



Technische Universität Hamburg-Harburg

Hamburg University of Technology

Institute for Software Systems



Dräger Medical

Section User Interfaces

Master Thesis

Ontology Based Data Access

for Separating User Interfaces
from Application Logic

Hannes Molsen

#20730260

First Supervisor: Prof. Dr. Ralf Möller

Second Supervisor: Prof. Dr. Helmut Weberpals

Issue Date: April 18, 2012

Filing Date: October 18, 2012

Erklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tage eingereichte Masterarbeit vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Alle Abbildungen in dieser Arbeit sind eigene Darstellungen. Wo Abbildungen übernommen und abgeändert wurden, ist die Quelle der ursprünglichen Abbildung angegeben.

.....

Lübeck, 18.10.2012

Hannes Molsen

All hyperlinks to resources on the Internet have been checked and last accessed on Monday, 2012-10-15.

Abstract

The separation of user interfaces from the underlying application logic is a constant research topic. The misuse of Reenskaug's Model View Controller as a system architecture provides only modular separation on source code level, while UI and application logic remain in tight semantic coupling.

To break this coupling, we will present an approach, which applies the highly efficient and scalable techniques of ontology based data access. It is thereby possible to provide an ontology as intermediate formal model to semantically separate the two system components, whilst still being performant enough to display large amounts of frequently changing continuous data.

The presented concept includes patterns to derive an entire OBDA system, including database schema, ontology and mapping rules from an entity relationship data model. In contrast to other existing separation approaches, the presented one can easily be used to enhance existing software systems. This is proved by adapting an existing medical decision support system to the implemented OBDA system by altering only a single line of existing source code.

Zusammenfassung

Die Trennung von Benutzerschnittstellen von der zugrundeliegenden Anwendungslogik ist ein konstantes Forschungsfeld. Durch den Missbrauch von Reenskaugs Model View Controller als Systemarchitektur ist lediglich eine modulare Trennung auf Quelltextebene erreichbar, während die Komponenten nach wie vor eng gekoppelt sind.

Um diese Kopplung aufzubrechen werden im dargestellten Ansatz die leistungsstarken und skalierbaren Techniken des ontologiebasierten Datenzugriff (OBDA) angewandt. Dadurch wird es ermöglicht, eine Ontologie als formales Zwischenmodell einzusetzen, um die Komponenten auf semantischer Ebene zu trennen, aber gleichzeitig den hohen Ansprüchen der großen Mengen sich schnell verändernder Daten gerecht zu werden.

Das hier vorgestellte Konzept enthält Muster, welche es ermöglichen von einem *Entity Relationship* Modell ein komplettes OBDA System abzuleiten, inklusive des Datenbankschemas, der Ontologie und den notwendigen Abbildungsregeln. Im Gegensatz zu anderen Ansätzen kann der präsentierte auf bestehende Softwareprodukte angewandt werden, ohne in den ersten Entwicklungsphasen berücksichtigt werden zu müssen. Um dies zu beweisen wurde das Konzept auf ein bestehendes medizinisches Entscheidungssystem angewandt. Dazu musste lediglich eine einzige Zeile existierenden Quelltextes angepasst werden.

Contents

1. Introduction	8
1.1. Motivation	8
1.2. Objectives	8
1.3. Contributions	9
2. Background	10
2.1. Description Logics	10
2.1.1. Languages	10
2.1.2. Terminological and Assertional Descriptions	11
2.1.3. Reasoning	12
2.1.4. The DL-Lite Family	14
2.2. Ontologies	14
2.2.1. Definitions	15
2.2.2. Languages and Syntaxes	15
2.2.3. Querying Ontologies	20
2.2.4. Ontology Editors	20
2.2.5. Applications and Types of Ontologies	22
2.3. Ontology Based Data Access	23
2.3.1. Conventional RDF Stores	23
2.3.2. Relational Data vs. Graph Data	24
2.3.3. Quest Reasoner	26
2.3.4. OWL 2 QL	28
2.3.5. Mappings	29
2.3.6. The -ontop- framework	29
2.4. User Interface Architecture	30
2.4.1. Reenskaug's Models-Views-Controllers	30
2.4.2. Separated Presentation	31
2.4.3. Limitations of Current Approaches	32
2.4.4. User Interface Data	36
2.4.5. Ontology-Enhanced User Interfaces	37
2.5. Related Work	38
2.5.1. Paulheim et al.	39
2.5.2. Tilly et al.	39
2.5.3. Summary	39
2.6. SmartPilot View	40
2.6.1. Problems	41

3. Concept	42
3.1. Overview	42
3.2. Data Flow	44
3.2.1. Application to User Interface	44
3.2.2. User Interface to Application	46
3.2.3. Eventing Mechanisms	47
3.3. Key Components	48
3.3.1. Data Model	48
3.3.2. Database	49
3.3.3. Ontology	51
3.3.4. Mapping Rules	53
3.4. Connections to the Core	54
3.4.1. Connecting the Application	54
3.4.2. Connecting the User Interface	56
4. Implementation	57
4.1. Data Model	57
4.2. Database	58
4.2.1. Autogenerated Primary Keys	58
4.2.2. Representation of Graph Data	58
4.2.3. Units of Measure	60
4.3. Ontology	60
4.3.1. Enumerations	61
4.3.2. Representation of Graph Data	61
4.3.3. Units of Measure	61
4.4. Mapping Rules	62
4.4.1. Representation of Graph Data	62
4.4.2. Units of Measure	63
4.5. Connecting the Application	64
4.5.1. Inverse Object Relational Mapping	64
4.5.2. Adapter	64
4.5.3. Automatically Generated Primary Key	65
4.5.4. Representation of Graph Data	66
4.5.5. Units of Measure	67
4.6. Connecting the User Interface	67
4.6.1. Java: Accessing Quest	67
4.6.2. SPARQL: Accessing the Data	68
4.6.3. Representation of Graph Data	69
5. Evaluation	71
5.1. Correct Result	71
5.2. Mapping Rules Completeness	72
5.3. Write Performance	73
5.4. Read Performance	74

5.5. Summary	77
6. Conclusion and Future Work	78
6.1. Conclusion	78
6.2. Future Work	79
A. Overview: Description Logics	95
A.1. Basic Languages	95
A.2. Extension Operators	95
A.3. Abbreviations	95
B. SmartPilot View: Ontology	96
C. SmartPilot View: UI Data Subset	99
D. System Overview	100
E. Evaluation Results	101
E.1. Mapping Completeness	101
E.2. Case	102
E.3. Drugs	103
E.4. NSRI Curves	104
E.5. Patient	105
E.6. Used Drugs	106

1. Introduction

1.1. Motivation

The separation of user interfaces from application logics is a constant research topic since the 1970s, when Reenskaug introduced Model View Controller (MVC). Over the decades, MVC has become the silver bullet of user interface separation and is often misused as entire system architecture. While it provides an efficient means of achieving modular separation on the source code level, this procedure inevitably leads to a tight semantic coupling between both components. Depending on the implementation, this architecture can make the deployment of the two components on physically different systems (spatial separation), the integration of multiple applications into one or more user interface, or the parallel development by independent teams difficult, if not impossible. To overcome the above shortcomings, semantic separation of user interfaces and application logics is a necessary condition.

Ontologies provide a conceptual access point to data, enabling to exploit knowledge about, e.g., the interrelations of such data individuals. In contrast to conventional data stores as relational databases, the semantics of the data is accessible at run-time through so called inference systems. Thereby, ontologies are predestined for being used as an intermediate layer to achieve a high degree of separation. But at the present time, no such inference system for native ontology stores is capable of handling the high amount of frequently changing data, which is exchanged between user interfaces and applications.

Ontology based data access (OBDA) is a comparably young technology, which is able to provide a conceptual view on relational data sources through an ontology. The -ontop-Quest reasoner is an inference system which enables such access, by using mapping rules to translate SPARQL ontology queries to SQL database queries. It is thereby possible to exploit both, the conceptual knowledge of the ontology, and the mature algorithms of query answering for relational data. In contrast to native ontology approaches, Quest is thereby able to maintain the high scalability and efficiency of the underlying relational database management system.

1.2. Objectives

The objectives of this work are to investigate, how ontology based data access can be applied to existing software systems in order to increase the degree of separation between user interfaces and application logic. Therefore, the requirements of the data exchange between the two system components have to be analyzed, and suitable OBDA

techniques have to be examined, to show the general feasibility of such an approach. On that basis, a new system architecture is to be developed, which results in the desired degree of separation. Often the requirements for software systems increase as they evolve. Therefore, in contrast to other existing approaches, the designed architecture has to be retrofitable to enhance existing software systems. This shall be proved by applying the concept to an existing medical decision support system, and measuring, whether the architecture is able to handle the high amounts of continuous data.

1.3. Contributions

The main contributions of this work are twofold. In the research topic of user interface architecture, we will present a new approach to achieve semantic separation between user interfaces and applications, which, at the same time, is able to integrate the data of different applications.

In the topic of ontology based data access, this work will contribute potentially automatable patterns for deriving an entire OBDA system, including database schema, ontology and mapping rules, from an entity-relationship data model.

2. Background

2.1. Description Logics

Description Logics are a family of logic based knowledge representation formalisms. Their foundations lie in the field of semantic networks, developed in the 1960s by R.F. Simmons and M.R. Quillan [Sim63, Qui63], which are the descendants of KL-ONE [SBI85]. In Description Logics a specific domain is described by its concepts (or classes), roles (or relationships/properties) and individuals. Most Description Logics are subsets of First Order Logic (FOL), but trade expressive power for decidability [Bor96] and reasoning performance [TH03]. In this Section, we will describe common Description Logic languages (2.1.1), and explain their allowed operators and the naming convention. We will illustrate the distinction between TBoxes and ABoxes (2.1.2), and introduce an example ontology, which will be referred throughout this chapter. The basic concepts of reasoning and query answering over ontologies will be listed in 2.1.3, and 2.1.4 gives an overview about the DL-Lite family, which is the foundation of OBDA systems.

2.1.1. Languages

With regard to First Order Logic, the concepts in Description Logics are equivalent to unary predicates, the roles to binary predicates, and the individuals to constants. These basics are valid for all DL languages. They differ in the set of operators or constructors that are allowed to build complex concept and role expressions in that language. This has an influence on the expressiveness as well as on the complexity of the Description Logic.

To distinguish between the different DLs, a naming convention has been agreed upon, which directly shows the allowed operators. There are three different types of basic logics, \mathcal{AL} , \mathcal{FL} , and \mathcal{EL} , as well as several possible extensions allowing more operators, each denoted by a single capital letter.

The most commonly used constructors are the ones that correspond to boolean operators, and the ones for quantification, i.e. complement / negation (\neg), concept conjunction (\sqcap), disjunction (\sqcup), universal (\forall), and existential (\exists) quantification. These are the basis for \mathcal{ALC} , which is one of the most important DLs, as it is the smallest propositionally closed one [SSS91]. The logic \mathcal{ALC} , together with \mathcal{R}_+ for transitive roles, often acts as the basis for further extensions, and is thus abbreviated with \mathcal{S} .

In the following explanations the letter a will be used for individuals, the letters C and D will be used for concepts, while r and s will denote roles. Its operators are shown in the upper part of Table 2.1.

Table 2.1.: Description Logics: Languages and semantics

DL	Name	Syntax	Semantics
\mathcal{ALC}	Top Concept	\top	$\Delta^{\mathcal{I}}$
	Bottom Concept	\perp	\emptyset
	Negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
	Conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
	Disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
	Value Restriction	$\forall r.C$	$\{d \in \Delta^{\mathcal{I}} \mid \forall e. (d, e) \in r^{\mathcal{I}} \rightarrow e \in C^{\mathcal{I}}\}$
	Existential Restriction	$\exists r.C$	$\{d \in \Delta^{\mathcal{I}} \mid \exists e. (d, e) \in r^{\mathcal{I}} \wedge e \in C^{\mathcal{I}}\}$
\mathcal{H}	Role hierarchy	$r \sqsubseteq s$	$r^{\mathcal{I}} \subseteq s^{\mathcal{I}}$
\mathcal{O}	Nominals	I	$I^{\mathcal{I}}$ Singleton
\mathcal{I}	Inverse roles	r^{-}	$\{(d, e) \mid (e, d) \in r^{\mathcal{I}}\}$
\mathcal{N}	Cardinality Restriction	$\leq nr$	$\{d \in \Delta^{\mathcal{I}} \mid \#\{(d, e) \in r^{\mathcal{I}}\} \leq n\}$
		$\geq nr$	$\{d \in \Delta^{\mathcal{I}} \mid \#\{(d, e) \in r^{\mathcal{I}}\} \geq n\}$
\mathcal{Q}	Qualified Cardinality Restriction	$\leq nr.C$	$\{d \in \Delta^{\mathcal{I}} \mid \#\{(d, e) \in r^{\mathcal{I}} \mid d \in C^{\mathcal{I}}\} \leq n\}$
		$\geq nr.C$	$\{d \in \Delta^{\mathcal{I}} \mid \#\{(d, e) \in r^{\mathcal{I}} \mid e \in C^{\mathcal{I}}\} \geq n\}$
(\mathcal{D})	Datatype Properties, Data Values or Data Types		

The semantics of Description Logics is based on so called *interpretations*. With $\Delta^{\mathcal{I}}$ being the domain of \mathcal{I} , which is a non-empty set, and $\cdot^{\mathcal{I}}$ an interpretation function, such an interpretation is a pair $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$. Therein, $\cdot^{\mathcal{I}}$ assigns an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ to each individual a , a set $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ to each concept name C , and a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ to each role name r . An overview of syntax and semantics of \mathcal{ALC} and frequently used extensions can be found in Table 2.1.

2.1.2. Terminological and Assertional Descriptions

Description Logics describe knowledge in terms of facts. An example knowledge base will formalize the facts that

- men are humans,
- fathers are men, which have a human child,
- John is a man, and
- Jack is a father.

The notions TBox and ABox [Tur10] are common to split the facts into two groups. While the TBox contains the terminological knowledge, being the statements about the concepts and their inter-relationships, the ABox contains the assertional knowledge, being facts about the individuals. Equation 2.1 shows an example for terminological

statements, and Equation 2.2 one for the assertional facts that represent the example knowledge base. It is also depicted as graph in Figure 2.1.

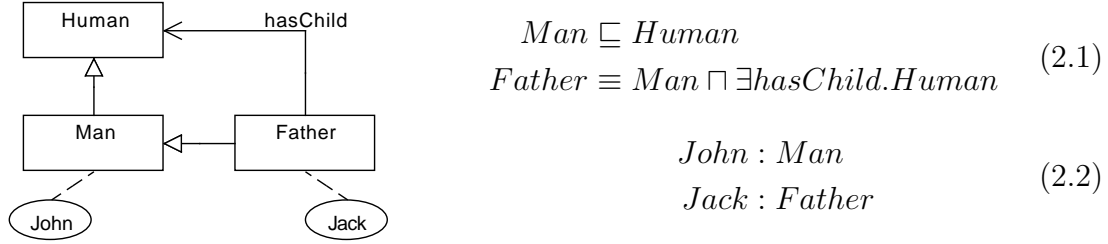


Figure 2.1.: Example Knowledge Base with TBox (2.1) and ABox (2.2)

2.1.3. Reasoning

TBox and ABox axioms together $\langle \mathcal{T}, \mathcal{A} \rangle$ form a knowledge base \mathcal{K} . This knowledge base (also called *DL-ontology*, cf. section 2.2) is accessed usually through an interface to an inference system. These systems are capable of making knowledge explicitly available, which is contained only implicitly in the TBox and ABox knowledge, i.e., the given facts do not contain this knowledge, but it can be derived from them. Using the example from Figure 2.1, “Jack is a man” is implicit knowledge and can be derived from the explicit facts that “Jack is a father” and “fathers are men”. As this knowledge has not been specifically expressed, it is also referred to as *new knowledge* [BL07], which has been derived from \mathcal{K} . This process is called *reasoning* and the underlying applications *reasoner*. The reasoning again can be split as well into the two parts *terminological reasoning* and *assertional reasoning*. While the first comprises only operations on the TBox, the latter uses the whole knowledge base pair $\langle \mathcal{T}, \mathcal{A} \rangle$. Baader and Lutz identify satisfiability and subsumption as the two fundamental inference problems for terminological, and consistency and instance checking for assertional knowledge [BL07]:

Satisfiability A concept description C is *satisfiable* with respect to a TBox \mathcal{T} if there exists a common model of C and \mathcal{T} .

Subsumtion A concept description C is *subsumed* by a concept description D with respect to a TBox \mathcal{T} if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every model \mathcal{I} of \mathcal{T} (written $C \sqsubseteq_{\mathcal{T}} D$).

Consistency An ABox \mathcal{A} is *consistent* with respect to a TBox \mathcal{T} if there exists a common model of \mathcal{A} and \mathcal{T} .

Instance Checking An individual name a in an ABox \mathcal{A} is an *instance* of a concept description C with respect to a TBox \mathcal{T} if $a^{\mathcal{I}} \in C^{\mathcal{I}}$ for all models \mathcal{I} of \mathcal{A} and \mathcal{T} (written $\mathcal{A} \models_{\mathcal{T}} C(a)$).

They state that some additional inference problems, such as *equivalence* can be reduced to the fundamental ones in a trivial way, while some others can only be reduced to a composition of the them. These are called compound inference problems, and need to be mentioned separately, as the naive serial execution of the basic problems would lead to intolerable run-time behaviour:

Classification Given a TBox \mathcal{T} , compute the restriction of the subsumption relation “ $\sqsubseteq_{\mathcal{T}}$ ” to the set of concept names used in \mathcal{T} .

Realization Given an ABox \mathcal{A} , a TBox \mathcal{T} , and an individual name a , compute the set $R_{\mathcal{A},\mathcal{T}}(a)$ of those concept names A that are used in \mathcal{T} , satisfy $\mathcal{A} \models_{\mathcal{T}} A(a)$, and are minimal with this property with respect to the subsumption relation “ $\sqsubseteq_{\mathcal{T}}$ ”.

Retrieval Given an ABox \mathcal{A} , a TBox \mathcal{T} , and a concept C , compute the set $I_{\mathcal{A},\mathcal{T}}(C)$ of individual names a used in \mathcal{A} and satisfying $\mathcal{A} \models_{\mathcal{T}} C(a)$.

The instance *retrieval* is the most basic form to query an ABox, that requires the relational structure of the query to be tree shaped [CDGL98]. A more generalized way of querying ABoxes, the *conjunctive query answering* has been first studied for DLs by Calvanese et al. in 1998 [CDGL98]. Given a concept C , a role r , variables v, v' , a conjunctive Query q is a finite set of expressions $C(v)$ or $r(v, v')$, called *atoms*. $Var(q)$ is the set of variables occurring in q , which can be split in answer variables $Ans(q)$ and (existentially) quantified variables $Exq(q)$.

Conjunctive Query Answering Given an ABox \mathcal{A} , a conjunctive query q , an individual a , an interpretation \mathcal{I} of \mathcal{A} , and $\pi : Var(q) \rightarrow \Delta^{\mathcal{I}}$ a total function, such that for every $v \in Ans(q)$ there is an a , such that $\pi(v) = a^{\mathcal{I}}$ (written $\mathcal{I} \models^{\pi} C(v)$ if $\pi(v) \in C^{\mathcal{I}}$, for roles analogously). Compute π such that $\mathcal{I} \models^{\pi} atom$ for all $atom \in q$.

Therefore, the aforementioned deduction that “Jack is a man” is an *instance check*: $\mathcal{A} \models_{\mathcal{T}} Man(“Jack”)$.

As mentioned in Section 2.1.1, there is always the trade-off between computational complexity, which is important for tolerable run-time behaviour of the reasoners, and expressive power, which in turn is important to model non-trivial knowledge bases.

The most commonly found reasoner implementations are tableau¹ [BS01] based reasoners. A tableau algorithm evaluates the explicitly given facts in a knowledge base and derives new facts for as long as new facts can be deduced or a contradiction is found. Some well-known tableau-based reasoning tools are *RACER* [HM01] for the *SHIQ* description logic, *Pellet* [SPG⁺07] for *SHOIN*(\mathcal{D}) and *FaCT++* [TH06] for *SHOIQ*(\mathcal{D}), restricted to string and integer datatypes. An extension to the classic tableau is the hypertableau, used by the reasoner *HermiT* [MSH09].

A second type of reasoner implementations is based on logic programming. The deduction of knowledge is carried out by translating the knowledge base into a program

¹the notion tableaux is also used in literature, both as singular and as plural

of a logic programming language, e.g. datalog. Query answering is thereby delegated to the corresponding language interpreters. The reasoners *KAON2* [MS05] for Horn²-*SHIQ* and *OntoBroker* [DEFS98] for Horn-Logic are famous representatives of this type.

2.1.4. The DL-Lite Family

The DL-Lite family was designed by Calvanese et al. [CDGL⁺07] for the special purpose to handle reasoning, especially (conjunctive) query answering, on knowledge bases with a large number of instances. It is the foundation of ontology-based data access, as it provides the possibility of terminological reasoning in polynomial time, and answering complex queries over large ABoxes in LOGSPACE data complexity [PLC⁺08].

One big advantage and the intention behind DL-Lite is, that by restricting Description Logics to the expressivity of, e.g., entity relationship diagrams (cf. Section 2.3.2), queries to the knowledge base can be rewritten into highly efficient SQL queries to a relational database, as it will be further described in Section 2.3 Ontology Based Data Access.

A core language called DL-Lite_{core} was developed as the basis for all languages of the whole family. In addition to that the two extensions DL-Lite _{\mathcal{F}} and DL-Lite _{\mathcal{R}} were proposed. DL-Lite _{\mathcal{F}} allows for cyclic assertions, is-a on concepts, inverses on roles, domain and range on roles, mandatory participation on roles, and functional restrictions on roles [PLC⁺08]. Beneath fully capturing the DL part of RDF Schema (see Section 2.2.2), DL-Lite _{\mathcal{R}} also allows mandatory participation on roles and disjointness between concepts and roles. It has been shown [CDGL⁺07] that these two languages of the DL-Lite family are the maximum subsets for efficient query answering, as any further extension to them, even the combination of both [PLC⁺08], would increase data complexity to at least NLOGSPACE and thus hinder the delegation of ABox query answering to database management systems. The fact, that the QL subset of OWL 2 (cf. Sections 2.2.2 Languages and Syntaxes and 2.3.4 OWL 2 QL) is underpinned by DL-Lite _{\mathcal{R}} [W3C09b] emphasizes the significance of this logic family.

Poggi et al. proposed a new member to the DL-Lite family, called DL-Lite _{\mathcal{A}} . It has been specifically designed for ontology-based data access, taking for example a clear distinction between objects and values into account. They extend DL-Lite _{\mathcal{R}} by features of DL-Lite _{\mathcal{F}} , like functional properties, but restrict them by, e.g., requiring the unique name assumption (UNA)³ to retain the LOGSPACE data complexity.

2.2. Ontologies

The term ontology originates from the philosophical study (*λογος*) of being (*οντος*) and reality, as well as the basic categorizations of things and relations between them. There, the singular form is used (the ontology, as in “the biology”), whereas the computer scientist’s notion accepts plural (ontologies, as in “databases”). Following the philosophical

²a horn clause is a disjunction with at most one positive (i.e. non-negated) literal

³Logics with UNA assume that different names never refer to the same entity in the world [RN02]

definition, the term ontology in computer science is often referred to as a formal model of a domain, although several different uses and definitions exist. This section will give an overview about different definitions of ontologies, the languages that are recommended by the World Wide Web Consortium to model such ontologies and how they are used in practice.

2.2.1. Definitions

Gruber [Gru93] gives the now frequently cited definition that “an ontology is an explicit specification of a conceptualization”. He refers to Genesereth’s and Nilsson’s [GN87] definition of conceptualization, and states, that this is an “abstract, simplified view of the world” containing “objects, concepts and other entities [...] and the relationships that hold among them.”

Guarino and Giaretta later discuss seven different interpretations of the term ontology in their article from 1995 [GG95]. Their result is that ontologies in computer science are clearly independent of the philosophical antecedents and they propose a simple glossary. Therein, the term *conceptualization* is described as “an intensional semantic structure which encodes the implicit rules constraining the structure of a piece of reality” and *ontology* as “(sense 1) a logical theory which gives an explicit, partial account of a conceptualization; (sense 2) synonym of conceptualization.”

Based on these results and the definitions used by Xiaomeng and Ilebrikke [SI06], Uschold and Groninger [UG96] and Gruber [Gru93], a combined definition, which fits for this work could be:

An ontology is a formal, partial representation of a conceptualization. This conceptualization includes a set of objects⁴, concepts⁵, and their inter-relationships⁶.

This definition includes that an ontology is usually not all-embracing, but only a partial model of a certain domain of interest. The ontology also has to be a formal representation in such way that machines are able to interpret it. Often the instances of a concept are not captured by the cited definitions of ontologies, but certainly play an important role, especially in reasoning over large ABoxes.

Ontologies can also be described as a new kind of data model, which includes the semantics of the data in the form of metadata and stores data on a graph basis. This will be further described in the following Section 2.2.2 Languages and Syntaxes.

2.2.2. Languages and Syntaxes

Ontologies are one of the pillars of the semantic web vision, introduced by Tim Berners-Lee et. al in 2001 [BLHL01]. The vision that knowledge is not only written as human

⁴also being referred to as instances or individuals in literature

⁵also being referred to as categories or classes in literature

⁶also being referred to as roles or properties in literature

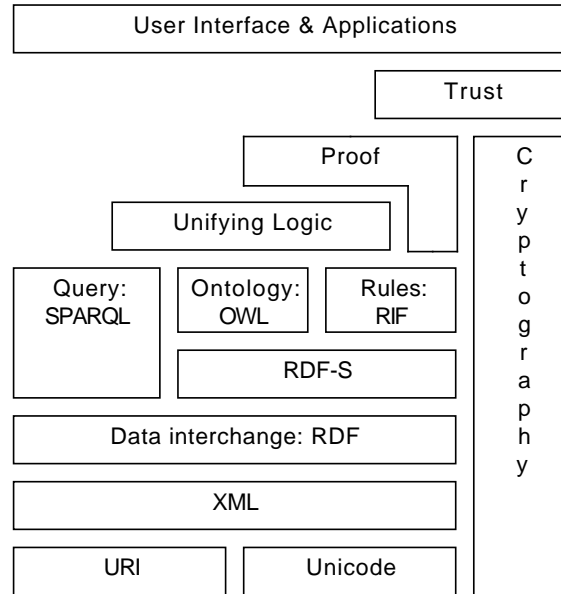


Figure 2.2.: The Semantic Web Stack [BL09]

readable text, but encoded in a way that machine agents can understand the meaning of the information. Therefore the World Wide Web Consortium W3C standardized a set of languages that meet the requirements to specify machine understandable knowledge. The foundations of those languages is the eXtensible Markup Language XML [BPM⁺08], as Tim Berners-Lee depicts in the semantic web stack, shown in Figure 2.2, but also other syntactic representations of these languages exist.

