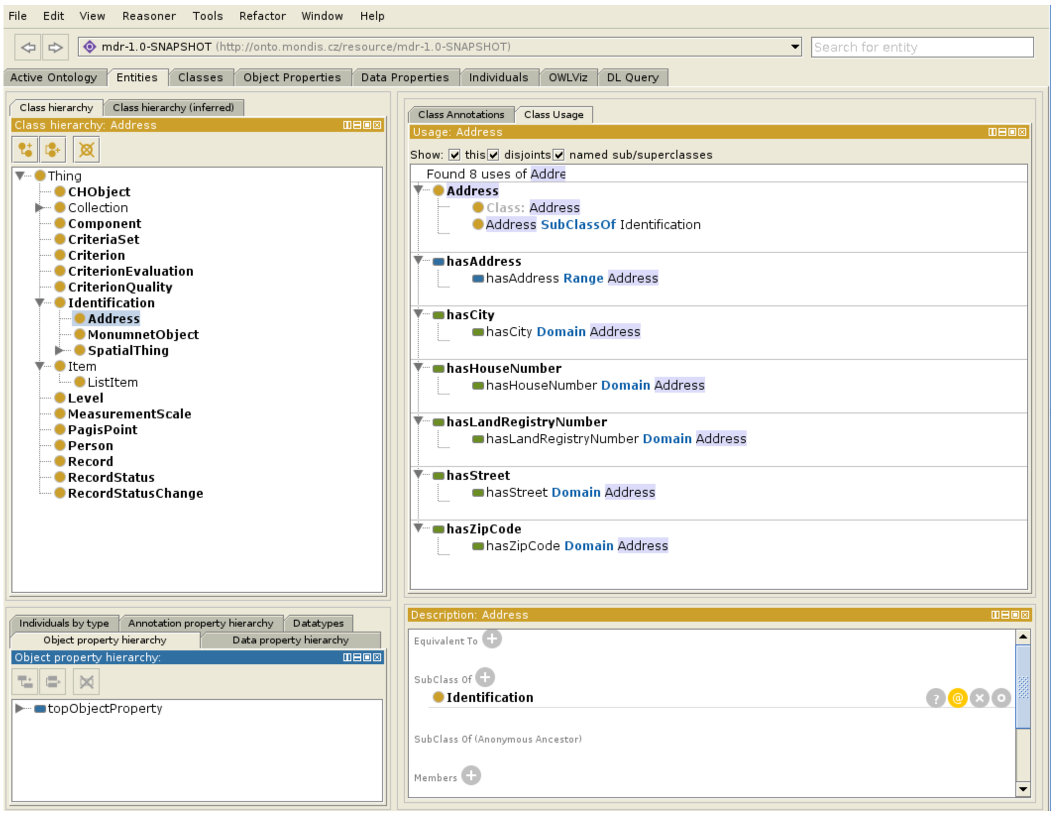
Thanks to recent work done in [15] JOPA includes OntoDriver layer which supports multiple OWL storage engines including OWLAPI, OWLDB, Jena, OWLIM.

2.3 Graphical interfaces for editing ontologies

In this section we are going to explore existing tools for editing ontology data. First we focus on OWL editors. It is also interesting to look into linked data editors. These do not have constraints provided by ontology, but the underlying data is similar because of the use of RDF.

2.3.1 Ontology Editors

Protégé[1] is a leading ontology editor. It is an open-source software developed at Stanford University. It features a plugin architecture. There are many plugins enhancing ontology visualization, reasoning and querying.



```
Thing
  CHObject
  Collection
  Component
  CriteriaSet
  Criterion
  CriterionEvaluation
  CriterionQuality
  Identification
    Address
    MonumnetObject
    SpatialThing
  Item
    ListItem
  Level
  MeasurementScale
  PagisPoint
  Person
  Record
  RecordStatus
  RecordStatusChange
```

```
Show: [✓] this [✓] disjoints [✓] named sub/superclasses
Found 8 uses of Addre
Address
    Class: Address
    Address SubClassOf Identification
hasAddress
    hasAddress Range Address
hasCity
    hasCity Domain Address
hasHouseNumber
    hasHouseNumber Domain Address
hasLandRegistryNumber
    hasLandRegistryNumber Domain Address
hasStreet
    hasStreet Domain Address
hasZipCode
    hasZipCode Domain Address
```

```
topObjectProperty
```

```
Equivalent To
SubClass Of
    Identification
SubClass Of (Anonymous Ancestor)
Members
```

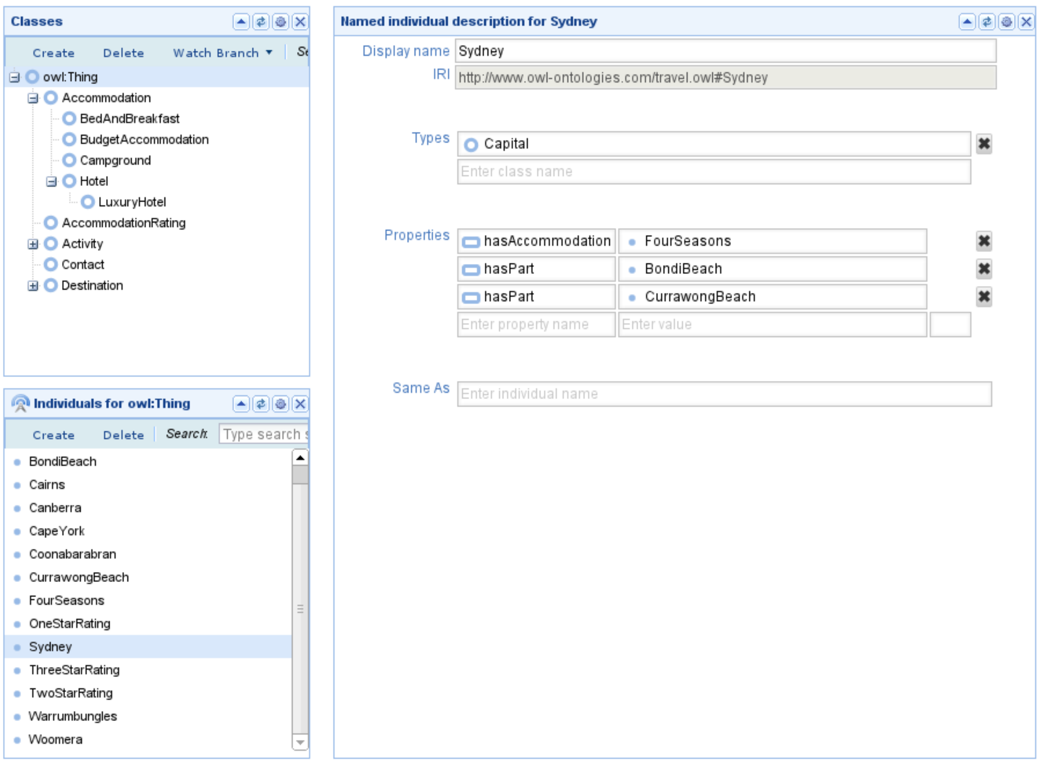
Figure 2.1: Protégé

WebProtégé[2] is a web-based version of Protégé. It does not have as many features as the desktop version. Its advantage is the hosted version where user can edit ontologies online without the need to install any software. WebProtégé offers many customization options. It supports interface for form-based editing[3]. However, the editing widgets need to be configured manually using XML configuration file.

TopBraid Composer[4] is a commercial multipurpose Semantic Web editor. It offers comprehensive application modeling. It is based on Eclipse IDE.

NeOn Toolkit[5] is another open-source ontology editor with many pluggins available. It is based on Eclipse. It seems it is not developed anymore, as last commit to the source code was in 2011.

[1] http://protege.stanford.edu/
[2] http://webprotege.stanford.edu/
[3] http://protegewiki.stanford.edu/wiki/PropertyFormPortlet
[4] http://www.topquadrant.com/products/TB_Composer.html
[5] http://neon-toolkit.org/


Classes
Create
Delete
Watch Branch
Individuals for owl:Thing
Search
Type search
Named individual description for Sydney

owl:Thing
- Accommodation
  - BedAndBreakfast
  - BudgetAccommodation
  - Campground
  - Hotel
    - LuxuryHotel
- AccommodationRating
- Activity
- Contact
- Destination

BondiBeach
Cairns
Canberra
CapeYork
Coonabarabran
CurrawongBeach
FourSeasons
OneStarRating
Sydney
ThreeStarRating
TwoStarRating
Warrumbungles
Woomera

Display name: Sydney
IRI: http://www.owl-ontologies.com/travel.owl#Sydney
Types: Capital
Enter class name

Properties	
hasAccommodation	FourSeasons
hasPart	BondiBeach
hasPart	CurrawongBeach
Enter property name	Enter value

Same As: Enter individual name

Figure 2.2: WebProtégé

A list of projects can be found is a survey published in 2007 [10]. Many of those are outdated and no longer developed.

