Semantic Web Enabled Software Engineering

Philipp Schügerl

A Thesis
In the Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy(Computer Science) at
Concordia University
Montreal, Quebec, Canada

September 2011

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Philipp Schügerl

Entitled: Semantic Web Enabled Software Engineering

and submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY (Computer Science)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. M. Pugh

_____ External Examiner
Dr. G. Antoniol

_____ External to Program
Dr. F. Khendek

_____ Examiner
Dr. S. Bergler

_____ Examiner
Dr. V. Haarslev

_____ Thesis Supervisor
Dr. J. Rilling

Approved by _____
Dr. V. Haarslev, Graduate Program Director

September 8, 2011 _____
Dr. Robin A.L. Drew, Dean
Faculty of Engineering & Computer Science

Abstract

Semantic Web Enabled Software Engineering

Philipp Schügerl, Ph.D.
Concordia University, 2011

Ontologies allow the capture and sharing of domain knowledge by formalizing information and making it machine understandable. As part of an information system, ontologies can capture and carry the reasoning knowledge needed to fulfill different application goals. Although many ontologies have been developed over recent years, few include such reasoning information. As a result, many ontologies are not used in real-life applications, do not get reused or only act as a taxonomy of a domain.

This work is an investigation into the practical use of ontologies as a driving factor in the development of applications and the incorporation of Knowledge Engineering as a meaningful activity into modern agile software development. This thesis contributes a novel methodology that supports an incremental requirement analysis and an iterative formalization of ontology design through the use of ontology reasoning patterns. It also provides an application model for ontology-driven applications that can deal with non-ontological data sources. A set of case studies with various application specific goals helps to elucidate whether ontologies are in fact suitable for more than simple knowledge formalization and sharing, and can act as the underlying structure for developing large-scale information systems. Tasks from the area of bug-tracker quality mining and clone detection are evaluated for this purpose.

Acknowledgments

Foremost, I would like to express my gratitude to my supervisor, Dr. Juergen Rilling, for giving me the opportunity to work on an interesting research project together with the Defence Research and Development Canada (DRDC) agency. His intellectual guidance, insights and patience have made this work possible and have shaped me, both as a person and researcher. My gratitude also goes out to Dr. Sabine Bergler for invaluable support over the years. Her suggestions from a research perspective outside the Software Engineering community have helped me keep this thesis balanced. I would also like to thank Dr. Volker Haarslev for his great feedback that has always encouraged me. Furthermore, my gratitude goes to my fellow master's and Ph.D. students in the research lab who have always given me great feedback. Last but not least, I want to thank my family and friends for their support, encouragement and inspiration during my studies — You are the source of my success.

This research was partially funded by DRDC Valcartier (contract no. W7701-081745 /001/QCV) and by NSERC Grant: 227680.

Dedication

To my parents and brother...

Table of Contents

List of Tables

List of Figures

Listings

Glossary

The following is a list of acronyms used throughout the dissertation:

AI Artificial Intelligence, the branch of computer science that aims to create intelligent machines.

DL Description Logic, a family of formal knowledge representation languages.

IR Information Retrieval, the area of study concerned with searching for documents and the information within documents.

SE Software Engineering, a systematic approach dedicated to designing, implementing, and modifying software.

KE Knowledge Engineering, an engineering discipline which involves the integration of knowledge into systems to solve complex problems.

XP Extreme Programming, an agile software development methodology.

AST Abstract Syntax Tree, a tree representation of the abstract syntactic structure of source code.

API Application Programming Interface, a particular set of specifications that software programs can follow to communicate.

CBR Case-Based Reasoning, the process of solving new problems based on the solutions of similar past problems.

IDE Integrated Development Environment, a software application that provides development facilities to computer programmers.

KBS Knowledge-Based Systems, AI tools that provides intelligent decisions with justification.

NLP Natural Language Processing, the field of linguistics concerned with the interactions between computers and human languages.

OWL Web Ontology Language, a family of knowledge representation languages for authoring ontologies.

OWA Open World Assumption, the assumption that the truth-value of a statement is independent of whether or not it is known.

RDF Resource Description Framework, a conceptual description of information that is implemented in web resources.

RUP Rational Unified Process, an iterative software development process framework.

UML Unified Modelling Language, a visual language for modeling the structure of software artifacts.

UNA Unique Name Assumption, the assumption that different names always refer to different entities in the world.

URI Uniform Resource Identifier, a string of characters used to identify a name or a resource on the Internet.

W3C the World Wide Web Consortium, the standards body for web technologies.

XML Extensible Markup Language, a set of rules for encoding documents in machine-readable form.

ASEG Ambient Software Evolution Group, a research group of Concordia University.

DRDC Defence Research and Development Canada, an agency for the scientific and technological needs of the Canadian Forces.

KBSE Knowledge-Based Software Engineering, the vision of software engineering as an AI-supported activity.

RDFS Resource Description Framework Schema, a set of classes with certain properties for RDF.

REST Representational State Transfer, a style of software communication for the Internet.

SPARQL SPARQL Protocol and RDF Query Language, an RDF query language for triple storages.

Chapter 1

Introduction

Creating software is a knowledge-intensive activity. In fact, it is the amount and scope of relevant knowledge that makes software so difficult. — Peter G. Selfridge, AT&T Bell Laboratories

Knowledge Engineering is an important aspect of Artificial Intelligence and Cognitive Science. It is the process of defining and formalizing information using knowledge representation techniques so that computers are able to process and use it [SBF98]. In particular, it is the knowledge engineer's task to define facts from which a computer can infer additional knowledge (reasoning services) [HKR09]. With the emergence of the semantic web, ontologies (a form of knowledge representation) have found their way into modern software development. The role of the knowledge engineer in a modern software project is, however, still largely undefined. While Software Engineering processes guide software development and provide activities, techniques and artifacts to developers, there exists little work on how to incorporate Knowledge Engineering into the software development life cycle. Problems such as the design, management and reuse of knowledge have emerged and there exists a need to make Knowledge Engineering an integral part

of modern incremental/iterative software projects. At the same time, problem domains which particularly benefit from Knowledge Engineering, and can be modeled and solved efficiently using knowledge representation techniques and reasoning services, need to be identified.

In this thesis, a novel ontology design methodology, which fits into an agile software application development process, is introduced. Ontology design is no longer treated as a separate activity but is incorporated into application development (therefore, throughout this thesis, also referred to as "ontology development"). User stories are leveraged as the starting point for modeling knowledge within an agile process. Iterations within Knowledge Engineering are facilitated by introducing ontology design patterns as reusable, best-practice solutions for ontology design. By applying ontology design patterns repeatedly throughout the development process an ontology can be incrementally refined while ensuring a satisfactory design quality and capturing individual responsibilities of user stories. To ensure reasoners can infer additional knowledge from the facts stated by the knowledge engineer, a novel type of ontology design patterns, namely structural reasoning patterns, are introduced. Additionally, an application model for the use of non-ontological data sources is introduced in this thesis as a proven solution for a particular set of problems that can be tackled using Knowledge Engineering and reasoning services.

1.1 Motivation and Objective

The work in this thesis is cross-disciplinary research which positions itself at the intersection of Software Engineering and Knowledge Engineering. The original (generally defined) motivation for this thesis was the question of "How can semantic web technologies

be of use for Software Engineering?" In particular, the "applicability of reasoning services on large Software Engineering ontologies" and "the feasibility of mining Software Engineering artifacts using semantic web technologies" were considered as motivational ideas. From this starting point, the development of different applications scenarios has led to the investigation of how developers can leverage reasoning service and, ultimately, how ontology design can be incorporated into agile software development. This refined perspective has led to the following set of objectives for this thesis which can be defined as:

1. To investigate how Knowledge Engineering can benefit the Software Engineering domain.

2. To propose a methodology for the integration of ontology design into an agile software process-skeleton.

3. To provide an application model for the proposed methodology.

4. To evaluate the methodology and application model through different application scenarios

This work is focusing on the development of ontology-driven applications (applications which make use of ontologies and reasoning services) and ultimately tries to lower the existing fear of first contact with semantic web technologies by allowing a seamless integration of ontologies into the widely used agile software processes and practices. The expected impact of the thesis is the two-fold: (1) the methodology defined in this thesis should allow project managers to quickly integrate ontologies into their agile software development process and (2) the best practices as well as the application model

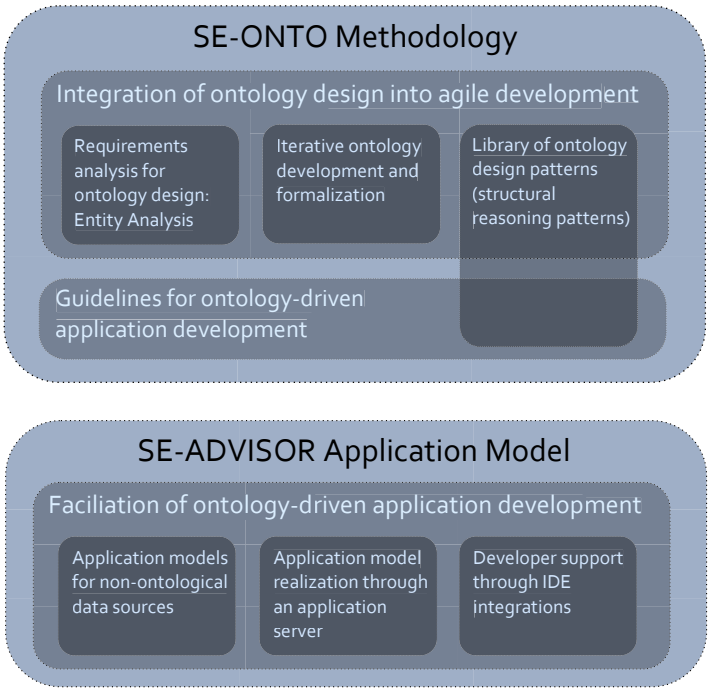

Figure 1.1: Main contributions of the thesis

should guide software developers in building ontology-driven applications that are of lower complexity than their traditionally built counterparts.

1.2 Contributions

The main objective of this thesis is to investigate to what extent the design of ontologies can be incorporated into modern Software Engineering processes, and how developers can leverage ontologies and reasoning services in order to solve application problems (develop "ontology-driven" applications). The thesis introduces a novel methodology, called SE-ONTO methodology, which incorporates ontology design activities into the agile SCRUM [Sch97] software development process-skeleton. It defines a set of techniques regarding the incremental requirement analysis (for ontology design) and the iterative development/formalization of ontologies in agile software processes which is facilitated by

a library of novel ontology reasoning design patterns. While the patterns foster a better understanding of what problems can be solved efficiently using ontologies and reasoning, the thesis also contributes a specific application model (called SE-ADVISOR) for the incorporation of existing non-ontological data sources into an ontology-driven application. Figure 1.1 depicts a general overview of the contributions of this thesis: (1) The integration of ontology design into agile development (SE-ONTO methodology). (2) Guidelines for ontology-driven application development, and (3) the facilitation of ontology-driven development through an application model (SE-ADVISOR) and supporting tools.

1.3 Outline

The remainder of this thesis is organized as follows: Chapter 2 describes the background of this thesis, including work from Software Engineering and Artificial Intelligence, as well as recent contributions in Knowledge Engineering and the modeling of knowledge (such as OWL). Chapter 3 reviews the state of the art in ontology design methodologies, and introduces the novel SE-ONTO methodology that can be incorporated into the agile SCRUM software process-skeleton. As part of this ontology design methodology that fosters the use of reasoning services, a novel type of ontology reasoning design patterns is presented in Chapter 4. An application model for the use of the SE-ONTO methodology in conjunction with non-ontological data sources is presented in Chapter 5. The methodology, application model and ontology reasoning design patterns are evaluated in terms of their performance and their ability to support multiple application scenarios in Chapter 6. The thesis concludes with a summary of the achieved results in Chapter 7.

Chapter 2

Background

It is not the aim of AI to build intelligent machines having understood natural intelligence, but to understand natural intelligence by building intelligent machines.
— Ipke Wachsmuth, The Concept of Intelligence in AI

While Software Engineering has emerged as an engineering approach to the professional development of software which is supported by many different processes, Knowledge Engineering and Artificial Intelligence in general have not yet seen wide-spread process oriented support by the scientific community. This lack of support can be partially addressed by developing a methodology that incorporates ontology design into agile software process-skeletons. Relevant background for this methodology, such as Software Engineering, Knowledge Engineering, and Artificial Intelligence, are explored in this chapter. Further, technologies and concepts which have been adapted in the context of this thesis are discussed and explained. Each section concludes with a review of related work and its relevance to the contributions of this thesis.