Resource Description Framework

The Resource Description Framework *RDF* [MM04] allows to formally describe knowledge. That is, expressing information about resources as logical statements, called facts. It is used to describe the data stored in an ontology. Such a fact consists of the three parts subject, predicate, and object. In “John is a man” (see Figure 2.1), *John* is the subject, *is a* is the predicate and *Man* is the object. Because of these three elements, RDF statements are referred to as triples, and therefore native storage solutions as triple stores. In each of those triples, subject, predicate and object are basically names for concrete or abstract entities in the real world, which have the URI [BLFM05] format, are literals, or blank nodes. A literal can be seen as a value, being just raw text data. In the example in Listings 2.1 and 2.2 the title “TUHH - Startseite” is such a literal. If the title itself is not known, but it is known that the resource has a title, it can be modeled using a blank node. The below listing shows this in Turtle syntax, where such a node is preceded with an underscore.

```
1 <http://www.tu-harburg.de/> dc:title _:a
```



```

1 <?xml version="1.0"?>
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dc="http://purl.org/dc/elements/1.1/">
    <rdf:Description rdf:about="http://www.tu-harburg.de/">
5      <dc:title>TUHH - Startseite</dc:title>
    </rdf:Description>
  </rdf:RDF>

```

Listing 2.1: RDF/XML syntax

```

1 @prefix dc: <http://purl.org/dc/elements/1.1/> .
  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

  <http://www.tu-harburg.de/> dc:title "TUHH - Startseite" .

```

Listing 2.2: Turtle syntax

RDF Syntaxes

To express RDF triples, different syntaxes can be used. The two most common ones are RDF/XML [Bec04] and Turtle, the Terse RDF Triple Language [Bec12]. The latter provides a less verbose syntax alternative to RDF/XML, and directly shows the triple character of the statements. This syntax was first introduced by Notation3, or short N3 [BLC11], which, beneath providing that syntax, is a superset of RDF, extending it by formulae, variables, logical implication and functional predicates. Thus Turtle is the RDF-only subset of N3. A comparing example below states that *http://www.tu-harburg.de* (subject) *has the title* (predicate) “*TUHH - Startseite*” (object), once in RDF/XML syntax (Listing 2.1) and once in Turtle syntax (Listing 2.2). The Turtle-Syntax provides the possibility to define *prefixes*, which allow for even less verbose and easy to read triples, as the long and often repeated part of the URIs is substituted and thus the focus remains on e.g. the concept or role descriptors. To avoid repetitions in Turtle, it is possible to combine multiple statements about the same subject or about the same subject and predicate. In the former, only different predicate-object pairs for one subject are delimited by a semicolon, while in the latter the different predicates are delimited by a comma. Examples of this can be found in the example ontology in Listing 2.4.

Web Ontology Language

In addition to the data description capabilities of RDF, using RDF Schema (RDFS) [GB04] and the Web Ontology Language OWL⁷ [MvH04] - the current version is OWL2 [W3C09b] - are used in the semantic web stack (cf. Figure 2.2) and allow to make metadata statements about concepts of, and the inter-relationships between resources.

⁷The abbreviation OWL instead of WOL was chosen for unambiguous pronunciation and the association to the wisdom of owls [Fin01]

While RDF was about the individuals (cf. ABox, Section 2.1.2), with RDFS and OWL it is possible to describe the TBox (cf. Section 2.1.2) classes, properties, their hierarchies as well as inheritance. For example “all men are humans”, or more formally:

```
1 :Man rdfs:subClassOf :Human .
```

Listing 2.3: All Men are Humans in Turtle Syntax

In OWL, the facts are not split explicitly into ABox and TBox statements, but if one expresses the triples in natural language, the distinction is easy: Using the singular (John is a man) usually refers to individuals and thus the ABox, whereas the plural (men are humans) usually describes concepts and thus the TBox.

Regarding the syntax, an OWL and RDFS ontology can be serialized in all notations that exist for RDF since these ontology languages are syntactically embedded into RDF. Especially RDF/XML has been chosen as interchange format between ontology software [W3C09b]. Nevertheless, there exist several others syntaxes for other purposes. The *Manchester Syntax* [HDG⁺06] for easier writing DL ontologies, *Functional Syntax* [MPPS09] to easily see the formal structure of ontologies, OWL/XML [PPSM09] for use by XML tools, and again Turtle for its RDF triple readability.

Description Logics, especially the \mathcal{SH} family (cf. Section 2.1.1), underlie the Web Ontology Language [HPSH03] which itself comes in three levels of expressive power in descending order, or in three levels of computational complexity in ascending order: OWL Full, OWL DL, and OWL Lite. One could say that OWL together with RDF provides a LISP-like syntax for the Description Logics, such that it can be written in the ASCII character set. As inference in OWL Full is undecidable [Hor05], it will not be further considered here. OWL DL is equivalent to the DL $\mathcal{SHOIN}^{(D)}$, while OWL Lite comprises the slightly simpler $\mathcal{SHIF}^{(D)}$ logic (see Table 2.1).

The new W3C recommendation *OWL 2* proposes three new profiles as subsets of its full capabilities, EL, RL and QL, all based on specific DLs which allow tractable reasoning [MFH⁺09]. The first has its basis in the Description Logic $\mathcal{EL}++$ [BBL05], the second is based on Description Logic Programs (DLP) [GHVD03], and the third, and most important for this work, is based on the description logic DL-Lite_R (cf. Section 2.1.4). Section 2.3.4 will pick up this profile, as it is the foundation of OBDA.

The reason for the proximity to Description Logics is that OWL ontologies are thereby able to exploit the already advanced research on DL, its properties and complexities, as well as to use the already implemented applications and algorithms, especially in terms of reasoners and inference systems.

Listing 2.4 shows the Turtle syntax for the example ontology from Section 2.1.2 Terminological and Assertional Descriptions.

```

1  @prefix : <http://www.draeger.com/onto/example#> .
   @prefix owl: <http://www.w3.org/2002/07/owl#> .
   @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
   @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
5  <http://www.draeger.com/onto/example> rdf:type owl:Ontology .

   :hasChild rdf:type owl:ObjectProperty ;
             rdfs:domain :Father ;
             rdfs:range :Human .

10  :name rdf:type owl:DatatypeProperty ;
       rdfs:domain :Human ;
       rdfs:range xsd:string .

15  :Father rdf:type owl:Class ;
        rdfs:subClassOf :Man ,
                        [ rdf:type owl:Restriction ;
                          owl:onProperty :hasChild ;
                          owl:someValuesFrom :Human
20                        ] .

   :Human rdf:type owl:Class .

   :Man rdf:type owl:Class ;
25   rdfs:subClassOf :Human .

   :Jack rdf:type :Man ,
           owl:NamedIndividual ;
   :name "Jack"^^xsd:string .

30  :John rdf:type :Father ,
           owl:NamedIndividual ;
   :name "John"^^xsd:string .

```

Listing 2.4: Example ontology in turtle syntax

Ontology Constructs

In the following, we will describe the subset of ontology constructs important for this work. For more information, see [W3C09a].

Class `owl:Class` A category for instances

Class Hierarchy `rdfs:subClassOf` Allows a hierarchy by adding specializations of such categories

Object Properties `owl:ObjectProperty` A relation between individuals of classes.

Datatypes `owl:DatatypeProperty` Relate individuals to data values. OWL can be used with the XML Schema datatypes [W3C04], like `xsd:string` or `xsd:integer`. The default datatype is a plain *Literal*.

```
1 PREFIX : <http://www.draeger.com/onto/example#> .  
  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .  
  
  SELECT ?x  
5 WHERE  
  {  
    ?x rdf:type :Human .  
  }
```

Listing 2.5: Querying for all Humans

2.2.3. Querying Ontologies

Data stored in the RDF triple format can be accessed with the query language SPARQL. The name is a recursive acronym for *SPARQL Protocol and RDF Query Language*, which has been standardized as W3C recommendation in 2008 [PS08]. With SPARQL it is possible to query required and optional graph patterns, as well as conjunctions and disjunctions of those.

Although SPARQL is roughly based on the SQL **SELECT-FROM-WHERE** structure, it is a pure query language, which means that the recommendation does not allow for updating, inserting or deleting in RDF graphs. It provides four different forms of queries:

SELECT returns the variables bound in a query pattern match

CONSTRUCT returns an RDF graph constructed by using the given graph template and the solutions to a query

ASK returns a boolean answer to the question if the query has a solution

DESCRIBE returns an RDF graph containing the known data about the resources from the query

For the construction of the queries the above described Turtle syntax is used, so that each triple can be seen explicitly. Variables in SPARQL have global scope and are prefixed with a question mark `?` or a dollar sign `$`. In the query in Listing 2.5, the example ontology is asked to return all humans. Following the SPARQL specification, the result of this query will be a table with one column for the variable `x`, and two rows for the individuals `:Jack` and `:John`.

To answer such queries, inference systems as described in section 2.1.3 are used. A simple **SELECT** query, for example, corresponds to a *retrieval* inference problem, more complex ones to *conjunctive query answering*.

2.2.4. Ontology Editors

To create and modify an ontology it is possible to use plain text or XML editors, but due to the possible complexity it is more convenient to fall back to more sophisticated tools. Several different tools exist, some examples of which are the open source NeOn

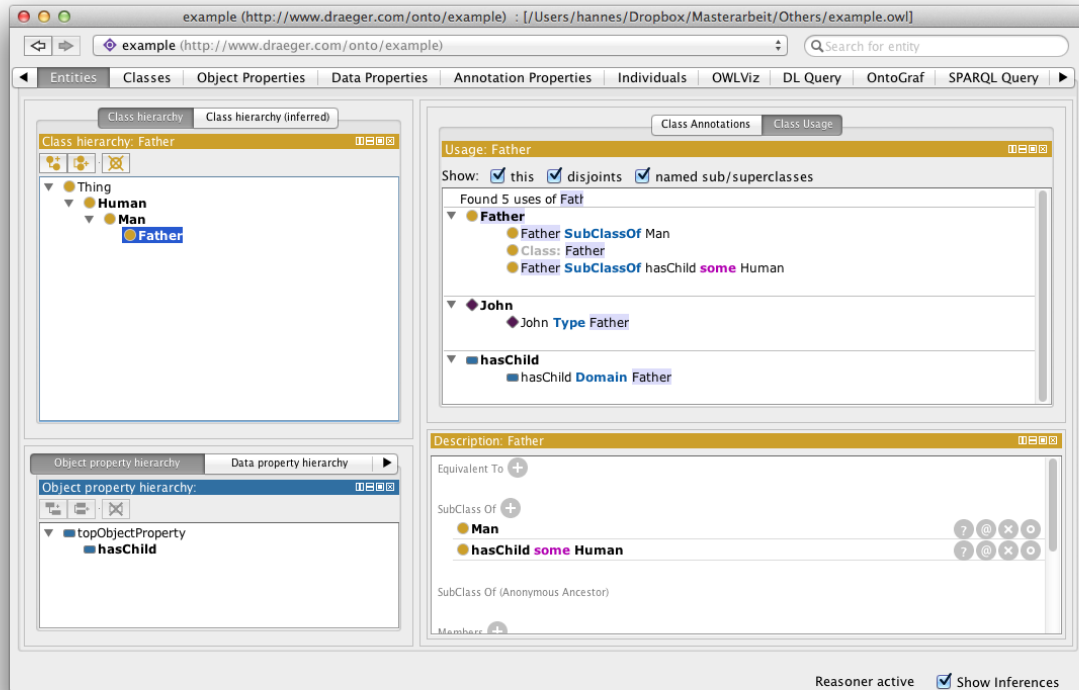


Figure 2.3.: The Protégé Ontology Editor

toolkit [HLSE08], developed as an EU IST⁸ project, the commercial OntoStudio⁹ or the discontinued SWOOP ontology editor [KSPH04] from the University of Maryland.

But with a market share of above two thirds in 2007 [Car07], the very mature Protégé ontology editor [GMF⁺03], developed at the universities of Stanford and Manchester, is the most frequently used. Despite being a five year old survey, this result seems to be still valid, since about 200 projects are listed in the Wiki¹⁰ and over 200.000 registered users¹¹ are using Protégé, which undergoes frequent updates and improvements. Figure 2.3 shows a screenshot of Protégé, displaying the aforementioned example ontology (cf. Figure 2.1).

The flexible architecture allows for plugins to be developed to extend the functionality and the field of application. An important representative for this work is the *-ontopPro-*plugin from the University of Bolzano [RLC08], decried in detail in Section 2.3.6 The *-ontop-* framework.

⁸IST is the abbreviation for Information Society Technologies <http://cordis.europa.eu/ist/about/about.htm>

⁹<http://www.semafora-systems.com/en/products/ontostudio/>

¹⁰<http://protege.cim3.net/cgi-bin/wiki.pl?ProjectsThatUseProtege>

¹¹<http://protege.stanford.edu/>

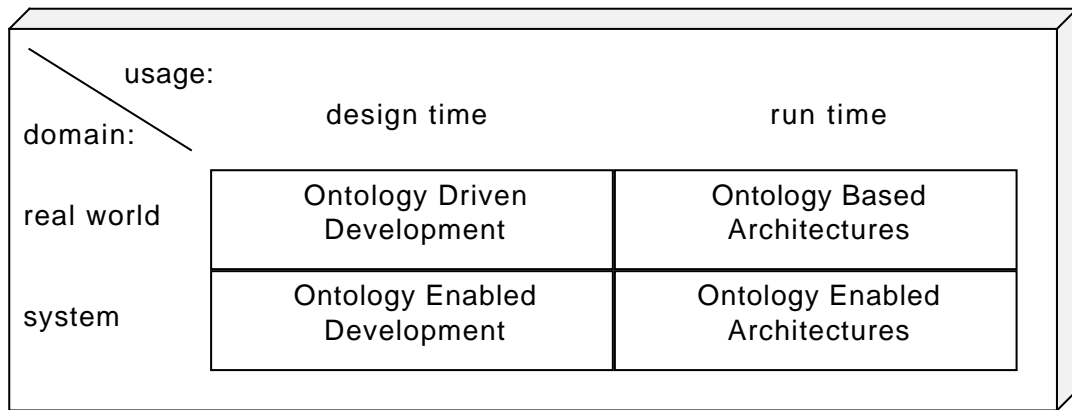


Figure 2.4.: Classification of ontologies, according to Happel and Seedorf [HS06]

2.2.5. Applications and Types of Ontologies

Ontologies can be used in many different application fields. To illustrate these uses, Happel and Seedorf [HS06] analyzed different use cases for ontologies and classified them by their usage in software engineering. The classification uses two dimensions: the ontology usage time and the ontology domain. This matrix is depicted in Figure 2.4.

Ontology Driven Development uses ontologies at design time to formalize the real world application domain.

Ontology Enabled Development uses ontologies at design time to formalize the components of the application.

Ontology Based Architectures use ontologies as central part of the application at run time.

Ontology Enabled Architectures use ontologies to achieve intelligent software infrastructure, such as semantic web services.

Guarino distinguishes ontologies by their level of dependence. In Figure 2.5, these levels are one below each other, and the arrows denote specializations. *Top-level* ontologies describe very generic concepts like space, time or event. *Domain* ontologies describe the vocabulary for a generic domain, like medicine and *task* ontologies a special task like diagnosing. The last kind is called *application* ontology. Its concepts may depend both on *task* and *domain* ontologies, and specify their concepts for a concrete application. These ontologies are therefore usually not reusable.

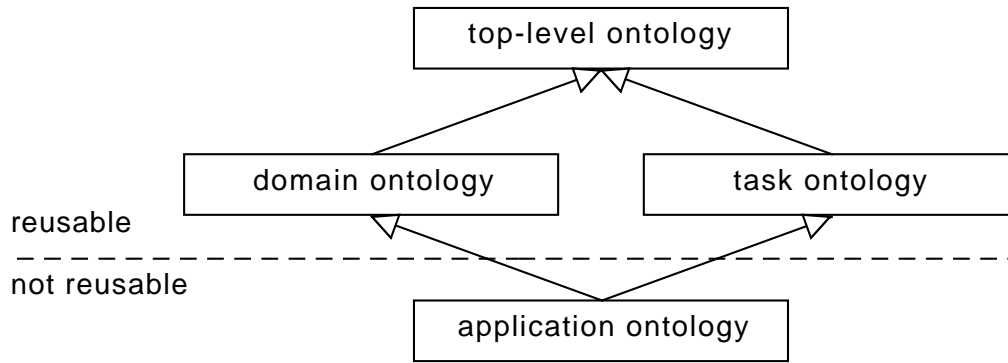


Figure 2.5.: Classification of ontologies, according to Guarino [Gua97]

Uschold and Gruninger [UG96] identified three main categories of uses for ontologies:

Communication between people and organisations, by reducing conceptual or terminological confusion between people with different viewpoints

Inter-Operability between systems, by providing an Inter-Lingua between different software systems

Systems Engineering reusable components, specification, reliability. In this context ontologies are used during system design and development rather than being used at runtime.

In the context of this work, we will make extensive use of the techniques of ontology-based data access as an intermediate layer between two software parts. The therefore created application ontology will be used as the central part of the application at runtime, which will provide access to the data for the user interface. With an ontology based architecture, the inter-operability between the UI component and the business logic component will be established.

We will further describe the use of ontologies, specifically in the context of user interfaces in Section 2.4.5 Ontology-Enhanced User Interfaces.

2.3. Ontology Based Data Access

2.3.1. Conventional RDF Stores

The triples that make up an ontology can be stored in several different ways. Well known RDF triple stores are for example the open source frameworks Apache Jena¹², Sesame¹³,

¹²<http://jena.apache.org>

¹³<http://www.openrdf.org/>

Virtuoso¹⁴ and Mulgara¹⁵, or the commercial tools OntoBroker¹⁶ from semafora or AllegroGraph¹⁷ from Franz Inc.

Basically there exist three categories of RDF storage solutions:

In-memory triple stores provide the entire RDF graph in main memory. Due to the space limitations, this is not feasible for large data sets as they occur in most real-world applications.

Native triple stores provide persistent storage by implementing own, purpose-tailored graph-based database systems.

Non-native triple stores use external, often relational database systems to store the RDF data.

While Mulgara is a pure native store, most frameworks like Apache Jena or Sesame offer all three storage solutions. For non-native storage they provide adapters for most common relational database systems.

2.3.2. Relational Data vs. Graph Data

Relational databases are one, if not the most mature way to store and query large amounts of data. The data is stored in tables, where each entry is a row, which has values for each named column. The most common query language for relational databases is SQL.

The idea to store data in that way grew in the late 1960s and had its breakthrough with Edgar Codd's publication in 1970 [Cod70]. Over forty years of constant research have improved relational database management systems and SQL query answering. Although there are other database types, like key-value, document, and object stores, which are usually summarized by the term *NoSQL*¹⁸, many applications still use the relational model, as it has proven itself to work, scale well and be reliable over decades. In addition to that, the model is backed by services, software products, and support of large companies like Oracle and IBM.

Figure 2.6 shows a direct comparison between the relational and the graph data model. This will be briefly discussed in the following.

In contrast to graph-based databases like triple stores, where the instances of concepts are stored as objects, relational databases store the values itself in cells. The discrepancy

¹⁴<http://virtuoso.openlinksw.com/>

¹⁵<http://www.mulgara.org>

¹⁶<http://www.semafora-systems.com/de/produkte/ontobroker/>

¹⁷<http://www.franz.com/agraph/allegrograph/>

¹⁸The term *NoSQL* was first used 1998 by Carlo Strozzi for a relational database system that intentionally does not use SQL. As this has nothing to do with the movement of not using relational data models, he proposes to rather use the term NoREL or something similar. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/Home%20Page

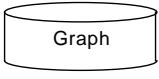
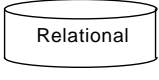
Model	Data	Identifier	Query	Metadata	Semantics At Runtime
	Cell Values	Primary Key	SQL	Columns / ERD	no
	RDF Objects	URI	SPARQL	RDFS / OWL	yes

Figure 2.6.: Data Model Comparison

between objects and values is known as *impedance mismatch*. If those systems are to be used together, this mismatch needs special attention.

Tables in RDBs must have one column or a composition of them as uniquely identifiable primary key to allow retrieval of entries and the modeling of relations via foreign keys. RDF stores use URIs to uniquely identify each object.

During the process of designing a database for a specific application, Entity Relationship Diagrams (ERDs) can be used to determine what data has to be stored and how it is grouped and related. While this information contained in the ERD, often referred to as *implicit knowledge*, is generated and used at the design time, it can not be exploited at runtime, as the following example illustrates.

Given the ontology already introduced in Figure 2.1, one could create a relational database that contains the modeled information, as depicted in Figure 2.7 and Figure 2.8. If we now query the database for all humans that have children, the answer would be empty, as there are no entries in the table `tb_hasChild` for the `hasChild` relation:

```
1 SELECT Father_ID FROM tb_hasChild
```

However, the correct answer would be 2 (**Jack**), as the semantics of our model imply, that every father has a child. Although this knowledge is contained in the ER diagram, there is no way to make use of it. Using a reasoning service on the ontology, the TBox information can be used and a SPARQL query (prefixes omitted) would give the correct answer:

```
1 SELECT ?x WHERE { ?x :hasChild _:child . }
```

Nevertheless, relational databases usually outperform triple stores in terms of data capacity and query answering performance. There exist several huge petabyte-sized relational databases, which have proved the model to be incredibly scalable and still provide acceptable run-time behaviours for query execution [Lai08]. In contrast to that, query answering in expressive description logics is limited to comparably small ABox sizes. In 2008 RacerPro [HM01], as an example, was able to cope with ABoxes of about 10.000 to 100.000 individuals [Möl08].

<table><tr><th colspan="2">tb_Father</th></tr><tr><th colspan="2">Human ID</th></tr><tr><td colspan="2">2</td></tr></table>	tb_Father		Human ID		2		<table><tr><th colspan="2">tb_Man</th></tr><tr><th colspan="2">Human ID</th></tr><tr><td>1</td><td>2</td></tr></table>	tb_Man		Human ID		1	2	<table><tr><th colspan="2">tb_Human</th></tr><tr><th>Human ID</th><th>Name</th></tr><tr><td>1</td><td>John</td></tr><tr><td>2</td><td>Jack</td></tr></table>	tb_Human		Human ID	Name	1	John	2	Jack	<table><tr><th colspan="2">tb_hasChild</th></tr><tr><th>Father ID</th><th>Human ID</th></tr><tr><td></td><td></td></tr></table>	tb_hasChild		Father ID	Human ID		
tb_Father																													
Human ID																													
2																													
tb_Man																													
Human ID																													
1	2																												
tb_Human																													
Human ID	Name																												
1	John																												
2	Jack																												
tb_hasChild																													
Father ID	Human ID																												

Figure 2.7.: Example Database

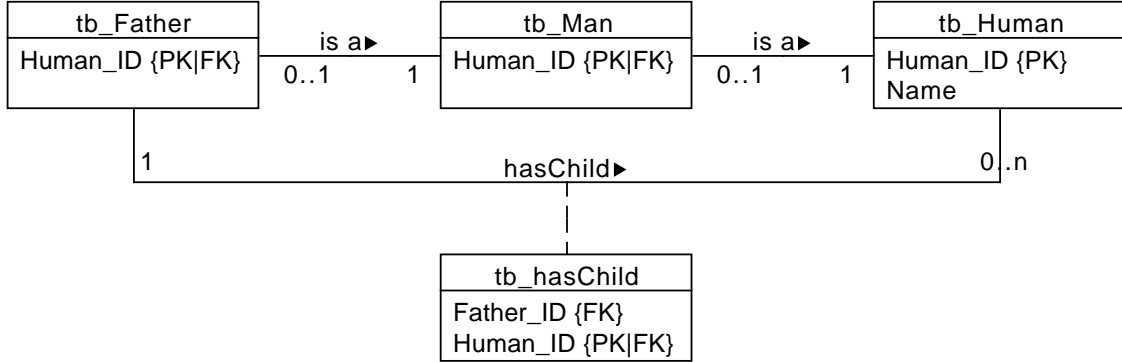


Figure 2.8.: Example Entity Relationship Diagram

2.3.3. Quest Reasoner

The idea of ontology-based data access is, to use data that is stored in relational databases, link it to a single ontology that contains a common conceptualization of the data, use this ontology as coherent access point to heterogeneous data, and utilize the sophisticated algorithms developed for relational data for fast query answering.

In contrast to extract the data from the database, transform it into triples and store it in conventional RDF stores (This process is called ETL by Rodríguez-Muro [RMC12], i.e., Extract, Transform, Load) with OBDA the data in the relational database is accessed directly, such that no data duplication, and thus, no synchronization between database and ontology is necessary. Data, which is utilized on the fly in such manner is called *virtual ABox*, and the difference to a ordinary ontology is depicted in Figures 2.9 and 2.10.

To allow this access, a reasoner called *Quest* [RMC12] has been developed at the University of Bolzano. *Quest* is specifically tailored for efficient query answering services and uses the techniques of query rewriting to retrieve answers to SPARQL queries from relational data sources. The semantic model contained in the TBox is exploited during the rewriting process, such that implicit knowledge can be obtained as in ordinary ontologies.

How a data store can be used by *Quest* is described in a so called *OBDA model*. Such

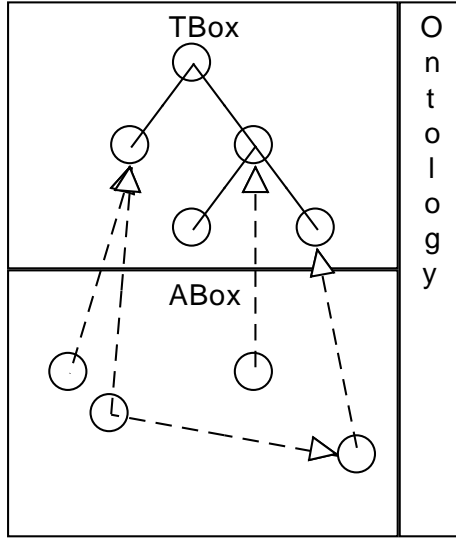


Figure 2.9.: Ordinary ABox

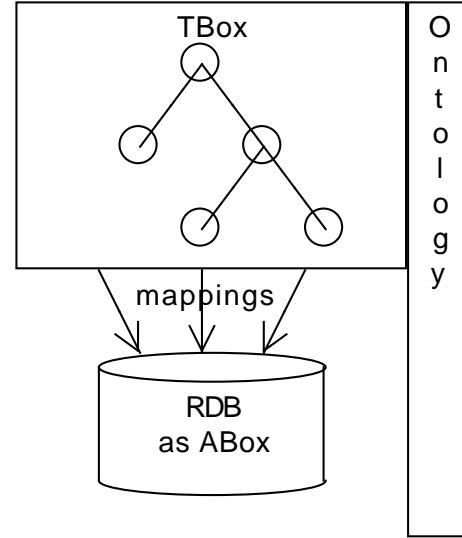


Figure 2.10.: Virtual ABox

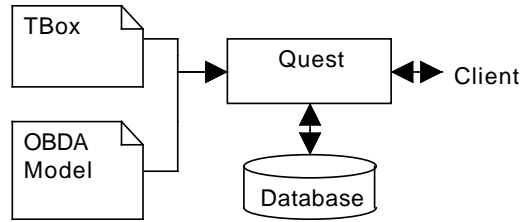


Figure 2.11.: Quest System Overview, from [RMC12]

a model consists of a *data source definition*, which contains information of how the data can be accessed (i.e., a JDBC¹⁹ connection), and a set of *mapping axioms*, which will be further explained in Section 2.3.5.