2.3.2 Linked data browsers and editors

Tabulator[6] is a geneic data browser and editor. It works as a standalone web application or Firefox extension. It uses outline with table layout to browse data. Nodes can be expanded to go deeper in the hierarchy.

OntoWiki[7] is a platform for semantic knowledge base. It enables creating linked pages and intuitive authoring of semantic content using RDF.

RDFaCE[8] is a semantic content editor which uses alternative approach. A user does not fill information using input fields. Instead a WYSIWYG editor is used and user annotates the text content.

OpenLink Data Explorer[9] is a browser extension that allows to browse semantic data included in web pages. It displays them the using tabular layout.

[6] http://www.w3.org/2005/ajar/tab
[7] http://aksw.org/Projects/OntoWiki.html
[8] http://rdface.aksw.org/
[9] http://ode.openlinksw.com/

Sig.ma[10] is a powerful online tool for visualizing semantic data combined from multiple sources. The user can curate information by selecting which properties or sources to show and which ones to hide.

Freebase editor[11] is an editor for Freebase, which is a collaborative knowledge base curated by community. Freebase editor allows to fill in information based on a schema or link topics to each other.

2.3.3 Form generators

MetaWidget is a component for generating user interfaces. It described in detail in the next section.

AspectFaces[12] is a tool to reduce development efforts by generating UI based on model inspection. It is an alternative approach similar to MetaWidget. It was a commercial software when work on this thesis started, now it has been released as open source.

2.4 MetaWidget

MetaWidget[13] is a component that generates user interface based on the object model. It has a very flexible architecture which allows it to be used with various backends and graphical systems. It does not force any technology to be used for implementation of an information system, rather it complements the technologies chosen by application developer. Author of metawidget describes it as Object/User Interface Mapping tool (OIM) [6]. The main technique of MetaWidget is inspecting object metadata and creating User Interface (UI) widgets based on the metadata.

It is provided under open source LGPL license which allows the use of Metawidget in open source and commercial projects [14]. Commercial licenses are also available.

When using MetaWidget for generating user interface first the backend architecture is inspected and then native widgets are created. A principle of MetaWidget is to work with existing technologies and not to force particular one. Therefore it supports multiple backend technologies [6]: *annotations, Bean Validation (JSR 303), Commons JEXL, Commons Validator, Groovy, Hibernate, Hibernate Validator, Jackson, JavaBeans, Java Persistence Architecture (JPA), Javassist, JBoss Forge, JBoss jBPM, JSON, JSON Schema, OVal, REST, Scala, Seam and the Swing AppFramework.*

After the object model was inspected the user interface can be rendered using one of the following frontend technologies [6]: *Android, Google Web Toolkit (including extensions such as ExtGWT), 'plain' HTML 5 (POH5), JavaScript (including extensions such as AngularJS, Bootstrap, JQuery Mobile, JQuery UI and Node.js), Java Server Faces (including extensions*

[10]http://sig.ma/
[11]http://wiki.freebase.com/wiki/Editing_topics
[12]http://www.aspectfaces.com/
[13]http://metawidget.org
[14]http://metawidget.sourceforge.net/doc/faq/licensing.php

such as Facelets, ICEfaces, PrimeFaces, RichFaces and Tomahawk), 'plain' Java Server Pages (including extensions such as DisplayTag), Spring Web MVC, Struts, Swing (including extensions such as Beans Binding, JGoodies, MigLayout and SwingX), SWT and Vaadin.

2.4.1 Architecture of MetaWidget

The architecture of MetaWidget allows reusing functionality, flexible customization and adding of new functionality. To achieve this MetaWidget implements pipeline architecture where custom plug-ins can be inserted. The pipeline has five stages as shown on the picture:

1. Inspector - inspects backends and extracts information. Multiple backends can be combined using composition. For Java inspection is usually using reflection.
2. InspectionResultProcessor - process and modifies inspection result. It can be used to sort or exclude properties.
3. WidgetBuilder - builds native widgets for specific frontend. Multiple widget builders can be composited. We can write custom builder which can fall back to builders shipped with standard distribution.
4. WidgetProcessors - process and modify each widget. These can be used to add data binding, event handlers, validation, tooltips and so on.
5. Layout - organizes widget in the layout on the screen. We can put widgets into table, tab panels or write custom layout.

2.4.2 Comparison to other projects

MetaWidget's website includes list of other projects with comparison[15]. They do not have all of the key goals of MetaWidget:

1. to create UI widgets by inspecting existing architectures
2. not to try to 'own' the entire UI, but to focus on creating native sub-widgets for slotting into existing UIs
3. to perform inspection either statically or at runtime, detecting types and subtypes dynamically

I encourage readers to visit MetaWidget's website to get more information.

[15]http://metawidget.sourceforge.net/doc/faq/comparison.php

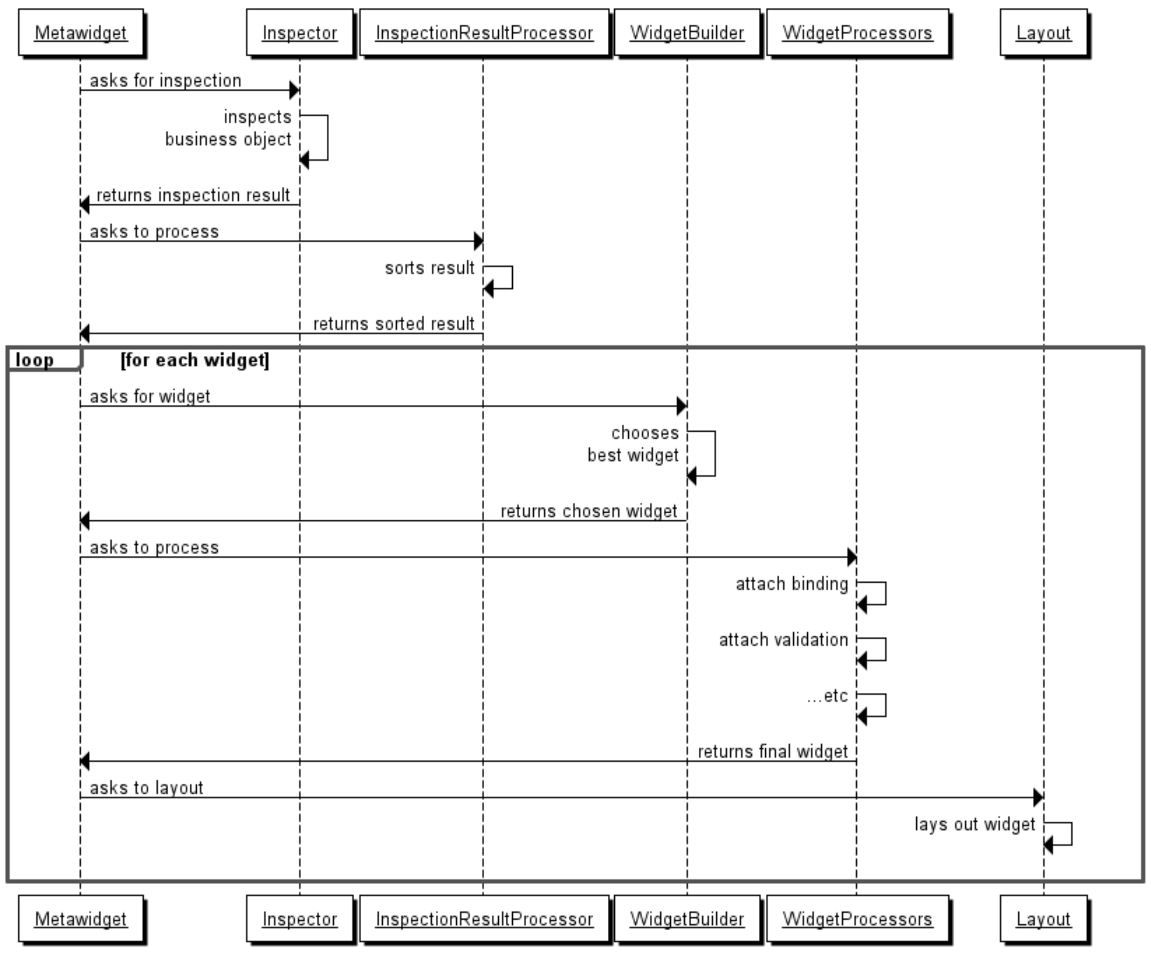


Figure 2.3: Metawidget architecture showing a five stage pipeline [13]

Chapter 3

Design

This chapter describe a design process of the system. First we will specify the software requirements. Then we will introduce and design the architecture. Lastly we will cover some user interface guidelines that will ensure easily usable interface.