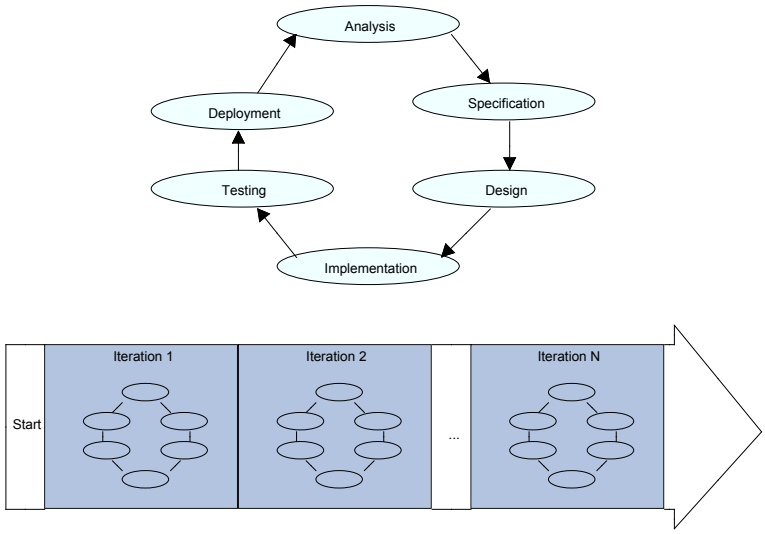

Figure 2.1: Iterative software development process

2.1 Software Engineering

Software Engineering (SE) is a systematic approach concerned with the development and maintenance of software systems. It covers areas such as requirements engineering, designing, coding, testing, maintaining, and assessing software. One of the key elements of SE is the need for a structured set of activities required for the development of a system, also called software process. In contrast to a life-cycle model, which only provides an outline of a project flow, a software methodology describes details (deliverables and artifacts) on how to build software [ZSG79].

Traditional models such as the sequential waterfall model [Roy70] or the V model [Ger92] have been replaced over time by iterative and incremental development methods such as the spiral model [Boe86]. Iterative models suggest multiple development iterations with frequent releases of the developed software and an incremental refinement of software over time (depicted in Figure 2.1). Common to these models is that they support

the development of software solutions for which requirements are expected to change periodically over time [BA96]. Consequently, the efficient management and execution of these changes are critical to software quality and evolution [BLP00]. Agile approaches, such as RUP [JBR99], XP [Bec99] or SCRUM [Sch97], are lightweight methodologies in the sense that they try to minimize the overhead forced upon a developer by a software process and focus on a working product. For this purpose the agile manifesto [Bec01] defines the following priorities:

- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Individuals and interactions over processes and tools
- Responding to change over following a plan

The observation that software is constantly changing and evolving has also been made by Lehman and Belady [Leh79] who describe that there needs to be "a balance between forces driving new developments on one hand, and forces that slow down progress on the other hand". The findings have been stated as Lehman's laws:

- FIRST LAW: A system that is used will be changed.
- SECOND LAW: An evolving system increases its complexity unless work is done to reduce it.

Life cycle models (e.g. Figure 2.2, [ZSG79]) have been introduced to model and guide the activities involved in software development.

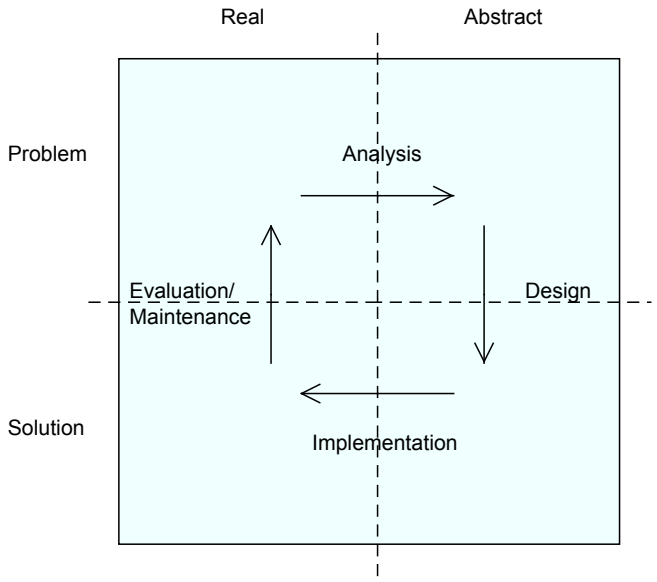

Figure 2.2: Software development lifecycle

As part of a software methodology, knowledge has to be continually integrated from different sources (including source code, repositories, documentation, and test cases), analyzed at different levels of abstraction (from single variables to system documentation), and shared among individuals. The need to integrate various resources represents a similar challenge to the one faced in Knowledge Engineering. Nevertheless, methodologies for Knowledge Engineering often miss the maturity of those of Software Engineering.

2.2 Artificial Intelligence and Software Engineering

The term Artificial Intelligence (AI) was formed at the Dartmouth conference in 1956 as the study and design of intelligent agents [HJK+04]. P. H. Winston [Win93] states that "AI is the study of the computation that makes it possible to perceive, reason, and act". Artificial Intelligence is often miss-interpreted as re-creating human intelligence,

although today's AI research community agrees on a broader perspective. Nevertheless, the late '60s and early '70s were marked by exactly such an understanding and the confidence to be able to imitate human intelligence within a short time. In 1965 H. A. Simon is quoted "Machines will be capable, within twenty years, of doing any work a man can do" [Cre93]. After initial success and heavy funding through the U.S. Department of Defense, AI research declined in the mid '70s with little progress made towards this final goal. The '80s were dominated by the success of expert systems (and knowledge-based systems in general). Today, AI can be found in many systems ranging from games to medical diagnosis and logistics. A driving factor of AI is the increasing power of personal computers (CPUs as well as GPUs), allowing for larger knowledge representations and more sophisticated reasoning within it.

While there are not many intersections between AI and SE on first sight, Figure 2.3 shows several main areas in which interdisciplinary work exists. In both domains, expressing knowledge and expertise is a fundamental aspect (although often less formal in SE). Especially Knowledge-Based Systems (KBS), Knowledge-Based Software Engineering (KBSE) and Ambient Intelligence (AMI) or research in the field of Knowledge Engineering build upon the methods explored in classical AI.

In the Article "Expert Systems for Software Engineering", Tsai et al. [TZ88] describe why SE cannot be easily supported by expert systems. They state that SE problems are ill-structured and that there exists little expertise in many SE areas. In contrast to the practices in place 20 years ago, today's software projects follow well-defined software processes. The focus and ideas of what an expert system can achieve (within the domain of SE) have changed: In contrast to a general purpose SE expert system, applicable to any

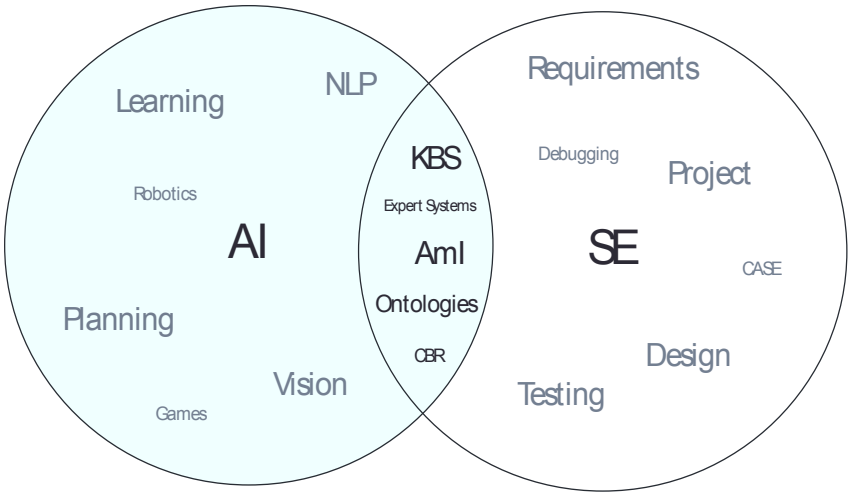

Figure 2.3: Overlapping research in AI and SE

software project, modern expert systems are highly specialized to a specific project and only share core constraints and concepts with other projects.

Software Engineering itself depends heavily on the experience of experts. Design patterns are an excellent example of expert knowledge, which is transferred between individuals [AJWH03]. Similarly, expertise in development methods, tools, and techniques needs to be captured and shared in every modern software project.

Software comprehension and reverse engineering have been an important aspect of SE since its beginning. The late '80s defined the systematic reuse and management of experiences, knowledge, products, and processes through the use of the so-called Experience Factory (EF) [Bas85] or also referred to as Learning Software Organization (LSO) [BCR94]. One of the first organizations to incorporate an EF was SEL (the NASA Software Engineering Laboratory) [RU89]. Other EF applications were developed in the USA and Europe [HSW98] [HSW91] [ABKD+02]. While EF methods focus on capturing organizational and process-oriented knowledge, synergies through collaboration methods and modern knowledge representation are left out of their scope.

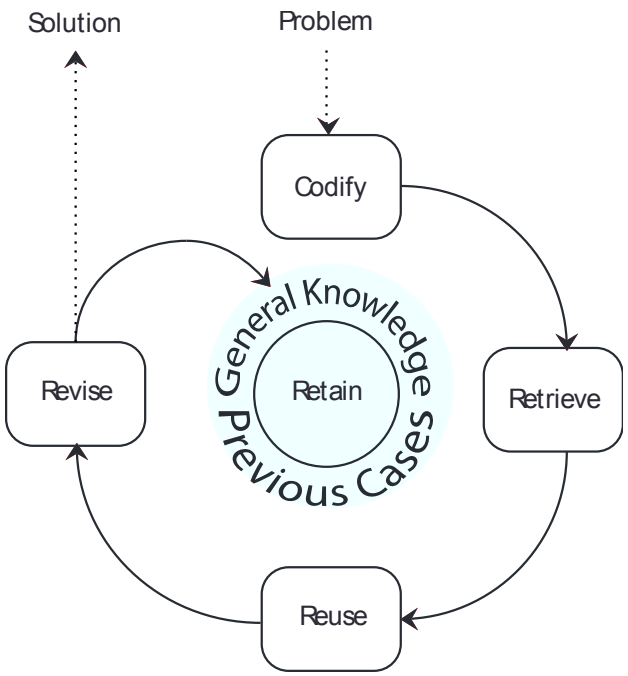

Figure 2.4: Case-based reasoning model

As one variant of a KBS, Case-Based Reasoning (CBR) emerged in the late '70s as a model for problem solving and learning [SA77]. As shown in Figure 2.4, CBR systems work by finding a (generalized) example of a problem within a Knowledge Base (KB) and suggest a solution based on this information. The steps performed are: (1) to codify the problem into a set of distinct (and comparable) features; (2) to retrieve from the existing cases the one that matches the closest; (3) to reuse the found case with the new information and suggest a solution; (4) to revise the case through testing or through an expert and (5) to retain the gained knowledge as an example for further cases. Additional knowledge is used to model domain specific ground rules, dependencies, etc. The use of similarity measurements is another key principle of CBR systems. The objective is to rank cases in decreasing order of similarity (the nearest k cases). Choosing an appropriate value for k is an ongoing research issue, which is further discussed in [KCS01]. Once

similar cases have been identified, they can be adapted to solve the problem case either by rules, a human expert or by a simple statistical procedure such as a weighted mean. In the latter case, the system is often referred to as using the k-nearest neighbor (k-NN) technique.

A basic similarity measurement often used in CBR is defined in [Aha91], where P is the set of n features, C1 and C2 are cases with numerical features:

$$SIM(C_1, C_2, P) = \frac{1}{\sqrt{\sum_{j \in P}(C_{1j} - C_{2j})^2}}$$

Examples of CBR systems in SE range from process effort [MVP92] and cost estimation [Kem87], to specification [Mai91] and component reuse [OHPDB92]. More recently, CBR systems have also been used with EFs [AN04]. An overview on relevant approaches for Knowledge-Based Software Engineering (KBSE) is given in [Cha01]. A fundamental limitation of existing CBR systems is their need for codification of knowledge into a feature vector which makes the resulting knowledge representation non-interchangeable. Furthermore, this representation does not support reasoning within existing knowledge.

Other types of KBS are referred to as rule-based reasoning systems [HK87]. In the literature the term expert systems and rule-based KBS are used interchangeably, although they differ in their objectives. The term expert system focuses on who creates information in contrast to the term rule-based KBS which focus on describing the methodology used to store and reason on the modeled data. Consequently, an expert system could also be a CBR - KBS by definition.

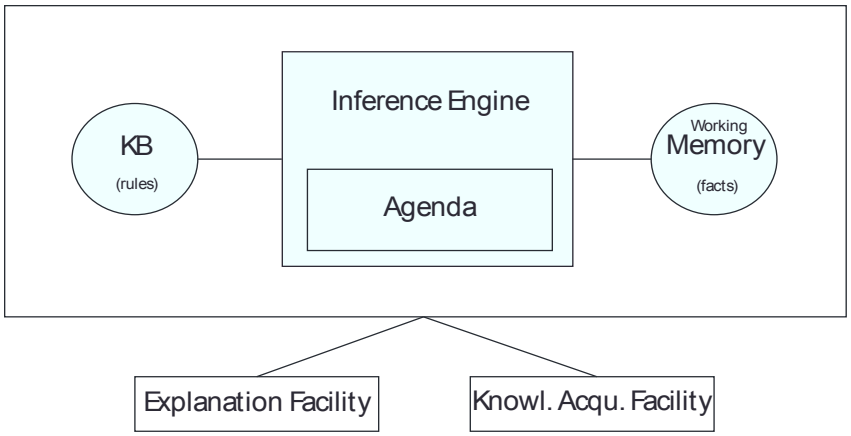

Figure 2.5: Rule-based reasoning model

Figure 2.5 [BCM+03] shows the architecture of a typical rule-based KBS. Premise-consequence rules (also referred to as condition-action rules) are stored in a knowledge base (*KB*). The *Working Memory* holds the initial facts from the *KB* and generated facts from the *Inference Engine*. In order to generate facts, the condition-part of the rules is matched against facts stored in the *Working Memory*. Rules with satisfied conditions are active rules and are placed on the *Agenda*. Among all active rules, one is selected (based on predefined priorities) as a next rule for execution ("firing") and the consequence of the rule is added as a new fact to the *Working Memory*. Most systems also contain an *Explanation Facility* that provides the details (reasoning steps) on how facts have been created in order to "explain" (to a user) how a solution was found. The cycle ends when no more rules are on the *Agenda*.