Figure 2.11 shows how quest is set up in virtual ABox mode. After Quest has performed an initialization process, a client can pose a query in SPARQL, and Quest processes it in three steps. It first rewrites it by using the knowledge from the TBox, then unfolds it into a single SQL query by using the mapping rules and finally executes this SQL query. The query results are then streamed to the client. A detailed description of Quest, its initialization steps as well as its rewriting and unfolding mechanisms can be found in Rodríguez-Muro and Calvanese’s article *Quest, an OWL 2 QL Reasoner for Ontology-Based Data Access* [Rod12a].

Quest is not only able to operate in virtual ABox, but also in classic ABox mode.

¹⁹Java Database Connectivity, <http://www.oracle.com/technetwork/java/overview-141217.html>

In both modes, the OBDA model is used for query answering, but while in virtual ABox mode the data is accessed directly in the relational database, classic ABox mode requires the Ontology to be filled also with the ABox triples. Quest then takes care of the storage, either *in memory* or in a *non-native* triple store (cf. Section 2.3.1), as well as the necessary mappings. But as this causes the data to be duplicated, which is accompanied by synchronization and performance issues, the virtual ABox mode was chosen for this work. The avoidance of negative impacts of these issues is an important factor for the use of OBDA techniques in conjunction with user interfaces, as the data, like graph data, is volatile and may constantly and rapidly change.

2.3.4. OWL 2 QL

In Order to make query rewriting possible for the reasoner, the TBox of the ontology must adhere to the OWL 2 profile QL. This profile is specifically designed for enabling LOGSPACE query answering with respect to data complexity, especially through passing rewritten queries to relational database management systems. The expressivity is chosen, such that conceptual models (e.g., entity-relationship diagrams) can be expressed. As described in Section 2.1.4, the foundation of the QL profile is the description logic $DL-Lite_{\mathcal{R}}$. OWL 2 QL supports the following axioms, constrained so as to be compliant with the mentioned restrictions on class expressions:

- subclass axioms (`SubClassOf`)
- class expression equivalence (`EquivalentClasses`)
- class expression disjointness (`DisjointClasses`)
- inverse object properties (`InverseObjectProperties`)
- property inclusion
(`SubObjectPropertyOf` not involving property chains and `SubDataPropertyOf`)
- property equivalence
(`EquivalentObjectProperties` and `EquivalentDataProperties`)
- property domain (`ObjectPropertyDomain` and `DataPropertyDomain`)
- property range (`ObjectPropertyRange` and `DataPropertyRange`)
- disjoint properties (`DisjointObjectProperties` and `DisjointDataProperties`)
- symmetric properties (`SymmetricObjectProperty`)
- reflexive properties (`ReflexiveObjectProperty`)
- irreflexive properties (`IrreflexiveObjectProperty`)
- asymmetric properties (`AsymmetricObjectProperty`)
- assertions other than individual equality assertions and negative property assertions (`DifferentIndividuals`, `ClassAssertion`, `ObjectPropertyAssertion`, and `DataPropertyAssertion`)

The above list, as well as a complete description of features supported by OWL 2 QL can be found in the corresponding W3C Recommendation [MFH⁺09].

2.3.5. Mappings

The mapping rules allow that the rewritten SPARQL queries can be unfolded into SQL queries. Up to now, these mappings have to be created manually and there exist no tools or best practices yet for automating the process, although such tools are announced to be developed in the near future [RMC12].

Quest uses the mapping language introduced by Poggi et al. [PLC⁺08]. Therein, a mapping consists of an SQL query and a so called *ABox assertion template*. Such templates are a set of RDF triples that are written in a syntax leaned on Turtle (cf. Section 2.2.2). The columns in the result of the SQL query can be referenced via variables in the subject and object of the triples, whereby the values in each result row can be used to generate virtual ABox assertions.

With the example database introduced in Figure 2.7 and the TBox (2.1) from Figure 2.1, we could define the following mapping, that makes all humans with their respective names accessible through the ontology. Please note, how the columns `Human_ID` and `Name` are referenced with a preceding `$` in the RDF triple, and how the URI for each Human is generated by using the primary key `$Human_ID` of the table `tb_Human`.

```

1  # Target
   <"&::human/{$Human_ID}"> rdf:type :Human ;
   :name $Name^^xsd:string .
2  # Source
5  select Human_ID, Name from tb_Human;
```

Listing 2.6: OBDA Mapping

There do exist other mapping languages, like *R2O* [BÓCGp04], *Virtuoso RDF Views* [Ope10], *D2RQ* [Biz04], or *D2RML* [DSC12]. Nine of them have been thoroughly compared by Hert et al. in 2011 [HRG11]. For this work the mapping language introduced above [PLC⁺08] has been chosen, as it comes with a powerful environment, the -ontop- framework, whose reasoner is known to outperform other systems²⁰, and will support other mapping languages like R2RML or D2RQ in the near future [RMC12].

2.3.6. The -ontop- framework

The -ontop- framework²¹ is a collection of tools for the tasks of ontology-based data access. The parts of -ontop- are the above described Quest reasoner (cf. Section 2.3.3), with its most recent version QuestOWL, and a plugin called -ontopPro- for the ontology editor Protégé (cf. Section 2.2.4). This plugin provides a graphical user interface for defining an OBDA model, as well as a SPARQL interface which allows to save and execute SPARQL queries on the ontology.

²⁰according to a blog post, a paper from the University of Manchester is about to be published, which states that Quest is usually 6-10 times faster than Virtuoso and the performance is almost as good querying the RDB directly [Rod12a].

²¹<http://obda.inf.unibz.it/protege-plugin/>

QuestOWL is a recently (2012-08-22) [Rod12b] released version of the Quest reasoner, which is compatible to the OWLAPI3 ²², by which it is possible to use the OBDA model directly from Java applications, without the previous need for Protégé. This was an important aspect during the environment decision process for the application presented in this work, as it enables convenient access to the data from any Java based user interface.

2.4. User Interface Architecture

User interfaces play one of the most important roles in current software products, during design and development as well as at runtime. They are responsible for up to 50% [MR92] (70% [KG03]) of the total application development and maintenance time and costs. As all information between the software system and the human user is passed through such interfaces [IEE90], they are the most perceived part of a software system by the user. Already in the 1970s, when graphical user interfaces were not quite common, there was the need to modularize the complex user interface and application code into several components, to allow for better overview and maintainability.

2.4.1. Reenskaug's Models-Views-Controllers

Since its first introduction in 1979 [Ree79b, Ree79a] by Trygve Reenskaug, Model View Controller is one of the, if not the most applied and cited idea for the design of user interfaces. It was conceived as “*a general solution to the problem of users controlling a large and complex data set,*” and was supposed to “*bridge the gap between the human user's mental model and the digital model that exists in the computer.*” [Ree11]

Reenskaug defined Model-View-Controller when he was working in the context of the object-oriented Smalltalk-80 programming language at Xerox PARC. By applying MVC, it was possible to add new presentations to an existing Model, which could be left untouched in the process. This can be seen in the dependency diagram in Figure 2.13, as both View and Controller depend on the Model, but not the other way around. He defined the three entities as follows [Ree79a]:

Models represent the knowledge contained in a system, i.e., it is responsible for the data.

Views are visual representations of the model. Views is attached to a model and can update it.

Controllers are the link between the system and the user. Thus, it translates between user output²³ and model data, as well as model data and user input.

²²<http://owlapi.sourceforge.net/>

²³Reenskaug uses user output for system input, and user input as system output

Additionally, he defines an optional fourth entity, the *Editor*, which works as interface between View and input devices, such as mouse and keyboard. In later implementation, he also uses the term *Tool* for a composition of *Controller* and *Views*, as depicted in Figure 2.12.

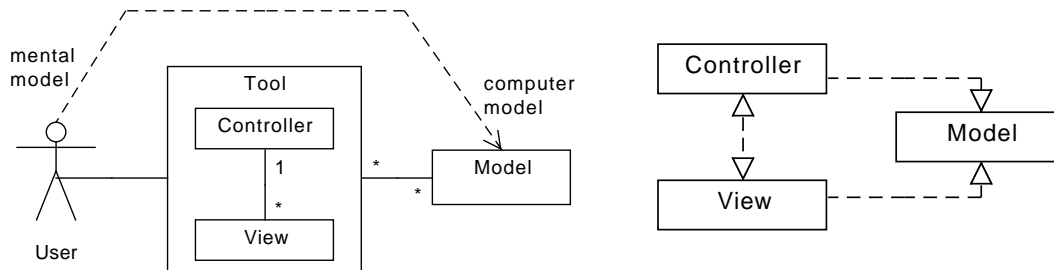


Figure 2.13.: MVC Dependencies

Figure 2.12.: MVC model, adapted from [Ree11]

2.4.2. Separated Presentation

Model View Controller can often be read, when investigating modern software systems. However, in today's applications it is inaccurate to refer to *the* MVC, as not only a large number of different implementations exist²⁴, but also a variety of interpretations. Model View Controller is referred to as a “triad of classes” [GHJV94], as “Metaphor” [KP88], “GUI Architecture” [Fow06], “Design Pattern” [Buc09], “Paradigm” [BHS07], “Philosophy” [Sha96], “Principle” [Fow03] or “Architecture Pattern” [McG04]. As example for implementation hints, the renowned Gang of Four book *Design Patterns* [GHJV94] splits MVC up into the collaboration of the Observer, Strategy and Composite pattern. Over the years, MVC has apparently become the silver bullet of user interface development for many authors, although, its meaning is often ill-defined and the interpretations of the three terms differ from Reenskaug's definition.

Fowler found out, that the idea of the separated presentation was “the most influential to later frameworks”, which, on the other hand, take other ideas from Reenskaug's MVC, freely interpret them, and re-describe them as MVC. Thereby, the understanding of MVC as a whole, as well as the understanding of the distinct entities and their responsibilities changed, and is not commonly agreed upon. Fowler explains this by the fact, that the original MVC does not “make sense for rich client systems these days” [Fow06].

There do exist a number of user interface design ideas beneath MVC. Examples include Presentation/Abstraction/Control [Cou87], which is quite similar to MVC [LR01], DocumentView²⁵, which sort of combines Model and Controller into a *Document*, Model-

²⁴see, e.g., the wikipedia comparison of web application frameworks http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks

²⁵[http://msdn.microsoft.com/en-us/library/4x1xy43a\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/4x1xy43a(v=vs.80).aspx)

View-Presenter [Pot96], a revised version of MVC for Java and C++, or Martin Fowler's separation of MVP into the two patterns Passive View²⁶ and Supervising Controller²⁷. They all address different shortcomings and provide specific improvements, like improved testability or the like.

The original MVC has several drawbacks, especially concerning the degree of separation, and all of the mentioned MVC-derivatives share these problems. But we especially criticize the misuse of MVC as an application architecture, which, induced by the above mentioned dependencies, inevitably leads to tightly coupled software components. Therefore, a new architecture will be proposed in this work, which improves or avoids these shortcomings, as it allows the user interface and the application logic to be completely separated from each other.

2.4.3. Limitations of Current Approaches

Distributed Systems

The original Model View Controller does not provide means for using it for distributed systems, like for common client-server architectures or even more complex systems. Nevertheless, there do exist a lot of frameworks for this purpose that claim to be "MVC". In these cases the three entities of MVC are redefined for nearly each framework that exists and the intersections in the understanding of the terms are often superficial: The Model somehow contains the data, and sometimes the entire application logic, which is located mostly on the server side. In contrast, the Views present the data to the user, sometimes as parts of the user interface like in Reenskaug's definition, sometimes as synonym for the entire user interface, and are located on the client side. In modern web application frameworks like, e.g., Ruby on Rails, the Controller is used as the "glue between model and view"²⁸, which is clearly not what Reenskaug intended. It is often imposed with the new obligations of this field, i.e., it also contains the communication logic and thus needs to be split up between client and server.

The Model and Controller code can be split in nearly any number of ways between client and server and it is up to the developers to decide, how to partition the application [LR01]. Fowler therefore writes that MVC is also one of the most misquoted patterns around [Fow06].

Also notice, how Reenskaug's plural form (Models, Views, Controllers) changed to the singular (Model, View, Controller). This falsely suggests that there exist just one instance of each of the three parts per application, which might be a mix-up with the widely accepted three layer model of software, introduced by Fowler [Fow03, pp. 97]. He therein distinguishes between the *data source layer*, the *business logic layer*²⁹, and the presentation layer. The data source layer in this case explicitly refers to access to some means of persistent storage, like databases. The Model in MVC can refer to either

²⁶<http://martinfowler.com/eaDev/PassiveScreen.html>

²⁷<http://martinfowler.com/eaDev/SupervisingPresenter.html>

²⁸guides.rubyonrails.org/getting_started.html

²⁹originally called *domain layer* by Fowler

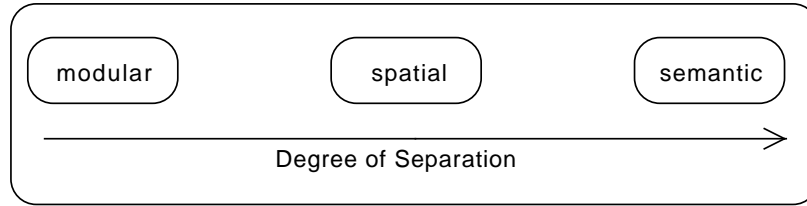


Figure 2.14.: Degrees of Separation

the data source layer, or volatile storage, as objects in the business logic layer. Not all applications do have a separate data source layer.

All this ambiguity of MVC causes confusion among developers and system designers, as it is never clear, what exactly is meant by the terms. Additionally, the controller is thereby imposed with way more than one task and thus violates the separation of concerns principle, which was the initial idea of MVC.

By the use of a new semantic intermediate layer between the user interface and the business logic, we will provide a clear point of separation for distributed systems, without another redefinition of the terms Model, View and Controller.

Degree of Separation

Although Reenskaug’s MVC splits an application in distinct entities, these are not independent of each other. There still exists a strong coupling, especially between the Views and the Model. This is due to the fact, that in order to pass data between Model and Views, the Views have to know much about the implementation of the model. In an article about user interfaces for object-oriented systems, Allen Holub argues that MVC therefore can not be considered as an object-oriented approach [Hol99]. Even Reenskaug itself states in his original MVC report that “*all these questions and messages have to be in the terminology of the model, the view will therefore have to know the semantics of the attributes of the model it represents.*” [Ree79a]

To further investigate the problem, we will adhere to a notion, introduced in 2010 by Tilly et al. They distinguish between *modular*, *spatial* and *semantic* separation as three degrees of separation (see Figure 2.14).

Modular Separation describes separation at source code level, such that different responsibilities can be found in different locations.

Spatial Separation is the separation between the actual deployment of the software, e.g., on physically different machines.

Semantic Separation introduces an intermediate semantic layer which allows full semantic decoupling between the user interface and the execution layers.

While, e.g. the original MVC provides good modular separation, as the source code of the distinct parts is clearly separated, today's reinterpretations of MVC also try to achieve spatial separation. But Tilli et al. argue that separation only on modular and spatial level cause an application to be designed, implemented and operated as a single monolithic unit. This coupling makes it hard to develop user interface and application logic in parallel by different developers, as both implementations strongly depend on each other, usually the UI more on the application logic than the other way around. A full agreement about the exchanged data, its labels, like variable names and the provided access functions must be derived from a higher level domain model. Therefore it is difficult to dedicate tasks to user interface experts that don't have knowledge about the application logic.

Nowadays, such parallel development is handled by design documents that are unreadable for machines. The developers rely on the common understanding of these design documents, supported by best practices like the RFC2119 [Bra97], which for example defines keywords for requirement levels.

By using a formal domain model as an intermediate language, the user interface will be completely independent from the implementation of the application logic and vice versa. Both parts of the system can be developed in parallel. It will be possible to extend functionality or to partially or completely rewrite either side, without the other side noticing it, due to the semantic separation of the components.

Cardinality

To further investigate the limitations, we have to distinguish between four different types of cardinality for the relationship between user interfaces and application logics on architectural level (see Figure 2.15).

- 1-to-1:** The most common relationship is 1-to-1. Here one application and one user interface form a closed unit. Usually it is not intended to change the relationship later on, and if nevertheless, it is often accompanied by massive redesign and refactoring.
- 1-to-n:** The general approach of most MVC frameworks is to allow multiple user interfaces for one application logic; be it to allow different representations of the same data, or to develop interfaces for different platforms. These can be shipped as n different 1-to-1 units, with the same, portable application logic, and platform specific user interfaces.
- n-to-1:** When multiple application logics are connected to a single user interface, this is called integration. According to Paulheim [Pau11, p. 9], this integration can be done on all three application levels (see Section 2.4.3), i.e., on data source, business logic or presentation level.
- n-to-m:** The most general way to interconnect user interfaces and application logics is n-to-m. An arbitrary number of applications can be connected in multiple ways to an arbitrary number of user interfaces.

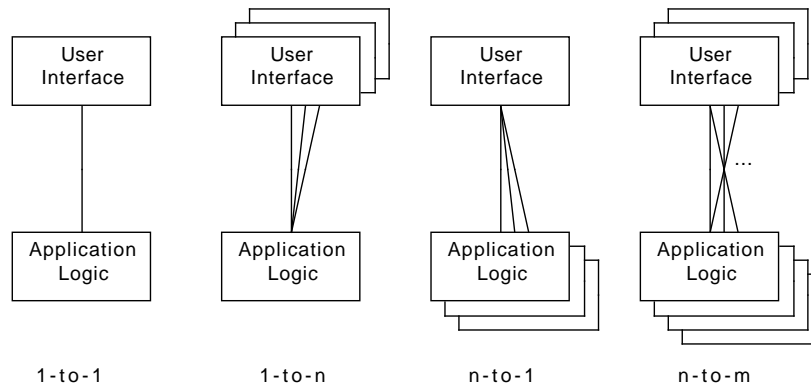


Figure 2.15.: User Interface to Application Logic Cardinality

Please note that the focus of this work is limited to single-user systems, and therefore the above mentioned n or m user interfaces are not specifically meant to be connected to the application logic simultaneously. However, as our approach is heavily based on relational databases, which provides multi-user access out of the box, we expect this extension to be feasible.

Current means of separation, like the above described MVC and its derivatives, be it in distributed or local environments, only cover *1-to-1* or *1-to-n* relationships. The OBDA approach will pave the way for connecting multiple application logics to the semantic intermediate layer, and thus allow for *n-to-1* or *n-to-m* relationships as well.

Application Logic Centered Design

Figure 2.16 shows the architecture of a common software system with data source and presentation, in analogy to [Fow03, pp. 97]. In this design the application is the central point, and the application developer has the responsibility for two interfaces: a connection to the data source and one to the presentation layer. This dependency prevents parallel and independent development of the user interface and the application to a certain extent, as the user interface relies on the data representation of the application logic.

Databases, in turn, are specifically designed for parallel access of multiple sources. Most of them, if not all, provide this feature out of the box. So we will present an approach as depicted in Figure 2.17, where the data source is the central element of the architecture, and save an interface in the application logic, as it is not directly connected to the user interface anymore.

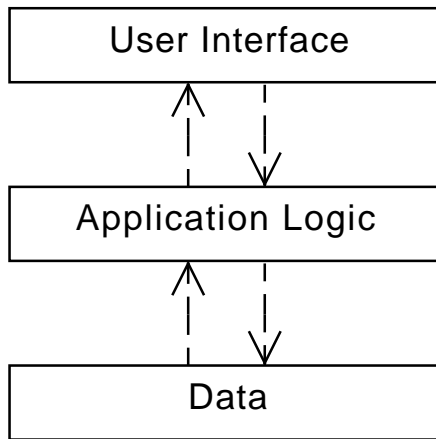


Figure 2.16.: Application Centered

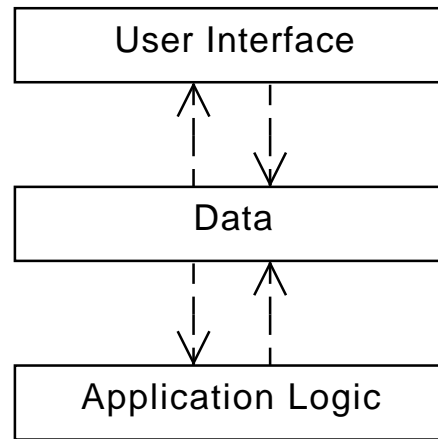


Figure 2.17.: Data Centered

Redundant Data Handling

In many application fields, especially in medical systems, it is important to maintain a complete history of the data that was generated during the runtime of an application for later reference. This persistence is either achieved by simply serializing the data to some data sink, e.g., disk storage, or by using a database. The in Section 2.4.2 presented separation techniques do not cover the possibility to persist the presented data.

In the presented approach, the two tasks of persisting and transferring data to the user interface will be combined into a single one, and as a result avoid this redundancy.

2.4.4. User Interface Data

After analyzing the data, which needs to be handled by graphical user interfaces, we distinguish between data that has to be displayed, and data that has to be modified. The displayed data comprises *quasi static*, *dynamic*, and *continuous data*. The data that has to be modified is usually only *quasi static data*.

Quasi Static Data comprises data that is not expected to change frequently and can be considered to be constant throughout a case. Examples of which include the age and weight of a patient, or the used drugs for this case. In other words, the probability of returning identical values when querying for a quasi static datum at different points in time is high.

Dynamic Data comprises data like heartbeat or blood pressure values. These are expected to change frequently and thus executing the same query at different times most likely returns different values.

Continous Data can be compared to streaming data, which is data, where existing values are continuously appended with new values. Continuous data extends the definition of streaming data, as it also allows for changing already existing data of the stream. This will be needed for prediction values of the NSRI and drug effect-site concentrations.

Transient Data comprises data like events and notifications sent between the application logic and the user interface.

2.4.5. Ontology-Enhanced User Interfaces

In Section 2.2.5 we presented classification proposals of ontologies in general. Now, we will extend the previous categorization specifically in the context of user interfaces.

In 2010, Heiko Paulheim and Florian Probst were the first to publish a big survey on the application of ontologies to improve user interfaces [PP10]. They argue that ontologies applied on the data source and business logic layer (see Section 2.4.3) are quite common, while the use on the presentation layer has not gained much attention.

They differ between ontology-enhanced and ontology-driven user interfaces. While the latter are using ontologies as a key element, the former may use them to improve, for example, just one single part in a large user interface. In their survey, they present a five dimensional categorization schema, depicted in Figure 2.18, for ontology-enhanced user interfaces and a definition of the term:

Ontology-Enhanced User Interfaces are user interfaces whose visualization capabilities, interaction possibilities, or development processes are enabled or (at least) improved by the employment of one or more ontologies.

In the following, only the definitions of Paulheim and Probst that are relevant for this work, and not already covered by the definition of Happel and Seedorf [HS06] (see Section 2.2.5) will be reflected. The interested reader is referred to [PP10, pp. 5] for complete definitions of the terms used in Figure 2.18.

Domain: Real World. The ontology characterizes a part of the real world, typically the one that the application is used in (e.g., banking, travel, etc.). The goal is to identify the central concepts and their relations.

Complexity: Medium. Ontologies of medium complexity also contain relations other than the subclass relation. Like for ontologies with low complexity, RDF-S and OWL Lite can be used.

Presentation: No presentation. The ontology is completely hidden, which means that the user of the system is not aware of the ontology in the back of the system.

Interaction: No interaction. The user cannot interact with the ontology, i.e., the ontology can't be extended, modified or altered.

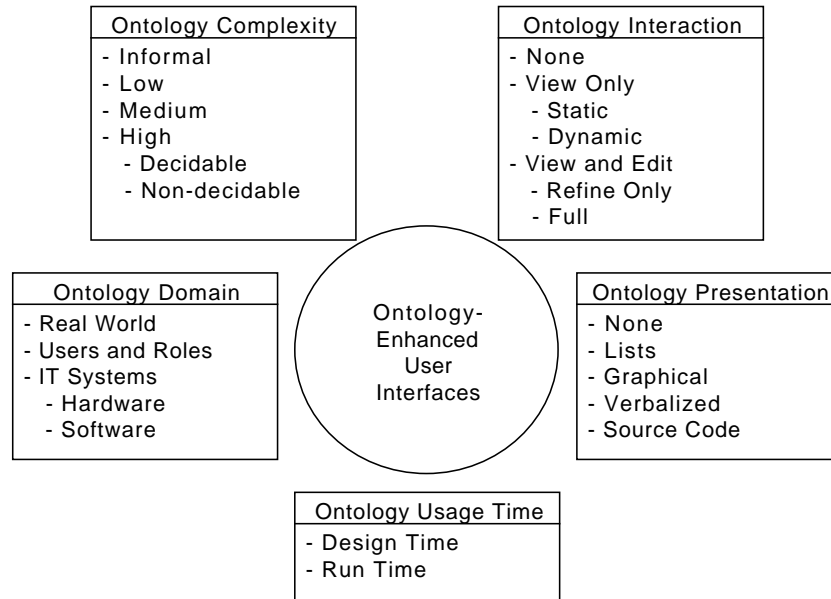


Figure 2.18.: Characterization of Ontology-Enhanced User Interfaces, from [PP10, p. 5]

Please note that the authors of this survey do not distinguish between concepts and individuals, and base their characterization scheme solely on the concepts, i.e., the TBox (see Section 2.1.2). Even if the user is able to change ABox data, he is neither able to alter, nor to see the TBox, which is why we classify our approach as “no presentation” and “no interaction”.

2.5. Related Work

As described by Paulheim and Probst in the survey [PP10] presented in Section 2.4.5, little work has been done to investigate the employment of ontologies in the area of user interfaces. Supported by a thorough literature review, specifically in the field of ontology-enhanced user interfaces and ontology-based data access, and discussions with developers of such systems, like Mariano Rodríguez-Muro from the -ontop- project at the University of Bolzano, we come to the conclusion that there have not been any attempts so far to use ontology-based data access to separate the presentation from the application logic.

Nevertheless, there have been approaches that are related to our research topic, such as ontology-based integration of applications into a single user interface [Pau11], as well as approaches on separating presentation from application logic by using ontologies [TP10].

2.5.1. Paulheim et al.

The former describes a system which uses ontologies to integrate different existing applications into a single user interface. The ontologies developed for that purpose contain formalized knowledge about user interface components as well as the data they process. In this case, the applications and their respective user interfaces remain one unit, only the user interface capabilities are semantically annotated in order to allow an integrated and uniform presentation, as well as event processing between formerly independent applications. As Paulheim et al. integrate applications solely on the user interface level, they only have to exchange events via the ontology-based part of the architecture. The data exchange between application and presentation logic takes place within the single applications and is therefore of no relevance for the authors.