3.1 Software Requirement Specification

3.1.1 Purpose

This section describes the requirements for the JOPA Forms software developed as a part of this thesis. The purpose of the JOPA Forms is to generate a user interface that is used to input data inside of an information system. The datamodel of an information system is represented in ontology using OWL. This section will also describe system constraints, interface and interactions with other software components of the system.

3.1.2 Scope

The JOPA Forms software generates user interface based on OWL ontology. It will present a dynamic form where user can fill in information for an information system. It will make easy for user to input various data types. When user is done the software will validate consistency of the data and present hints for correction to the user.

The data will be persisted to knowledge store interfacing with the JOPA layer. JOPA will be using for reasoning and checking a consistency of the data.

The application developer can override visual representation of the UI to match application specific requirements for a particular information system.

The goal of this software is to provide a UI framework for developing an information system. The software is flexible and not tied to a specific application. The result is not an complete information system but an sample application demonstrating features and usage of this framework. A application developer can learn from this example application and use it to develop a complete information system.

3.1.3 User characteristics

There are three types of users that interact with the system: *end users*, *ontology designers* and *application developers*. Each type of these users use system differently and has different requirements.

End user

This is an user of an information system developed using the ontology framework. The user wants to input data into the system, list entries in the system and view their details. There is also additional functionality that is specific for each information system. This custom functionality must be implemented by an application developer and is not in the scope of this software.

Ontology designer

The ontology designer creates an ontology which serves as a data model for an information system. The ontology creation process is out of scope of this work and we will not focus on ontology designer's requirements. They can use tools described in chapter 2.

Application developer

The application developer takes an ontology created by an ontology designer and use it as a data model for development of an information system. JOPA Forms framework will provide classes and methods for creating a data model for the target programming language. There will be classes that generate the user interface and integrations that save the data filled in by end users to persistent storage. The framework will allow to be integrated into a information system and will provide ways to extend user interface and change its visual appearance.

3.1.4 Functional requirements

3.1.4.1 User interface

1. List entities

   A user should see list of all the entities stored in the information system.

2. View details of an entity

   A user should view details of selected of an entity selected from list.

3. Edit information

   A user should be edit information. This includes editing properties of existing entities, creating and deleting entities.

4. Specify multiple values

   A property of an entity can be assigned multiple values if the datamodel supports it. The user interface should guide user to add information that does not invalidate integrity constraints.

5. Show inferred values for selection

   The system should present a list of possible values for particular property. These values are results inferred by a reasoning engine. The interface should take the object hierarchy into consideration.

6. Validate values

   The user interface should accept only valid values for a given datatype. The interface should hint the user that data is not valid.

7. Check integrity constraints

   The system should check integrity constraints. It should perform a fast check on the application level. After that it should perform a consistency check using reasoning engine. The system should hint the user to correct the information when integrity constraints are violated.

3.1.4.2 Application Programming Interface

1. Generate datamodel from an ontology

   The system should generate a datamodel in the native programming language from an OWL ontology. A developer can use the generated classes to implement custom business logic.

2. Generate UI from datamodel

   The system should generate a user interface from the datamodel. A developer can integrate this UI into an application. The generated UI should not interfere with other custom UI created specifically for the application.

3. Support extending or adding UI components

   A developer should be able to extend default functionality. If needed the developer can create new components and use them instead of the default ones. The visual appearance of the UI can be customized. This includes changes in layout, fonts, colors etc.

4. Check IC and infer values with a reasoner

   The software should implement integration with a reasoning engine. The reasoner should be used to get inferred values and check whether the integrity constraints are not violated. In case of an violation the transaction should be rolled back so that consistency of the data is guaranteed.

5. Persistence layer

   A developer should be able to select which one of the supported backend technologies is used and configure the storage options to suit the needs of an particular information system.

3.1.5 Non-functional requirements

1. **User interface**

   The user interface will be implemented as a web-based application.

2. **System environment**

   The backend will use Java programming language and JOPA framework for persistence storage. The application implementation will be based on J2EE. It will support Tomcat or Glassfish application servers.

3. **Performance and security**

   Because we are not developing complete application, hardware or security requirements are not part of this specification. These requirements depend on software requirements of a specific information system and are not thus considered.

3.1.6 Documentation

The software is documented within this thesis. It will cover architecture design and implementation details. There will be short instructions how to setup development environment and integrate the software in an application. There will also be a list of OWL constructs used to generate integrity constraints.

As the user facing application is not part of the scope, user manual is not included.

3.2 Architecture

Using JOPA as our persistence layer influences the architecture. We can see overall JOPA architecture in the picture. This thesis focuses on the frontend side. The goal is to implement the presentation logic which is diagrammed in a box on the bottom right side of the image. This is a part that all applications share. It is the need to provide interface for user to edit and modify data.

Another part of the frontend is business logic that is specific for each application. This logic will be written and integrated by an application developer.

3.2.1 Backend

I will be building the software on the foundation of JOPA backend. JOPA will provide following functionality:

- It will generate datamotel from the OWL ontology. This is done using OWL2Java tool which is included in JOPA. The Java object model will have annotations describing underlining OWL properties.

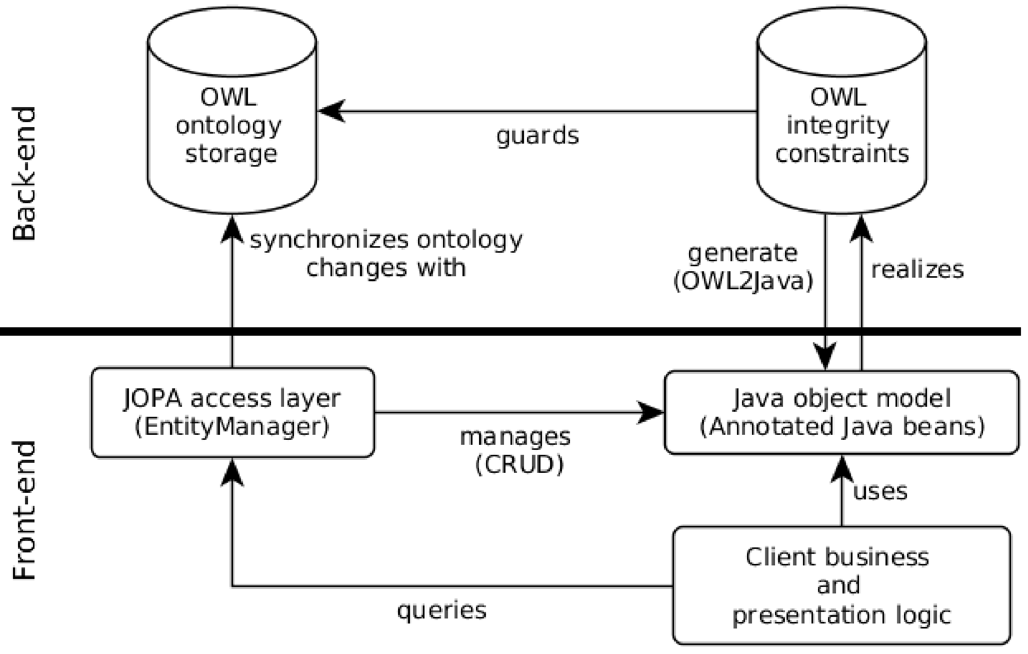

Back-end
Front-end
OWL ontology storage
OWL integrity constraints
guards
synchronizes ontology changes with
generate (OWL2Java)
realizes
JOPA access layer (EntityManager)
manages (CRUD)
Java object model (Annotated Java beans)
uses
Client business and presentation logic
queries

Figure 3.1: JOPA achitecture [14]

- It provides storage layer with API similar to JPA 2.0 standard. It supports multiple storage backends. Our main interaction will be with the EntityManager class.
- It will check whether integrity constraints are not violated. JOPA uses Pellet reasoning engine.
- We will get list of possible values using SPARQL query.