Different methods of reasoning and rule activation within rule-based KBS[Jac99] exist:

Forward-Chaining: In this method the inference engine starts with the available facts and uses the rules to conclude more facts until a conclusion is reached. Because the

data available determines which inference rules are used, this method is also called data driven. The method is often used for real-time expert systems in monitoring and control. Examples of systems using forward-chaining are CLIPS and OPS5.

Backward-Chaining: Starting from a hypothesis (query), supporting rules and facts are sought until all parts of the antecedent of the hypothesis are satisfied. Because the list of goals determines which rules are selected and used, this method is also called goal driven. The method is often used for diagnostic and consultation systems. An example of a system using backward-chaining is EMYCIN.

A main focus in AI lies on the representation of information and reasoning within it [RN03]. In order to solve problems, formalized knowledge about the domain of discourse, such as objects, their properties, and relations between them, is required. Although early small KBS showed promising results, many large (and commercial) implementations failed due to missing proper development (design) and maintenance processes, a situation similar to the SE "software crisis" in 1968. This ultimately lead to the establishment of Knowledge Engineering as a separate discipline.

2.3 Knowledge Engineering

Knowledge Engineering (KE) is closely related to SE and deals with the development of expert-systems and knowledge repositories as well as knowledge representation techniques and their methodologies. It has the goal to bring the process of constructing KBS to a well-defined "engineering discipline". A sub-discipline of KE, focusing on the development of specific knowledge repositories, is ontology engineering [GPFLC04]. Ontologies have their origin in philosophy, where they correspond to a theory about the

nature of existence and the categories of things that exist. In computer science, ontologies are an important part of knowledge modeling and sharing, by acting as a common language. This becomes imminent in the context of semantic web technologies, of which ontologies are a fundamental building block.

> "A conceptualization is an abstract, simplified view of the world that we wish to represent for some purpose. [...] An ontology is an explicit specification of a conceptualization." [Gru93]

Ontologies have been widely used in computer science in order to formally define domains of discourse and can be described as a conceptualization of explicit information [BCM+03]. They consist of concepts (also often referred as classes) and their properties and relations between them. Ontologies emphasize engineering quality aspects such as communication and interoperability [UG96]. In contrast to databases, ontologies allow to define the semantics associated with a described domain, allowing for reasoning services to automatically infer knowledge out of explicitly stated facts. Additionally, their formal representation is easy extendable and interchangeable [Gru93].

Implementations of ontologies can vary in several ways, such as their degree of quality, formality, reusability or reasoning capabilities. In general one can distinguish reference ontologies and application ontologies [Obe04]. Reference ontologies only act as a specification of a domain and provide a taxonomy as well as unique identifiers to specify knowledge unambiguously. Application ontologies, on the other hand, serve an application-specific purpose and can use reasoning services to infer knowledge about stated facts (classify instances, check consistency of facts, or answer queries). A more detailed categorization of ontologies is defined in [Gua97]:

Domain ontology: A domain ontology acts as a specification of the world in a specific context (the domain). Its main purpose is the disambiguation of terms and the definition of relations between them. For example, a "car ontology" might have a different meaning in the domain of Formula-1 race cars and regular street cars, even though some properties and terms might overlap.

Application ontology: Application ontologies capture a particular problem (and solution). The reusability of application ontologies is generally low as they are built with a specific purpose in mind. Application ontologies can be built on top of a domain ontology.

Upper ontology: An upper (sometimes also called "generic") ontology captures general concepts that may appear in application and domain ontologies across many fields. Such concepts are independent of a usage scenario and intended to be shared in a community of knowledge engineers as a common basis (e.g. the concept "event"). An example for an implementation of such an ontology is the PROTo ONtology[1] (a basic subsumption hierarchy for indexing and annotation).

Core ontology: As an intermediate between upper and domain ontology, generic reusable values for a set of domains can be expressed in a core ontology. The distinction between what constitutes a core, upper or domain ontology thereby is fluent. An example for a core ontology is the SIOC[2] (Semantically-Interlinked Online Communities) ontology.

[1] http://proton.semanticweb.org/
[2] http://sioc-project.org/

A further classification of ontologies can be made according to their expressiveness: *Heavyweight ontologies* are very detailed and extensively axiomatized. In many cases they are used for reasoning activities and carry the application specific knowledge needed for specific tasks. An example for such an ontology is the SNOMED ontology [SPSW01] which describes clinical terms. *Lightweight ontologies*, on the other hand, are simple taxonomies with only primitive structural relations. Their value lies in the agreement on a common terminology. ProdLight [Hep07], an ontology for production descriptions, is an example for such a lightweight ontology design.

Given the increasing popularity of ontologies, a question arises as to what extent ontologies differ from expert systems. Due to their different abstraction level, ontologies and expert systems cannot directly be compared. Instead ontologies should be compared with the knowledge representation part of an expert system. Welty et al. [Wel00] state that a main difference between expert systems and ontologies is that simple expert systems are usually toys while simple ontologies may already be extremely useful. He explains this by the focus of ontologies as means to share and query knowledge. A second difference can be seen in the available tool support. Expert systems usually need to be built by experts, while ontologies are a relatively comprehensible technology with plenty of tools available (also thanks to their standardization/specification). Also, in contrast to expert system, ontologies can be generated semi-automatically. A representational difference between ontologies and expert systems is that, the former use popular formats such as XML, whereas knowledge systems are relying on their proprietary formats.

2.4 Knowledge Modeling Technologies

For machines to understand and reason about knowledge, it needs to be represented in a well-defined language. The semantic web is an initiative of the W3C[3] that has the goal to represent knowledge on the Internet in such a language. Due to the emergence of the semantic web vision[BLHL02] ontologies have been attracting much attention recently. Along with this vision, new technologies and tools have been developed for ontology representation, machine-processing, and ontology sharing. There exist, however, a large number of knowledge representation languages that differ mainly in their semantics, syntax and expressivity. In this thesis, the Web Ontology Language (OWL) has be selected as the knowledge representation language as it is quickly becoming the standard for KE due to its use in the emerging area of the semantic web. Further, it is superior to other representation languages, as it is layered on top of existing well-established technologies [HPPSH05]. It uses Extensible Markup Language (XML) related standards like Uniform Resource Identifier (URI) and Unicode for its data layer, which is followed by a general resource description layer called Resource Description Framework (RDF) that can represent graphs (acting as the basic assertion language). This is the underlying model for the Resource Description Framework Schema (RDFS) and eventually OWL which both add a specific meaning to certain URIs and elements (e.g. `rdfs:subClass`). The different layers are shown in Figure 2.6.

A subset of OWL is based on Description Logic (DL), a formal knowledge representation language that is a decidable fragment of first order logic [BCM+03]. This section therefore starts with an introduction into DL before detailing its relation to OWL.

[3] http://www.w3.org/standards/semanticweb/

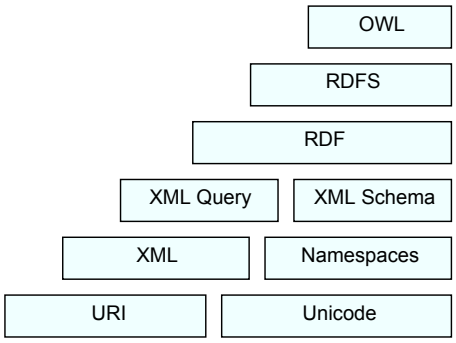

Figure 2.6: Layers of the semantic web

2.4.1 Description Logics

In order to define a domain of discourse, a knowledge representation formalism has to be used. Logic-based approaches use predicate calculus to define facts (in comparison to non-logic based approaches that model knowledge in ad-hoc data structures). Description Logic (DL) is a logic-based formalism that can describe a domain formally. A further emphasis within DL lies on reasoning services. The "is-a" relationship (subsumption) is a fundamental building block of DLs that allows to categorize a domain into sub and super-concepts, creating a taxonomy.

There exist different implementations of DLs with different syntax. Attributive Language (AL) represents a minimal set of constructs on which most other DLs are built (C and D denote concept descriptions):

	C	atomic concept	$\neg C$	atomic negation (not)
$C, D \rightarrow$	$\top$	universal concept (top)	$C \sqcap D$	intersection
	$\bot$	bottom concept	$\forall R.C$	value restriction (for all)

Further extensions allow more expressive DLs such as:

$$
\begin{array}{rll}
 & C \sqcup D & \text{union} \\
C, D \rightarrow & \exists R.C & \text{existential quantification (exists)} \\
 & \neg C & \text{concept negation (not)}
\end{array}
$$

Or introduce numerical restrictions such as:

$$
\begin{array}{rll}
C, D \rightarrow & \geq_n R & \text{numerical restriction (at least)} \\
 & \leq_n R & \text{numerical restriction (at most)}
\end{array}
$$

The formal semantics of the definitions given above can be expressed through the following as defined in [BCM$^+$03]: Given is an interpretation $\mathcal{I}$ that consists of a non-empty set $\Delta^{\mathcal{I}}$ (the domain of the interpretation) and an interpretation function which assigns to every atomic concept C a set such as $C^{\mathcal{I}} \subset \Delta^{\mathcal{I}}$ and to every atomic role R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The interpretation function can then be extended to concept descriptions by the following inductive definitions:

$$
\begin{aligned}
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} &= \emptyset \\
(\neg C)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(\forall R.C)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} \quad | \quad \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\} \\
(\exists R.\top)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} \quad | \quad \exists b.(a,b) \in R^{\mathcal{I}}\}
\end{aligned}
$$

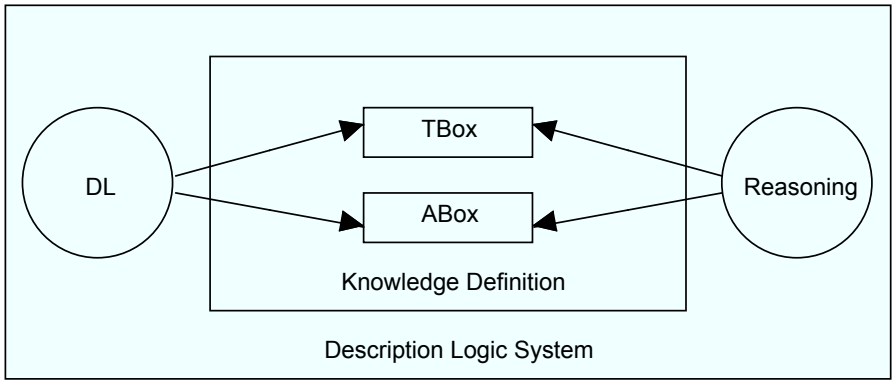

Figure 2.7: DL system architecture

$$
\begin{aligned}
(C \sqcup D)^{\mathcal{I}} &= C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(\exists R.C)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} \quad | \quad \exists b.(a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \\
(\neg C)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}} \\
(\geq_n R)^{\mathcal{I}} &= \left\{ a \in \Delta^{\mathcal{I}} \quad \middle| \quad \left|\{b|(a,b) \in R^{\mathcal{I}}\}\right| \geq n \right\} \\
(\leq_n R)^{\mathcal{I}} &= \left\{ a \in \Delta^{\mathcal{I}} \quad \middle| \quad \left|\{b|(a,b) \in R^{\mathcal{I}}\}\right| \leq n \right\}
\end{aligned}
$$

DL-based knowledge systems allow for implementation of reasoning services. The purpose of reasoning is to explicate knowledge that is stored implicitly in a given knowledge base. Tableau algorithms are a commonly used procedure. Most reasoners thus go beyond the expressiveness of the basic $\mathcal{ALCN}$ Description Logic described before.

DL systems provide the means to set-up a knowledge base and reason about their content. A typical architecture is shown in figure Figure 2.7 [BCM$^+$03]. A knowledge base consists of a TBox $\mathcal{T}$, defining the terminology of a domain (as formalized above), and an ABox $\mathcal{A}$, defining assertions about named individuals (nominals). Individuals

denote a specific state of an ontology. The following definitions for assertions over every individual I in a set $I^{\mathcal{I}} \subset \Delta^{\mathcal{I}}$ can be made:

$$(I)^{\mathcal{I}} = I^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} with |I^{\mathcal{I}}| = 1$$

$$(C(a))^{\mathcal{I}} = a^{\mathcal{I}} \in C^{\mathcal{I}}$$

$$(R(b,c))^{\mathcal{I}} = (b^{\mathcal{I}}, c^{\mathcal{I}}) \in R^{\mathcal{I}}$$

Distinct individual names (e.g. a and b), usually denote distinct objects ($a^{\mathcal{I}} \neq b^{\mathcal{I}}$). $\mathcal{I}$ satisfies an ABox $\mathcal{A}$ with respect to a TBox $\mathcal{T}$ if it is a model of $\mathcal{A}$ and $\mathcal{T}$ [BCM$^+$03].