2.5.2. Tilly et al.

The latter, Tilly et al., propose a system following a service oriented approach, which separates user interface components and application logic³⁰ by using static documents, which they call *Semantic User Interface (SUI) Documents*, and domain ontologies. The SUI Documents contain descriptions about how event- and data sources are associated to pieces of application logic. Their aim is corresponding with this work's vision, namely to achieve a stronger separation of the different application parts. They describe an application design scenario, in which there are three actors: domain analysts, user interface designers and component (i.e., application logic) developers. They state that

in theory they do not have to directly communicate with each others, since every relevant pieces of information is stored in domain ontologies, which they can read, thus all can do his own part without knowing about the parts of the others.

The domain ontologies in this case contain the knowledge necessary to build and use the Semantic User Interface Documents. It is not described on how the data is actually transferred between the execution layers and the user interfaces, but [TP10, p. 6, Fig. 3] shows that Tilly et al. adhere to Model View Controller and the semantic separation takes place between the Model on the one, and View and Controller on the other side.

2.5.3. Summary

The above described approaches do cover the issue of semantic separation, but rely on prototypic heavyweight frameworks. As a consequence, applications which want to employ these results usually need to be rebuilt from the ground up. It is not possible to apply these techniques on already existing software systems. As many of these could profit from a semantically separated user interface (see Section 2.4.3 Degree of Separation for advantages), we will present an approach, which can easily be implemented into existing applications.

³⁰Tilly et al. also use the terms execution layers or underlying system infrastructure.

2.6. SmartPilot View

SmartPilot View (SPV) is a software application by Dräger, which supports anesthetists in making decisions during surgery. For this purpose, it displays the current, past, and predicted depth of anesthesia by performing calculations based on widely accepted pharmacodynamic and pharmacokinetic patient models. An anesthesia comprises the dispensation of a combination of hypnotics to induce sleep, and opioids to reduce pain. By processing the data obtained automatically by connected syringe pumps or manual user input, SmartPilot View is able to compute and display the effect-site concentrations³¹ (Ce) of active ingredients of drugs, and derive, among others, the Noxious Stimulation Response Index (NSRI). The NSRI is an index, which represents the anesthetic depth on a range from 100 to 0. Before actually applying a drug to the patient, the anesthetist can provide the information to the SmartPilot View application, which in turn displays a dashed “What if...” curve. This is called a *presetting*.

The interested reader is referred to [LSV⁺10] for further information on the NSRI and to the Dräger SmartPilot View Brochure³² for further information on the software capabilities. A screenshot of SmartPilot View in simulator mode can be seen in Figure 2.19.



Figure 2.19.: SmartPilot View, Screenshot

³¹The effect-site is the immediate milieu where the drug acts upon the body, which is usually not the application site, e.g., the blood plasma for intravenous administration.

³²http://www.draeger.de/sites/assets/PublishingImages/Products/ane_smartpilot_view/Attachments/smartpilot_view_br_9066336_de.pdf

2.6.1. Problems

This application will be the basis for the prototypic implementation of the OBDA semantic separation techniques. It uses Model View Controller (see Section 2.4.1) as application architecture to a certain extend, and thereby suffers most of the limitations described in Section 2.4.3 Limitations of Current Approaches.

Degree of Separation, Distributed Systems

The SmartPilot View application was initially designed to operate on a single hardware configuration of a medical device. While the source code of the application and the user interface is separated (cf. modular separation, Section 2.4.3 Degree of Separation), no further separation can be achieved with the current architecture.

As requirements change, it is now desired to be able to access the user interface of SPV from a physically different machine, whereby the application would become a distributed system, and the MVC architecture reaches its limits.

Cardinality

Being a plain 1-to-1 relationship between SPV's application and the user interface, it is on the one hand desired to be able to connect multiple different presentations to the SmartPilot View application logic. On the other hand, SPV already displays information of other medical devices and applications, like the heart rate of the patient. With the current architecture, this integration will always be the task of the application logic, as it is not possible to connect multiple applications to a single user interface.

Application Logic Centered Design

Data like the heart rate or the blood pressure are of no importance to the SmartPilot Views calculations. The data is only passed through the application logic to be displayed on the user interface. The problem is that the application is the central point of the architecture, and lies between the data and the presentation (see Figures 2.16, 2.17).

Redundant Data Handling

The SPV application also stores the computed data to a log file on the disk. At the same time, this data is sent to the user interface to be displayed. This redundancy of data output can be avoided by using a data centered design.

3. Concept

3.1. Overview

The intention behind ontology-based data access (OBDA) (see Section 2.3) is to enhance existing relational data with a TBox (see Section 2.1.2). Thereby the relational data is accessible through an inference system and the advantages of semantic storage, like deduction of implicit knowledge (see Sections 2.1, 2.2), can be exploited. In the presented approach, this technology will be applied to create an intermediate layer between applications and user interfaces and thereby fully separate these two parts of a software system.

There exist different approaches for semantically separating graphical user interfaces from applications. As summarized in Section 2.5, it is hard to apply these approaches to existing applications. But as many software systems already exist with tightly coupled user interfaces, and the need for a higher degree of separation might arise only later, a solution for these systems would be desirable.

By performing this separation through an OBDA system, it will be possible to extend existing software systems with this technology, without having to consider such a user interface in early design phases. This is an important point, because many applications require a higher degree of separation only later, when the system evolves.

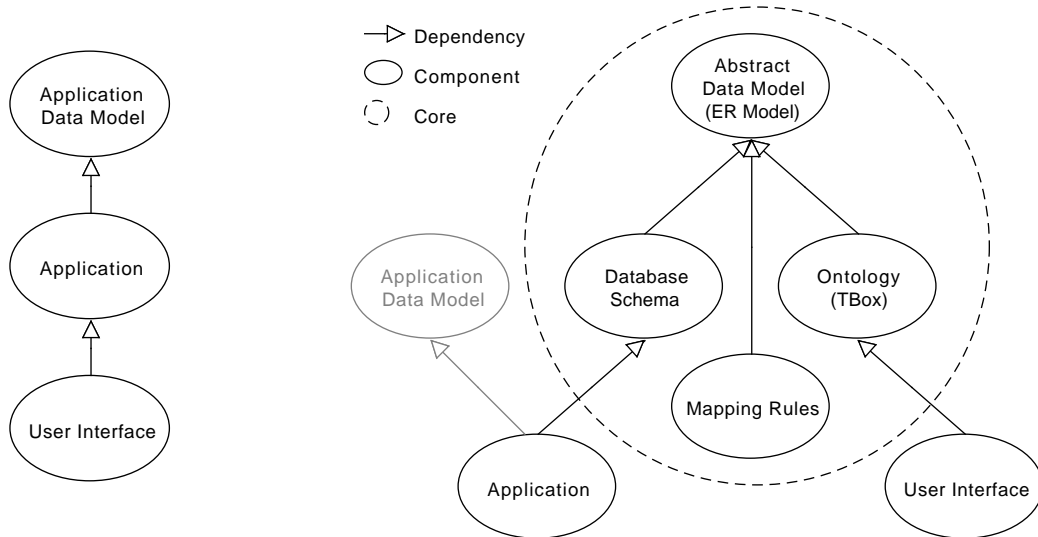


Figure 3.1.: Dependencies Before (left) and After (right)

The semantic separation will enable to clearly partition responsibilities of the different developers, which we will call domain experts. To employ our approach the needed experts are *application logic experts*, *optional database experts*, *OBDA experts*, and *user interface experts*. These developers only need to have profound know-how in their respective domain, with no, or very little expertise in the adjacent domains.

The separation will be achieved by cutting the direct dependency between the user interface and the application, as depicted in the *Before* part of Figure 3.1. The right side shows the dependencies between the different components after employing the OBDA system. Therein, no dependency between the user interface and the application can be found. Both depend on the abstract data model, which is the basis for the database schema and the ontology, as well as the mapping rules between them. As the application still depends on its own data model, an adapter has to be developed, as described in Section 3.4.1.

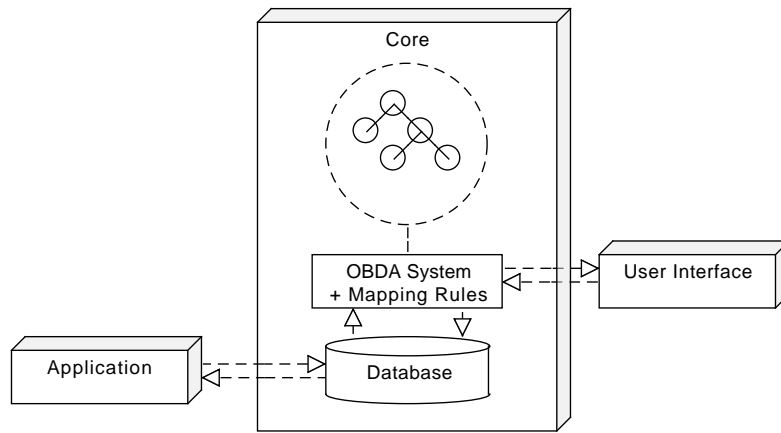


Figure 3.2.: Conceptual System Overview

Figure 3.2 shows an overview of the proposed system at a high level of abstraction. The application logic and the user interface are two entirely separated systems which do not share any code among each other. There exists a clear separation, even in terms of data access. While the application logic is only communicating with the relational database, the user interface accesses the data solely through the ontology. The OBDA system acts as an intermediate layer and connects the two parts. This will lead to a data centric design, as depicted in Figure 2.17, which avoids the shortcomings explained in Section 2.4.3.

Both Figure 3.1 and Figure 3.2 are partial views on the system. Appendix D shows the designed system as a whole, including the adaption of the SmartPilot View application, which will be described in Chapter 4 Implementation.

The presented approach is divided into seven layers, which are ordered in the following list by the data flow from the application to the user interface. The topics important for each layer are outlined.

Application The application, which is to be enhanced with a fully separated user interface.

Application-Database The connection between the application and the relational database, an *inverse object relational mapping*

Database The relational database itself, the requirements regarding functionality and performance, as well as the designed database schema

Database-Ontology The mapping rules, which are needed to provide access to the data stored in the database

Ontology The classes, including their interrelationships, their hierarchy, object- and datatype properties

Ontology-User Interface The Quest reasoner and the OWLAPI 3 interface

User Interface The actual presentation of the data

We will propose a generally applicable procedure for designing the core and connecting applications and user interfaces to it, which will then be applied to the presented SmartPilot View (see Section 2.6) software system in Chapter 4.

In the first part of this chapter, Section 3.2, we will examine the aforementioned data flow between the application and the user interface. It will be discussed what technology is necessary for the presented approach, what technology is already existing, and how it can be applied. The second part in Section 3.3 will elaborate, how the *core* is designed, which comprises the key components database (3.3.2), ontology (3.3.3) and mapping rules (3.3.4). Finally, we will show how applications (3.4.1) and user interfaces (3.4.2) can be connected to this core.

3.2. Data Flow

This section will describe, how the data is passed between applications and user interfaces. It will give an overview on how the proposed system is intended, designed and what design decisions were made for the prototype implementation, which will be further described in Chapter 4. It will be shown, how the distinct types of data, introduced in 2.4.4 User Interface Data, are handled by the system. Examples for those types of data are taken from the SmartPilot View application.

3.2.1. Application to User Interface

Quasi Static and Dynamic Data

Quasi static data, like the age or the weight of a patient, as well as dynamic data, like the patients heart rate or the applied drugs, need to be displayed by the user interface. If such data is generated or modified within the application, these changes need to be

propagated to the it. Therefore, the data is emitted by the application, and stored in a relational database. This is done by an object relational mapping, further described in Section 3.4.1. The user interface, in turn, can access this data through an OBDA system (see Section 2.3). Thereby the database as actual storage point is hidden, as the only access point is the ontology. But how does the user interface know that data is emitted by the application?

Basically, there exist two commonly known principles: polling versus pushing. Polling, in this case, means that the user interface is responsible to check on a regular basis if new data is available, while pushing means that the user interface receives either the new data itself, or a notification that it is available by the application or the OBDA layer. Although a pushing solution would be desirable for a productive system, it is not realizable yet, as further described in Section 3.2.3. Furthermore, polling is sufficient for the prototype presented in this work, as the data on the screen can be updated, for instance, once per second. If the load caused by naive polling leads to intolerable run time behaviour, the data in the ontology can be extended by metadata as the expected update frequency of values. This could be used to reduce unnecessary polls.

Continuous Data

Data like the NSRI and the computed effect site concentrations (see Section 2.6) need to be displayed as graphs over time. The latest value of each such graph is calculated by the SmartPilot View (see Section 2.6) application every second. There exist approaches, which are purpose tailored for accessing streaming data through ontologies, wherefore extensions to SPARQL (see Section 2.2.3), like C-SPARQL [BBCG10] or streamingSPARQL [BGJ08] have been proposed. To access data streams, which are stored in a relational database in OBDA environments, Calbimonte et al. developed SPARQL_{Stream} along with the mapping language S₂O [CCG10]. Calbimonte's approach is based on the ODEMapster¹ OBDA environment, and is thus not immediately compatible to the presented approach, which uses the -ontop- environment. But besides that issue, all proposed SPARQL extensions do not cover the requirements of the continuous data of SmartPilot View.

To display a data stream, instead of querying a sequence of single values it is desired to directly query for a window of data. Both C-SPARQL and streamingSPARQL provide such window operators, but fix the upper bound of this window to the query execution time [CCG10, p. 14]. In other words, it is only possible to retrieve the latest n values. This problem has been considered in SPARQL_{Stream}, which also allows to freely set the upper bound, and thus retrieve data sets from the past. This is valuable, as use cases exist, where the user of SmartPilot View is interested in such values from the past, but it is still not enough. SmartPilot View also makes a prediction of future values, but the proposed query language extensions do not provide operators to query for them.

In addition to that, there exist typical use cases that cause data values from the past to be recalculated. For example in SmartPilot View, when the anesthetist applies a drug

¹<http://neon-toolkit.org/wiki/ODEMapster>

manually, but inserts the information into the system not until some minutes later, all graph data of those minutes from the past have to be recalculated. This violates the definition of streaming data, as therein data in the past is fixed and the only allowed operation is appending new data.

We propose to exploit the capabilities of the PostgreSQL relational database, which can work efficiently with arrays². The entire data stream, including the prediction values is stored within an array. As SmartPilot View uses a relative time value, the *case time*, which is an integer counting the seconds from the beginning of a case (see Section 2.6), the array index can be used to access individual values or entire ranges. With purpose tailored classes in the ontology (see Section 4.3) and the corresponding mappings (see Section 4.4) it is possible to access this continuous data without just plain SPARQL.

3.2.2. User Interface to Application

Not only in the specific case of SmartPilot View, but also in most other software systems, a user interface emits mostly quasi static and transient data. Therefore, dynamic and continuous data do not need to be considered in this data flow direction. Transient data is described separately in Section 3.2.3, therefore the following concentrates on quasi static data.

If the user changes patient data or adds information about a recently applied drug via the user interface, this information needs to be propagated to the application. As described in Section 2.2.3, SPARQL is a pure query language, without operators for inserting, updating or deleting data in an ontology. But also in this case, there exists an extension, which is called SPARQL/Update, and a current Working Draft of the W3C [GS12].

To use SPARQL/Update together with OBDA systems, Hert et al. published ONTOACCESS [HRG10], an approach comprising the update aware mapping language *R3M* together with algorithms to translate SPARQL/Update queries into SQL queries. As this is an isolated solution, which has no interfaces to any OBDA framework or environment, Eisenberg et al. recently proposed a different solution [EK12], which extends the D2RQ platform (see Section 2.3.5) and the D2RQ mapping language with D2RQ/Update, and thereby also translate SPARQL/Update to SQL statements.

Both proposed systems are not compatible with the -ontop- framework, which is used for the developed prototype. As Eisenberg's approach is based on the D2RQ mapping language, which is announced to be supported by the -ontop- framework in the near future (see Section 2.3.5), we assume that the data flow from the user interface to the application will not be an issue in the future. According to the developers of -ontop-, plans for SPARQL/Update support exist and are scheduled by the end of 2013. Nevertheless, working systems exist, and it is already possible to use them parallel to the -ontop- framework, as depicted in Figure 3.3. For the proof of concept implementation in Chapter 4, which is highly dependent on the performance of access to continuous data, we will focus on the data flow direction from the application to the user interface.

²<http://www.postgresql.org/docs/9.2/static/arrays.html>

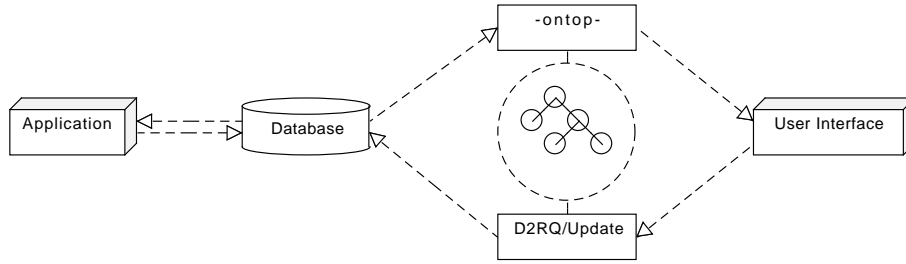


Figure 3.3.: Data Flow with Parallel OBDA Systems

3.2.3. Eventing Mechanisms

As described in Section 2.4, all information between the software system and the user is passed through the user interface. On the one hand, the user interface is a representation of the data generated and stored by the application. On the other hand it allows the user to interact with the application by adding new data or triggering actions. To enable this functionality, a mechanism is necessary, which allows the application and the user interface to notify each other. The application must notify the user interface, for example, if data has changed, to keep the displayed data up to date, and the user interface must notify the application if, for example, the user triggered a calculation. These mechanisms are often referred to as eventing or notification mechanisms and are usually realized using the publish / subscribe pattern.

The ideal solution for the presented approach would be that no system besides the used OBDA system is needed. But with existing technology these notifications are only realizable in one direction, as depicted in Figure 3.4. FirebirdSQL events³ or the Database Change Notifications of Oracle DB⁴ allow applications to be notified, when the result of an SQL query changes. This, in conjunction with SPARQL/Update (see Section 3.2.2), can provide the needed functionality for updates to be sent from the user interface to the application. Currently there exists an approach called sparqlPuSH [PM10], which is based on the PubSubHubbub⁵ protocol. sparqlPuSH can be plugged onto a SPARQL endpoint and enables to register a SPARQL query. If the result to this query changes, sparqlPuSH notifies all interested parties. As this approach relies on the data to be updated via SPARQL/Update at the same endpoint, it is not possible to use it in conjunction with an OBDA system, because the design intends the application to be unaware of the ontology. Therefore it can only update the data via the relational database and not via the SPARQL endpoint, which is why sparqlPuSH would not be able to notify the user interface.

Therefore, the only possibility to realize notifications with the currently available

³http://www.firebirdsql.org/file/documentation/papers_presentations/Power_Firebird_events.pdf

⁴http://docs.oracle.com/cd/E14072_01/java.112/e10589/dbchgnf.htm

⁵<http://code.google.com/p/pubsubhubbub/>

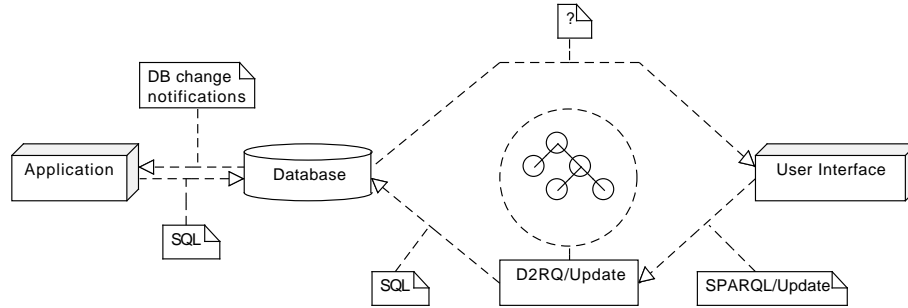


Figure 3.4.: Notifications through OBDA

technology is to use a separate system solely for that purpose. An example for such a system is the open source Distributed Publish/Subscribe Event System⁶.

As an implementation with such a workaround would not benefit the research of this work, we decided for a different solution. SmartPilot View is an existing fully functional application, therefore a user interface is already existing. This has not been replaced by the semantically separated user interface, but for testing the critical continuous data, these two UIs run in parallel. Thereby, the old interface is reused for the interaction between the user and the application, while the new one accesses the data.

3.3. Key Components

As depicted in Figure 3.2, the core of the suggested architecture comprises three key components, which are the database, the ontology and the mapping rules. The mapping rules are, besides the data source definition, part of the OBDA model, as described in Section 2.3.3 (see Figure 2.11).

3.3.1. Data Model

In the presented approach, the foundation for the core is laid by an entity relationship model. This model contains all the data that has to be passed between the applications and the user interfaces and how this data is interrelated. This model is supposed to be an abstraction of the data and should be created without considering the implementation of the application, which contains the data. As the data model has a direct influence on the ontology, which will be the only interface of the data that is visible to the consumer (the user interface), it must be free of implementation details of the application. The adapter used to overcome this gap will be described in Section 3.4.1.

In Section 3.3.2 we will show, how the database schema can be created according to the model. Then the procedure of deriving an OWL 2 QL compliant TBox from the

⁶<http://pubsub.codeplex.com/>

model will be presented in Section 3.3.3. To link the ontology to the database, a pattern for creating a complete set of mapping rules will be proposed in Section 3.3.4.

The in Figure 3.5 presented example entity relationship model consists of two sample entities, **Case** and **Patient**, which are related by a one-to-many relationship, such that a **Patient** can have zero or more **Cases**. Entity names consist of words starting with capital letters and separated by spaces.

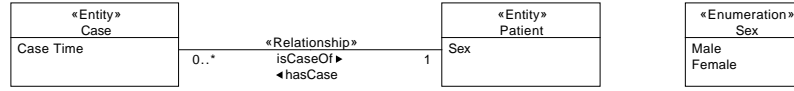


Figure 3.5.: Concept Entity Relationship Model

3.3.2. Database

The presented approach requires all data relevant to the user interface to be stored in a relational database. For one thing, this technology has been chosen for its performance and robustness (see Section 2.3.2), but also because of its high prevalence. This prevalence ensures, on the one hand that many application developers are already familiar with the topic of mapping application data to relational data stores. On the other hand, there exist a huge variety of database APIs and relational mapping frameworks in most, if not all common programming languages. Thereby it is easy for the experts of the application logic, which are already working on the software product, to implement the necessary changes.

Some applications already use a relational database to record computed values, e.g., as log or protocol for later reference. The schema of such databases is often tightly coupled to the single application. But as in the presented approach the database is part of the core, which can be seen as an abstraction layer between applications and user interfaces, it is not recommended to reuse those existing databases.

In common software systems, the data exchanged between the application and the user interface is volatile, if the application does not provide any additional means of storing it. As in the presented approach a database system is used for the exchange, this opens the opportunity to maintain a complete history of the past cases for free. To uniquely identify a case and the related objects, like graphs or used drugs, the database tables contain columns for primary and foreign keys. The key generation can be handled either by the application or the database itself. To guarantee a consistent database, we strongly suggest to use the features provided by the database system, to add an automatically incremented primary key to inserted rows.



Figure 3.6.: Concept Database Schema

Datatypes

The various components used in this work, namely the application, database, ontology and user interface do all have their own representation of datatypes. The mapping between them has to be done carefully, but is usually straightforward. An example is a decimal value, like the height of a person. The representation in the application might be a `C# Double`, in the database it is a `double precision`, the OWL 2 QL representation is a `xsd:decimal`, which is, at the end of the journey, a Java `double`. The problems that might arise with this are not part of this work, but the interested reader is referred to [MSS05].

The data may contain entries, which are limited to certain values. An example, to be seen in Figure 3.5 is the sex of the patient. In programming languages such restrictions are usually represented by enumerations, but for the database a string representation has been chosen. These strings need to be consistent between the application and the mapping rules of the OBDA model, but will be transparent to the consumer of the data, the user interface. This will be described in Section 3.3.3.

Creating the Schema

We now propose a pattern, to derive the database schema from the entity relationship model. This pattern consists of four steps.

1. Create one table per *Entity*.
prefix: `tb_`
name: lowercase plural of the entity name, underscores for whitespaces
example: *Case* → `tb_cases`
2. Create an auto generated primary key column for each table with outgoing “many” relationships. These need to be referenced later as foreign keys of other tables.
name: lowercase singular of the entity name, with underscores for whitespaces
postfix: `_id`
example: *Case* → `case_id`
3. Create a foreign key column for each outgoing “one” relationship of a table.
name: the referenced primary key **example:** *Case* → `patient_id`

4. Mark all foreign key columns of tables, which only have incoming “many” relationships, as composite primary key.

example: see `tb_used_drugs` as an example in Figure 4.2

3.3.3. Ontology

The developed TBox is the access point to the data for the user interface and makes the information of the entity relationship model usable at runtime. For the reasoning process the ontology has to comply with the OWL 2 QL (see Section 2.3.4) recommendation.

Datatypes

As described in Section 2.2.5, there exist different classifications of ontologies. With regard to Guarino’s distinction, the created TBox will be a representative of a *application ontology*. Such an application ontology by itself can only be used in a closed environment, as the contained concepts lack context. An example would be the sex of a patient. While this concept is understood within such an environment, where all using parties agreed upon constraints (like, *f* for female and *m* for male representing the sex), the concept must not necessarily be understood outside this environment. In a different environment, the same concept might be called *gender*. Therefore *domain ontologies* exist. When two application ontologies in two different environments link their classes to the classes of a common domain ontology, it can be derived that gender and sex are equivalent in the two environments. Although it is possible to link the datatype properties to a domain ontology, this is not possible for each allowed value, as for example the string ‘f’ can’t be linked to a concept “Female” of the domain ontology. Therefore, in contrast to the database representation, the in Section 3.3.2 described enumerations are represented as classes, with each allowed value being a subclass thereof.

Creating the Ontology

There do exist approaches to automatically generate OWL Lite ontologies from entity relationship models, as for example [Fah08]. But these approaches are exceeding the expressivity allowed by OWL 2 QL, e.g., with cardinality constraints. Therefore, analogous to the database schema creation, we now propose a new pattern to derive the ontology from the entity relation model. Nevertheless, this pattern is leaned on [Fah08, pp. 327], and results, for the presented example, in the ontology of Figure 3.7.

1. Create one class per *Entity*.
name: the entity name, in CamelCase, omitting whitespaces
example: *Used Drug* → `:UsedDrug`
2. Create one class per *Enumeration*.
prefix: Enum **name:** the enumeration name, in CamelCase, omitting whitespaces
3. Create one class per enumeration value
name: the enumeration value, in CamelCase, omitting whitespaces

```

1  :Sex rdf:type owl:ObjectProperty ;
    rdfs:domain :Patient ;
    rdfs:range   :EnumSex .