3.2.2 Frontend

For the rendering of user interface we will use MetaWidget library described in chapter 2. A implementation of a storage layer using JOPA will be implemented. The system will use JavaServer Faces (JSF) framework. The main logic will be implemented inside a Managed Bean.

We will also need to implement custom components for displaying value suggestions and extend UI related to validation. Details of implementation will be described in the next chapter.

3.3 User Interface

3.3.1 Ontology visualization

A survey describing different methods for ontology visualization is presented in [12]. This survey categorize characteristics of ontology visualization techniques and lists software implementations where they are applied. Each group of methods provide specific features. Their usefulness depends on the type of an application and requirements. However, methods grouped in one category may have elements of the other categories.

Each category can be also further divided between 2-dimensional and 3-dimensional interfaces. Generally 3D interfaces have ability to present more information in a limited space but add more complexity for the user interaction. 2D interfaces are generally easier to use and users are more familiar with them.

Indented list

Indented list is used by most of the visualization systems, including Protégé. It is the same concept used in numerous file browsers which the users are familiar with. Another advantage is simplicity of implementation and representation. Node labels do not overlap as it is often using other methods. It is also convenient for quick browsing.

One problem of *indented list* method is that it represents tree and not a graph. This is well suited for displaying inheritance relations but role relations are not always clear. However it has been proven in several evaluations that this type of visualization seems to perform better than other visualizations used for hierarchies [12].

Node–link and tree

Node–link and tree techniques represents ontologies as a set of interconnected nodes in a tree layout. Nodes are displayed in a top down (or left to right) positioning which offers a good overview of hierarchical structures. The tree node-link methods typically make inefficient use of screen space as root side of the tree is completely empty and opposite side is overcrowded or requires excessive scrolling. Several methods try to address this issue. One way to remove clutter is to allow retracting and expanding of subhierarchies. This method is seems to be effective for representing an overview of the hierarchy, however, only for small trees.

Zoomable

Zoomable visualizations present the nodes in the lower levels of the hierarchy nested inside their parents with smaller size than their parents. Changing the current viewing level is done by zooming in and enlarging child nodes. Zoomable interfaces (ZUIs) seem to bee effective for locating specific nodes. However, they do not offer and effective overview of the hierarchical structure and they do not support user in forming a mental image of the hierarchy.

Space-filling

Space-filling techniques use the whole screen space by subdividing the space among its children. Well-known example of this category is a Treemap. One of its problems is that no space remains for the internal nodes of the tree which makes it difficult to reconstruct the hierarchical information.

Focus + context or distortion

Focus + context or distortion methods are based focused node being in center and other nodes are placed around it reduced in size as they get further from the center. When another node is selected it is centered and whole layout repositioned. An advantage is that every node of interest can be easily moved towards the center of the tree to show more details. However, they do not maintain a constant positioning of the nodes, which may be disorienting.

3D Information landscapes

3D Information landscapes are commonly used in virtual reality environments for document management. The nodes are placed on a plane as 3D objects using color and size for encoding properties. They benefit from having extra dimension where more information can be presented. They have not been used much in practice and they lack extensive evaluations.

3.3.2 Form design

The Web contains a wide range of different form design solutions for similar aspects and problems. An overview of published research and guidelines how to make forms more usable is presented in [9]. To simplify many explored aspects of form design, authors classify different topics as follows: *(1) form content, (2) form layout, (3) input types, (4), error handling and (5) form submission.* In the following text we summarize some of those guidelines that can be applied for design of the ontology forms.

Form content

The way the form should be designed depends on the information we need to ask from the users. It is suggested to keep questions in an intuitive sequence, *e.g., first ask for the name, then the address and, at the end, for the telephone number.* A strategy for user-centered design is to map the natural environment familiar to the user as closely as possible to the virtual one.

We should indicate which information is essential and which is optional. This can be achieved by separating required from optional fields and use color and asterisk to mark required fields. This leads to faster filling of the forms.

An example application of these guidelines is to present fields for an address in the same order as it is written on a conventional mail envelope. Fields with required cardinality will have highlighted labels.

Form layout

Forms consist mainly of labels and input fields which can be placed in different variations. People fill the forms fastest when the labels are placed above the corresponding input fields. However, it is also recommended to use left-aligned labels for unfamiliar data where one wants users to slow down and consider their answers.

A form layout should not be divided into more than one column. Only one question should be asked per each row. A length of a field is recommended to match the length of the expected answer. This helps to guide users to what kind of answer is expected.

Input types

To restrict the number of of options checkboxes, radio buttons and dropdown menus can be used. The use of checkboxes instead of list boxes for multiple selection items is preferred. For single selection items use radio buttons for up to four options, otherwise use dropdown menu to save screen space. The options should be ordered in an intuitive sequence (e.g., weekdays in the sequence Monday, Tuesday, etc.). If no meaningful sequence is possible, an alphabetical order should be considered.

The results revealed that using a drop-down menu for a date selection is best when format errors must be avoided. Using a simple input field and placing the format requirements left or inside the text box leads to faster completion time.

Error handling

The users should be guided through forms as quickly and error-free as possible. Providing input restrictions (such as minimum password length, date entry format, etc.) in advance leads to fewer errors.

Error messages should be polite and explain to the user in familiar language that a mistake has occurred. It should clearly describe what the mistake is and how it can be corrected. Errors must be noticeable at a glance, colors and icons are suggested for highlighting.

If the error occurs, the already completed fields should never be cleared.

Form submission

To ensure optimal form submission, the submit button should get disabled as soon as it has been clicked to avoid multiple submissions. It is a good practice to show confirmation after the from has been sent. It is not recommended to provide reset buttons, as they can be clicked by accident. If used anyway, they should be visually distinctive and placed away from submit buttons.