Typical reasoning tasks on concepts are satisfiability checks, to determine whether a description can have individuals, or subsumption tests, to examine whether one description is more general than another one. Concepts can be organized into a terminology hierarchy according to their generality through the use of subsumption reasoning. Other TBox reasoning includes classification and consistency checks. Basic reasoning for the ABox includes instance checking (i.e., whether a given individual is an instance of a certain concept), tuple retrieval, and instance realization.

2.4.2 Web Ontology Language

The Web Ontology Language (OWL)[4] has been standardized by the World Wide Web Consortium (W3C) and has paved the way for a machine-understandable Internet. More recently, ontologies have found their way into most KE application domains and are now widely accepted as an proven method for knowledge representation.

[4]http://www.w3.org/2004/OWL/

RDFS/OWL2 Constructor	DL Syntax	Semantics
owl:Thing	$\top$	$\Delta^{\mathcal{I}}$
owl:Nothing	$\bot$	$\emptyset$
rdf:type (individual)	$C(I_1)$	$I^{\mathcal{I}} \in C^{\mathcal{I}}$
rdf:resource (relation)	$P(I_1, I_2)$	$(I_1^{\mathcal{I}}, I_2^{\mathcal{I}}) \in P^{\mathcal{I}}$
rdfs:subClassOf	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
rdfs:subPropertyOf	$P_1 \sqsubseteq P_2$	$P_1^{\mathcal{I}} \subseteq P_2^{\mathcal{I}}$
owl:equivalentClass	$C_1 \equiv C_2$	$C_1^{\mathcal{I}} = C_2^{\mathcal{I}}$
owl:equivalentProperty	$P_1 \equiv P_2$	$P_1^{\mathcal{I}} = P_2^{\mathcal{I}}$
owl:complementOf	$\neg C$	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
owl:disjointWith	$C_k \sqsubseteq C_j \equiv \bot$	$C_i^{\mathcal{I}} \subseteq C_j^{\mathcal{I}} = \emptyset, k \neq j$
owl:intersectionOf	$C_1 \sqcap \ldots \sqcap C_2$	$C_1^{\mathcal{I}} \cap \cdots \cap C_2^{\mathcal{I}}$
owl:unionOf	$C_1 \sqcup \ldots \sqcup C_2$	$C_1^{\mathcal{I}} \cup \cdots \cup C_2^{\mathcal{I}}$
owl:oneOf	$\{I_1, \ldots, I_n\}$	$\{I_1\}^I \cup \cdots \cup \{I_n\}^I$
owl:sameAs	$I_1 = \ldots = I_n$	$I_j^{\mathcal{I}} = I_k^{\mathcal{I}}$
owl:differentFrom	$I_1 \neq \ldots \neq I_n$	$I_j^{\mathcal{I}} \neq I_k^{\mathcal{I}}$
owl:TransitiveProperty	$Tra(P)$	$(R^{\mathcal{I}})^+$

Table 2.1: RDFS/OWL2 constructs for concepts translated to DL

OWL terms differ slightly from those used in DLs: The definitions for the terms `owl:class`, `owl:ObjectProperty`, `owl:DatatypeProperty`, as well as the terms `owl:Individual` and `owl:Datatype` correspond to concept, role, concrete role, object and concrete domain in DLs. Table 2.1 shows RDFS and OWL constructs and their DL representation; Table 2.2 lists restriction types for concepts. C, C_1, C_2 are OWL classes, P, P_1, P_2 denote an OWL property and I_1, I_2 are OWL individuals [HPSH03]. The given semantics of the DL syntax is used throughout this thesis.

The Web Ontology Language Version 2 (OWL2)[5] is an extension of OWL developed by the same W3C working group. It enriches OWL with features such as simpler meta-modeling, additional property and qualified cardinality constructors and more flexible datatypes. Examples of OWL2 can be found in the OWL2 primer[6]. The `owl:hasKey`

[5] http://www.w3.org/TR/owl2-overview/
[6] http://www.w3.org/TR/owl2-primer/

RDFS/OWL2 Constructor	DL Syntax	Semantics
owl:someValuesFrom	$\exists P.C$	$\{x \vert \exists_y (x,y) \in P^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
owl:allValuesFrom	$\forall P.C$	$\{x \vert \forall_y (x,y) \in P^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
owl:hasValue	$\exists P.\{I\}$	$\{x \vert (x, I^{\mathcal{I}}) \in P^{\mathcal{I}}\}$
owl:hasSelf	$\exists P.Self$	$\{x \vert \exists_y (x,y) \in P^{\mathcal{I}} \wedge y = x\}$
owl:minCardinality	$\geq n\ P.C$	$\{x \vert \#(y \vert (x,y) \in P^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}) \geq n\}$
owl:maxCardinality	$\leq n\ P.C$	$\{x \vert \#(y \vert (x,y) \in P^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}) \leq n\}$

Table 2.2: OWL2 restrictions translated to DL

is a new feature of OWL2 that allows to specify a set of keys which identify an individual uniquely. It is implemented separately as a rule in most reasoners and cannot be directly translated to DL. For simplicity, it is expressed as $Key(C, u_1, u_2, ...)$ in this thesis, whereby C denotes a concepts and $u_1 ... u_n$ denote a data or object property. Another new functionality in OWL2 is the introduction of a property chain axiom (represented as $\circ$) that allows the definitions of object property chains which are more expressive than simple transitivity (e.g. $hasSister \circ hasChild \rightarrow isUncleOf$).

The original OWL specification is separated into three sub-languages with different degrees of expressiveness:

OWL Lite: OWL Lite supports users which primarily need hierarchy classification and simple constraints. It can be considered a minimal useful subset of language features that are easily implemented by tool developers.

OWL DL: OWL DL, a sub-language of OWL, is based on DL. It supports users who want maximum expressiveness of a description language but still a system that terminates in finite time. OWL DL includes all language constructs specified in OWL but only allows to use them under certain restrictions (e.g., in comparison to OWL Full, a class cannot be an individual). The resulting sub-language is still

relatively easy to implement for tool developers.

OWL Full: OWL Full allows the unrestricted use of RDF constructs and is undecidable. Limited support of tool developers regarding OWL Full exists.

OWL2 is separated into different tractable language profiles[7] that are better suited for the implementation of reasoners with different runtime characteristics (a main criticism of the original OWL sub-languages):

OWL EL: The profile has been created in response to the development of very large ontologies (used so far mainly in the medical domain) that do not need the full expressivity of OWL and benefit from class satisfiability and subsumption checking in polynomial time. It is based on $\mathcal{EL}$++ that allows class intersections, existential quantifications, property chains, keys and transitivity of object properties. However, universal quantification, negation and disjunctive class descriptions as well as more advanced properties (such as symmetric or inverse object properties), are out of the scope of the profile. Reasoning in $\mathcal{EL}$++ is known to scale well and has in particular been shown to be distributable to multiple machines [MM10] [LD09].

OWL QL: The QL profile has been developed to allow for conjunctive query answering using traditional database systems which can be performed in logarithmic time. The profile is more restrictive than OWL EL and does not allow existential quantifications, keys or transitive object properties. However, symmetric, inverse and reflexive object properties are allowed. Reasoning in OWL QL remains in the polynomial time complexity class and the profile is therefore only beneficial to very large ABox datasets with a static TBox.

[7] http://www.w3.org/TR/owl2-profiles/

OWL RL: The profile is a subset of OWL that can be expressed using rule languages. It therefore can be reasoned about using rule-based reasoners. It is more restrictive than OWL QL but guarantees ontology consistency, class expression satisfiability, subsumption, instance checking, and conjunctive query answering in polynomial time.

Two interesting aspects of ontologies are the Unique Name Assumption (UNA) and Open World Assumption (OWA):

UNA: While two different object names normally denote tow different objects, the same does not hold true for ontologies. Stating that $R(a, b)$ holds and $R(c, b)$ holds with the restriction that b is only related to one other object through R will not result in the ontology being inconsistent but rather a and c denoting the same object.

OWA: Unspecified knowledge is not assumed to not exist (be "false") as in databases or other logic systems, but rather treated as *unknown*. This prevents a reasoner from making decisions that might not be correct (e.g. just because the information that "Paul and Paulina have a child" is not specified does not mean it is not true) and therefore makes OWL reasoning ideal for the WWW.

As a knowledge representation language, OWL has already been applied in many applications within the SE domain, such as model-driven software development (e.g. [TPO+06]), reverse engineering tool integration (e.g. [JC05]) and component reuse (e.g. [HKST06]). There exists relevant work on conceptualizing the SE domain to support teaching of SE, e.g., [ABH+00]. Petrenco et al. [PPRB07] used open-source software systems in teaching software evolution. Falbo et al. [FNM+03] reported on the shared

conceptualization for integrated tool development, and Deridder et al. [DWL00] have used ontologies for linking artifacts at several phases of the development process. The SWEBOK project[8] applies ontologies in SE to provide pointers to relevant literature on each of its concepts. Current web-based learning approaches [Pol03] focus on reusability in their content design. Wongthongtham et al. [WCDS05] [AWWH08], introduced a SE ontology for the collaborative nature of SE. Ankolekar et al. [ASH+06] modeled bugs and software components using an ontology. In [DE05] a software design patterns OWL ontology that supports the identification of design pattern in source code is presented. [Wel97] introduced a system called Code-Based Management Systems (CBMS) which uses a representation of source-code to detect programming side effects (e.g. erroneously changed global variables). Common to all of these approaches is that their main intend to support, in one form or another, the conceptualization of knowledge, mainly by standardizing the terminology to support knowledge sharing based on a common understanding. These approaches typically fall short on adopting and formalizing a process model that supports connecting knowledge resources. The presented models do not explicitly consider the advantages of reasoning services and lack a concrete "added value" in comparison to other forms of knowledge representation.

Software development is a knowledge-intensive activity [Sel92] which has lead to different implementations to support the development process. Such systems are often referred to as KBSE tools. LaSSIE [PRPB91] is such a tool that was introduced to support the development of the Bell telephone system. Studies have shown that LaSSIE could reduce time and costs in building software systems but was also cumbersome to

[8] http://www.computer.org/portal/web/swebok/

maintain due to the manual creation of relations within the domain model. In addition to KBSE tools, there exist further traces of knowledge modeling in SE. In [WF99] an ontology that can capture an architectural layer of a system is presented. [M̈96] describes how object oriented concepts are modeled as a functional layer for DLs and Berardi et al. [BCG01] uses DLs to perform consistency checks on formalized Unified Modelling Language (UML) models. However, no methodology for producing OWL ontologies is given in the mentioned approaches. Instead, ontologies are created ad-hoc and based on "intuition". This lack of existing processes in developing application specific ontologies has motivated the development of the SE-ONTO methodology and the SE-ADVISOR application model, which are described later in this thesis.

Chapter 3

Methodology

The main objective of this thesis is to investigate to what extent the design of ontologies can be incorporated in modern Software Engineering processes and how developers can leverage ontologies and reasoning services in order to build ontology-driven applications. In order to address this objective, a methodology, which incorporates ontology design into an agile software development process (namely SCRUM), is presented in this chapter. The so-called SE-ONTO methodology has a strong focus on designing ontologies together with an application in order to guide the ontology design and ensure the goal of an "ontology-driven" application, in which some responsibility is handled by an ontology and reasoning services, is met. In contrast to other approaches, responsibilities and reasoning goals are constantly monitored throughout the design.

The outline of the chapter is as follows: There are several aspects to what constitutes good ontology design which are discussed in Section 3.2. Section 3.3 reviews existing methodologies in the Knowledge Engineering community and compares them to the proposed methodology. The novel SE-ONTO methodology is presented in Section 3.4,

whereby details of the incremental requirement analysis are presented in Section 3.4.1 and the iterative development/formalization of ontologies is explained in Section 3.4.2.

3.1 Term Disambiguation

The terms methodology, process, activity and task are used vaguely throughout the literature. In this thesis, a composite relationship between the terms is assumed. This is in accordance with the definition used in the IEEE glossary of Software Engineering terminology [IEE90] and IEEE Software Project Management guidelines [IEE98] that state:

> **Methodology:** "A set of ordered process steps and techniques that describe the creation of a service by certain quality criteria."
>
> **Process:** "A sequence of steps performed for a given purpose."
>
> **Activity:** "A collection of work tasks."
>
> **Task:** "The smallest unit of work."

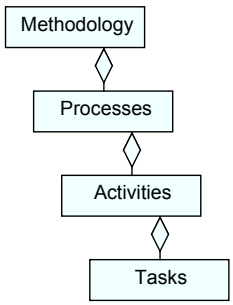

Figure 3.1: Methodology related term disambiguation

3.2 Ontology Design Quality

In SE there exist many definitions for quality. The Institute of Electrical and Electronics Engineers (IEEE) defines quality as "the degree to which a system, component or process meets specified requirements" [IEE90]. The International Organization for Standardization (ISO) provides the definition as "the degree to which a set of inherent characteristics fulfills a need or expectation that is stated, general implied or obligatory" [Hoy01]. Several generic quality models have been introduced in the past (e.g. [ISO01]). Nevertheless, no agreement exists on what constitutes good design quality within the research community. Some common properties are:

Maintainability: The ease with which a design can be modified or adapted to a changed environment.

Extensibility: The ability to add further/supplementary properties to the design.

Portability/Reusability: The ability of the design to represent different model implementations.

Integrability: The ability to combine the design with another.

Testability: The ease with which the design can be tested.