```

Listing 3.1: Example: Enumeration Property

4. Create subclass relationships between all enumeration value classes and their respective name classes.
example: `:Female rdfs:subClassOf :Sex .`
5. Make all enumeration value classes of the same enumeration disjoint with each other.
example: `:Female owl:disjointWith :Male .`
6. Create object properties for each entity attribute that is an enumeration
name: the enumeration class name without the prefix
example: `:Patient :Sex :EnumSex7`
7. Create datatype properties for each entity attribute, according to the required type.
name: the same as the attribute, in CamelCase, omitting whitespaces
example: `:Case :CaseTime xsd:nonNegativeInteger`
8. Create two object properties for each relation of two entities, one per direction
From the *one* to the *many* entity
prefix: `has`
name: the *many* class name
example: `:Patient :hasCase :Case .`
From the *many* to the *one* entity
prefix: `is`
name: the *many* class name
postfix: `Of`
example: `:Case :isCaseOf :Patient .`

Please note that for one-to-one relationships the naming procedure will be the same as described above, but depending on the context the direction can be chosen arbitrarily. As many-to-many relationships can always be avoided by transforming them into two one-to-many relationships with an intermediate entity, we don't consider them separately.

⁷Please note: the notation used is no valid OWL construct, but for better readability we use the following format for property examples: `:DomainClass :Property :RangeClass .` See Listing 3.1 for the valid OWL construct.

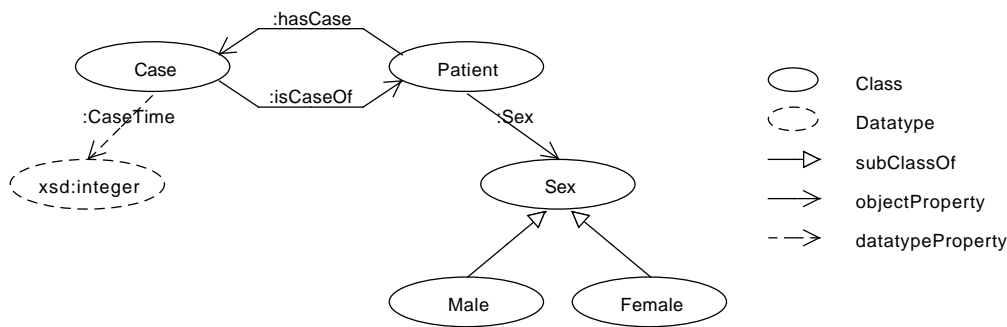


Figure 3.7.: Concept Ontology

3.3.4. Mapping Rules

To enable access via SPARQL to the data in the database, mapping rules according to the -ontop- mapping language have to be defined, which link between the ontology and the database entries, as illustrated in Section 2.3.5. These mapping rules need to be created carefully, as missing or incorrect mappings lead to unexpected empty or false results of SPARQL queries.

1. Choose a base URI and use it as default prefix.
name: arbitrarily
example: @prefix : <http://www.draeger.com/ontologies/spv#>
2. Create the URI template for each entity.
name: the base URI plus the entity name, in CamelCase, omitting whitespaces
separator: -
name: the primary key of the respective database table
example: <"&;Case-{\$case_id}">
note: concatenated by dash for composite keys
example: <"&;UsedDrug-{\$drug_id}-{\$case_id}">
3. Create one rule per entity, containing three parts: one `rdf:type` mapping the entity to the class, one `owl:datatypeProperty` for each attribute except enumerations, and one `owl:objectProperty` for each outgoing *one* relation, i.e. the `isEntityOf` relationship.
name: the entity name
example:

```

1  name:    Case
   target:  <"&;Case-{$case_id}"> a :Case;
           :CaseTime $case_time;
           :isCaseOf <"&;Patient-{$patient_id}"> .
5  source:  select case_id,patient_id,case_time from tb_case

```

4. Create one rule per entity per enumeration value.

name: the entity name, the enumeration name and the enumeration value, concatenated by blanks

example:

```
1 name:    Patient Sex Female
   target: <"&;Patient-{$patient_id}"> :Sex :Female .
   source: select patient_id from tb_patient where sex='f'
```

3.4. Connections to the Core

Once the core has been set up, it is possible to connect applications as data producers and user interfaces as consumers to it. Applications connect to the database, and store all UI relevant data. This process will be described in Section 3.4.1, while the procedure to connect user interfaces to the core will be shown in Section 3.4.2.

3.4.1. Connecting the Application

To connect an object oriented application to the core, the data of the application has to be written to the database. In object-oriented environments the process of keeping the instances of an application synchronized with the data of a database is called *object relational mapping* (ORM). There exists a variety of mechanisms to achieve an ORM, ranging from architecture patterns, as described by Fowler [Fow03], to entire frameworks, which handle the desired synchronization, like ADO.NET⁸ for C#, or Hibernate for Java⁹. The intention of such frameworks is, to hide the database access from the application developer, who just uses classes, which are extended by methods to synchronize the data they contain automatically with the respective database entries.

Inverse Object Relational Mapping

In the way, ORM frameworks are generally used, the starting point is the application and its classes, which then have a strong influence on the database structure and therefore the entity relationship diagram. In the presented use case this would violate the constraints that we want to impose on the system. As described in Section 3.3.1, every component depends, directly or indirectly, on the created data model, namely the entity relationship diagram. By applying ORM in the traditional way, this dependency would be reversed, such that the data model depends on the database schema, which in turn depends on the application implementation. This would lead to two major drawbacks. First, thereby the ontology and thus the user interface would depend on the application, which would contradict the objective of this work, to fully separate the user interface from the application. And further, as the database is based on one application, it would hinder

⁸[http://msdn.microsoft.com/en-us/library/h43ks021\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/h43ks021(v=vs.100).aspx)

⁹<http://www.hibernate.org/>

```

1 public class tb_casesObject : DbObjectBase
{
    protected DbObjectValue<System.Int32> p_case_id =
        new DbObjectValue<System.Int32>();
5    protected DbObjectValue<System.Int32?> p_case_time =
        new DbObjectValue<System.Int32?>();
    protected DbObjectValue<System.Int32?> p_patient_id =
        new DbObjectValue<System.Int32?>();
    // ...
10 }

```

Listing 3.2: Class Representation of Case in CSharp

other applications in a similar context but with different implementations to efficiently use this data structure (see Section 2.6.1 Cardinality).

Therefore we present a basic idea, which we call *inverse object relational mapping*. Therein, the existing database schema is used to generate classes. How exactly these classes look like highly depends on the programming language, the developers flavor or the used tools. Basically, each table maps to a class, with each column being represented as a class attribute, as depicted in Listing 3.2. This set of classes is depicted in Figure 3.8 as *iORM Classes*.

Adapter

Now we have a library or package, which can be used to tap and transform the actual data from the application, which depends solely on the abstract data model of the core. The data, which is contained in the application needs to be tapped and transformed from the application data model to the abstract data model. Therefore, an *adapter*, as depicted in Figure 3.8 has to be developed. As this part is very application specific, further details can be seen in the Chapter 4 Implementation in Section 4.5.

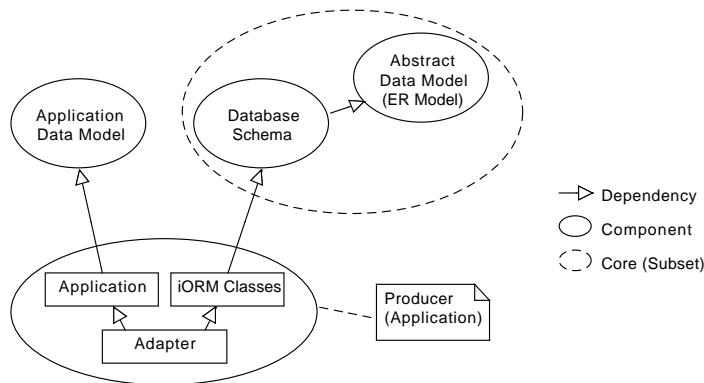


Figure 3.8.: Connecting the Application: Components and Dependencies

3.4.2. Connecting the User Interface

The last part necessary for a running system is the user interface itself. With the presented Java interface to the Quest reasoner via OWLAPI 3 (see Section 2.3.6) it is possible to develop a user interface directly in the Java programming language. But it would also be possible to create a web service in Java, which provides a SPARQL endpoint according to the SPARQL Protocol for RDF recommendation [CTF08]. In this way it is not only possible to create user interfaces in other programming languages, but to provide access to the application data over HTTP, and thereby over network.

The focus of this work, nevertheless, lies on the core, described in Section 3.3. Therefore, in Section 4.6, we will only briefly show that it is possible to retrieve the data. That this is also performant will be proved in Chapter 5.

4. Implementation

This chapter will show the application of the patterns proposed in Chapter 3 Concept. The structure will be according to the structure of the concept. This means, we will first show the implementation of the core, comprising the data model in Section 4.1, the database in Section 4.2, the ontology in Section 4.3, and the mapping rules in Section 4.4. Thereafter, we will show how applications (Section 4.5) and user interfaces (Section 4.6) can be connected to the core.

Each part contains an overview over the result of the applied patterns, e.g., the database schema or the ontology, as well as some details specific to the SmartPilot View application, e.g., retaining unit/value pairs or how to efficiently represent frequently changing graph data.

4.1. Data Model

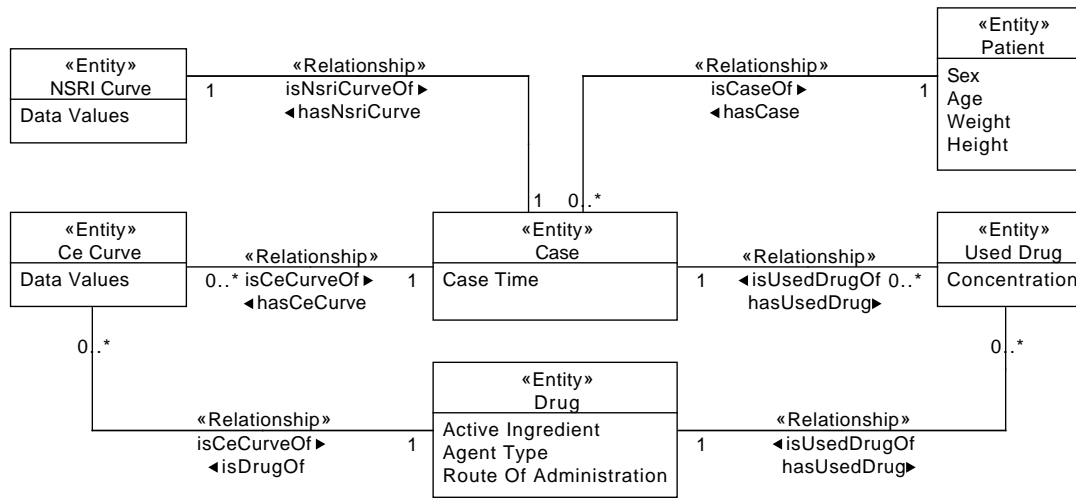


Figure 4.1.: SmartPilot View - Data Model

The central entity in the data model of SmartPilot View is the *Case*. It represents one single anesthesia. The *Patient* is the person, which undergoes it. The patient can be anesthetized in multiple cases. This *Patient's* depth of anesthesia in this *Case* is displayed by an *NSRI Curve*.

A *Drug* is characterized mainly by its active ingredient, but also by its agent type, namely hypnotic, opioid or relaxant, and the route of administration, e.g., intravenous or inhaled. There may be different compounds for the same active ingredient, differing by concentration (e.g., 10 mg of Propofol per mL Diprivan®). Therefore, specializations of a drug, the *Used Drugs*, are used during a *Case*. Nevertheless, the graphs for the effect site concentrations *Ce Curve* are computed combined per *Drug*, i.e., if two *Used Drugs* are of the same *Drug*, they are displayed in one *Ce Curve*.

As it can be seen on the screenshot in Figure 2.19, there exists some other data like the two-dimensional graph on the upper left side, the event timeline at the bottom of the figure, or additional rate and presetting graphs (see Section 2.6) included in the *Ce Curves*. This additional data can be ignored for the prototype implementation, as the chosen subset of the data is sufficient. That is, because all three displayable data types distinguished in Section 2.4.4 are covered by it. The patient data is an example for *quasi static data*, the case time for *dynamic data*, and the NSRI and effect site concentration curves are *continuous data*. Adding the application rate of a drug, or the presetting values of any curve is trivial, as it can be derived from the implemented NSRI curve. As described in Section 3.2.3, no *transient data* needs to be handled by the prototype.

Table C.1 in Appendix C gives an overview and short descriptions of this important fragment of the data SmartPilot View (see Section 2.6) is using and generating for the user interface. From the textual description above and in Table C.1, the data model in Figure 4.1 has been created. As described in Section 3.3.1, this will be the basis for the following implementations.

4.2. Database

According to the pattern described in Section 3.3.2, the database schema has been derived from the data model. But apart from applying the pattern, several data specific design decision have been made. The resulting database schema can be found in Figure 4.2.

4.2.1. Autogenerated Primary Keys

In common software systems, the data exchanged between the application and the user interface is volatile, if the application does not provide any additional means of storing it. As in the presented approach a database system is used for the exchange, this opens the opportunity to maintain a complete history of the past cases for free. When a new case is started, instead of losing all data of the past

4.2.2. Representation of Graph Data

The database system needs to be chosen according to the specific needs of the data that is to be displayed. While *quasi static* and *dynamic data* (see Section 2.4.4) can

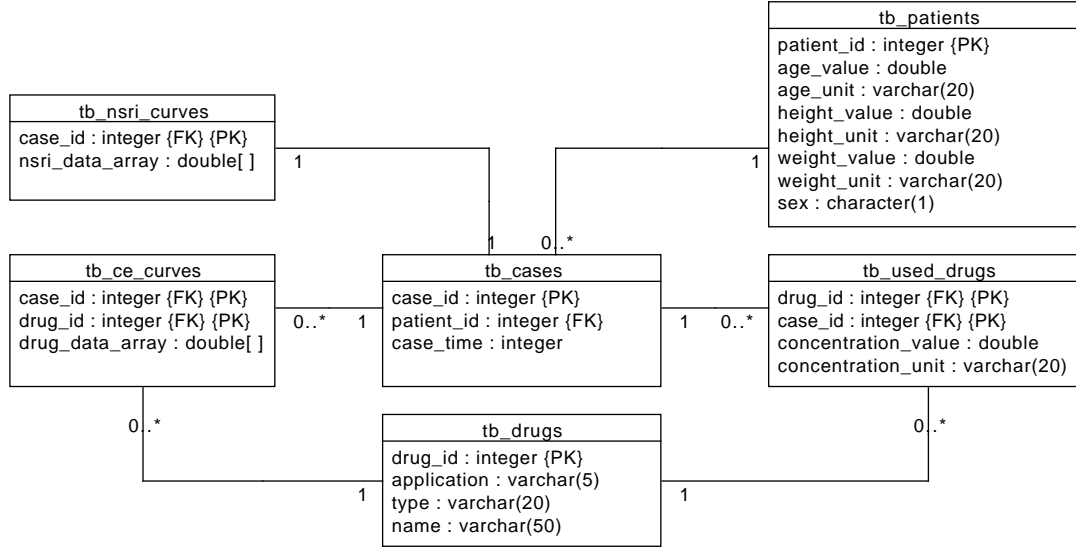


Figure 4.2.: SmartPilot View - Database Schema

be stored trivially in any database, particular attention has to be paid to *continuous* and *transient data*, in the present case this specifically means the graph data. As the amount of data is constantly increasing, and the values may change frequently, these kinds of data need efficient data structures and update mechanisms to enable the high performance needed for fluent display on user interfaces. In the early design phases it was discussed to represent each single data point as one row in the database, and to handle efficient retrieval by means of database functionality. As the user interface needs to retrieve not single data points, but a sequence of such, for displaying a graph, we tried to use clustering¹. Thereby, when such data points were written to the database, the database management system (DBMS) made sure that subsequent data was actually stored subsequently to the physical disk. In terms of the NSRI curve this meant, that a relevant sequence of NSRI data points could be read with high efficiency, as they could be read by the DBMS with only one read access to the physical disk. While this indeed decreased the access time, it imposes a new issue. Clustering in PostgreSQL, for example, is a one time operation, which means that new data stored to the table is not clustered until the clustering operation is manually executed again. As for the graph data usually the most recent data is read, this is also the data which is actually not clustered. Therefore, clustering does not solve the performance issues, which is, for example, explicitly stated in Microsoft's SQL Server documentation: "Clustered indexes are not a good choice for columns that undergo frequent changes"².

We solved this issue by using arrays³ as datatype for the graph data. These provide

¹<http://www.postgresql.org/docs/9.2/static/sql-cluster.html>

²[http://msdn.microsoft.com/en-us/library/aa933131\(v=sql.80\).aspx](http://msdn.microsoft.com/en-us/library/aa933131(v=sql.80).aspx)

³<http://www.postgresql.org/docs/9.2/static/arrays.html>

efficient read and write access of either the whole array, ranges or even single entries, while the issues of physical storage is dedicated to the DBMS entirely. In Section 5.3 we will prove that this solution provides the performance needed for fluent user interfaces.

4.2.3. Units of Measure

Type safety is a critical point of most software systems. But sometimes using compatible datatypes is not enough. Some data of the SmartPilot View application represents values that have a unit, e.g., the age, the concentration, or the Ce values. Basically, there exist two ways to handle units in a software system. Either units are omitted, and some design documents specify them outside the application, or the units are stored together with the value within the system. As the core of the proposed system provides an interface to the data, accessible by arbitrary applications and user interfaces, it was critical to provide a possibility to handle such units.

To store unit value pairs in the database, we decided to create one column for the unit, which is a string with at most 20 characters (`varchar(20)`), and a value of arbitrary data type. We constrained the unit string to be compliant to the Regenstrief Unified Code for Units of Measure *UCUM*, which is a code system for ASCII representations of “all units of measures being contemporarily used in international science, engineering, and business”⁴.

4.3. Ontology

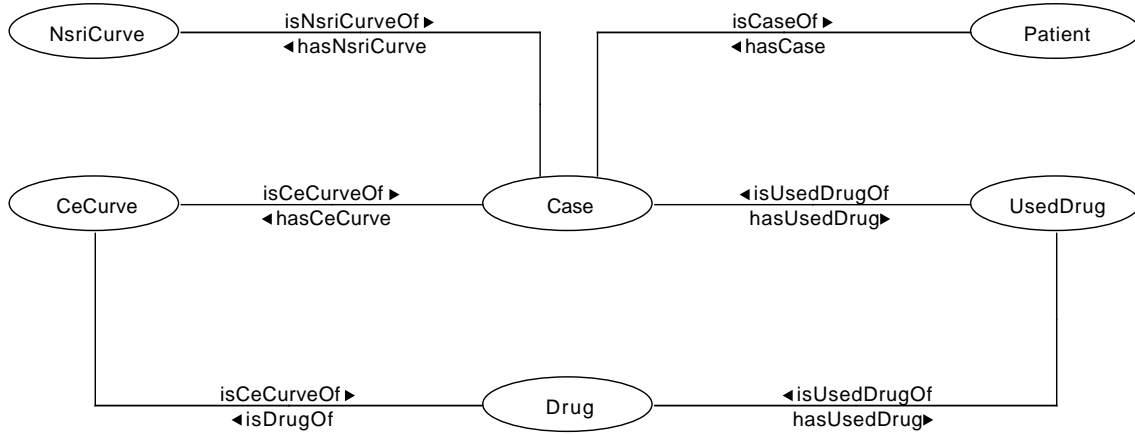


Figure 4.3.: SmartPilot View: Ontology

⁴<http://unitsofmeasure.org/>

The main classes of the ontology directly derived from the data model (see Section 4.1) according to the pattern (see Section 3.3.3) can be seen in Figure 4.3. For the sake of clarity, the complete ontology has been split into smaller pieces, according to these main classes, and can be found in the Appendix B.

4.3.1. Enumerations

There are two possibilities to represent enumerations in the ontology. The one is, to use a datatype property and restrict it to specific values, as it can be seen in Listing 4.1. Although this would have been the straightforward way, we decided to use class - subclass constructs to represent such enumerations. This provides the possibility to link the developed *application ontology* to a widely accepted *domain ontology* (see Figure 2.5 in Section 2.2.5), which enables other applications and developers to understand the meaning of the classes used in the created non-reusable application ontology.

The structure of the drug attributes, for example, was taken from the OpenGALEN⁵ medical ontology.

```

1  DataPropertyRange(
    :ActiveIngredient DataOneOf(
        "Propofol"^^xsd:string
        "Enflurane"^^xsd:string
5     "Fentanyl"^^xsd:string
        ...
        "Rocuronium"^^xsd:string
    )
)

```

Listing 4.1: Active Ingredients as Enumeration in OWL Functional Syntax

4.3.2. Representation of Graph Data

For the access to the graph data we provide two types of classes in the ontology. The one, called `PointCurve` represents the entire graph, giving access to the entire data array as `PlainLiteral`, or a subset of this data array as `PlainLiteral`, while the other, called `Point` provides access to single data points. An excerpt of the entire ontology, containing the mentioned classes, can be found in the Appendix B, Figure B.6. More information of this will be given in Section 4.4.

4.3.3. Units of Measure

To represent unit / value pairs, we enhanced the ontology by `Struct` classes, instead of adding two datatype properties. For example, instead of adding a datatype property `ConcentrationValue`, and one `ConcentrationUnit`, we found it to be more clearly

⁵<http://www.opengalen.org/>

to create such a struct class. This class would be called **StructConcentration** and contains itself the datatype properties **Unit** and **Value**, while the class that possesses this attribute is linked to the struct class via an object property, called **Concentration** in this case. Examples of this can be seen in the Appendix B in Figure B.3 or B.5. As with the graph data, Section 4.4 will give more insight on this.

4.4. Mapping Rules

4.4.1. Representation of Graph Data

Accessing the Array

Listing 4.2 shows exemplary the mapping of curve data by means of the NSRI curve. Therein, the entire data array is mapped to the `:DataArray` datatype attribute. Variable access to arbitrary ranges of the array by the user interface is not yet possible through the ontology, due to the lack of suitable SPARQL extensions, as described in Section 3.2.1. But by extending the `PointCurve` class by a datatype attribute, which hard codes a window of, e.g., 60 minutes, we could show that access to such ranges is indeed possible with the mapping rules. In the present case of SmartPilot View, we know that the application produces prediction values for 20 minutes into the future, and we know that the user interface is going to display a history of 40 minutes and a prediction of 20 minutes. Thus, as the source for the attribute `:DataArrayWindow40to20`, the array has to be queried for 60 minutes, which are 3600 values, at the end of the array (`array_upper(...)`).

Although we do induce a dependency of the ontology and the mapping rules to both, application and user interface, we found it to be a suitable intermediate solution.

```

1  # Target
   <"&::NsriPointCurve-{$case_id}"> a :NsriPointCurve ;
   :DataArray $nsri_data_array ;
   :DataArrayWindow40to20 $nda_window ;
5  :isNsriCurveOf <"&::Case-{$case_id}"> .

# Source
select
  case_id,
10 nsri_data_array,
   nsri_data_array[array_upper(nsri_data_array,1)-3600:
                        array_upper(nsri_data_array,1)] as nda_window
from tb_nsri_data

```

Listing 4.2: Mapping Curve Data

Accessing the Points

To show that the ontology representation is indeed independent of the database representation, we implement access to single data points through the ontology. Therefore, we already added the required classes in Section 4.3, and are now able to use the `unnest(...)` function of PostgreSQL's arrays⁶ to display one data row for each element of the array. As these points need to be uniquely identified, we added a column with the index of the array element, which also corresponds to the respective case time of the data point (see Section 3.2.1). This column can be automatically generated by PostgreSQL with the function `generate_subscript(...)`⁷, and in conjunction with the respective `case_id` works as a unique identifier for the data point.

Thereby, it is actually possible to retrieve a specific range of the graph by executing the respective SPARQL queries on the single points, using `FILTER(...)` to specify the range. Although being possible, our performance tests in Section 5.4 showed that the execution times for such queries are not tolerable, yet.

```

1  # Target
   <"&::NsriPoint-{$case_id}-{$point_id}"> a :NsriPoint ;
   :Value $value ;
   :Timestamp $point_id ;
5  :isPointOf <"&::NsriPointCurve-{$case_id}"> .

   # Source
   select
   case_id,
10  unnest(nsri_data_array) as value ,
   generate_subscripts(nsri_data_array,1) as point_id
   from tb_nsri_data

```

Listing 4.3: Mapping Point Data

4.4.2. Units of Measure

In addition to the mapping rules created with the pattern described in Section 3.3.4, we create one mapping for each struct class (see Section 4.3.3). Listing 4.4 shows, how the value and the unit of a patient's age is mapped to the respective tables in the database. The `StructAge` class is related to the `Patient` by an `ObjectProperty` called `Age`.

To access these values with SPARQL from the user interface, an intermediate variable is needed for the struct, which can be used in a triple to access the members, like unit or value. An example of this can be seen in Appendix E, Listing E.5.

⁶<http://www.postgresql.org/docs/9.2/static/functions-array.html>

⁷<http://www.postgresql.org/docs/9.2/static/functions-srf.html>

```

1 # Target
  <"&::Age-{$patient_id}"> a :StructAge;
    :Value $age_value;
    :Unit $age_unit .
5
# Source
select patient_id, age_value, age_unit from tb_patient

```

Listing 4.4: Mapping Unit Value Pairs

4.5. Connecting the Application

As described in Section 3.4.1, an inverse object relational mapping and an *Adapter* are necessary to fill the database with the application data, and thus connect the application to the core.

4.5.1. Inverse Object Relational Mapping

For the inverse object relational mapping we were able to apply the open source C# code generator *pgorm*⁸. It works exactly in the proposed direction, as it connects to an existing PostgreSQL database and generates a triad of classes for each database table. The naming convention is according to the following description, with *table* being replaced by the actual table name, like *tb_cases*.

tableObject An instance of this class is the representative of a database entry, containing each column as attribute. See Figure 3.2 for an example.

tableFactory This class wraps the operations that can be executed on the database for the respective table, e.g., *UpdateBy_case_id(...)*, *DeleteBy_case_id(...)*, or *GetBy_case_id(...)*.

tableRecordSet A *RecordSet* is filled by the *GetAll()* *Factory* method and then contains all *Objects* of the database table.