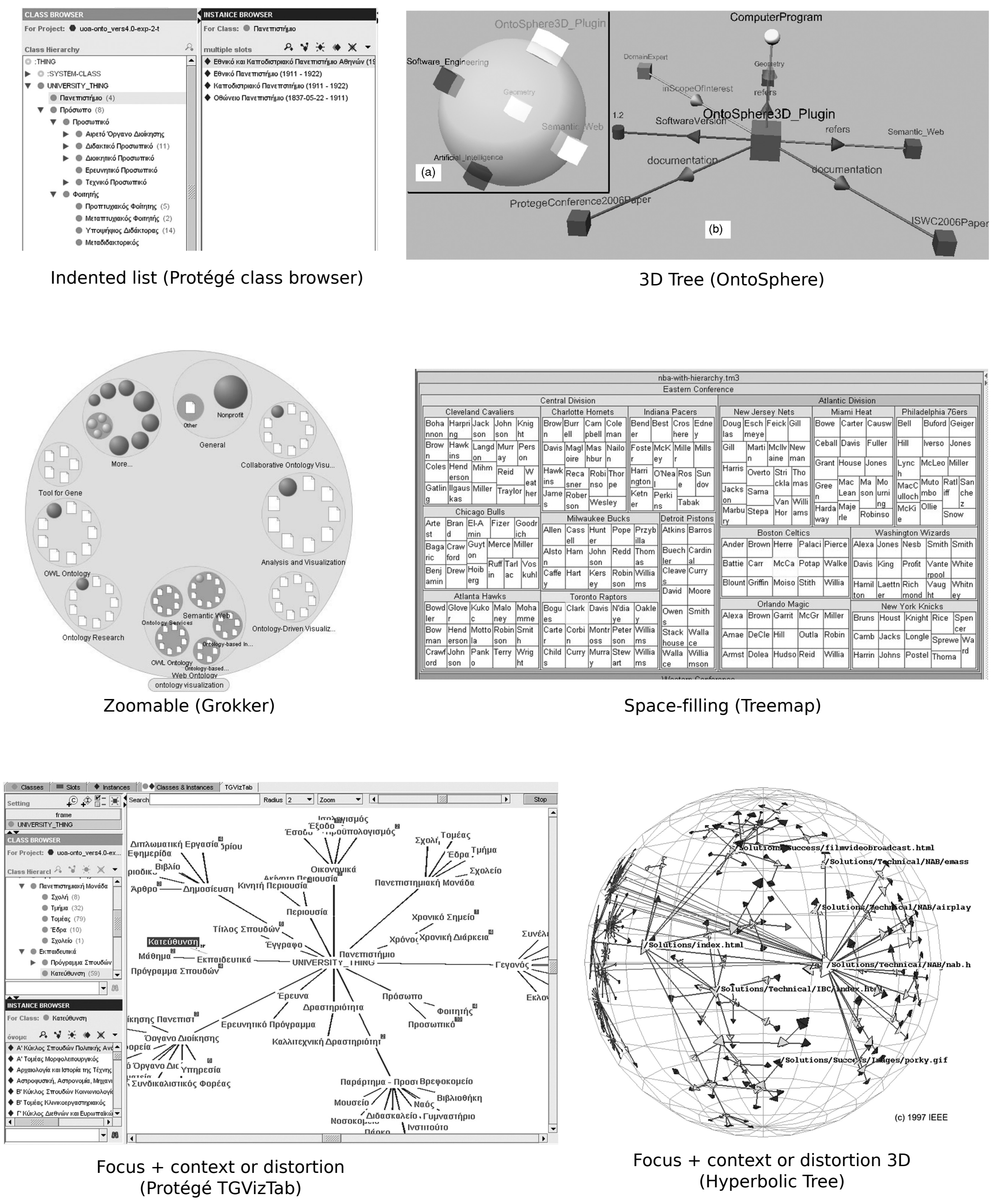

Indented list (Protégé class browser)
3D Tree (OntoSphere)
Zoomable (Grokker)
Space-filling (Treemap)
Focus + context or distortion
(Protégé TGVizTab)
Focus + context or distortion 3D
(Hyperbolic Tree)

Figure 3.2: Example of different ontology visualization techniques [12]

Chapter 4

First implementation

An implementation of this software can be divided into three main parts.

1. Extracting annotations
   First we need to extract ontology annotations from the data model. This will serve as metadata for widget builder which will generate the user interface.
2. Ontology controller
   Then we need to implement a controller that will feed the metadata into UI generator, load and store ontology data.
3. Customizing widgets
   The last part is to customize widgets to support desired behavior specific to ontology data.

4.1 Extracting annotations

We will describe how integrity constraints from OWL are translated into Java annotations using OWL2Java tool which is part of JOPA. The we will go through strategy how to inspect these annotations to extract metadata for graphical builder.

4.1.1 Annotations description

Id

Annotates attribute that is used as a primary key. It can have `generated` argument which declares that a primary key should be automatically generated when it is not explicitly set during object creation. This is analogous to `auto-increment` functionality used in relational database.

Example:

```
@Id(generated = true)
protected String id;
```

OWLAnnotationProperty

This annotates how basic RDF types are mapped onto object properties. In the example we see that `rdfs:label` is stored using `name` attribute:

```
@OWLAnnotationProperty(iri = CommonVocabulary.RDFS\_LABEL)
protected String name;
```

OWLClass

This annotation is used to annotate Java class to IRI represented in the ontology:

```
@OWLClass(iri = Vocabulary.s\_c\_Address)
public class Address {
  ...
}
```

OWLDataProperty

A data property declares a primitive type like numbers, strings or dates. It is usually used in combination with `ParticipationConstraints`.

Example:

```
@OWLDataProperty(iri = Vocabulary.s\_p\_hasAltitude)
protected double hasAltitude.
```

OWLObjectProperty

An object property declares relation to other object. It is usually used in combination with `ParticipationConstraints`.

Example:

```
@OWLObjectProperty(iri = Vocabulary.s\_p\_hasAddress)}
protected Address hasAddress;
```

ParticipationConstraints

The participation constraints declare integrity constraints for a given property. The `min` and `max` arguments specify minimum and maximum cardinality:

```
@ParticipationConstraints({
  @ParticipationConstraint(owlObjectIRI = Vocabulary.s_d_PlainLiteral,
                           min = 1, max = 1)
})
```

Properties

Annotates property used to store additional properties. This servers as a work-around for the Java static typing.

```
@Properties
protected Map<String, Set<String>> properties;
```

Types

Annotates property used to store additional Type information. This servers as a workaround because Java does not support multiple inheritance.

```
@Types
protected Set<String> types;
```

Here we can see the example annotations for a Person class:

```
@OWLClass(iri = Vocabulary.s_c_Person)
public class Person {

    @OWLAnnotationProperty(iri = CommonVocabulary.RDFS_LABEL)
    protected String name;
    @OWLAnnotationProperty(iri = CommonVocabulary.DC_DESCRIPTION)
    protected String description;
    @Types
    protected Set<String> types;
    @Id(generated = true)
    protected String id;
    @Properties
    protected Map<String, Set<String>> properties;
    @OWLDataProperty(iri = Vocabulary.s_p_hasUserName)
    @ParticipationConstraints({
        @ParticipationConstraint(owlObjectIRI = Vocabulary.s_d_Literal,
                                 min = 1, max = 1)
    })
    protected String hasUserName;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setDescription(String description) {
        this.description = description;
    }
    public String getDescription() {
        return description;
    }

    public void setTypes(Set<String> types) {
        this.types = types;
    }

    public Set<String> getTypes() {
        return types;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }

    public void setProperties(Map<String, Set<String>> properties) {
        this.properties = properties;
    }

    public Map<String, Set<String>> getProperties() {
        return properties;
    }

    public void setHasUserName(String hasUserName) {
        this.hasUserName = hasUserName;
    }

    public String getHasUserName() {
        return hasUserName;
    }

}
```

4.1.2 Annotations inspector

MetaWidget already implements inspector for Java objects. As we described in previous chapters, MetaWidget includes composite inspector which combines results of multiple inspectors. We can utilize this for extracting integrity constraints. It is done by creating `JopaAnnotationsInpector` which extends `BaseObjectInspector`.

The recommended way is to override `inspectProperty` of the `BaseObjectInspector`. However, the base inspector follows Java Bean convention and properties are inspected on the getter method. Consider following example:

```
@OWLClass(iri = Vocabulary.s_c_Person)
public class Person {

    // property declaration with OWL annotations
    @OWLDataProperty(iri = Vocabulary.s_p_hasUserName)
    @ParticipationConstraints({
        @ParticipationConstraint(owlObjectIRI = Vocabulary.s_d_Literal,
                                 min = 1, max = 1)
    })
    protected String hasUserName;

    // setter method
    public void setHasUserName(String hasUserName) {
        this.hasUserName = hasUserName;
    }

    // getter method
    public String getHasUserName() {
        return hasUserName;
    }
}
```

MetaWidget's inspector uses `getHasUserName()` getter method which does not contain the annotations. We need to use the property declaration with annotations. After looking into MetaWidget's code I found out we need to override `inspectTraits` method instead.

4.1.3 Configuration file

To make MetaWidget use this inspector we will append JopaAnnotationsInpector into configuration file in `/WEB-INF/metawidget.xml`
(notice the line containing <`jopaAnnotationsInspector xmlns="java:cz.kbss.jopa"/`
>).

```
<?xml version="1.0"?>
<metawidget xmlns="http://metawidget.org"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://metawidget.org
    http://metawidget.org/xsd/metawidget-1.0.xsd" version="1.0">

  <htmlMetawidget xmlns="java:org.metawidget.faces.component.html">
```

```
    <inspector>
      <compositeInspector xmlns="java:org.metawidget.inspector.composite"
        config="CompositeInspectorConfig">
        <inspectors>
          <array>
            <propertyTypeInspector
              xmlns="java:org.metawidget.inspector.propertytype"/>
            <metawidgetAnnotationInspector
              xmlns="java:org.metawidget.inspector.annotation"/>
            <facesAnnotationInspecto
              xmlns="java:org.metawidget.inspector.faces"/>
            <xmlInspector xmlns="java:org.metawidget.inspector.xml"
              config="XmlInspectorConfig"></xmlInspector>
            <jopaAnnotationsInspector xmlns="java:cz.kbss.jopa"/>
          </array>
        </inspectors>
      </compositeInspector>
    </inspector>
  </htmlMetawidget>

</metawidget>
```

4.2 Ontology controller

The second part is to implement the controller. Its responsibility is to interact with the EntityManager and provide data for the view templates. It must handle object life-cycle, load an object in a cache, handle edits from the form and persist to the EntityManager when user hits the save button. In case integrity constraints are violated, the controller needs to pass the error messages to the view. The controller also provides the data for form autocompletion like list of instances and classes.

4.3 Customizing widgets

MetaWidget includes annotations that influence visual style. The annotations can modify the way how widgets are rendered. Placing them in the Java code is not desirable, because it is automatically generated from ontology and will be overwritten each time the ontology is changed. Instead we can utilize XML inspector and add extra metadata in file `/WEB-INF/metawidget-metadata.xml`.