Similarly, the evaluation of design quality in the ontology community is a widely discussed subject. One of the fundamental rules in ontology design is that "there is no one correct way to model a domain, there are always viable alternatives. The best solution almost always depends on the application that you have in mind and the extensions that you anticipate" [NM01]. While design quality criteria can provide helpful insights into

different aspects of ontology design, a compromise between expressiveness and performance is nearly always required.

An important aspect of any ontology design methodology is the quality of the produced ontology. In order to evaluate what constitutes a good ontology, several aspects must be taken into consideration. In general, the evaluation of an ontology can be grouped into three categories [GP94]:

- Ontology Validation - All ontology definitions must be necessary and sufficient to represent the ontology's purpose.
- Ontology Verification - The process of ensuring that the ontology specification and requirements function correctly.
- Ontology Assessment - The evaluation of quality aspects from a user-perspective such as usability, comprehensibility, generality, etc.

Gruber et al. [Gru95] introduced one of the first extensive criteria catalogs and design guidelines. Their catalog consists of the following items:

Clarity: The ontology must use a clear description of any used terminology (in natural language).

Coherence: The ontology must pass any consistency tests by reasoners. Additionally, the ontology (and its inferred axioms) should be checked against examples given in natural language in the documentation for contradicting statements.

Extensibility: The ontology must be designed for evolution. Adding and removing of terms should have as little side effects as possible on the ontology.

Minimal encoding bias: The ontology should not make any assumptions of how it is being used and only use necessary conditions in its definition.

Minimal ontological commitment: The ontology should make minimal assumptions about the world in regards to the modeled domain.

While Gruber states that there is a trade-off between ontological commitment and clarity as well as extensibility, his criteria are solely motivated by seeing ontologies as a medium to transfer knowledge. In particular, the minimal encoding bias has led to the current state of ontology design which is not motivated by use-cases but maximal reusability. This has led to a race for the most general ontologies that only provide concepts without questionable clarity.

Further criteria for the evaluation of ontologies are defined by Gomez-Perez et al. [GP99]. The inspection of an ontology taxonomy focuses mainly on the detection of static errors that are partially discoverable by reasoners:

Consistency: No knowledge that is contradictory to the ontology specification must be inferable. An example of a consistency problem is a "Partition Error".

$$\begin{aligned} \text{paritionError} \leftrightarrow \quad & \exists C_1, C_2, C_3 \in \mathcal{C} \\ & \texttt{subClassOf}(C_1, C_2) \wedge \\ & \texttt{subClassOf}(C_1, C_3) \wedge \\ & \texttt{disjointWith}(C_2, C_3) \end{aligned}$$

Completeness: The ontology does not lack any information necessary for its function-

ality. Examples of criteria for incomplete ontologies are “Incomplete Disjoints”.

$$\begin{aligned}\text{incompleteDisjoints} \leftrightarrow\ & \exists C_1 \in \mathcal{C}\ I_1, I_2, I_3 \in \mathcal{I} \\ & \texttt{instanceOf}(I_1, C_1) \wedge \\ & \texttt{instanceOf}(I_2, C_1) \wedge \\ & \texttt{instanceOf}(I_3, C_1) \wedge \\ & \texttt{disjointWith}(I_1, I_2) \wedge \\ & \neg\texttt{disjointWith}(I_2, I_3)\end{aligned}$$

Conciseness: No unnecessary (in regards to the ontology’s responsibilities) knowledge is specified in the ontology or can be inferred from it. Redundancies, such as the “Indirect Subclass Repetition”, are a violation of this criterion.

Baumeister et al. [BS05] introduced further criteria to check the structure of an ontology outside a reasoner.

3.3 Ontology Design Methodologies

Building a good ontology is a challenging task [SS09]. Over recent years, many ontologies have surfaced with only very few being actively used. The semantic web community has been divided into those who advocate ontologies as a means to exchange information in a standardized way (as linked data), and those who see ontologies as the underlying framework for information systems and reasoners. Not surprisingly, this has also led to different standards for ontology design. Many of the existing ontology design methodologies focus solely on the knowledge sharing aspect of ontologies and only consider reasoners for simple consistency checks. Most methodologies are detailed about the

definition of concepts, relations and attributes but fail to address the usability aspect of ontologies.

In ontology design, criteria such as extensibility and coherence are of high importance [Gru95]. However, design details always depend on the intended use of the ontology. Generic (upper) ontologies focus on a minimal ontological commitment; the reusability quality criterion is of most importance. Application ontologies, on the other hand, have a strong ontological commitment and are tailored to a certain use-case. Most methodologies focus on domain ontologies, which have a strong focus on reusability but trade some ontological commitment and extensibility to describe certain aspects of a domain in detail.

The following sections outline a number of methodologies to create ontologies. The methodologies are ordered by their publication date and listed under their primary author.

3.3.1 Gruninger et al.

The Gruninger and Fox methodology [GF94] is based on a motivating scenario that illustrates the use of an ontology. The scope of the ontology is informally defined. In a preliminary step, ontologies that could be reused are identified and evaluated. Next, informal competency questions (in natural language) are defined that express the requirements that the ontology needs to meet. The formalized ontology must represent all these questions using its terminology and provide their answers through axioms and definitions. The final questions are usually simple questions (e.g. starting with “Who”, “What”, “Why”). These are first used to specify the terminology vocabulary (its concepts, relations and axioms) and then serve as the basis for concepts in first-order logic themselves. An ontology is considered complete when it is able to answer every formal

competency question.

The methodology consists of the following individual steps (in order):

1. Identify and evaluate existing ontologies for reuse
2. Gather informal competency questions
3. Specifying the ontology terminology formally
4. Specifying the competency questions formally using the ontology terminology
5. Verification of completeness

3.3.2 Uschold et al.

The design of an ontology according to the Uschold and King ("skeletal model") methodology [UK95] starts with the identification of the ontology scope and purpose (range of intended users and example scenario). The authors suggest different strategies be used to identify concepts and relations: bottom up (from the most specific), top down (from the most general) and middle out (combined). Unambiguous textual definitions are stored in a concept dictionary. In a further step, the captured definitions are "coded" into an ontology language (such as OWL or Prolog). Existing ontologies can be considered during this step. The methodology concludes with an evaluation phase which is similar to the one proposed by Gruninger and Fox (competency questions). All assumptions made during the development of the ontology must be documented.

The methodology consists of the following individual steps (in order):

1. Identifying purpose and scope

2. Ontology building

    - Capturing concepts and relations informally

    - (En)Coding of captured knowledge using the ontology terminology

    - Integrating existing ontologies

3. Evaluating the ontology

4. Documenting the ontology

3.3.3 Fernandez et al.

The so-called *METHONTOLOGY* of Fernandez, et al. [FLGPSS99] starts, similar to other approaches, with the specification of the intended purpose, scope and level of formality. In a knowledge acquisition stage, (un-)structured interviews with experts and text analysis are carried out. A top-down knowledge specialization is suggested. Next, existing ontologies have to be checked for reuse and are evaluated from a knowledge representation point of view. In the conceptualization activity, a glossary of terms (concepts, relations, instances, attributes, ...) is constructed which is then used to build a concept taxonomy and define a binary relations diagram. The ad hoc binary relations, instances and class attributes are then incrementally detailed and organized into tables. Once all elements have been defined, formal axioms and rules are added to infer values. The ontology is formalized in an ontology language. Ontology evaluation concludes the methodology whereby the ontology, its associated software environment and the documentation is analyzed with respect to a frame of reference. “The frame of reference may be requirements specifications, competency questions, and/or the real world” [GP94].

The methodology consists of the following individual steps (in order):

1. Knowledge acquisition

2. Ontology reuse

3. Conceptualization activity

    - Build a glossary of terms

    - Create concept taxonomies and establish ad hoc binary relations

    - Incrementally detail binary relations, instances and class attributes

    - Describe formal axioms and rules

    - Add instances and infer values

4. Ontology formalization

5. Ontology evaluation

3.3.4 Swartout et al.

In the *SENSUS* methodology by Swartout, Knight, Russ and Rey [SKRR97], an ontology is built by matching seed terms against a generic high-level ontology of more than 50000 concepts (inspired by Cyc [GL90] and WORDNET [Fel98]). This abstract ontology of synonyms and related concepts is linked by hand to seed terms from the modeled domain. The selection of seed terms (that should be relevant to the domain) is also carried out manually. Subtrees/graphs in the *SENSUS* ontology, which have many links, are completely carried over to the target ontology.

The methodology consists of the following individual steps (in order):

1. Select relevant seed terms in domain

2. Match seed term to *SENSUS* ontology

3. Carry over parts from *SENSUS* ontology to target domain ontology

3.3.5 Sure et al.

The aim of the *On-To-Knowledge* methodology by Sure, Schnurr, Studer and Staab [SSS00] is to further develop Ushold's and Fernandez's methodologies into a full ontology life-cycle model. Best practices, such as competency questionnaires and the need for ontology evaluation, are taken over from the two models and are refined with examples. In a "kickoff" phase, the goal, domain and scope of an ontology are identified. Sources of knowledge (e.g. persons) and applications to be supported by the ontology must also be noted. A list of users, usage scenarios and competency questions concludes the phase. In a refinement phase, a seed taxonomy is detailed until a final ontology design is reached. The ontology is formalized and assessed in an evaluation phase (against a requirements specification document and the target application environment). The maintenance preparation phase documents the ontology for future changes.

The methodology consists of the following individual steps (in order):

1. Identification of the goal, domain, scope, sources of knowledge, applications, usage scenarios and competency questions

2. Refinement of a baseline taxonomy into an ontology that can be formalized

3. Evaluation against requirements and the target application

4. Maintenance preparations

3.3.6 Hristozova et al.

The so-called *EXPLODE* methodology of Hristozova and Sterling [HS03] collects requirements through competency questions and defines system constraints (depending on the use and application of the ontology). In a planning stage, the scope and proposed concepts and relations are detailed and functional and quality requirements are both formalized and prioritized as competency questions. Starting from a baseline (the "simplest possible ontology" capturing the architectural outline and core competency questions of the application), the ontology is refined to address additional questions and constraints. The methodology concludes with an acceptance testing phase where all constraints and competency questions are used to verify the system.

The methodology consists of the following individual steps (in order):

1. Requirements analysis and planning (through competency questions and system constraints)
2. Baseline implementation of core competency questions
3. Refinement (addressing more questions and system constraints)
4. Acceptance testing

3.3.7 NeOn Project

In the "eXtreme Design" (*XD*) methodology [SFBD+09], a test-driven approach to ontology design is introduced that borrows some practices from Extreme Programming (XP). The methodology defines the use of ontology design patterns as "solutions to typical modeling problems". Selected requirements from a specification document are trans-

formed into use cases. Simple sentences from those requirements are then transformed into competency questions and further into queries for unit tests. Next, content patterns (an ontology design pattern type that addresses some domain specific problems) are matched against the complete competency question. This is repeated until all competency questions are covered. Once the ontology is populated with instances, unit tests queries are run.

The methodology consists of the following individual steps (in order):

1. Use cases are extracted from the specification
2. Competency questions (queries) and unit tests are created
3. Content patterns are matched against competency questions
4. Population of ontology and execution of queries as unit tests

3.3.8 Summary

Many ontology design methodologies have been proposed over the last few decades involving an ever-increasing level of detail and sophistication. At the same time, a growing trend towards the incorporation of more SE best practices is taking place and software development methodologies have influenced many of the life-cycles for ontology development. However, the disconnect between ontology design and agile software development remains a challenge.

	Gruninger	Uschold	Fernandez	Swartout	Sure	Hristozova	NeOn
Ontology reuse	No	Yes	No	Partially	Yes	No	No
Ontology design patterns	No	No	No	No	No	No	Some
Non-ontological data sources	No	No	No	No	No	No	Partially
Support for reasoning services	No	No	No	No	No	No	No
Level of detail	Low	Low	High	Low	Medium	Low	High
Life-cycle model	No	Yes	Yes	No	Yes	Yes	Yes
Project management guidelines	No	No	No	No	No	No	Partially
Application model	No	No	No	No	No	No	No
Application domains	One	One	Multiple	Multiple	One	One	Multiple
Testing and evaluation	CQ[1]	CQ[1]	CQ[1]	No	RSD[2]	CQ[1] + SC[3]	CQ[1]

Table 3.1: Comparison of ontology design methodologies

[1] Competency Questions

[2] Requirements Specification Document

[3] System Constraints

Table 3.1 shows an overview over the different ontology design methodologies. Notably, Swartout et al. [SKRR97] is the only methodology which does not use competency questions. Competency questions were first introduced by Gruninger et al. in 1994 [GF94] and have since been reused in most methodologies as either a starting point for the development of an ontology or as an evaluation basis. They are abstract enough to capture demands/queries for an ontology without the need to know its final design (concepts, relations, etc.). The process of identifying competency questions, however, is usually carried out with a customer who has little or no knowledge about ontology design and imposes his own knowledge of the processed data on the formulated questions. Competency questions are therefore ill-suited to validate or guide the complete ontology design. Ontologies developed in such a way tend to be 1-1 transformations from entity-relationship diagrams or database schemas, which limit their advantages to those of RDF and linked data. This 1-1 transformation, most of the time, defies the very purpose of developing an ontology and results in customer dissatisfaction.