4.5.2. Adapter

To tap the data of the application, which is necessary for the user interface, we were especially looking for a way, which does not need much of the existing application code to be touched. Therefore we examined the architecture of SmartPilot View and found that all data which is to be displayed by the user interface is part of some events. Figure 4.4 can be divided into two parts. The upper part shows the data flow within the SmartPilot View application. This part is taken from the architecture documentation of SPV. All user interface relevant data is passed via event mechanisms from the business process to the user interface. Figure 4.5 shows a part the class diagram of SmartPilot

⁸<http://code.google.com/p/pgorm/>

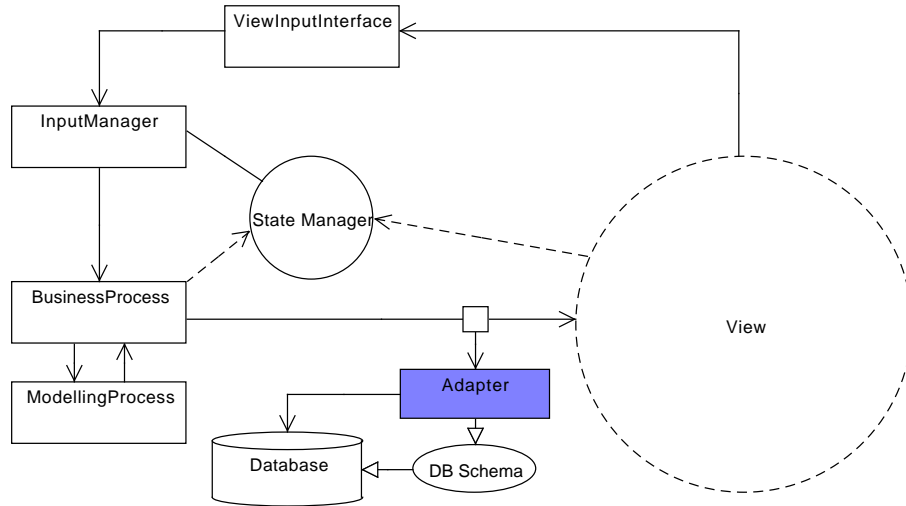


Figure 4.4.: SPV Adapter: Tapping Point

View. The Business Process from Figure 4.4 is represented as `BpState`, while the View is represented as `ViewControlState`. The states of the implemented state machine are `Operate` and `Standby`. As the interesting data is displayed while the system operates, we added the adapter as specialization of the `ViewControlOperate` state. This enables the adapter to register to the respective events to collect the necessary data, and at the same time keep the existing user interface running in parallel, as intended in Section 3.2.3.

In this way, we were able to limit the number of lines of existing source code that have to be modified to write the adapter to *one*, namely `StateMachine.cs:277`:

```
1 ViewControlState = new SemAdViewControlOperate(ViewControl);
```

There, the state `ViewControlOperate` was replaced by `SemAdViewControlOperate`. All adaptations then happen by overriding the respective event handlers, and depending on the event extract the data, like the age of a patient, pass it to a helper class (`dbFunctions` in Listing 4.6), which in turn writes the data to the aforementioned `tableObject` classes.

4.5.3. Automatically Generated Primary Key

As described in Section 3.3.2, we suggest to leave the generation of the primary keys to the database. The following listing shows that the application first creates a new instance of the case, then sets the attributes, and finally saves it to the database. The return value of the `Save(...)` method is the instance, also containing the automatically generated primary key `case_id`, which can then be used in foreign key attributes of other entities, like the NSRI curve.

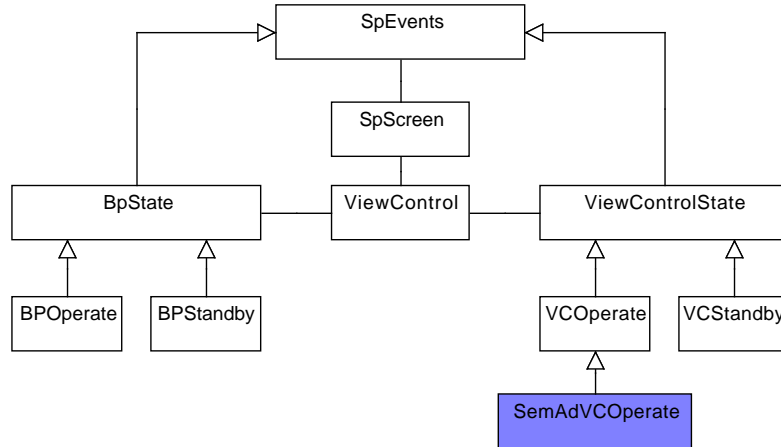


Figure 4.5.: SmartPilot View: Class Diagram (Subset)

```

1 private void initializeCase() {
    currentCase = new tb_casesObject();
    currentCase.case_time = 0;
    currentCase = tb_casesFactory.Save(currentCase);
5 }
private void initializeNsriCurve() {
    nsriCurve = new tb_nsri_curvesObject();
    nsriCurve.case_id = (int?) currentCase.case_id;
    tb_nsri_curvesFactory.Save(nsriCurve);
10 }

```

Listing 4.5: Using the Generated Primary Key

4.5.4. Representation of Graph Data

Although *pgorm* was able to handle basic datatypes, no support for arrays was implemented by the authors. The library *npqsql*, which is responsible for the connection to the database, is in turn capable of handling arrays. We were thus able to manually add support for arrays, currently limited to the element types `int` and `double`. The necessary change, shown in Listing 4.7, takes place in the table independent part of *pgorm*, the `DataAccess` class. Thereby, the automatically generated classes are able to use arrays, as shown in Listing 4.8. Please see the user manual of *npqsql*⁹ for additional information on the topic.

NSRI and effect site concentration (Ce) curves are very similar, both in the representation of the data and in their change frequency. Therefore, we only implemented the NSRI curves on the application side, as all conclusions drawn from them can directly be applied to the Ce curves as well.

⁹<http://npqsql.projects.postgresql.org/docs/manual/UserManual.html>

```

1 public override void HandleIncomingModelSetting
    (ModelSettingEventArgs args) {
    base.HandleIncomingModelSetting(args);
    if (IsValidPatientModellingData(args))
5        HandleIncomingPatientData(args);
    }
    // ...
    private void HandleIncomingPatientData
        (ModelSettingEventArgs args)
10 {
    switch (args.Id) {
        case ModelSettingId.AgeSetting:
            dbFunctions.updatePatientData(
                (UnitValue)args.Data,
15                UnitValue.Zero,
                UnitValue.Zero,
                Gender.Unknown);
                break;
                // ...
20    }
    }

```

Listing 4.6: SemAdViewControlOperate.cs: Overriding Event Handlers

4.5.5. Units of Measure

In the SmartPilot View application, the `UnitValue` class handles values, units and conversions between them. The adapter contains a simple method to convert from the unit of SmartPilot View's `UnitValue` class to the respective UCUM string, an excerpt can be seen in Listing 4.9.

4.6. Connecting the User Interface

The connection of the user interface to the core can be grouped into two parts. The first part comprises the access to the Quest inference system (see Section 2.3.3), and the second part consists of the SPARQL queries (see Section 2.2.3) to the knowledge base, i.e., the OBDA system. The main difference between those parts is that the interface to quest is bound to the Java programming language, while the SPARQL queries themselves are the language independent part.

4.6.1. Java: Accessing Quest

The access to the Quest reasoner is described in detail and with helpful examples in the -ontop- framework documentation¹⁰. As this has no impact on the presented work, it won't be presented further.

¹⁰<https://babbage.inf.unibz.it/trac/obdapublic/wiki>

```

208 public static DbParameter NewParameter(string name, object val)
    {
210     NpgsqlParameter pm;
    if (val.GetType().IsArray)
    {
        Type t = val.GetType().GetElementType();
        if (t == typeof(double)) {
215             pm = new NpgsqlParameter(name,
                NpgsqlDbType.Array | NpgsqlDbType.Double);
        } else if (t == typeof(int)) {
            pm = new NpgsqlParameter(name,
                NpgsqlDbType.Array | NpgsqlDbType.Integer);
220        } else {
            // no further types supported, yet
            return null;
        }
        pm.Value = val;
225    } else {
        pm = new NpgsqlParameter(name, val);
    }
    return pm;
}

```

Listing 4.7: DataAccess.cs: Adding Support for Arrays with pgorm

4.6.2. SPARQL: Accessing the Data

To retrieve the data from the application, we presented a system which makes it accessible via the SPARQL query language. In the following we will show some example on how the data can be retrieved by the user interface.

If a new case is started, it is added to the database, and automatically receives a unique primary key. But as the data of the preceding cases is still present in the database, the user interface needs to know what data set to access. As the primary key is automatically generated by the database system and is thus strictly increasing, the user interface must first query for the largest value, to make sure that the displayed data is the one of the currently active case. This problem is not present in common software systems, where the user interface is connected directly to the application.

```

1  PREFIX : <http://www.draeger.com/ontologies/spv#>
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    select ?curCaseNumber where
    {
5  ?case rdf:type :Case ;
        :CaseNumber ?curCaseNumber .
    } order by desc(?curCaseNumber) limit 1

```

Due to the constraint that the key is constantly increasing, the current case is always the one with the maximum key. The above listing shows one possible way to retrieve this value. Please note that aggregate functions, such as `max(...)` are not yet supported by Quest, although it is already part of the roadmap.

```

1 public class tb_nsri_curvesObject : DbObjectBase
{
  // ...
  protected DbObjectValue<System.Int32?> p_case_id =
5      new DbObjectValue<System.Int32?>();
  protected DbObjectValue<double[]> p_nsri_data_array =
      new DbObjectValue<double[]>();
  // ...
}

```

Listing 4.8: tb_nsri_curvesObject.cs: Using Arrays with pgorm

```

1 private string getUcumFromSpvUnit(UnitId unit_id)
{
  switch (unit_id)
  {
    // ...
5     case UnitId.Cm:
        return "cm";
        case UnitId.MilligramPerMilliliter:
            return "mg/mL";
10    // ...
  }
  return "UNIT_UNKNOWN";
}

```

Listing 4.9: SemAdEnhancedDbFunc.cs: Generating UCUM Strings

4.6.3. Representation of Graph Data

Assuming that the current case has the ID 135, the Query in Listing 4.10 will return the data array of the respective NSRI curve, as shown in Listing 4.11. In the Java application the PlainLiteral data array arrives as string. This string has the format "{value1,value2,value3,...}" and has to be parsed into the desired Java representation of such data structure, as for example a `List<double>`.

Special attention has to be paid on the data beyond the current case time. The SmartPilot View application does not compute a prediction value for each second, but only for every eighth. The seven values in between are not set, and thus returned as `NaN` in the array.

In Chapter 5 Evaluation, we will provide more examples of SPARQL queries and compare them by their execution time.

```
1 PREFIX : <http://www.draeger.com/ontologies/spv#>
  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  select ?case ?caseTime ?nsriCurve ?array where
  {
5 ?case :CaseNumber 135 ;
   :CaseTime ?caseTime .
  ?nsriCurve :isNsriCurveOf ?case ;
   :DataArray ?array .
  }
```

Listing 4.10: SPARQL Query for Data Array of Case 135

```
1 <http://www.draeger.com/ontologies/spv#Case-135>,
  "2373"^^xsd:integer,
  <http://www.draeger.com/ontologies/spv#NsriPointCurve-135>,
  "{100,100,...,99.999984741210895,99.999855041503906,...,100}"
```

Listing 4.11: SPARQL Query Result for Data Array of Case 135

5. Evaluation

By applying the patterns from Chapter 3, we were able to implement a system, which allows user interfaces and applications to connect to it, with those being entirely separated among each other. By accessing the data in Section 4.6 Connecting the User Interface through the implemented system, we were able to show that it is possible, but neither that the retrieved data is correct, nor that the whole system is performant enough.

In Section 2.4.4 we presented a possible distinction of user interface data. With reference to this distinction, we expect the *continuous data* to be the critical part of the application. The data arrays for the curves are extended by 3,600 data points per hour per graph. Displaying 5 graphs at the same time, which is the current limitation of the original SmartPilot View application, 18,000 new data points per hour would be generated. Regarding this immense amount of data, we assume the access of the comparably small amount of *quasi static* and *dynamic data* to be also accessible in tolerable time, if we can show that the *continuous data* is performant enough.

As one new data point is added to the graph every second, the time for the datum to be passed from the application via the adapter to the database, and then to the user interface via Quest, should be considerably smaller than $1/n$ seconds, with n being the number of displayed graphs.

5.1. Correct Result

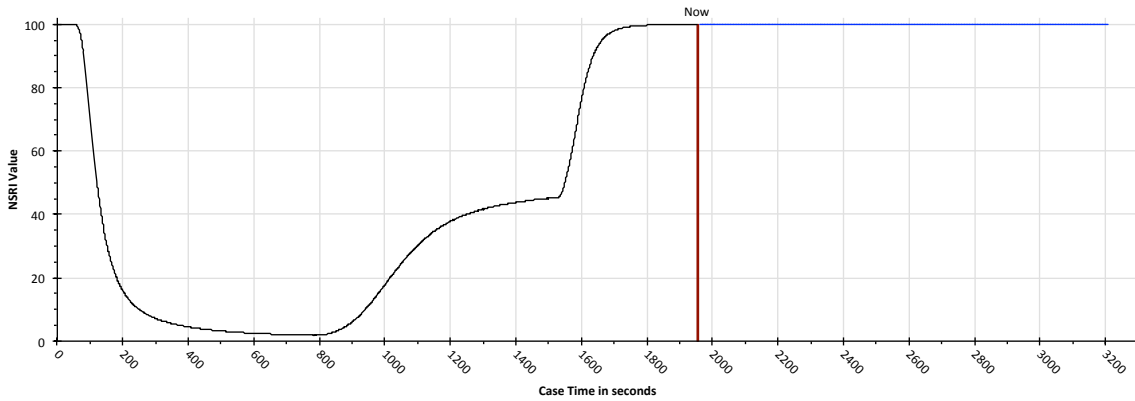


Figure 5.1.: Plot of the Retrieved NSRI Data at Case Time 1954s (00:32:34)



Figure 5.2.: Screenshot of SmartPilot View at Case Time 1954s (00:32:34)

To prove that the data transmitted through the system stays correct, we did a black box test. This means that we queried the data from the Java application (i.e., the connected user interface, see Section 4.6), and compared it to the data sent by the application. As the existing user interface of SmartPilot View is running in parallel (see Section 3.2.3), it was easily possible to compare the data. Being identified as the most critical type of data, Figure 5.1 shows the NSRI curve on the user interface side, while the screenshot in Figure 5.2 shows the expected NSRI curve. A brief comparison of the figures, as well as a thorough comparison of the data values shows, that the result is indeed correct. All other values, e.g., the age of the patient, are trivial to compare, and have been checked with the presented queries in Section 5.2.

5.2. Mapping Rules Completeness

While the mapping process is usually straightforward, no best practices or even (semi-) automated tools exist, yet. Furthermore, there exist no tools to check whether the created mappings are sound and complete. And although they are checked for syntax errors by the -ontop- plugin, as they otherwise couldn't be parsed, they can't be checked for semantic errors. The biggest problem of such semantic errors is that they are very hard to detect. When the result set of a query is empty it can be for two indistinguishable reasons: Either there exist no corresponding entries in the database, in which case the empty result is correct, or there do exist entries, but the mappings are incorrect, in

which case the empty result indicates an error.

To check, whether the created mappings fully capture the information present in the database and listed in the data model, we created a test bench in Protégé, consisting of one SPARQL query per Class. These queries were created according to the following pattern.

1. Create one SPARQL query per main *Class*.
2. Create one triple stating the `rdf:type`, add the variable to the select clause.

```
select: ?case
where{}: ?case rdf:type :Case .
```
3. Create one triple for each attribute, except structs, but including enumerations.

```
select: ?caseTime
where{}: ?case :CaseTime ?caseTime .
```
4. Create one triple for each relation, i.e., object property

```
select: ?patient ?nsriCurve ?usedDrug
where{}: ?case :isCaseOf ?patient;
        :hasNsriCurve ?nsriCurve ;
        :hasUsedDrug ?usedDrug .
```
5. Create triples with intermediate variables to access struct values.

```
select: ?age_v ?age_u
where{}: ?patient :Age ?age .
        ?age :Value ?age_v ;
        :Unit ?age_u .
```

The thereof created queries and the respective result sets can be found in Appendix E. The results of the queries show that indeed all data in the database is accessible via the provided SPARQL interface, and therefore the mapping creation pattern from Section 3.3.4 results in a complete set of mappings. Please note that the pattern does not include struct classes, which needed to be included in the query test bench to check for the correctness of the results (see Section 5.1).

5.3. Write Performance

To check, whether the implemented solution is performant enough to display continuous data, we implemented a test which measured the time to write the array data to the database from the C# application, as shown in the following listing.

```
1  DateTime timeBefore = DateTime.UtcNow;
   // update local objects and write to database
   double nsri2dbTime_ms =
       (DateTime.UtcNow - timeBefore).TotalMilliseconds;
```

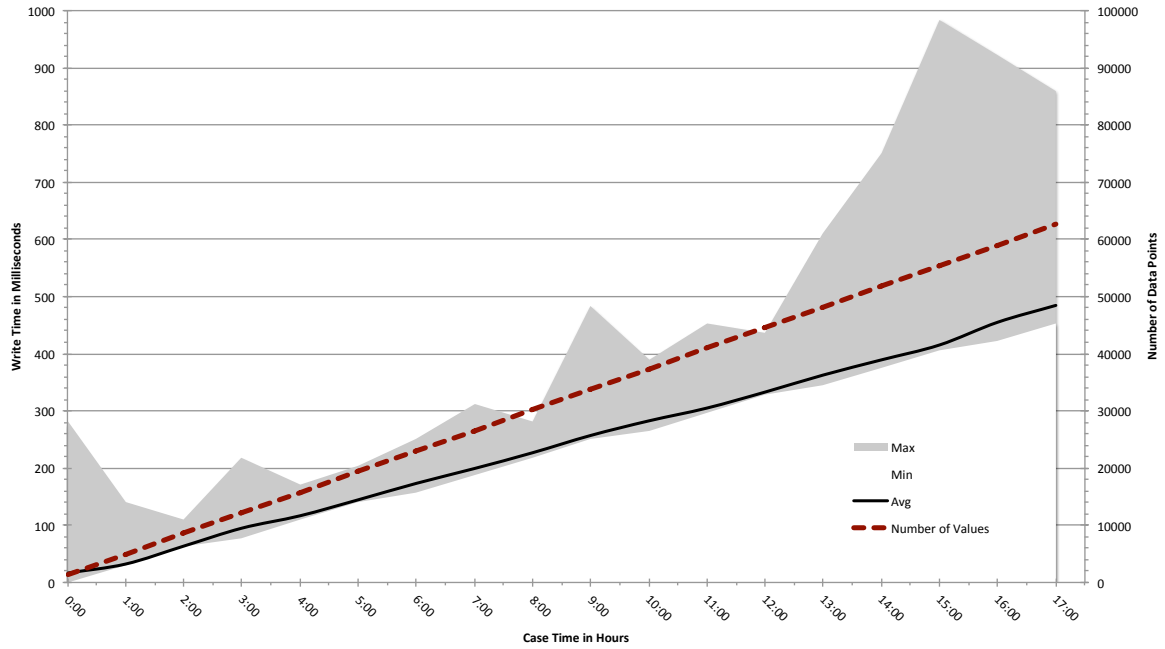


Figure 5.3.: NSRI Write Performance

Due to the current naive implementation, where the array stored as local object is always written completely to the database. Thus, we expected the time necessary for the write process to increase linearly with the size of the array. The test ran over 17 hours, starting with write times around 15 to 30ms for 2000 data points, and increasing to about 500ms for nearly 65000 data points. As PostgreSQL also offers data manipulation on specific array ranges, we expect it to be possible to reduce the write times drastically, once the classes for the inverse object relational mapping (see Section 4.5) are extended by the functionality of updating only the differing ranges in the array. Thereafter the write time will be independent of the size of the array, and we can expect constant write times below 30ms, as each second only one data point needs to be changed in the array. For a recalculation (see Section 3.2.1) a range of 1200 prediction values plus the values from the past, up to the point in time where the data is entered into the system needs to be updated. Therefore the data updates stay below 2000 data points, and for recalculations we can expect the time to be below 30ms, as well.

5.4. Read Performance

According to the test of the write performance, we measured the time of the query execution in the Java application. The time span does not include any further operations on the data, like generating a `List<double>` representation of the array. This is just the retrieval time of the plain string.

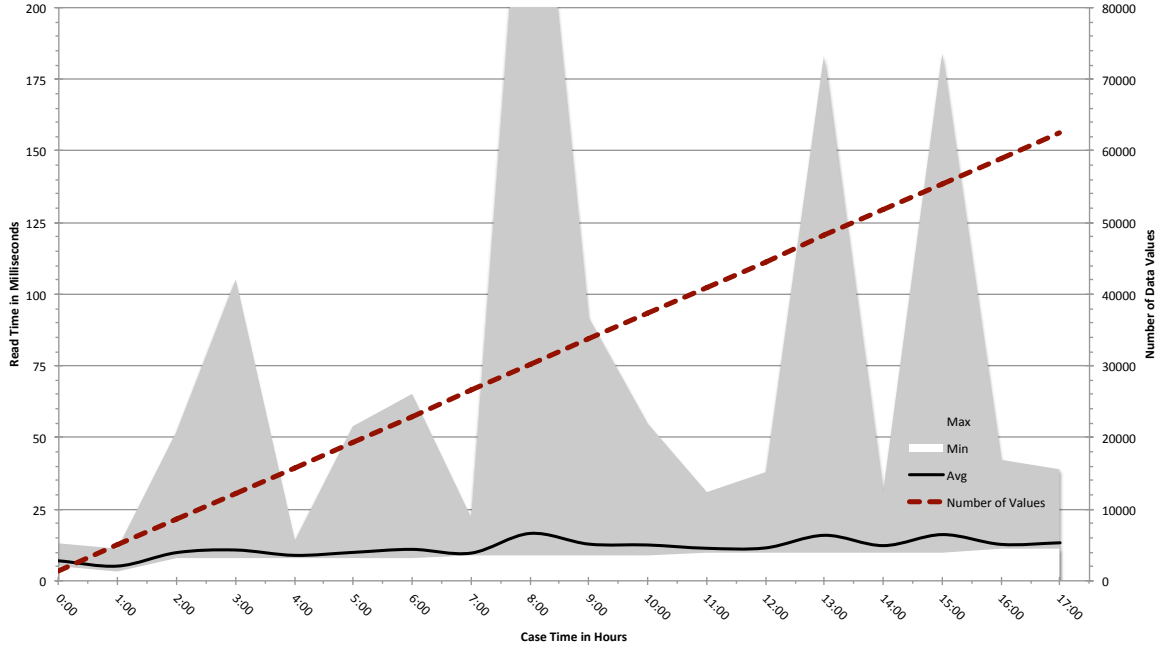


Figure 5.4.: NSRI Read Performance

```

1  long timeBefore = System.nanoTime();
   // execute query
   long onto2nsriTime_ms =
       (System.nanoTime() - timeBefore) / 1000 / 1000;

```

As Figure 5.4 depicts, the time to retrieve the data values from the system is much less affected by the number of data points compared to the write performance, even though we retrieve the whole data array. The average reading times are increasing from around 10ms to only 15ms for the 65000 data points. Although there exist some outliers, with read times of around 300ms, they are very rare (16 outliers in 17 hours run time), and can thus be ignored.

The implemented prototype supports three different ways of accessing the data array. The first is, to retrieve the whole data array, as depicted in Figure 5.4; the second is described in Section 4.4.1, namely the hard coded access to the 60 minute range via the datatype property: `DataArrayWindow40to20`. The third and last is presented in the same Section, which is to access the curve data as single data points. A comparison of the access times is shown in Figure 5.5.

Access to single data points is, as expected, infeasible, as the retrieval time increases to over 8 seconds for not even 5000 data points in the array, depicted by the nearly vertical gray line in the graph. We did further investigation on this, and were able to track down the reason for this to disadvantageous query unfolding. The execution of the following query for *all* points of *all* cases takes around 450ms to complete.

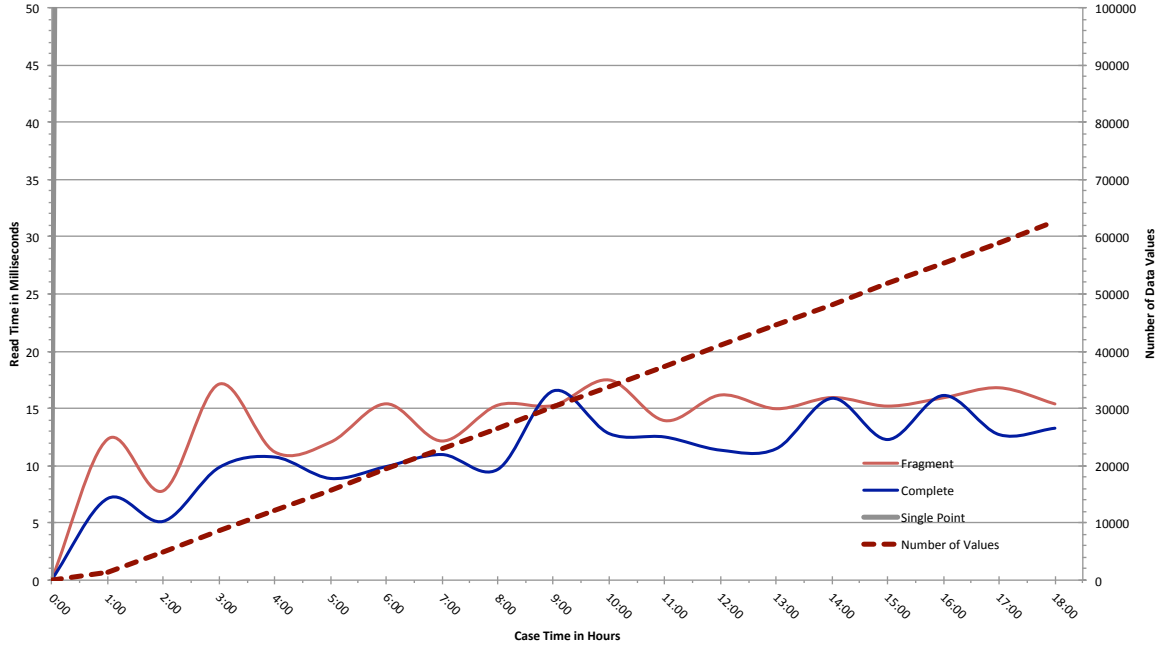


Figure 5.5.: NSRI Read Performance Comparison

```

1  select distinct ?case ?time ?val where {
    ?nsri rdf:type :NsriPoint;
        :Timestamp ?time;
        :Value ?val;
5     :isPointOf ?curve .
    ?curve :isNsriCurveOf ?case .
}

```

But as soon as we add the triple below to retrieve only the points of a specific case, the unfolded query requires the database to execute a five times nested loop, which causes the execution time to explode (6.5 seconds for Case 131, which contains 3392 data points).

```

1  ?case :CaseNumber 131 .

```

The access to the 60 minute fragment tends to be slightly slower than the access to the whole array, which is mainly caused by the additional computations necessary for the database to extract the desired range.

In the presented implementation, all systems are running on the physically same machine. If the data is to be accessed over a network, we expect the retrieval time to increase, especially for the whole array, as the returned string representation of the 65000-element array has a size of 1.17 MB, which is infeasible to be transported over network frequently. Therefore we consider fragment-wise access to the data to be most suitable in terms of read performance.

5.5. Summary

We could show that the system works correctly, as both the data provided by the application is accessible by the user interface, and the data sent by the application coincides with the data received by the user interface.

Currently the performance of the system is mostly determined by the inefficient way to write the continuous data to the database. With the measurements presented in this chapter and the suggested modifications to the data writing procedure, we can reasonably expect an average transmission time of data from the application to the user interface below 50ms. This is a tolerable value and provides enough buffer for further computations on the user interface side.