4.3.1 Custom metadata

Lets use the metadata to order the fields of an address to match the convention on the mail envelope. We want the fields in the following order: Street, House Number, City, Zip code. We will use the `comes-after` option to achieve that:

```
<?xml version="1.0"?>
<inspection-result xmlns="http://metawidget.org/inspection-result"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://metawidget.org/inspection-result
    http://metawidget.org/xsd/inspection-result-1.0.xsd" version="1.0">

  <entity type="cz.mondis.sampleapp.model.Address">
    <property name="hasHouseNumber" comes-after="hasStreet"/>
    <property name="hasCity" comes-after="hasHouseNumber"/>
    <property name="hasZipCode" comes-after="hasCity"/>
  </entity>

</inspection-result>
```

We can use various other options, for example to hide fields, change their type, override item label, divide fields into sections and others. In addition to custom options can use custom WidgetBuilder to use custom components for autocompletion and limiting number of values in arrays.

4.3.2 Custom widget

We can build custom widgets using a WidgetBuilder mechanism. A WidgetBuilder must implement the following interface:

```
W buildWidget( String elementName, Map<String, String> attributes, M metawidget )
```

where W is a widget type and M is the metawidget type (it is different for example for Swing and JSF applications). Thw WidgetBuilder reads attributes returned by the inspector and instantiates a widget based on it.

4.3.3 Evaluation

In spite MetaWidget's well-thought and stream-lined architecture I wasn't able to put all the pieces together and create a working application. I found that the application logic scattered across multiple XML files is very hard to debug.

Then I decided implement a custom solution from scratch to compare the complexity. It will not be as general as MetaWidget but I got further with it.

Chapter 5

Second implementation

5.1 Architecture

There two main approaches to design the architecture. First one is to render everything on the server and then transfer HTML which is displayed in the browser. Second approach is to transfer only data and render it in browser using Javascript. Second approach has advantage that user interface can respond faster, because it can handle most of the actions right in the browser without the need to transfer and get response from the server. It is more flexible and allows more advanced features.

Also by transferring only the data, we will decouple the implementation. It will be not tied to particular language. An API will be specified to describe data transfer between server an client. By clear separation we can then reuse both parts separately. We can use the API for other frontends. We can also use the form component for information systems written in other languages than Java.

Lets look now at the architecture stack on image 5.1. It consists of three main layers.

First we have data model described in OWL files. We use OWL2Java tool from JOPA suite to convert it to Java files that contain model classes. Java can be utilized for writing business logic. Everytime ontology changes the tool can be automatically rerun to update the Java model.

Server-side layer extracts the schema from generated Java classes using reflection. A manager based on JOPA handles loading and persisting entities to a permanent storage. It also handles serialization which converts Java objects into data that can be transfered over network and back. We will use JSON format because it is native Javascript data format and it is well supported in other languages as well.

HTTP layer servers the scripts and wraps the above functionality into API. We use Java Server Faces (JSF) and a Java application server (Glassfish or Tomcat).

On the client side the data gets deserialized and forms are rendered based on the schema. Browser will cache changes to objects. A client side validation will be performed. When user hits the save button, data objects are serialized and sent to server for persisting. JOPA

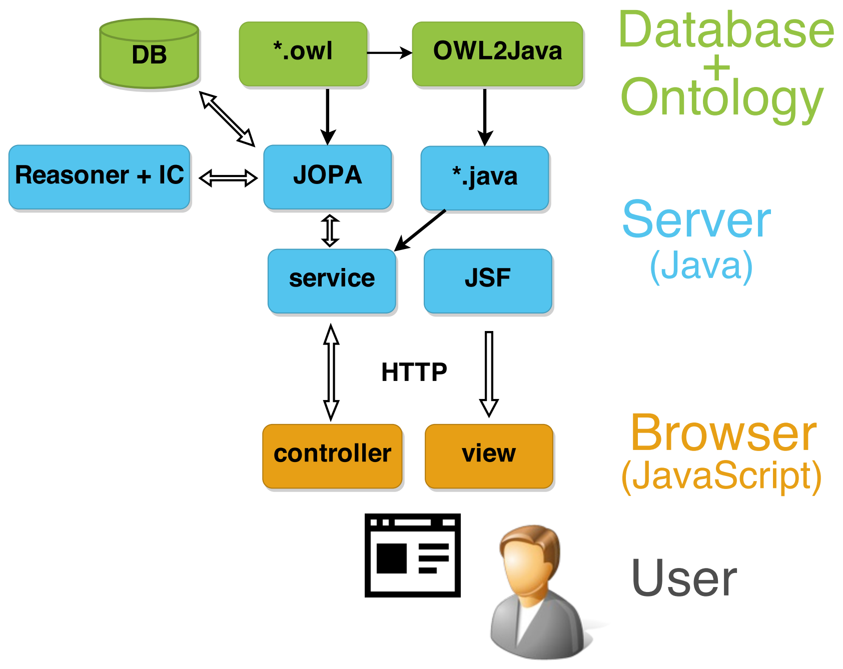


Figure 5.1: Architecture of the client-heavy JOPA Forms implementation

layer is using a reasoning engine to ensure the data consistency. If integrity constraints are violated the transaction gets rolled-back.

5.2 Backend

This section describes main components on the backend side. Key methods of their interfaces are described.

5.2.1 Extractor

The Extractor goes through a specified package or list of classes and extracts the schema.

- `Extractor(Class[] classes)`
  Constructor, sets which classes to work with
- `Extractor(String namespace)`
  Constructor, sets classes from a specified namespace
- `JSONObject getSchema()`
  Returns a schema for the list of classes passed in constructor
- `JSONObject getFields(Class inputClass)`
  Returns a schema for the specified class

- `Class[] getClasses()`
  Gets a list of classes. This method is useful when a namespace is passed and we need the list of classes to use in other components.

5.2.2 Converter

The Converter handles conversion from Java objects to JSON and back. It uses Gson library[1]. A tricky part is to handle circular references among objects. This is achieved by traversing the object tree and replacing the references with special attribute `$ref`. When the data gets back, we need to traverse the JSON and replace back the `$ref` attribute with references.

Another thing is to specify a class. When there is an inheritance the system cannot know child class to instantiate. We use a special `$class` attribute to explicitly provide that information.

An alternative implementation that prefers XML instead of JSON could be done by reimplementing this class.

- `Converter(Class[] classes)`
  Constructor, we need to specify classes for introspection to overcome Java's static typing system

- `String toJson(Object o)`
  Converts object to JSON string containing given object graph

- `JsonElement toJsonTree(Object o)`
  Converts object to JSON, useful for further manipulation

- `Map<String,Object> fromJson(String json)`
  Parses object graph encoded in JSON

5.2.3 Manager

The manager acts as an interface to EntityManager. It is used for basic create, read, update and delete (CRUD) operations. It also handle listing all entities for a given class by using SPARQL query against the ontology storage.

- `Manager(EntityManager em)`
  Constructor, pass the entity manager used for an ontology storage

- `String persist(String data)`
  Persist a given serialized data

- `Object find(Class cls, String id)`
  Fetches a specified entity

[1] https://code.google.com/p/google-gson/

- `remove(Class cls, String id)`
  Removes the specified entity

- `List<String> getEntities(String iri)`
  Gets list of entities for a class by given IRI

5.3 HTTP API

This section describes the HTTP API which acts as a layer between a backend and a frontend. The API is provided by the `RestService` class and it wraps the functionality of the Manager.

- `GET /schema`
  Gets the JSON schema

- `POST /find {cls},{id}`
  Finds an entity. This endpoint uses POST method instead of more appropriate GET because the application server does let through URL as a parameter (the id of an entity takes a form of an URL).

- `POST /persist`
  Persist an set of objects

- `POST /remove {cls},{id}`
  Removes an entity. This endpoint uses POST method instead of more appropriate DELETE because the application server does let through URL as a parameter.

- `GET /instances/{class}`
  Get list of instances for a given class.

- `GET /subclasses/{class}`
  Get list of subclasses for a given class.