Although the *XD* methodology of the NeOn project incorporates ontology design patterns, it fails to address the application specific constraints which often dictate concrete classes, relations and axioms and hinder the reuse of generalized design tidbits. As stated by Poveda-Villalon et al. [PVSFGP10]: "[We] realized the difficulty of applying the above mentioned method because of the lack of detailed guidelines in some of the tasks". While design patterns have been successfully applied in other domains, the design pattern categories suggested in the methodology are too narrowly set which has led to many design pattern incarnations with little or no value for ontology application development [Hep05]. This is further explored in Section 4.2 of this thesis.

The NeOn project also addresses non-ontological resources outside of their methodology in "Scenarios for Building Ontology Networks". Nevertheless, their approach is limited to 6 non-ontological resource types (glossaries, dictionaries, lexicons, classification schemas, taxonomies and thesauri) which are easily transferable into standardized ontology schemas. In contrast, the methodology introduced in this thesis does not use such resources as a starting point for ontology development and reusability, but rather as a separate conversion/transformation process, independent from ontology development.

3.4 SE-ONTO Methodology

The semantic web, as a general knowledge modeling approach where ontologies can be exchanged and combined, is often limited by the complexity of ontology design. Most ontology design methodologies cannot be easily incorporated into agile software processes due to this inherent complexity. Additionally, there exists a gap between "how ontologies are designed" and "how ontologies can be useful for application development", that is not filled by existing approaches. Ontology development should be raised to a similar well-established process as software development by promoting the reuse of best practices and existing ontologies. The large number of unused and unfocused ontologies available on the Internet justifies and encourages the development of a novel methodology. Such a methodology must be based on the experiences gained from existing, well-established development methodologies in the traditional SE domain. Furthermore, the methodology should promote the iterative and incremental building of ontologies in modern agile development projects, a property which is often neglected in existing publications. These goals have motivated the development of SE-ONTO and distinguish it

from the methodologies introduced in Section 3.3, which solely focus on the design aspect of mostly generic ontologies. As a main contribution in this thesis, a methodology is introduced with the objective to promote a paradigm shift in ontology design away from "designing broad for every possible usage scenario" to "designing for and with an application".

The SE-ONTO methodology promotes the reuse of best practices from the agile SCRUM [Sch97] software process-skeleton rather than defining a completely new process from scratch. SCRUM is an incremental, iterative approach to software development that has proven itself against traditional waterfall-style development practices. Instead of an extensive upfront requirement analysis and a single development-deployment cycle, SCRUM (and other agile approaches) suggest an iterative and incremental process. Each iteration (called a "sprint" in SCRUM) thereby includes multiple steps and is enriched with proposed activities and suggested artifacts. SCRUM applies particularly to Knowledge Engineering as it is a pure product development process and does not require any code-related activities (for example pair-programming from XP) or restrictive roles and artifacts (such as in RUP). Furthermore, it is widely supported by project management tools and has found large support from both open-source communities as well as companies (e.g. Google). These basic SCRUM principles are applied in SE-ONTO for the Knowledge Engineering process without modification of the process-skeleton. Regarding roles, SE-ONTO suggests that at least one domain expert, depending on the project type and assuming a team size of $7(\pm 2)$, must be part of the SCRUM development team. The domain expert should be familiar with ontologies as he is also responsible for keeping the ontology aligned with the terminology in the customer domain. He assists one or

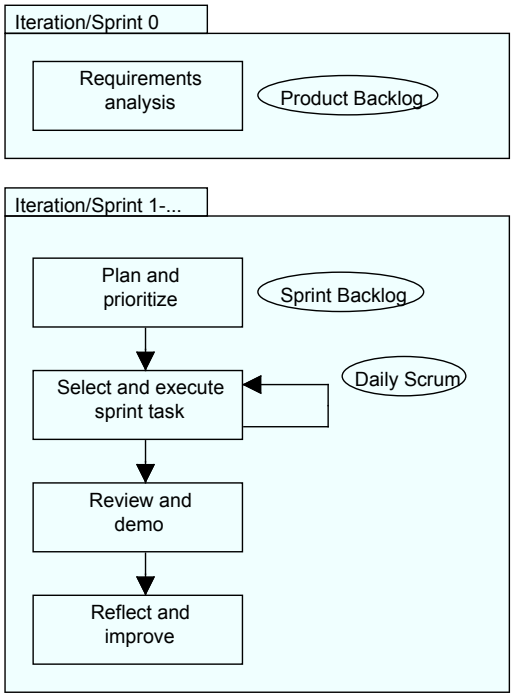

Figure 3.2: SCRUM analysis and sprint

more knowledge engineers as part of the development efforts.

A general outline for a SCRUM sprint with some of its major artifacts/practices is shown in Figure 3.2. A usual time-frame for each iteration/sprint is about one month. Each sprint task from the SCRUM backlog is no longer than 8 hours working time and can be handled by a single person. It has to be noted that requirements analysis will happen throughout the project whenever there is a need to adapt the software. However, the first sprint (iteration 0) intensively analyzes the requirements in order to get a better understanding of the product to be developed. For a more detailed description of SCRUM, the reader is referred to the excellent overviews in [DBL10] and [RJ00].

The main contributions of the SE-ONTO methodology, interwoven into the software development life cycle, are shown in Figure 3.3. The methodology specifies the gathering

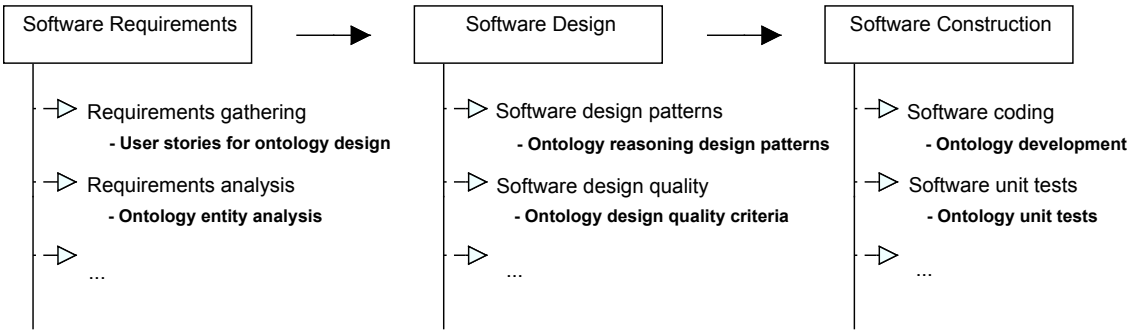

Figure 3.3: Contributions in the software development life cycle

of requirements in regards to ontology design and defines how such requirements can be transformed into a usable specification for incremental ontology design (the "entity analysis" phase). In an "ontology development" phase, the previously specified entities are formalized using a set of ontology design patterns. SE-ONTO makes use of a novel set of ontology reasoning patterns whose goal is to encourage the use of reasoning services during ontology design.

The following sections list the activities and work products (proposed by the SE-ONTO methodology) that can be used by following an agile development process (such as the SCRUM process-skeleton) and that allow for an iterative and incremental ontology design. As mentioned earlier, the methodology focuses on ontology-driven application development and the incorporation of reasoning services.

3.4.1 Ontology Entity Analysis

The main purpose of the analysis phase in SE-ONTO is the collection of requirements from the customer for the ontology design. It is therefore part of the requirement analysis phase of a software project. As suggested by the SCRUM process, user stories are an excellent mechanism to capture a customer's goals and wants. A best practice is to

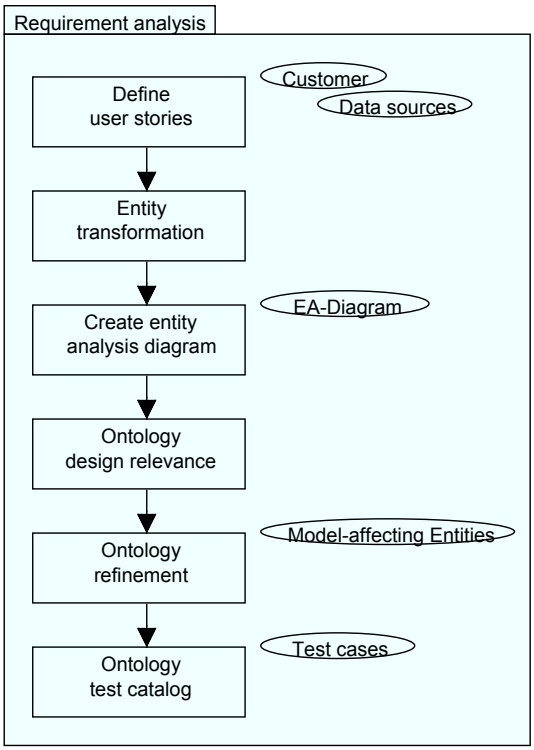

Figure 3.4: Requirements analysis for ontology design

define user stories in the format: “As ...(who)... I want/become/change ...(what)... so that ...(why)”. This captures the important context of the stakeholder (who) as well as a textual description of the reason (why) a certain task needs to be achieved. For existing data sources (such as databases), concepts and relations are often already available. Nevertheless, it is suggested to create user stories with customer involvement to capture the important semantics behind data and discourage both 1-1 mappings and ontologies that are too generic to be useful for a specific application context. This also benefits the verification of existing data structures that have often grown over time and are not necessarily well designed. Figure 3.4 summarizes all steps of the requirements analysis phase.

For user stories to be transformed into an ontology, a knowledge engineer first needs to extract the data model from the description into a less ambiguous representation that shows the inter-connected structure of the data. This also helps to reduce, at an early stage of the development cycle, potential misunderstandings from a customer perspective regarding the application knowledge requirements. In contrast to other methodologies, SE-ONTO does not differentiate between concepts, individuals and properties in the entity analysis. They are instead seen as implementation details which should not be considered in this stage of the development process. The term "entity" is used to talk about an element that will be represented in the data model (either as a concept, individual, ...).

SE-ONTO transforms user stories into entities by analyzing their text (using noun and adjective analysis). The objective of this analysis is to determine for each part of the user story: "What is the most concrete class that describes the part of the sentence?". The "most concrete class" is a descriptive name for a category of things (multiple elements should, at least in theory, be able to be part of/participate in the class) that is as concrete as possible (limits the amount of things that it represents to a minimum, but greater than one). The transformation is carried out manually by inspecting sentences from user stories but can be supported by Natural Language Processing (NLP) tools. For example, the user story "As a retailer, I want red cars to cost more because they sell better" identifies the entities `Retailer`, `RedCar`, `BetterSellingCar` and `CarCost`. Each class can have multiple participants, such as concrete retailers for `Retailer` or concrete cars for `RedCar`. The identified potential candidates are matched against a list of existing entities to prevent synonymous terms from being included and to identify similar entities, such as `BlueCar`. Entities extracted from a user story are placed in an entity analysis

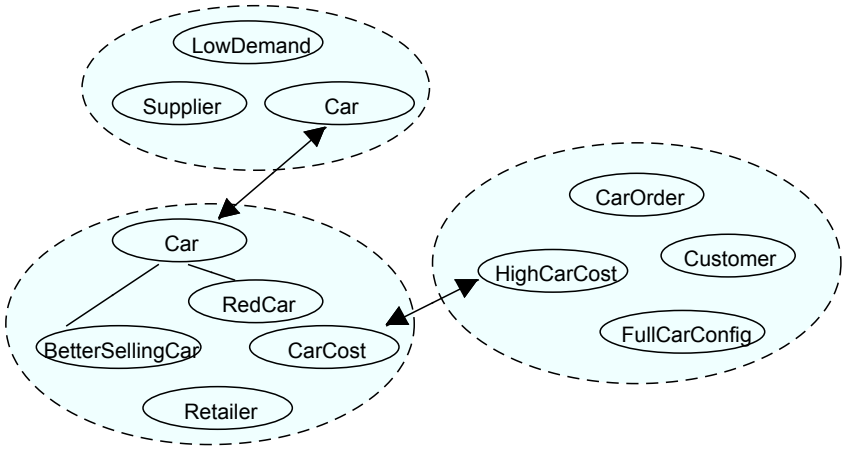

Figure 3.5: Entity analysis diagram

diagram which shows inter-connections. Entities are connected through their user story (dotted circles) and each entity has connections to similar terms (solid lines). The resulting entity analysis diagram is shown in Figure 3.5. Creating an entity analysis diagram is a supporting activity that allows a knowledge engineer to gain a better understanding of the domain and the user stories to be modeled. The entities identified during this stage are, however, not final and can change throughout the ontology development.

For each user story, the knowledge engineer must determine its relevance for the ontology design. A relevant user story is thereby defined as "containing an entity that (potentially) affects the underlying data model". It is important to clarify at this point, whether a problem described in the user story should be modeled by an ontology or better solved elsewhere (e.g. in application logic/program code). The responsibilities of the ontology for each user story must be specified in the task description that is added to the SCRUM backlog artifact (together with the entity analysis diagram).

Ontology design is different from classical database design. Concepts, data and ob-

ject properties (relations) can be easily added at any point. An entity `ItemName` might, for example, have no influence at all on the ontology design but acts only as a storage slot. Consequently, it is not as important to identify all entities in the initial iteration of the ontology design. In order to limit the amount of entities that need to be considered for the ontology design, it can be practical to keep two lists of entities: one primary list for potentially model-affecting entities and a secondary for all other entities that can be added later (if required). Model-affecting entities have the following characteristics:

- They are dependent on or affecting other entities in the data model.
- They do not have a composition relationship to an entity. They are not part of an entity and can exist without it.
- They do not model external information and/or solely relate to Information Retrieval- (IR) like tasks.