Finally, the user interface does indeed not share any code with the application logic. The user interface developer does not need to know anything about the implementation of the application. In fact, it could be replaced entirely without the user interface even noticing, as the only access point to the data is on the conceptual level through the ontology.

6. Conclusion and Future Work

6.1. Conclusion

In this work we introduced a system based on the techniques of ontology based data access, which enables user interfaces to be separated from the user interface on a high, semantic level.

We therefore analyzed Model View Controller, its original intention, as well as its misuse as architecture pattern, which, against the intention, causes the application and the user interface to be tightly coupled, and thus often be developed as a monolithic unit. In Section 2.4.3, we presented the detected limitations of such MVC approaches, namely the lack of support for distributed systems, the low degree of separation, the incapability for interconnecting an arbitrary number of applications and user interfaces, as well as the redundant processing of data by the application for both the presentation and the data source layer.

Ontologies are predestined to be used as intermediate layer between the user interface and the application, and thereby lead to an increased degree of semantic separation. As current reasoning systems are not yet able to handle the high amount of data interchanged between those two software system layers, we suggested to apply the techniques of ontology based data access. This technology is able to translate SPARQL ontology queries to SQL database queries, including the derived TBox knowledge. Thereby, all query execution is delegated to the relational database system, which is highly capable of handling very large amounts of frequently changing data.

The interchanged data has been split up into the four types of *quasi static*, *dynamic*, *continuous*, and *transient data*. As ontology based data access is a fairly recent research topic, it was necessary to perform a thorough literature research, which enabled us to acquire the required technology. Although various research prototypes are existing, and are able to handle all but one of the required data flows, they all are developed across heterogeneous OBDA environments. Thus, it is not yet possible to develop an event based user interface with only one system. We figured out that the -ontop- framework from the University of Bolzano is, in terms of performance, ongoing research and development, as well as provided functionality, meeting most of the requirements for the presented approach.

The thereby generated results made it possible to introduce a concept with according patterns to create the database schema, the ontology and the mapping rules from a formal entity relationship model of the interchanged data.

Based on an existing medical decision support system, which is implemented using the aforementioned Model View Controller system architecture, we were able to prove the

feasibility of our approach. While the existing user interface was tightly coupled with the application logic, the derived ontology now provides a semantically annotated access point to the data for the user interface and thereby drastically increases the degree of separation. Using the PostgreSQL relational database system, the performance of the system has been proved as being sufficient for the requirements of frequently changing continuous user interface data.

The developed core is also capable by design to integrate the data of multiple applications, and thereby saves the application the duty to loopthrough such data. On the other hand, multiple user interfaces can be connected to the core, which can be designed and developed by developers that need no insight on the implementation of the application itself, neither on the internal data representation, nor on the programming language.

By measuring the time the data needs to pass through the system, and showing that this is below 50ms, we could prove that the implemented solution is performant enough to display large amounts of frequently changing user interface data.

6.2. Future Work

Usually, OBDA systems are developed on top of an existing database. We introduced an approach, where an entire OBDA system, including ontology, database schema and mapping rules could be designed from scratch. The therefore introduced patterns in Section 3.3 are generally applicable for designing entire OBDA systems, not only in the context of separating user interfaces from applications. We expect that the generation process of the key components can most likely be automated. With some additional research on the soundness and completeness of the generated mappings, it can thus be possible to derive a fully functional and complete OBDA system automatically from just the specification of an entity relationship data model.

Despite the fact that ontology based data access is a comparably young field in computer science, several publications exist that extend the functionality of basic query translation to also handle data manipulation via SPARQL/Update or ontology based access to streaming data sources. But although the research is supported by various prototypic implementations, they all rely on different frameworks, mapping languages, and reasoners, which makes them incompatible with each other. Additional research needs to be done on how these ideas can be integrated into a single system.

While the Quest reasoner is accessible from the Java programming language, which is at least platform independent, access to the ontology could be desired from different programming languages as well. It is possible to develop a so called SPARQL endpoint as Java application to allow access to the data via HTTP, but additional research is needed to prove that this approach is performant enough to handle user interface data, or if there are other possibilities to access the data from various programming languages.

By using OBDA as intermediate system between the user interface and the application, it is possible to distribute the different parts of it. The three parts are the application,

the core and the user interface. While easily possible to cut off the application and access the database over network, whereby these components can easily be deployed on physically different machines, the current implementation is not yet able to split up the user interface from the reasoning system. But it is desired not to impose the client running the user interface with the duty of performing the reasoning. This computation intensive task should be executed on a therefore dedicated server. This is another reason why the previously suggested research on the feasibility of providing a SPARQL endpoint is needed, as it could solve this problem as well.

Especially in medical systems, provable correctness is a necessary requirement. Up to now there exist no debugging environments for the design, development and use of OBDA systems. We especially identified the problems of empty result sets for false or incomplete mappings in Section 3.3.4 as a problem. Thereby the user is not able to determine whether the empty result set means that there are no results, or that an error occurred. We think that this might be an issue for the productive use of such system in safety critical environments, as for example in medicine or aviation.

Additionally, although datatypes are supported by programming languages, database systems and the ontology language OWL, the internal representation of such datatypes can vary greatly between the different systems, and at the end, Quest returns basically a string to Java, leaving the transformation into the correct datatype to the Java application. As these numerous transformations into various internal representations can endanger the type safety, we suggest additional research on this topic, especially for safety critical systems.

List of Figures

2.1. Example Knowledge Base with TBox (2.1) and ABox (2.2)	12
2.2. The Semantic Web Stack [BL09]	16
2.3. The Protégé Ontology Editor	21
2.4. Classification of ontologies, according to Happel and Seedorf [HS06] . . .	22
2.5. Classification of ontologies, according to Guarino [Gua97]	23
2.6. Data Model Comparison	25
2.7. Example Database	26
2.8. Example Entity Relationship Diagram	26
2.9. Ordinary ABox	27
2.10. Virtual ABox	27
2.11. Quest System Overview, from [RMC12]	27
2.12. MVC model, adapted from [Ree11]	31
2.13. MVC Dependencies	31
2.14. Degrees of Separation	33
2.15. User Interface to Application Logic Cardinality	35
2.16. Application Centered	36
2.17. Data Centered	36
2.18. Characterization of Ontology-Enhanced User Interfaces, from [PP10, p. 5]	38
2.19. SmartPilot View, Screenshot	40
3.1. Dependencies Before (left) and After (right)	42
3.2. Conceptual System Overview	43
3.3. Data Flow with Parallel OBDA Systems	47
3.4. Notifications through OBDA	48
3.5. Concept Entity Relationship Model	49
3.6. Concept Database Schema	50
3.7. Concept Ontology	53
3.8. Connecting the Application: Components and Dependencies	55
4.1. SmartPilot View - Data Model	57
4.2. SmartPilot View - Database Schema	59
4.3. SmartPilot View: Ontology	60
4.4. SPV Adapter: Tapping Point	65
4.5. SmartPilot View: Class Diagram (Subset)	66
5.1. Plot of the Retrieved NSRI Data at Case Time 1954s (00:32:34)	71
5.2. Screenshot of SmartPilot View at Case Time 1954s (00:32:34)	72

5.3. NSRI Write Performance	74
5.4. NSRI Read Performance	75
5.5. NSRI Read Performance Comparison	76
B.1. SmartPilot View: Ontology - Key	96
B.2. SmartPilot View: Ontology - Case	96
B.3. SmartPilot View: Ontology - Patient	97
B.4. SmartPilot View: Ontology - Drug	97
B.5. SmartPilot View: Ontology - Used Drug	98
B.6. SmartPilot View: Ontology - Curves and Points	98
D.1. SPV with OBDA: Complete Overview	100

List of Tables

2.1. Description Logics: Languages and semantics	11
C.1. SmartPilot View, Relevant Data Subset	99
E.1. Cases	102
E.2. Drugs	103
E.3. NSRI Curves	104
E.4. Patient	105
E.5. Used Drug	106

Listings

2.1. RDF/XML syntax	17
2.2. Turtle syntax	17
2.3. All Men are Humans in Turtle Syntax	18
2.4. Example ontology in turtle syntax	19
2.5. Querying for all Humans	20
2.6. OBDA Mapping	29
3.1. Example: Enumeration Property	52
3.2. Class Representation of Case in CSharp	55
4.1. Active Ingredients as Enumeration in OWL Functional Syntax	61
4.2. Mapping Curve Data	62
4.3. Mapping Point Data	63
4.4. Mapping Unit Value Pairs	64
4.5. Using the Generated Primary Key	66
4.6. SemAdViewControlOperate.cs: Overriding Event Handlers	67
4.7. DataAccess.cs: Adding Support for Arrays with pgorm	68
4.8. tb_nsri_curvesObject.cs: Using Arrays with pgorm	69
4.9. SemAdEnhancedDbFunc.cs: Generating UCUM Strings	69
4.10. SPARQL Query for Data Array of Case 135	70
4.11. SPARQL Query Result for Data Array of Case 135	70

Bibliography

- [BBCG10] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. An execution environment for c-sparql queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 441–452, New York, NY, USA, 2010. ACM.
- [BBL05] Franz Baader, Sebastian Brand, and Carsten Lutz. Pushing the EL Envelope. In *In Proc. of IJCAI 2005*, pages 364–369. Morgan-Kaufmann Publishers, 2005.
- [Bec04] Dave Beckett. RDF/XML Syntax Specification (Revised). W3C recommendation, W3C, February 2004. <http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/>.
- [Bec12] David Beckett. Turtle - Terse RDF Triple Language. W3C working draft, W3C, July 2012. <http://www.w3.org/TR/2012/WD-turtle-20120710/>.
- [BGJ08] Andre Bolles, Marco Grawunder, and Jonas Jacobi. Streaming sparql extending sparql to process data streams. In *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, ESWC'08, pages 448–462, Berlin, Heidelberg, 2008. Springer-Verlag.
- [BHS07] F. Buschmann, K. Henney, and D.C. Schmidt. *Pattern Oriented Software Architecture: On Patterns and Pattern Languages*. Wiley Series in Software Design Patterns. John Wiley & Sons, 2007.
- [Biz04] Christian Bizer. D2rq - treating non-rdf databases as virtual rdf graphs. In *In Proceedings of the 3rd International Semantic Web Conference (ISWC2004)*, 2004.
- [BL07] Franz Baader and Carsten Lutz. 13 Description Logic. In Patrick Blackburn, Johan Van Benthem, and Frank Wolter, editors, *Handbook of Modal Logic*, volume 3 of *Studies in Logic and Practical Reasoning*, pages 757–819. Elsevier, 2007.
- [BL09] Tim Berner-Lee. Semantic Web and Linked Data. World Wide Web Consortium, 2009. <http://www.w3.org/2009/Talks/0120-campus-party-tbl>.
- [BLC11] Tim Berners Lee and Dan Conolly. Notation3 (N3): A readable RDF syntax. W3C teamsubmission, W3C, March 2011. <http://www.w3.org/TeamSubmission/2011/SUBM-n3-20110328/>.

- [BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. Standards track, IETF, January 2005. <http://www.ietf.org/rfc/rfc3986.txt>.
- [BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.
- [Bor96] Alex Borgida. On the Relative Expressiveness of Description Logics and Predicate Logics. *Artificial Intelligence*, 82:353–367, 1996.
- [BPM⁺08] Tim Bray, Jean Paoli, Eve Maler, François Yergeau, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, November 2008. <http://www.w3.org/TR/2008/REC-xml-20081126/>.
- [Bra97] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. Best current practice, IETF, March 1997. <http://www.ietf.org/rfc/rfc2119.txt>.
- [BS01] Franz Baader and Ulrike Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.
- [Buc09] James Bucanek. *Learn Objective-C for Java Developers*. Learn Series. Apress, 2009.
- [BÓCGp04] Jesús Barrasa, Óscar Corcho, and Asunción Gómez-pérez. R2o, an extensible and semantically based database-to-ontology mapping language. In *In Proceedings of the 2nd Workshop on Semantic Web and Databases (SWDB2004)*, pages 1069–1070. Springer, 2004.
- [Car07] Jorge Cardoso. The semantic web vision: Where are we? *IEEE Intelligent Systems*, 22(5):84–88, September 2007.
- [CCG10] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair J. G. Gray. Enabling ontology-based access to streaming data sources. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ISWC’10, pages 96–111, Berlin, Heidelberg, 2010. Springer-Verlag.
- [CDGL98] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the decidability of query containment under constraints. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS ’98, pages 149–158, New York, NY, USA, 1998. ACM.
- [CDGL⁺07] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The dl-lite family. *Journal of Automated Reasoning*, 39:385–429, 2007.

- [Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.
- [Cou87] Joelle Coutaz. PAC: An object oriented model for implementing user interfaces. *SIGCHI Bull.*, 19(2):37–41, October 1987.
- [CTF08] Kendall Grant Clark, Elias Torres, and Lee Feigenbaum. SPARQL protocol for RDF. W3C recommendation, W3C, January 2008. <http://www.w3.org/TR/2008/REC-rdf-sparql-protocol-20080115/>.
- [DEFS98] Stefan Decker, Michael Erdmann, Dieter Fensel, and Rudi Studer. Ontobroker: Ontology based Access to Distributed and Semi-Structured Information. In *Database Semantics: Semantic Issues in Multimedia Systems*, pages 351–369. Kluwer Academic Publisher, 1998.
- [DSC12] Souripriya Das, Seema Sundara, and Richard Cyganiak. R2RML: RDB to RDF Mapping Language. W3C proposed recommendation, W3C, August 2012. <http://www.w3.org/TR/2012/PR-r2rml-20120814/>.
- [EK12] Vadim Eisenberg and Yaron Kanza. D2rq/update: updating relational data via virtual rdf. In *Proceedings of the 21st international conference companion on World Wide Web, WWW '12 Companion*, pages 497–498, New York, NY, USA, 2012. ACM.
- [Fah08] Muhammad Fahad. Er2owl: Generating owl ontology from er diagram. In *Intelligent Information Processing IV*, volume 288 of *IFIP Advances in Information and Communication Technology*, pages 28–37. Springer Boston, 2008.
- [Fin01] Tim Finin. Re: NAME: SWOL versus WOL. Web Ontology Working Group Public Mailing List, 2001. <http://lists.w3.org/Archives/Public/www-webont-wg/2001Dec/0169.html>.
- [Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- [Fow06] Martin Fowler. *Patterns of Enterprise Application Architecture - Development Version*. <http://martinfowler.com/eaDev/uiArchs.html>, 2006.
- [GB04] Ramanathan V. Guha and Dan Brickley. RDF vocabulary description language 1.0: RDF schema. W3C recommendation, W3C, February 2004. <http://www.w3.org/TR/2004/REC-rdf-schema-20040210/>.
- [GG95] N. Guarino and P. Giaretta. Ontologies and Knowledge Bases: Towards a Terminological Clarification. In N. J. I. Mars, editor, *Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing*, pages 25–32. IOS Press, Amsterdam, 1995.

- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1994.
- [GHVD03] Benjamin N. Grosz, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: combining logic programs with description logic. In *Proceedings of the 12th international conference on World Wide Web, WWW '03*, pages 48–57, New York, NY, USA, 2003. ACM.
- [GMF⁺03] John H. Gennari, Mark A. Musen, Ray W. Fergerson, William E. Grosso, Monica Crubézy, Henrik Eriksson, Natalya F. Noy, and Samson W. Tu. The evolution of Protégé: an environment for knowledge-based systems development. *Int. J. Hum.-Comput. Stud.*, 58(1):89–123, January 2003.
- [GN87] M.R. Genesereth and N.J. Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann, 1987.
- [Gru93] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. In *International Journal of Human-Computer Studies*, pages 907–928. Kluwer Academic Publishers, 1993.
- [GS12] Paul Gearon and Simon Schenk. SPARQL 1.1 update. W3C working draft, W3C, October 2012. <http://www.w3.org/TR/2012/WD-sparql11-update-20120105/>.
- [Gua97] Nicola Guarino. Semantic Matching: Formal Ontological Distinctions for Information Organization, Extraction, and Integration. In *Information Technology, International Summer School, SCIE-97*, pages 139–170. Springer Verlag, 1997.
- [HDG⁺06] Matthew Horridge, Nick Drummond, John Goodwin, Alan L. Rector, Robert Stevens, and Hai Wang. The manchester owl syntax. In Bernardo Cuenca Grau, Pascal Hitzler, Conor Shankey, and Evan Wallace, editors, *OWLED*, volume 216 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.
- [HLSE08] Peter Haase, Holger Lewen, Rudi Studer, and Michael Erdmann. The NeOn Ontology Engineering Toolkit, 2008. http://watson.kmi.open.ac.uk/Downloads%20and%20Publications_files/neon-toolkit.pdf.
- [HM01] V. Haarslev and R. Möller. RACER System Description. In R. Goré, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, Siena, Italy*, pages 701–705. Springer-Verlag, 2001.
- [Hol99] Allen Holub. Building user interfaces for object-oriented systems, part 1. *JavaWorld*, January 1999. <http://www.javaworld.com/javaworld/jw-07-1999/jw-07-toolbox.html?page=4>.

- [Hor05] Ian Horrocks. Owl: a description logic based ontology language for the semantic web, 2005. <http://www.cs.ox.ac.uk/ian.horrocks/Publications/download/2005/Horr05c.pdf>.
- [HPSH03] Ian Horrocks, Peter F. Patel-Schneider, and Frank Van Harmelen. From SHIQ and RDF to OWL: The Making of a Web Ontology Language. *Journal of Web Semantics*, 1:7–26, 2003.
- [HRG10] Matthias Hert, Gerald Reif, and Harald C. Gall. Updating relational data via sparql/update. In *Proceedings of the 2010 EDBT/ICDT Workshops*, EDBT ’10, New York, NY, USA, 2010. ACM.
- [HRG11] Matthias Hert, Gerald Reif, and Harald C. Gall. A comparison of rdb-to-rdf mapping languages. In *Proceedings of the 7th International Conference on Semantic Systems*, I-Semantics ’11, pages 25–32, New York, NY, USA, 2011. ACM.
- [HS06] Hans-Jörg Happel and Stefan Seedorf. Applications of Ontologies in Software Engineering. In *2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006), held at the 5th International Semantic Web Conference (ISWC 2006)*, 2006.
- [IEE90] IEEE Std 610.12-1990:IEEE Standard Glossary of Software Engineering Terminology, 1990.
- [KG03] Alexander Kleshchev and Valeriya Gribova. From an Ontology-Oriented Approach Conception to User Interface Development. *International Journal of Information Theories and Applications*, 10, 2003.
- [KP88] Glenn E. Krasner and Stephen T. Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *Technical note, ParcPlace Systems*, 1988.
- [KSPH04] Aditya Kalyanpur, Evren Sirin, Bijan Parsia, and James Hendler. Hypermedia inspired ontology engineering environment: SWOOP. In *In: Proc. International Semantic Web Conference (ISWC) (2004)*, pages 7–11, 2004.
- [Lai08] Eric Lai. Size matters: Yahoo claims 2-petabyte database is world’s biggest, busiest. *Computerworld*, May 2008. <http://www.computerworld.com/s/article/9087918>.
- [LR01] Avraham Leff and James T. Rayfield. Web-application development using the model/view/controller design pattern. In *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*, EDOC ’01, pages 118–127, Washington, DC, USA, 2001. IEEE Computer Society.

- [LSV⁺10] Martin Luginbuehl, Peter Schumacher, Pascal Vuilleumier, Hugo Vereecke, Bjoern Heyse, Thomas Bouillon, and Michel Struys. Noxious Stimulation Response Index : a Novel Anesthetic State Index Based on Hypnotic-Opioid Interaction. *Anesthesiology*, 112(4):872–880, 2010.
- [McG04] James McGovern. *A Practical Guide to Enterprise Architecture*. Coad Series. Prentice Hall PTR, 2004.
- [MFH⁺09] Boris Motik, Achille Fokoue, Ian Horrocks, Zhe Wu, Carsten Lutz, and Bernardo Cuenca Grau. OWL 2 web ontology language profiles. W3C recommendation, W3C, October 2009. <http://www.w3.org/TR/2009/REC-owl2-profiles-20091027/>.
- [MM04] Eric Miller and Frank Manola. RDF primer. W3C recommendation, W3C, February 2004. <http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>.
- [MPPS09] Boris Motik, Bijan Parsia, and Peter F. Patel-Schneider. OWL 2 web ontology language structural specification and functional-style syntax. W3C recommendation, W3C, October 2009. <http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/>.
- [MR92] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '92, pages 195–202, New York, NY, USA, 1992. ACM.
- [MS05] Boris Motik and R. Studer. KAON2 – A scalable reasoning tool for the semantic web. *Proceedings of the 2nd European Semantic Web Conference (ESWC,05), Heraklion, Greece*, 2005.
- [MSH09] Boris Motik, Rob Shearer, and Ian Horrocks. Hypertableau Reasoning for Description Logics. *Journal of Artificial Intelligence Research*, 36:165–228, 2009.
- [MSS05] Nick Mitchell, Gary Sevitsky, and Harini Srivivasan. The Diary of a Datum: An Approach to Modeling Runtime Complexity in Framework-Based Applications. Technical report, IBM Research Report, 2005. RC23703.
- [MvH04] Deborah L. McGuinness and Frank van Harmelen. OWL web ontology language overview. W3C recommendation, W3C, February 2004. <http://www.w3.org/TR/2004/REC-owl-features-20040210/>.
- [Möl08] Ralf Möller. Reasoning for Ontology Engineering and Usage, Part 1 - Introduction to Standard Reasoning. In *Tutorial at 7th International Semantic Web Conference ISWC 2008*. Presented at the 7th International Semantic Web Conference ISWC 2008, Karlsruhe, Germany, October 2008.

- [Ope10] OpenLink. RDF Views of SQL Data (SQL Schema to RDF Ontology Mapping). Technical report, OpenLink Software, 2010. <http://virtuoso.openlinksw.com/whitepapers/relational%20rdf%20views%20mapping.html>.
- [Pau11] Heiko Paulheim. *Ontology-Based Application Integration*. Springer, 2011.
- [PLC⁺08] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. In Stefano Spaccapietra, editor, *Journal on data semantics X*, pages 133–173. Springer-Verlag, Berlin, Heidelberg, 2008.
- [PM10] Alexandre Passant and Pablo Mendes. sparqlPuSH: Proactive notification of data updates in RDF stores using PubSubHubbub. In *Proceedings of 6th Workshop on Scripting and Development for the Semantic Web*, CEUR Workshop Proceedings, Aachen, Germany, 2010. CEUR.
- [Pot96] Mike Potel. MVP: Model-View-Presenter, 1996. <http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>.
- [PP10] Heiko Paulheim and Florian Probst. Ontology-Enhanced User Interfaces: A Survey. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 6, 2010.
- [PPSM09] Bijan Parsia, Peter F. Patel-Schneider, and Boris Motik. OWL 2 web ontology language XML serialization. W3C recommendation, W3C, October 2009. <http://www.w3.org/TR/2009/REC-owl2-xml-serialization-20091027/>.
- [PS08] Eric Prud’hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>.
- [Qui63] R. Quillan. *A notation for representing conceptual information: an application to semantics and mechanical English paraphrasing*. Systems Development Corporation, 1963.
- [Ree79a] Trygve Reenskaug. Models-Views-Controllers. *Technical note, Xerox PARC*, December 1979.
- [Ree79b] Trygve Reenskaug. Thing-Model-View-Editor. *Technical note, Xerox PARC*, May 1979.
- [Ree11] Trygve Reenskaug. MVC, XEROX Parc 1978-79, 2011. <http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>.
- [RLC08] Mariano Rodriguez-Muro, Lina Lubyte, and Diego Calvanese. Realizing ontology based data access: A plug-in for protégé. In *Information Integration Methods, Architectures and Systems 2008*, pages 286–289, April 2008.

- [RMC12] Mariano Rodriguez-Muro and Diego Calvanese. Quest, an owl 2 ql reasoner for ontology-based data access. In *Proceedings of the 9th Int. Workshop on OWL: Experiences and Directions (OWLED 2012)*, volume 849 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, 2012.
- [RN02] Stuart Russell and Peter Norvig. *Artificial Intelligence : A Modern Approach*, page 333. Prentice Hall, Upper Saddle River, 2 edition, 2002.
- [Rod12a] Mariano Rodriguez. New build is now available. The -ontop- framework blog, August 2012. <http://obdalib.blogspot.de/2012/08/new-build-is-now-available.html>.
- [Rod12b] Mariano Rodriguez. [Obdalib-announcements] QuestOWL is now available. The Obdalib-announcements Mailing List, August 2012. <https://mail.inf.unibz.it/pipermail/obdalib-announcements/2012-August/000008.html>.
- [SBI85] James G. Schmolze, Bolt Beranek, and Newman Inc. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.
- [Sha96] Alec Sharp. *Smalltalk by Example: The Developer’s Guide*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1996.
- [SI06] Xiaomeng Su and Lars Ilebrikke. A Comparative Study of Ontology Languages and Tools. In Anne Pidduck, M. Ozsu, John Mylopoulos, and Carson Woo, editors, *Advanced Information Systems Engineering*, volume 2348 of *Lecture Notes in Computer Science*, pages 761–765. Springer Berlin / Heidelberg, 2006.
- [Sim63] R.F. Simmons. *Synthetic Language Behavior*. System Development Corporation, 1963.
- [SPG⁺07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51 – 53, 2007.
- [SSS91] Manfred Schmidt-Schaub and Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, February 1991.
- [TH03] Dmitry Tsarkov and Ian Horrocks. Dl reasoner vs. first-order prover. In *In Proc. of the 2003 Description Logic Workshop (DL 2003)*, volume 81 of *CEUR (http://ceur-ws.org)*, pages 152–159, 2003.
- [TH06] Dmitry Tsarkov and Ian Horrocks. FaCT++ Description Logic Reasoner: System Description. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 292–297. Springer Berlin / Heidelberg, 2006.

- [TP10] Károly Tilly and Zoltán Porkoláb. Semantic User Interfaces. *International Journal of Enterprise Information Systems*, 6(1):29–43, September 2010.
- [Tur10] Anni-Yasmin Turhan. Reasoning and explanation in EL and in expressive description logics. In *Proceedings of the 6th international conference on Semantic technologies for software engineering*, ReasoningWeb’10, pages 1–27, Berlin, Heidelberg, 2010. Springer-Verlag.
- [UG96] Mike Uschold and Michael Gruninger. Ontologies: Principles, methods and applications. *Knowledge Engineering Review*, 11:93–136, 1996.
- [W3C04] W3C. Xml schema part 2: Datatypes second edition. W3C recommendation, W3C, October 2004. <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/>.
- [W3C09a] W3C. OWL 2 web ontology language primer. W3C recommendation, W3C, October 2009. <http://www.w3.org/TR/2009/REC-owl2-primer-20091027/>.
- [W3C09b] W3C OWL Working Group. OWL 2 web ontology language document overview. W3C recommendation, W3C, October 2009. <http://www.w3.org/TR/2009/REC-owl2-overview-20091027/>.