5.4 Frontend

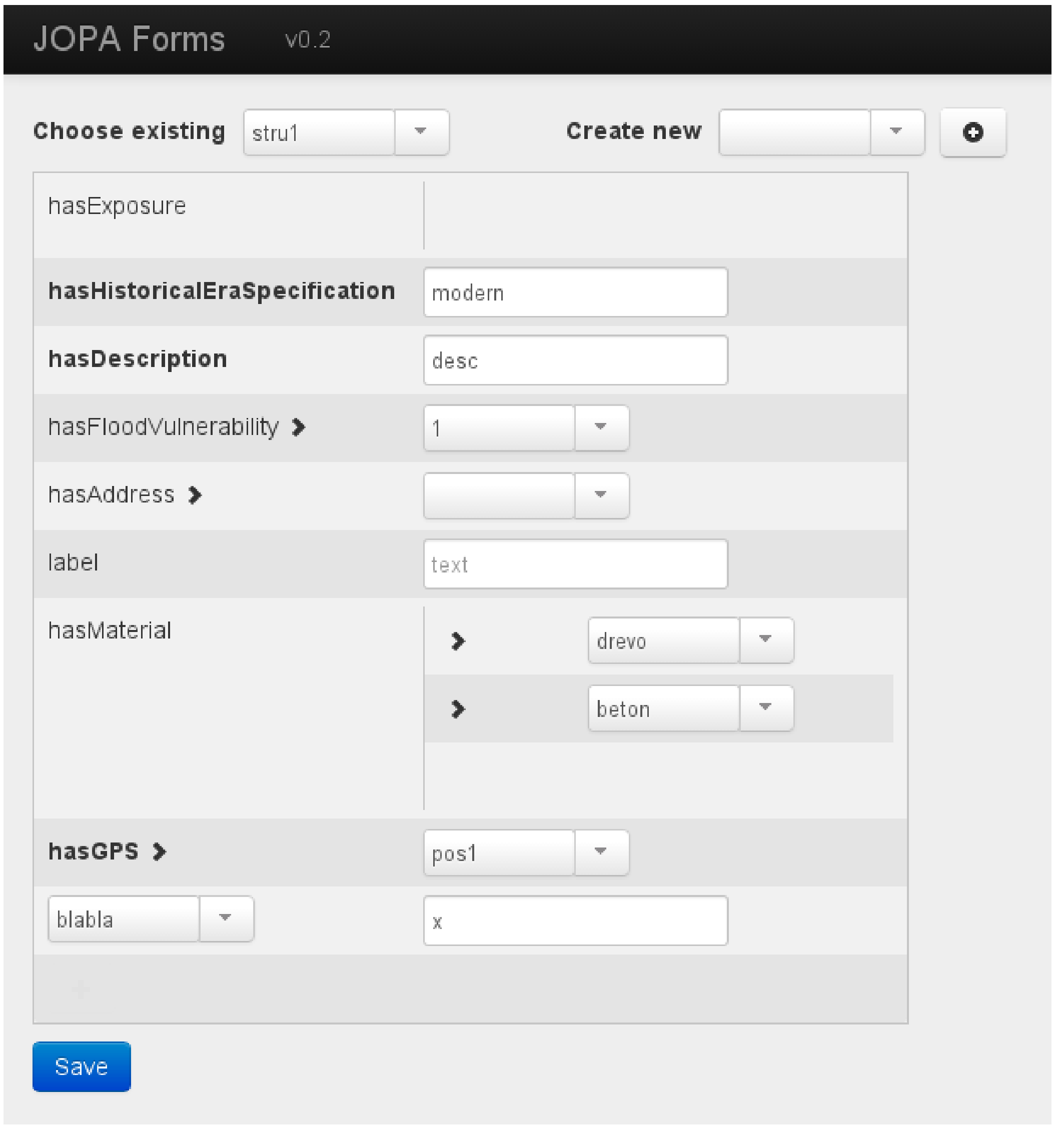
The frontend is implemented using Model-View-Controller (MVC) pattern using Backbone.js[2] library. Each widget is a separate view. There are views for primitive values like *strings*, *numbers* and *dates*. There are also two composite views - *object* and *array*. These create nested child views. For deciding which child view to use is the `dispatch` method that returns a view based on the schema type.

There are static methods in the *jopa* namespace that provide interface to the HTTP API for loading and saving the data.

[2]http://backbonejs.org/

To change a visual appearance it is possible to change the HTML template for a given view or use CSS. To add additional functionality one can create another view and add it to dispatch method.

In the picture 5.2 we can see the example interface. The bold items indicate required fields. Object properties are indicated with the right chevron symbol (>), they can be expanded to edit the properties of a nested object. There is a button on the bottom left side.



JOPA Forms v0.2

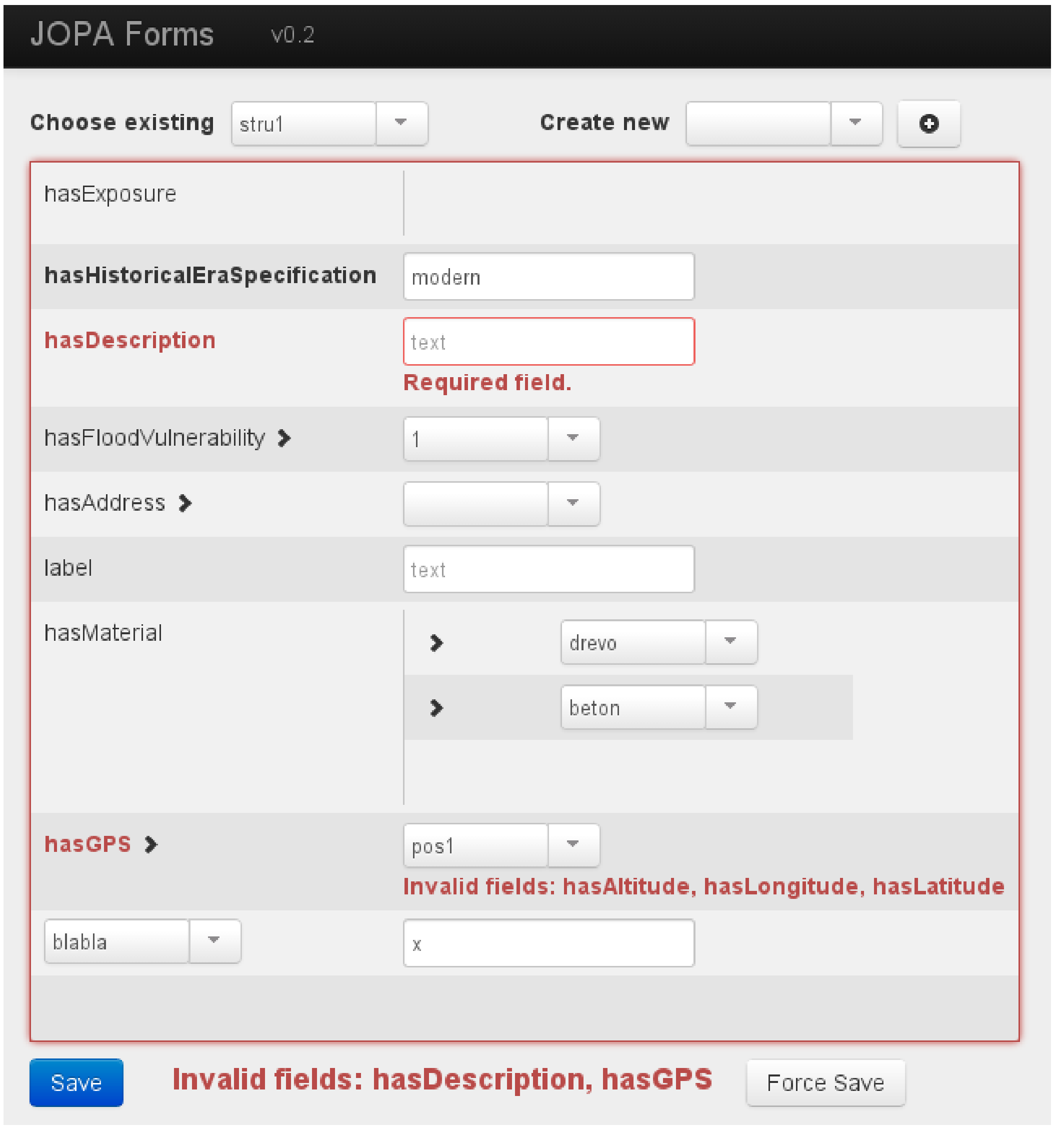
Choose existing stru1 **Create new**

hasExposure	
hasHistoricalEraSpecification	modern
hasDescription	desc
hasFloodVulnerability >	1
hasAddress >	
label	text
hasMaterial	> drevo
	> beton
hasGPS >	pos1
blabla	x

Save

Figure 5.2: JOPA Forms

In the picture 5.3 we can see the forms validation after a save button was pressed. We can see that *hasDescription* is invalid because it is missing. We can also see that *hasGPS is invalid*, because its nested fields are invalid. Integrity constraint violations are traced up the object hierarchy.