Consider the following two user stories (taken from an IBM project [1] and Westboro Systems agile training[2]):

> "As a sales person, when an item is scanned, I want a short description of the item and its price."
>
> "As a customer, when I purchase more than $5000 in goods, I become a preferred customer so that I can receive a 10% discount on all prices."

From the first statement the following entities can be identified: `SalesPerson`, `ScannedItem`, `ItemDescription` and `ItemPrice`. Following the previous definition one can identify `ItemDescription` and `ItemPrice` as non-model-affecting

[1] http://www.ibm.com/developerworks/java/library/j-jmod1023/
[2] http://www.westborosystems.com/2010/02/user-story-estimation/

entities as they are part-of `ScannedItem` and do not affect each other, or are affected by `SalesPerson`. In the second statement the following can be found: `Customer`, `MoreThanFiveThGoods`, `PreferredCustomer` as well as `TenPerDiscount-Price`. This suggests a revision of `ItemPrice` as it now is a model-affecting entity and should be added to the corresponding list. Note that, although SE-ONTO suggests the use of existing terminology, it is sometimes not clear which definition to use. Instead of `TenPerDiscountPrice`, the term `DiscountTen` could for example be used to represent the "10% discount on all prices". Nevertheless, such ambiguities are usually resolved in the later steps of the process.

User stories for Knowledge Engineering are usually larger and more complex than regular user stories. This can be explained by the fact that modeled knowledge usually spans over multiple user stories. With small user stories, it is more important to keep all suggested artifacts (the entity analysis diagram and entity lists), in order to not let any modeling gaps occur. To limit the amount of overlooked entity connections in the breaking down of requirements, SE-ONTO suggests to look at two user stories at a time (pair-wise inspection), to detect possible modeling connections. This also helps with identifying inconsistencies in the usage of the application terminology.

Once the entity analysis diagram is complete and all descriptive entities have been eliminated, SE-ONTO proposes a refinement technique that engages customers in a revision of the requirements by creating multiple examples. Thereby, connected clusters of entities (from the entity analysis diagram) are selected as a group and presented to the customer to express one of the following:

- A hierarchy or equality. Ask the key question: *"Is every ... also a ..."*. E.g. "Is

every `MoreThanFiveThGoods` also a `ScannedItem`?". This also helps to eliminate synonyms.

- A completion or negation. Ask the key question: *"Something not a ... is ..."*. E.g. "Something not a `TenPerDiscountPrice` is?". This helps to identify incomplete/missing or disjoint entities.

- A connection in the form: *"If I would/am ... and/or ... then I should/be ... and/or ..."*. E.g. "If I am a `PreferredCustomer`, then do I get `10PerDiscount`?"

Missing entities are first searched in the list of entities (in case of a missing relation in the entity analysis diagram) or added if not present. Besides clarifying the meaning of individual entities this process might also lead to new findings. In the previous example, a `Customer` might also be a `SalesPerson` who, in that case, might be able to obtain an additional discount. Relevant new findings are added to the task description. As a last step in the entity analysis for ontology design, examples from the requirements revision are included as part of the test catalog in the form of testable statements.

The entity analysis phase captures domain knowledge but cannot ensure that all facts about the domain (captured from the user stories) are complete and correct. It is the task of the knowledge engineer to discover missing gaps or contradicting statements. The SE-ONTO methodology facilitates domain modeling by providing a concrete work product (the entity analysis diagram) and a set of questions and rules that can help identifying modeling problems. However, further modeling problems can arise while formalizing knowledge (ontology development phase) and a refinement of the discovered statements can be necessary.

As a knowledge engineer, it is important to emphasize that ontology design (and Knowledge Engineering in general) is a process of creating information rather than simply retrieving information from a data store. During the requirements analysis phase the following questions should be answered, as they encourage more complicated ontologies:

1. How can a reasoner enrich and/or complete the existing data?

2. How can a reasoner verify the correctness of the existing data?

3. How can a reasoner support or model an application problem?

This concludes the entity analysis phase of the SE-ONTO methodology. The inter-connected user stories (with their part of the entity analysis diagram and list of model-affecting entities) are added to the SCRUM product backlog together with their test-cases and are ready to be transformed/formalized iteratively in the "ontology development" phase. In case the number of inter-connected user stories is large and the resulting workload would exceed the maximum of 8 hours allowed by SCRUM, these large stories need to be broken up. SE-ONTO suggests a cutting point with the least inter-connections between entity groups/user stories. The resulting separation of modeled knowledge can be refactored incrementally.

3.4.2 Ontology Development

As SCRUM is an iterative process, the design, whether it is application or knowledge related, must be executed in an iterative and incremental way. This incremental application development allows one to address ever-changing requirements and adapt an application

to the real needs of a customer. In SE-ONTO, the ontology formalization resembles a programming/development task in terms of the produced output: a "working" ontology. It is therefore also called an "ontology development" task in the SE-ONTO methodology. This is in contrast to database or application design, which merely serves as a supporting by-product of development and is generally discouraged or limited in agile processes (partly due to the overhead associated with design and the uncertainties of future developments). Ontology development is a building block of semantic applications and not a supporting by-product. It is important, however, that the ontology design cannot dictate the architecture of an application but must work together with program code.

In order to support an iterative knowledge-modeling approach that does not design the complete ontology upfront, it is necessary to decompose the knowledge to be modeled. For this reason, the entity analysis (described in the previous section) splits the knowledge to be modeled into manageable parts (clusters in the entity analysis diagram), which can be handled in sprint tasks. Each sprint task for ontology development starts with a task description and an entity analysis diagram that contains a subset of the knowledge to be modeled. This subset represents a number of user stories — an aspect of the application under development. Figure 3.6 shows the steps performed during ontology development.

Ontology design patterns play an important role in the SE-ONTO methodology and individual patterns are described in Chapter 4. SE-ONTO makes use of so-called ontology reasoning design patterns, which are reusable ontology fragments, that are enabled by semantic web reasoners. In particular, structural reasoning patterns (domain-independent architectural-modeling practices for ontologies) are used as refactorable de-

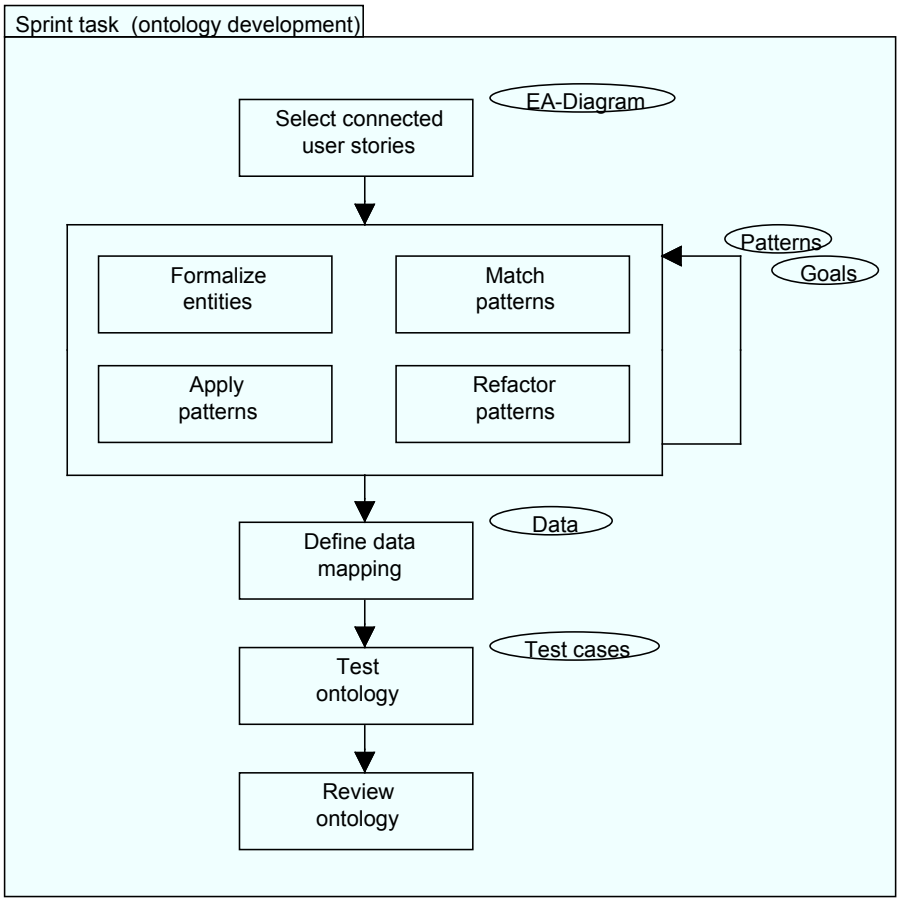

Figure 3.6: Sprint task for ontology development

sign solutions with a clear "solved-problem" description.

As discussed earlier, during the ontology entity analysis phase, a decision has been made whether a user story is model-affecting and should be implemented in the ontology (the ontology responsibility). The knowledge engineer should select (if possible) a reasoning pattern depending on the goal of a user story. There exist different patterns for different goals/responsibilities, such as "identifying the similarity between entities" or "modeling a hierarchy". Once a suitable pattern is found, it is refactored in order to fit into the current domain and the current ontology. Ontology design patterns can also

be helpful *during* the refactoring process. For example, the *Property-Class Commonality* pattern (see Section 4.3) allows of a concept class to be combined with an object property.

In case there is no applicable ontology design pattern, the knowledge engineer must formalize the entities of the user stories manually. SE-ONTO suggests a bottom-up (from the most specific to the most general) process. In a first step, every entity that has not yet been represented in the current ontology is added as an individual. Exceptions to this rule are entities that form a hierarchy — they are represented as concepts. The knowledge engineer then tries to complete the responsibilities of the user story by adding axioms such as object properties (relations) and existential quantifications (restrictions). Ontology design patterns can be applied in the refactoring process to transform the ontology, while the knowledge engineer has to ensure that the selected patterns do not impact performance (or other constraints such as the selected ontology language and profile).

The formalization step is illustrated by revisiting the previous example from the entity analysis phase (assuming no preexisting ontology that otherwise could be used as a starting point):

> "As a customer, when I purchase more than $5000 in goods, I become a preferred customer so that I can receive a 10% discount on all prices."

`Customer` and `PreferredCustomer` form a hierarchy and are created as concepts. `MoreThanFiveThGoods` and `TenPerDiscountPrice` are added as individuals. In the example, the user story is responsible for identifying preferred customers in the ontology and detecting the type of discount that is applied. One could therefore

add the following axioms:

$$PREFERREDCUSTOMER \equiv \exists purchased.\{MoreThanFiveThGoods\}$$

$$PREFERREDCUSTOMER \sqsubseteq \exists receives.\{TenPerDiscountPrice\}$$

Note that the agile principle of modeling dictates only to add what is needed to solve a concrete user story (and not more). One could easily argue for a presentation of `TenPerDiscountPrice` as a concept class that represents multiple reduced prices, however, this is not required by the user story and therefore not implemented. The use of refactoring and structural patterns will result in the transformation of the individual to a concept class if required (e.g. through the *Restriction Generalization* pattern, see Section 4.3).

By applying the SE-ONTO methodology, the resulting ontology design will not be directly linked to the design of the application logic. Ontology design should not be confused with object-oriented design. Instead, the knowledge engineer has to define a mapping between the ontology terminology and application objects/classes. For example, the individual for `TenPerDiscountPrice` is most likely modeled (in application logic/program code) as a class "Discount" which has an "Integer" attribute for the percentage (following object-oriented design principles). A mapping can define one or more individuals/concepts for every object in an application or group objects together. It is also possible to store only the part of application data in the ontology that is relevant to reasoning and use other storage solutions (e.g. a relational database) in the application. Once a mapping is defined, a so-called ontology population process can convert concepts

from the application to the ontology.

As part of the best practices/techniques from XP, continuous integration is encouraged by SCRUM and therefore also by the SE-ONTO methodology. The ontology usually is checked out from a version control system for a specific sprint and can then be modified. Upon check-in, an automatic build of the application using the ontology is triggered. As the entity analysis defines tests for the ontology design, these tests are added to a test repository together with other unit and integration tests. As the ontology is developed together with an application, tests can be implemented in the application logic. The ontology design is committed to the current development tree as often as possible.

Following agile principles, the ontology must be inspected for its design quality before finishing the sprint task. SE-ONTO suggests to use clarity, coherence as well as minimal ontological commitment, as defined by Gruber et al. [Gru95]. The clarity and minimal commitment criteria are thereby implicitly embodied in the methodology that encourages "doing what is necessary (but not more)". Furthermore, coherence should be emphasized through the integration tests over the ontology. In contrast to Gruber et al. [Gru95], extensibility and minimal encoding bias are of less importance. A commitment to a certain use of the ontology is instead seen as beneficial to solving application goals in the SE-ONTO methodology.

As suggested by most agile approaches, the ontology documentation follows the same rules as the design: "as little as possible, as much as needed". Examples which show reasoning results on small parts of the ontology should be kept in a repository. In case of large ontologies (if management of these ontologies becomes extensively difficult), more documentation may be required.