A. Overview: Description Logics

A.1. Basic Languages

\mathcal{AL} Attributive language allows:

- Atomic negation
- Concept intersection
- Universal restrictions
- Limited existential quantification

\mathcal{FL} Frame based description language allows:

- Concept intersection
- Universal restrictions
- Limited existential quantification
- Role restriction

\mathcal{EL} allows:

- Concept intersection
- Existential restrictions

A.2. Extension Operators

\mathcal{F} Functional properties

\mathcal{O} Nominals

\mathcal{E} Full existential qualification

\mathcal{I} Inverse properties

\mathcal{U} Concept union

\mathcal{C} Complex concept negation

\mathcal{N} Cardinality restrictions

\mathcal{H} Role hierarchy

\mathcal{Q} Qualified cardinality restrictions

\mathcal{R} Limited complex role inclusion axioms;
reflexivity and irreflexivity; role dis-
jointness

(\mathcal{D}) Datatype properties, data values or
data types

A.3. Abbreviations

\mathcal{S} is the abbreviation for \mathcal{ALC} with \mathcal{R}_+ transitive roles.

B. SmartPilot View: Ontology

Each class in Figure 4.3 has attributes or further relationships. These can be seen in the following detailed figures. The key is given in Figure B.1.

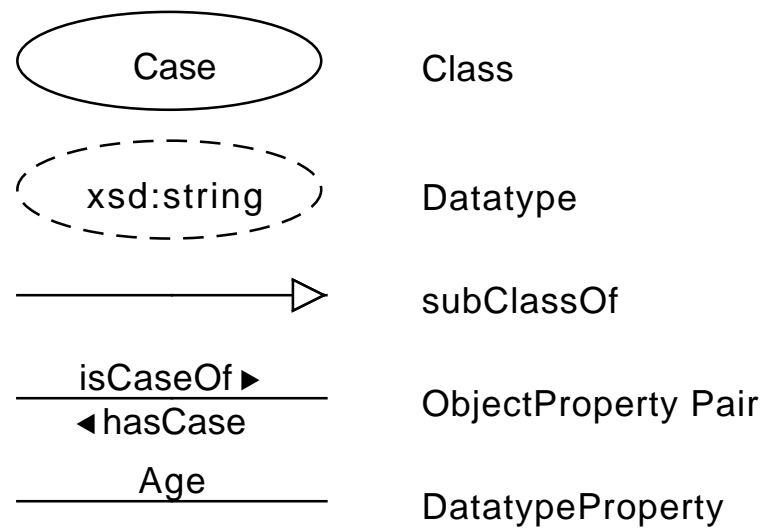


Figure B.1.: SmartPilot View: Ontology - Key

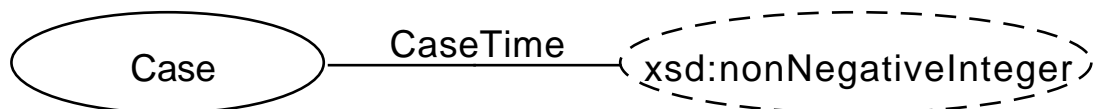


Figure B.2.: SmartPilot View: Ontology - Case

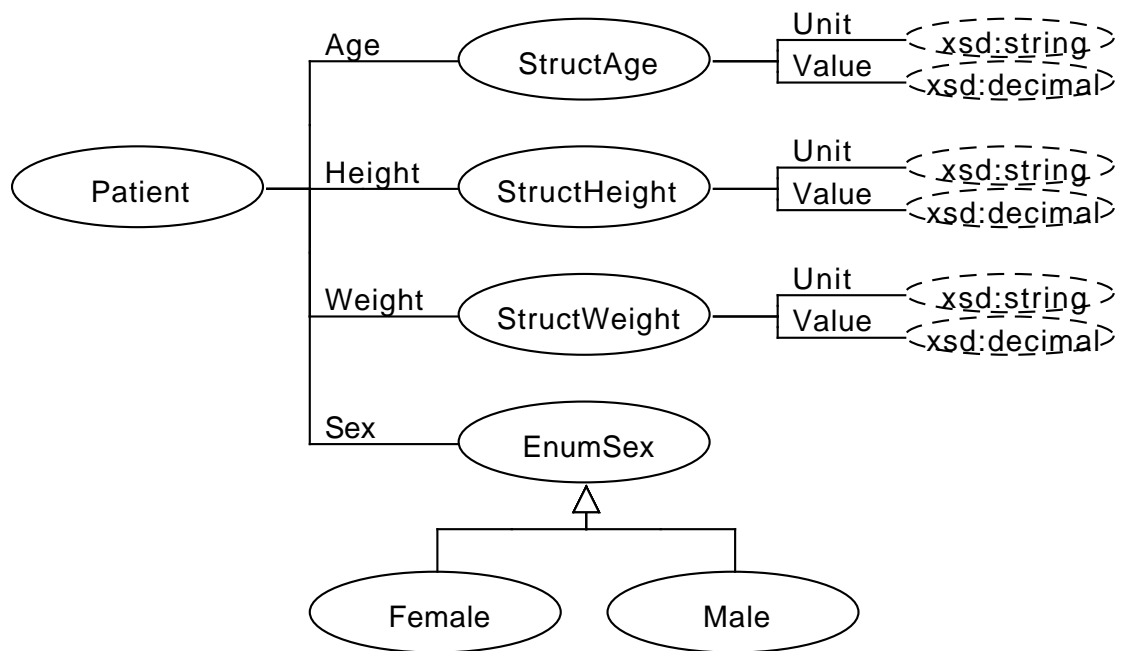


Figure B.3.: SmartPilot View: Ontology - Patient

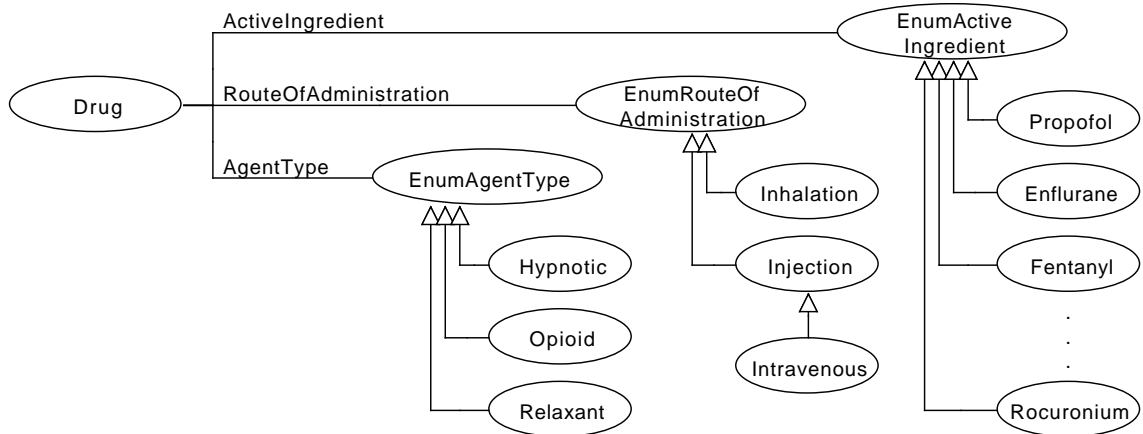


Figure B.4.: SmartPilot View: Ontology - Drug

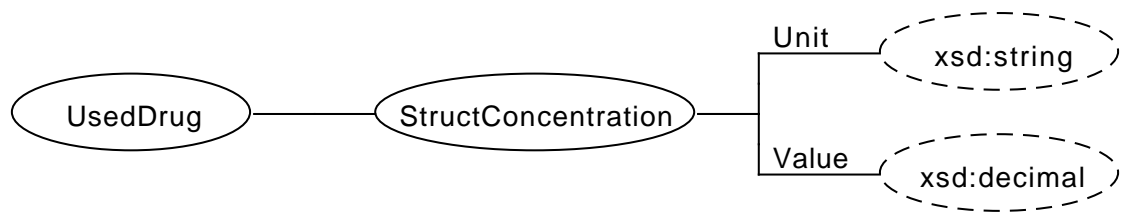


Figure B.5.: SmartPilot View: Ontology - Used Drug

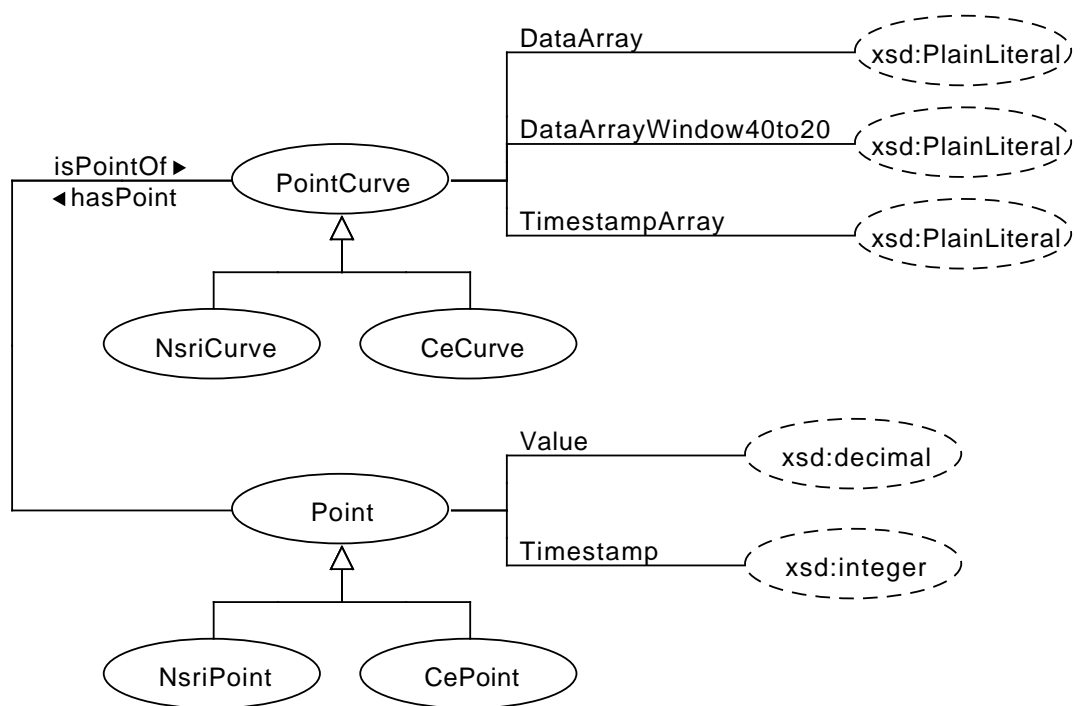


Figure B.6.: SmartPilot View: Ontology - Curves and Points

C. SmartPilot View: UI Data Subset

Table C.1.: SmartPilot View, Relevant Data Subset

Data	Description
Case	A single anesthesia for one patient is called a case
Case Time	The case time starts with 0 at the beginning of a case
Patient	undergoes anesthesia in the current case
Sex	} The displayed patient data
Age	
Weight	
Height	
Drug	having certain properties
Active Ingredients	The active ingredients known by SmartPilot View
Propofol	Fluid hypnotic
Desflourane	} Volatile hypnotics
Enflourane	
Halothane	
Isoflourane	
Sevoflourane	
Alfentanil	} Fluid opioids
Fentanyl	
Remifentanil	
Sufentanil	} Fluid muscle relaxants
Pancuronium	
Rocuronium	
Used Drugs	used in a certain case with the respective concentration
Concentration	how the used drug is diluted
NSRI Graph	Noxious Stimulation Response Index, see Section 2.6
Graph Data	contains points of NSRI over time
Ce Graphs	Effect Site Concentrations (Ce), One graph for each active ingredient used in the current case, see Section 2.6
Graph Data	contains points of Ce over time

D. System Overview

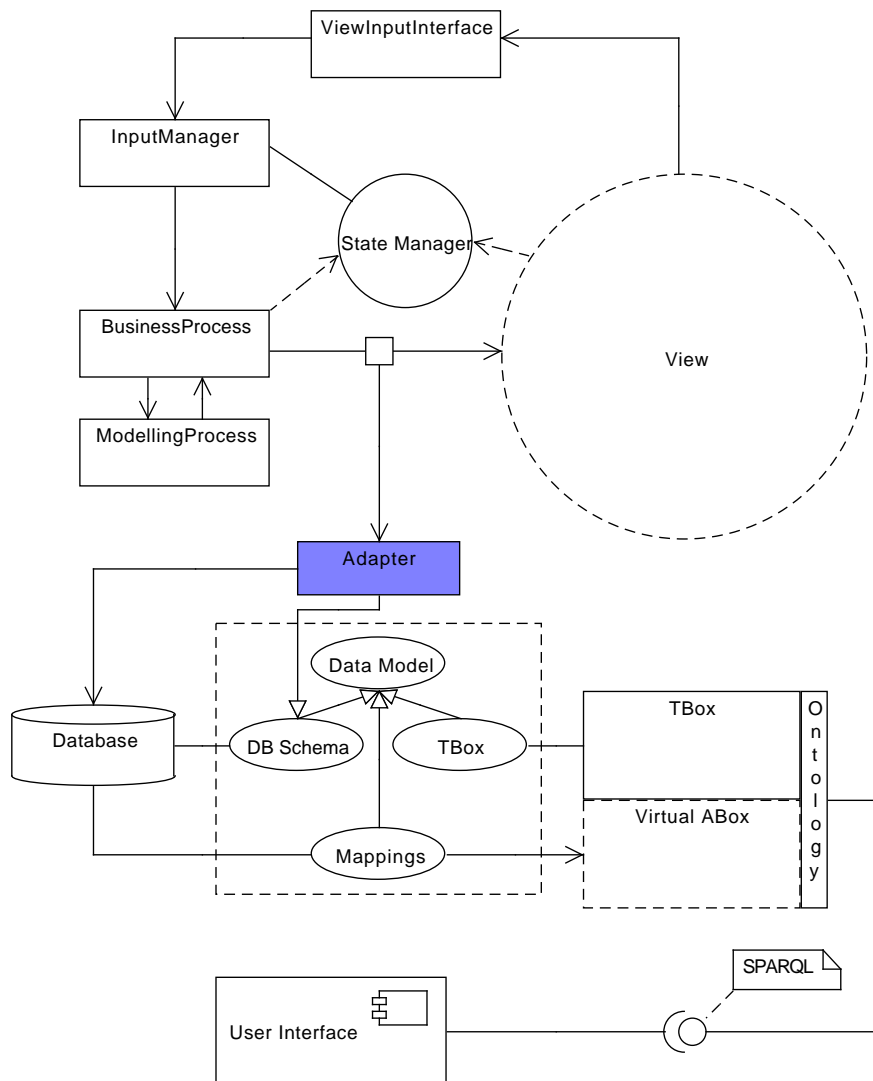


Figure D.1.: SPV with OBDA: Complete Overview

E. Evaluation Results

E.1. Mapping Completeness

This Appendix shows the results of the SPARQL queries described in Section 5.2. For better readability the following adaptations were made.

1. Arrays are shortened to show only the first two and the last element.
2. Decimal values are cut off at the third fractional digit. Thereby 12 digits are omitted.
Short: 99.835
Original: 99.835731506347699
3. Datatypes are provided in the column head
Short: "213"
Original: "213"^^xsd:integer
4. URIs are shortened by their prefix.
Short: :Case-113
Original: <http://www.draeger.com/ontologies/spv#Case-113>
5. Only the first 30 results are shown.

Please note that the Ce Curve class and its respective properties can be ignored for the tests and the evaluation, as described in Section 4.5.4.

E.2. Case

```

1 PREFIX : <http://www.draeger.com/ontologies/spv#>
  select ?case ?caseTime ?patient ?nsriCurve ?usedDrug where
  {
    ?case a :Case ;
5     :CaseTime ?caseTime ;
     :isCaseOf ?patient ;
     :hasNsriCurve ?nsriCurve ;
     :hasUsedDrug ?usedDrug .
  }

```

Table E.1.: Cases

case	caseTime xsd:integer	patient	nsriCurve	usedDrug
:Case-113	"213"	:Patient-85	:NsriPointCurve-113	:UsedDrug-11-113
:Case-113	"213"	:Patient-85	:NsriPointCurve-113	:UsedDrug-7-113
:Case-114	"59"	:Patient-86	:NsriPointCurve-114	:UsedDrug-11-114
:Case-114	"59"	:Patient-86	:NsriPointCurve-114	:UsedDrug-7-114
:Case-115	"428"	:Patient-87	:NsriPointCurve-115	:UsedDrug-11-115
:Case-115	"428"	:Patient-87	:NsriPointCurve-115	:UsedDrug-7-115
:Case-116	"13"	:Patient-88	:NsriPointCurve-116	:UsedDrug-11-116
:Case-116	"13"	:Patient-88	:NsriPointCurve-116	:UsedDrug-7-116
:Case-117	"127"	:Patient-89	:NsriPointCurve-117	:UsedDrug-11-117
:Case-117	"127"	:Patient-89	:NsriPointCurve-117	:UsedDrug-7-117
:Case-118	"16"	:Patient-90	:NsriPointCurve-118	:UsedDrug-11-118
:Case-118	"16"	:Patient-90	:NsriPointCurve-118	:UsedDrug-7-118
:Case-119	"371"	:Patient-91	:NsriPointCurve-119	:UsedDrug-11-119
:Case-119	"371"	:Patient-91	:NsriPointCurve-119	:UsedDrug-7-119
:Case-122	"482"	:Patient-94	:NsriPointCurve-122	:UsedDrug-11-122
:Case-122	"482"	:Patient-94	:NsriPointCurve-122	:UsedDrug-7-122
:Case-123	"179"	:Patient-95	:NsriPointCurve-123	:UsedDrug-11-123
:Case-123	"179"	:Patient-95	:NsriPointCurve-123	:UsedDrug-7-123
:Case-124	"192"	:Patient-96	:NsriPointCurve-124	:UsedDrug-11-124
:Case-124	"192"	:Patient-96	:NsriPointCurve-124	:UsedDrug-7-124
:Case-125	"161"	:Patient-97	:NsriPointCurve-125	:UsedDrug-11-125
:Case-125	"161"	:Patient-97	:NsriPointCurve-125	:UsedDrug-7-125
:Case-126	"8"	:Patient-98	:NsriPointCurve-126	:UsedDrug-11-126
:Case-126	"8"	:Patient-98	:NsriPointCurve-126	:UsedDrug-7-126
:Case-127	"10"	:Patient-99	:NsriPointCurve-127	:UsedDrug-11-127
:Case-127	"10"	:Patient-99	:NsriPointCurve-127	:UsedDrug-7-127
:Case-128	"112"	:Patient-100	:NsriPointCurve-128	:UsedDrug-11-128
:Case-128	"112"	:Patient-100	:NsriPointCurve-128	:UsedDrug-7-128
:Case-129	"29"	:Patient-101	:NsriPointCurve-129	:UsedDrug-11-129
:Case-129	"29"	:Patient-101	:NsriPointCurve-129	:UsedDrug-7-129

E.3. Drugs

```
1 PREFIX : <http://www.draeger.com/ontologies/spv#>
  select ?drug ?activeIng ?agentType ?route ?usedDrug where
{
  ?drug a :Drug ;
5   :ActiveIngredient ?activeIng ;
   :AgentType ?agentType ;
   :RouteOfAdministration ?route ;
   :hasUsedDrug ?usedDrug .
}
```

Table E.2.: Drugs

drug	activeIng	agentType	route
:Drug-10	:Sufentanil	:Opioid	:Intravenous
:Drug-2	:Enflurane	:Hypnotic	:Inhalation
:Drug-9	:Alfentanil	:Opioid	:Intravenous
:Drug-12	:Rocuronium	:Relaxant	:Intravenous
:Drug-8	:Fentanyl	:Opioid	:Intravenous
:Drug-3	:Isoflurane	:Hypnotic	:Inhalation
:Drug-4	:Sevoflurane	:Hypnotic	:Inhalation
:Drug-7	:Remifentanil	:Opioid	:Intravenous
:Drug-13	:Pancuronium	:Relaxant	:Intravenous
:Drug-5	:Desflurane	:Hypnotic	:Inhalation
:Drug-1	:Halothane	:Hypnotic	:Inhalation
:Drug-11	:Propofol	:Hypnotic	:Intravenous

E.4. NSRI Curves

```
1 PREFIX : <http://www.draeger.com/ontologies/spv#>
select ?nsriCurve ?dataArray ?dataWindow where
{
  ?nsriCurve a :NsriPointCurve ;
5   :dataArray ?dataArray ;
   :dataArrayWindow40to20 ?dataWindow .
}
```

Table E.3.: NSRI Curves

nsriCurve	dataArray	dataWindow	case
:NsriPointCurve-116	"{100,100,...,100}"	"{100,100,...,100}"	:Case-116
:NsriPointCurve-114	"{100,100,...,100}"	"{100,100,...,100}"	:Case-114
:NsriPointCurve-123	"{100,100,...,99.835}" ¹	"{100,100,...,99.835}"	:Case-123
:NsriPointCurve-125	"{100,100,...,99.902}"	"{100,100,...,99.902}"	:Case-125
:NsriPointCurve-126	"{100,100,...,99.446}"	"{100,100,...,99.446}"	:Case-126
:NsriPointCurve-127	"{100,100,...,NaN}"	"{100,100,...,NaN}"	:Case-127
:NsriPointCurve-128	"{100,100,...,99.805}"	"{100,100,...,99.805}"	:Case-128
:NsriPointCurve-132	"{100,100,...,NaN}"	"{100,100,...,NaN}"	:Case-132
:NsriPointCurve-129	"{100,100,...,NaN}"	"{100,100,...,NaN}"	:Case-129
:NsriPointCurve-130	"{100,100,...,99.768}"	"{100,100,...,99.768}"	:Case-130
:NsriPointCurve-117	"{100,100,...,100}"	"{100,100,...,100}"	:Case-117
:NsriPointCurve-118	"{100,100,...,99.723}"	"{100,100,...,99.723}"	:Case-118
:NsriPointCurve-115	"{100,100,...,100}"	"{100,100,...,100}"	:Case-115
:NsriPointCurve-135	"{100,100,...,NaN}"	"{78.275,78.863,...,NaN}"	:Case-135
:NsriPointCurve-122	"{100,100,...,99.778}"	"{100,100,...,99.778}"	:Case-122

E.5. Patient

```

1 PREFIX : <http://www.draeger.com/ontologies/spv#>
  select ?patient ?sex ?age_v ?age_u ?weight_v ?weight_u
    ?height_v ?height_u ?case where
{
5   ?patient a :Patient ;
      :Sex ?sex ; :Age ?age ; :Weight ?weight ; :Height ?height ;
      :hasCase ?case .
      ?age :Value ?age_v ;
        :Unit ?age_u .
10  ?weight :Value ?weight_v ;
        :Unit ?weight_u .
      ?height :Value ?height_v ;
        :Unit ?height_u .
}

```

Table E.4.: Patient

patient	sex	age_v decimal	age_u string	weight_v decimal	weight_u string	height_v decimal	height_u string	case
:Patient-3	:Female	"39.0"	"a"	"74.0"	"kg"	"174.0"	"cm"	:Case-3
:Patient-8	:Female	"19.0"	"a"	"52.0"	"kg"	"188.0"	"cm"	:Case-8
:Patient-32	:Female	"30.0"	"a"	"48.0"	"kg"	"150.0"	"cm"	:Case-40
:Patient-29	:Female	"40.0"	"a"	"40.0"	"kg"	"200.0"	"cm"	:Case-37
:Patient-2	:Male	"36.0"	"a"	"76.0"	"kg"	"176.0"	"cm"	:Case-2
:Patient-17	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-25
:Patient-45	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-54
:Patient-30	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-38
:Patient-10	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-14
:Patient-4	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-4
:Patient-7	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-7
:Patient-18	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-26
:Patient-28	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-36
:Patient-37	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-46
:Patient-5	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-5
:Patient-19	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-27
:Patient-46	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-55
:Patient-20	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-28
:Patient-6	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-6
:Patient-48	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-57
:Patient-21	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-29
:Patient-12	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-20
:Patient-13	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-21
:Patient-38	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-47
:Patient-14	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-22
:Patient-15	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-23
:Patient-35	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-43
:Patient-23	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-31
:Patient-42	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-51
:Patient-31	:Male	"40.0"	"a"	"75.0"	"kg"	"175.0"	"cm"	:Case-39

E.6. Used Drugs

```

1 PREFIX : <http://www.draeger.com/ontologies/spv#>
select ?usedDrug ?conc_v ?conc_u ?caseordrug where
{
  ?usedDrug a :UsedDrug ;
5   :Concentration ?conc ;
   :isUsedDrugOf ?caseordrug .
  ?conc :Value ?conc_v ;
   :Unit ?conc_u .
}

```

Table E.5.: Used Drug

usedDrug	conc_v xsd:decimal	conc_u xsd:string	caseordrug
:UsedDrug-11-89	"10.0"	"mg/mL"	:Case-89
:UsedDrug-9-89	"0.7"	"mg/mL"	:Case-89
:UsedDrug-7-89	"40.0"	"ug/mL"	:Case-89
:UsedDrug-11-89	"10.0"	"mg/mL"	:Drug-11
:UsedDrug-9-89	"0.7"	"mg/mL"	:Drug-9
:UsedDrug-7-89	"40.0"	"ug/mL"	:Drug-7
:UsedDrug-11-90	"10.0"	"mg/mL"	:Case-90
:UsedDrug-7-90	"40.0"	"ug/mL"	:Case-90
:UsedDrug-11-90	"10.0"	"mg/mL"	:Drug-11
:UsedDrug-7-90	"40.0"	"ug/mL"	:Drug-7
:UsedDrug-11-91	"10.0"	"mg/mL"	:Case-91
:UsedDrug-7-91	"40.0"	"ug/mL"	:Case-91
:UsedDrug-11-91	"10.0"	"mg/mL"	:Drug-11
:UsedDrug-7-91	"40.0"	"ug/mL"	:Drug-7
:UsedDrug-11-92	"10.0"	"mg/mL"	:Case-92
:UsedDrug-7-92	"40.0"	"ug/mL"	:Case-92
:UsedDrug-11-92	"10.0"	"mg/mL"	:Drug-11
:UsedDrug-7-92	"40.0"	"ug/mL"	:Drug-7
:UsedDrug-11-93	"10.0"	"mg/mL"	:Case-93
:UsedDrug-7-93	"40.0"	"ug/mL"	:Case-93
:UsedDrug-11-93	"10.0"	"mg/mL"	:Drug-11
:UsedDrug-7-93	"40.0"	"ug/mL"	:Drug-7
:UsedDrug-11-94	"10.0"	"mg/mL"	:Case-94
:UsedDrug-7-94	"40.0"	"ug/mL"	:Case-94
:UsedDrug-11-94	"10.0"	"mg/mL"	:Drug-11
:UsedDrug-7-94	"40.0"	"ug/mL"	:Drug-7
:UsedDrug-11-95	"10.0"	"mg/mL"	:Case-95
:UsedDrug-7-95	"40.0"	"ug/mL"	:Case-95
:UsedDrug-11-95	"10.0"	"mg/mL"	:Drug-11
:UsedDrug-7-95	"40.0"	"ug/mL"	:Drug-7