JOPA Forms v0.2

Choose existing stru1 Create new

hasExposure	
hasHistoricalEraSpecification	modern
hasDescription	text **Required field.**
hasFloodVulnerability ›	1
hasAddress ›	
label	text
hasMaterial	› drevo › beton
hasGPS ›	pos1 **Invalid fields: hasAltitude, hasLongitude, hasLatitude**
blabla	x

Save **Invalid fields: hasDescription, hasGPS** Force Save

Figure 5.3: JOPA Forms showing error messages for violated integrity constraints

Chapter 6

Usage

In this chapter we briefly describe how to use the OWL ontologies with integrity constraints.

6.1 Creating Ontologies

The easiest start is to model an ontology in Protégé editor. After we are done we save it as RDF/XML. Then we add declarations to use integrity annotations:

```
<!DOCTYPE rdf:RDF [
...
<!ENTITY ic "http://krizik.felk.cvut.cz/ontologies/2009/ic.owl#" >
...
]>

<rdf:RDF xmlns="http://krizik.felk.cvut.cz/ontologies/2013/ic-company.owl#"
...
xmlns:ic="http://krizik.felk.cvut.cz/ontologies/2009/ic.owl#"
...
>

<owl:AnnotationProperty rdf:about="&ic;isIntegrityConstraintFor"/>
```

After that we add the integrity constraints. Note the *isIntegrityConstraintFor*, it means which set of IC we will use as there can be more that one set in the file. We can use *owl:qualifiedCardinality*, *owl:minQualifiedCardinality* or *owl:maxQualifiedCardinality* to specify cardinality.

```
<owl:Axiom>
  <ic:isIntegrityConstraintFor>company-0.1</ic:isIntegrityConstraintFor>
  <owl:annotatedSource rdf:resource="&company;Entity"/>
  <owl:annotatedProperty rdf:resource="&rdfs;subClassOf"/>
  <owl:annotatedTarget>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Restriction>
          <owl:onProperty rdf:resource="&company;hasResidence"/>
          <owl:allValuesFrom rdf:resource="&company;Address"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&company;hasResidence"/>
          <owl:onClass rdf:resource="&company;Address"/>
          <owl:qualifiedCardinality
          rdf:datatype="&xsd;nonNegativeInteger">1</owl:qualifiedCardinality>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:annotatedTarget>
</owl:Axiom>
```

Then we run the OWL2Java tool:

```
java -cp ... cz.cvut.kbss.jopa.owl2java.OWL2Java transform -m mapping-file \
      -p example.modelcompany -c company-0.1 -d ./src \
      http://krizik.felk.cvut.cz/ontologies/2013/ic-company.owl
```

Where mapping file is used to translate IRIs to local files:

```
http://krizik.felk.cvut.cz/ontologies/2013/ic-company.owl > ./ic-company.owl
```

6.2 Data Types

OWL datatypes are mapped to Java using `cz.cvut.kbss.jopa.owlapi.DatatypeTransformer` class. There is an overview in the following table:

OWL	Java type	JSON representation
OWL2Datatype.RDF_PLAIN_LITERAL OWL2Datatype.XSD_STRING OWL2Datatype.RDF_XML_LITERAL	java.lang.String	string
OWL2Datatype.XSD_BOOLEAN	java.lang.Boolean	boolean
OWL2Datatype.XSD_INT OWL2Datatype.XSD_INTEGER	java.lang.Integer	integer
OWL2Datatype.XSD_DOUBLE OWL2Datatype.XSD_FLOAT	java.lang.Double	number
OWL2Datatype.XSD_DATE_TIME OWL2Datatype.XSD_DATE_TIME_STAMP	java.util.Date	datetime

Chapter 7

Conclusion

I have explored possibilities for automatic generation of user interfaces for ontology-based information systems. It is a very complex task with many obstacles.

I was not able to put all the changing and evolving pieces together to produce a working result. I was able only to create a prototype. Therefore there is also no documentation other than this thesis. The main contribution of this work is the exploration and process itself.

First of all the ontology does not imply how the use cases and the ways should interact with the system. Every information system can have various requirements that might not be supported and customization must be made. Sometimes the customization of an existing interface can be more complicated than writing the interface from scratch.

The JOPA framework is in a rapid development and changes broke the system several times during development. This should get better now. Also programmers documentation should be written.

For the future work I recommend top-down approach instead of the bottom-up. For this work I've been using bottom-up approach, I tried to design and implement general architecture and then tried to fit it into and ontology. I think better way would be the top-down when use requirements for the information system are collected and ontology is created. Then the user interface development is based on those requirements and ontology does specifies only the data model and not user interaction.

From an implemented information system the user interface components can be abstracted and generalized. There are also many different widget libraries which use different conventions. In MetaWidget's example it tries to support many of them but then it needs to be common denominator of those technologies and it does not use them to their full potential.

The ontology-based systems is a field with many research opportunities. I think it will play important role in the future. Ontology systems can save cost when developing information systems. However, it is important to keep in mind that the cost of implementing the new ontology-based paradigm must not be higher than the cost of savings it can provide.

Bibliography

[1] OWL web ontology language guide [online], http://www.w3.org/TR/owl-guide/, 2004.

[2] OWL web ontology language overview [online], http://www.w3.org/TR/owl-features/, 2004.

[3] OWL web ontology language reference [online], http://www.w3.org/TR/owl-ref/, 2004.

[4] SPARQL query language for RDF [online], http://www.w3.org/TR/rdf-sparql-query/, 2008.

[5] OWL 2 web ontology language document overview [online], http://www.w3.org/TR/owl2-overview/, 2012.

[6] Metawidget user guide and reference documentation [online], http://metawidget.sourceforge.net/doc/reference/en/html-single/index.html, 2013.

[7] RDF 1.1 concepts and abstract syntax [online], http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/, 2014.

[8] RDF schema 1.1 [online], http://www.w3.org/TR/rdf-schema/, 2014.

[9] J. Bargas-Avila, O. Brenzikofer, S. Roth, A. Tuch, S. Orsini, and K. Opwis. Simple but crucial user interfaces in the world wide web: introducing 20 guidelines for usable web form design. *User Interfaces*, page 1–10, 2010.

[10] J. Cardoso. The semantic web vision: Where are we? *IEEE Intelligent Systems*, 22(5):84–88, 2007.

[11] S. Grimm, A. Abecker, J. Völker, and R. Studer. Ontologies and the semantic web. In J. Domingue, D. Fensel, and J. A. Hendler, editors, *Handbook of Semantic Web Technologies*, pages 507–579. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[12] A. Katifori, C. Halatsis, G. Lepouras, C. Vassilakis, and E. Giannopoulou. Ontology visualization methods—a survey. *ACM Computing Surveys*, 39(4):10–es, Nov. 2007.

[13] R. Kennard. *Derivation of a General Purpose Architecture for Automatic User Interface Generation.* PhD thesis, University of Technology, Sydney, 2011.

[14] P. Křemen. *Building Ontology-Based Information Systems.* PhD thesis, Czech Technical University in Prague, Feb. 2012. 10.

[15] M. Ledvinka. *Application Access to Persistent Ontologies*. Master thesis, Czech Technical University in Prague, Faculty of Information Technology, 2013.

[16] B. A. Myers and M. B. Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, page 195–202, 1992.

[17] R. Studer, V. R. Benjamins, and D. Fensel. Knowledge engineering: principles and methods. *Data & knowledge engineering*, 25(1):161–197, 1998.

Appendix A

List of acronyms

API Application programming interface

CRUD Create, read, update and delete

CWA Closed World Assumption

DL Description Logic

IRI Internationalized resource identifier

JOPA Java OWL Persistence API

JPA Java Persistence API

JSF JavaServer Faces

MDA Model-driven architecture

MVC Model-View-Controller

OIM Object Interface Mapping

OIS Ontology-based Information System

OOP Object-oriented programming

OWA Open World Assumption

OWL Web Ontology Language

RDF Resource Description Framework

RDFS RDF Schema

ZUI Zoomable user interface

Appendix B

Contents of the CD

```
|- ui               Source for the frontend prototype
|- app              Source of the prototype backend
|- dundalek.pdf     Text of this thesis
\- contents.txt     List of CD contents
```