Chapter 4

Ontology Design Patterns

This thesis introduces a library of ontology design patterns as part of the SE-ONTO design methodology. The methodology, as described in Chapter 3, focuses on an incremental process of building ontologies, in which ontology design patterns are repeatedly applied to create and refactor the ontology, until a problem within a domain is solved, while ensuring a satisfactory design quality through continuous testing. As the main contribution in this chapter, Section 4.3 introduces a set of ontology design patterns, namely structural reasoning patterns. Ontology reasoning design patterns focus on the creation of reusable patterns that are enabled by semantic web reasoners. Structural reasoning patterns are domain independent, refactorable (can be adopted to specific purposes), ontology design solutions with a clear "solved-problem" description that focus on the (architectural) structure of the designed ontology.

All patterns are specified in the OWL2 EL profile, a subset of OWL DL with polynomial time complexity that can be used to reason over large amounts of data. While the primary objective of ontology reasoning patterns is to guide the development of an ontol-

ogy and provide best practices and common solutions, a secondary objective is to inform the developer about the introduced impact a pattern has on the developed application. Ontology reasoning patterns are tested fragments for which the runtime complexity and impact on reasoners is known (see Section 6.1). By choosing design patterns based on their performance, one can prevent ontologies which need extensive optimization after the initial design.

This chapter starts with an introduction to the visualization and nomenclature of patterns (Section 4.1) and then reviews the state of the art in ontology design patterns (Section 4.2). In conclusion, a novel type of ontology design pattern called structural reasoning patterns is introduced in Section 4.3.

4.1 Ontology Visualization

There is a need for a standardized visualization technique for ontologies. While UML, as a standardized modeling visualization technique, is well-suited to describe the static structure of concepts and their attributes, it was not developed to show graph-based structures. Although OWL-UML transformation systems are readily available, transformation results produce overloaded UML diagrams for even relatively small ontologies. In a recent survey by Katifori et al. [KHL+07], over 15 currently used methods have been analyzed for their expressiveness in displaying ontologies. It is stated that the "issue of coupling visualization and reasoning has not yet been sufficiently treated in existing literature and very few methods support it". OntoGraph [LN03] is mentioned as one of the only tools supporting reasoners. Nevertheless, only editing problems detected by the reasoner are taken into consideration and are displayed in red.

	Layout	Export	Distinctiveness	Reasoning	License
OntoGraf	Proprietary	Graph	Color, Symbols	No	GPL
OntoViz	Proprietary	Image	Color	No	MPL
OWLViz	Dot	Image	Color (no individ.)	Subclasses	LGPL
OntoTrack	Proprietary	Image	Color	Errors	Proprietary
IsAviz	Dot	Vector	Color, Graphics	No	Apache

Table 4.1: Ontology visualization techniques

In order to develop a representation suitable to display patterns together with related reasoning information, some of the existing tools have been analyzed. Node-link and tree visualizations are the most suitable ontology representations [KHL+07], offering more descriptiveness than indented lists while maintaining ease of navigation compared to, for example, 3D representations. Table 4.1 shows a list of some of the best tools available.

Graphviz[1], and its *Dot* language for graphs together with its automated layout formatter, has seen some support in ontology tools. While the *Dot* language allows for extensive attributes to differentiate between different classes of elements, tools often do not make use of them.

In this thesis, the *Dot* language and layout format is adopted and a set of *Dot* language rules to standardize the formatting of ontology graphs is introduced. This includes explicit rules for the visualization of reasoning results. The following formatting rules for ontologies are represented in *Dot* language and rendered into a vector format by the *Dot* layout scheme:

- Concepts are shown in *CAPITAL* letters surrounded by ellipses. Words are separated by underscores for readability.

- Individuals are shown as *CamelCased* words starting with a capital letter and are

[1] http://www.graphviz.org/

surrounded by boxes. To distinguish multiple similar individuals, numbers are added to the end of the name.

- Relations are also *camelCased* words but start with a non-capital letter. They are rendered in a smaller font size. Regular relationships are rendered as a solid line with an arrow denoting the direction of the relation.

- The special relationship, "individual of", is rendered as a dotted line without a label while the relationship, "has subclass", is rendered as a bold solid line without a label. The "same as" relationship for individuals is rendered as a bold dashed line without a label and direction arrow; the "equivalence" of classes is rendered as a bold solid line without a label and direction arrow.

- Subclasses and individual relationships with the concept "Thing" can be omitted. Any other omissions have to be explicitly stated.

- Results obtained through reasoning services are highlighted through a distinct version of the graph. To establish a sense of what information has been inferred, the original graph is grayed out. Trivial inferences (such as inherited membership through subclass relationships) are omitted.

The *Dot* layout scheme tries to obtain the best possible (in terms of readability) layout for a given *Dot* language file. In order to produce two distinct, but partially identical versions (one with reasoning results and one without) the reasoning results are marked as unconstrained ("constraint=false") and set invisible in the primary graph. With this method, they do not add to the overall layout of the graph and can be easily made visible in the secondary graph displaying the reasoning results. A tool that automatically

annotates *Dot* language files exported from Protege has been created for this purpose. Listing 4.1 shows a simple exported graph with and without reasoning results:

Listing 4.1: *Dot* graph example definitions

```
digraph g {
  "CONCEPT"
  "OTHER_CONCEPT"
  "Individual01" [shape=box]
  "Individual02" [shape=box]

  "CONCEPT" -> "OTHER_CONCEPT" [style=bold]
  "CONCEPT" -> "Individual01" [style=dotted]
  "CONCEPT" -> "Individual02" [style=dotted]
  "OTHER_CONCEPT" -> "Individual01" [constraint=false color=transparent style=dotted]
  "Individual01" -> "Individual02" [fontsize=10.0 label="relationship"]
  "Individual02" -> "Individual01" [color=transparent fontcolor=transparent fontsize=10.0
  label="relationshipInfered"]
}
```

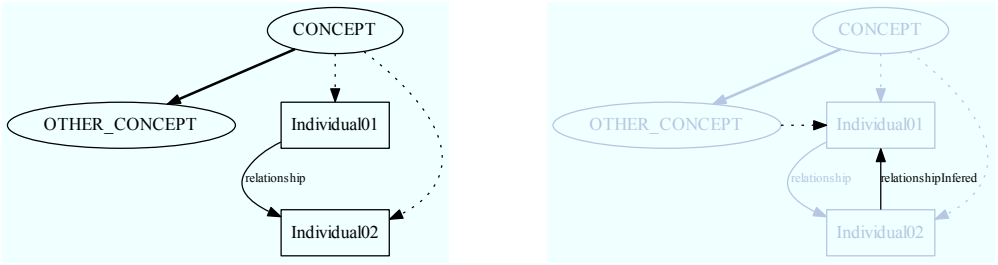

Figure 4.1: Design pattern visualization

While *Dot* visualizations can become hard to read for very large graphs/ontologies, the purpose of the chosen layout scheme is to visualize patterns, which are compact ontology fragments. Patterns tend to capture relatively small re-usable design solutions for which the *Dot* layout scheme can automatically find a good layout. Bigger patterns

might have to be edited manually.

4.2 Design Patterns

The purpose of software design patterns is to capture proven solutions and best practices for reoccurring problems. Examples of such patterns are the Gang of Four (GoF) patterns [GHJV94] (e.g. Composite Pattern). The core elements (axioms) used by software design patterns are classes/interfaces, their relations as well as methods. Design patterns can improve the structure and implementation/maintenance of software, and act as a communication tool to convey implicit and explicit design decisions. There exist structural, creational and behavioral patterns in software design. Generally, patterns are known to provide reuse, guidance and communication benefits. Design patterns have also been applied in other domains such as User Interface (UI) design and E-learning. According to [SFdCB+08] the practice of using design patterns for ontologies is not widespread because of little research in the area and the associated lack of education and pattern repositories.

Ontology design patterns follow the same idea as software design patterns by capturing reoccurring patterns of knowledge in concepts, relations and axioms. Patterns for knowledge modeling have been proposed by Clark et al. [CTP00] in 2000. Since then, catalogs of concrete ontology patterns have been suggested by various groups which are summarized in this section.

Gangemi et al. [Gan05] have analyzed different proposed design patterns for ontologies and created a categorization similar to the GoF patterns in software design. The following categories exist:

Structural Pattern: Such patterns include logical constructs that cannot be expressed using axioms as well as architectural patterns that deal with the overall design of the ontology (e.g. in terms of its computational complexity).

Correspondence Pattern: Patterns that cover mappings and transformations (defined as mappings with changes to logical types) between different ontologies.

Reasoning Pattern: Patterns that have the goal to obtain certain reasoning results are called reasoning patterns. Examples include: classification, subsumption, inheritance, etc.

Presentation Pattern: A pattern that improves the readability/usability of an ontology is called a presentation pattern. For example, certain entity naming conventions fall into this category.

Besides these patterns, Gangemi et al. define *content patterns* as an instance of structural patterns for a specific domain with an explicit vocabulary for this domain. They are examples of implementations that are reused by applying specialization, extension and composition. Similarly, the *lexico-syntactic patterns* are an even further specialized category for the NLP domain (e.g. to model word orderings). Unfortunately, the authors blur the line between the different patterns here and do not specify a similar exemplifying category for reasoning patterns.

Most of the work in [PGD+08] and [PG08] focuses on the above classification. A list of ontology design patterns is provided, and an online platform for a collaborative evaluation of ontology design patterns has been introduced under the "NeOn project",

which is still actively maintained[2]. So far, no actual reasoning design patterns have been proposed. Furthermore, the largest group of patterns (content patterns) introduced by [PGD+08] carry only very little design information. While regular software design patterns are very focused solutions to common problems, the patterns presented in the work of the "NeOn project", although following some of design quality criteria mentioned in Section 3.2, are not targeting any solution space. Their aim is to be as general as possible to allow for reusability without the need for refactoring. This is in contrast to the contributions of this thesis to provide reasoning patterns that are aimed to be refactored for a specific domain.

As the work of the "NeOn project" is closely related to this thesis, two of their patterns are analyzed to stress the difference between their contributions and the contributions in this thesis. Table 4.2 and Figure 4.2, taken from [PGD+08] and their website[3], show the *Agent Role* content pattern with its intended use as a link between an agent (person, object, ...) and a role the agent plays. In Table 4.3 and Figure 4.3, a correspondence pattern (taken from [PG08][4]) is shown which transforms an adjacency list (the representation of all edges or arcs in a graph as a list) into an ontology.

A critical analysis of the two patterns reveals some obvious shortcomings of the proposed ontology design patterns. The *Agent Role* pattern contains too little information to be reusable by itself. Clearly, one could argue for an alternative design of the agent and role relations. The competency questions are general and could easily match other implementations. The separation of agent and role reflects a design decision, where an agent can have multiple roles at the same time, but this is rather a property of object-oriented

[2]http://www.gong.manchester.ac.uk/odp/html/

[3]http://ontologydesignpatterns.org/wiki/Submissions:AgentRole

[4]http://ontolo.../Submissions:Classification_scheme_-_adjacency_list_model_-_to_Taxonomy

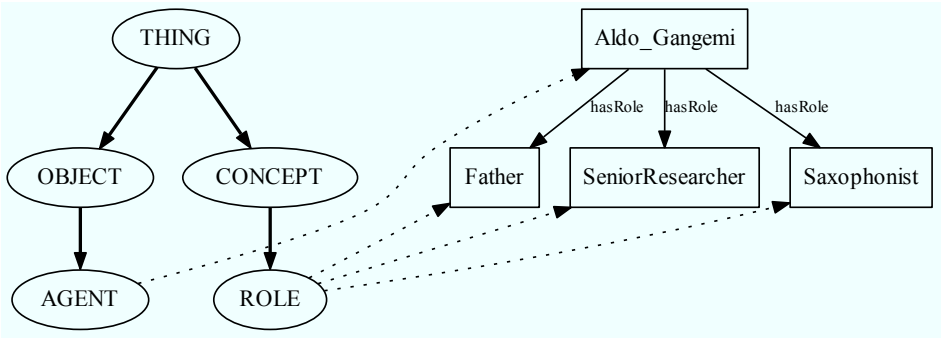

Figure 4.2: *Agent Role* pattern example [PGD+08]

Name	Agent Role
Submitted by	Valentina Presutti
Intent	To represent agents and the roles they play.
Domains	Management, Organization, Scheduling
Competency Questions	Which agent does play this role? What is the role that played by that agent?
Consequences	This CP allows designers to make assertions on roles played by agents without involving the agents that play that roles, and vice versa. It does not allow to express temporariness of roles.
Scenarios	She greeted us all in her various roles of mother, friend, and daughter.

Table 4.2: Example definition of the *Agent Role* pattern [PGD+08]

design principles and not captured in the ontology design pattern or its competency questions. The *Classification to Taxonomy* transformation pattern uses a non-ontological resource as an input and is therefore more clearly defined. Nevertheless, the transformation rules are too simple to provide an added-value and, as with the content pattern, no concrete problem is solved by the pattern.

Patterns addressing concrete modeling problems have been introduced by the Se-

Name	Classification Scheme to Taxonomy
Submitted by	Boris Villazon-Terrazas
Intent	Transformation of an adjacency list to an ontology.
Input	A classification scheme is a rooted tree of concepts, in which each concept groups entities by some particular degree of similarity. A list of items with a linking column associated to their parent items.
Ontology	Each category in the classification scheme is mapped to a class, and the semantics of the relationship between children and parent categories are mapped to subClassOf relations.

Table 4.3: Example definition of the *Classification to Taxonomy* pattern [PG08]

ID	Name	Parent
20000	Water area	
21000	Environmental area	20000
24020	Jurisdiction area	20000
21001	Inland/marine	21000

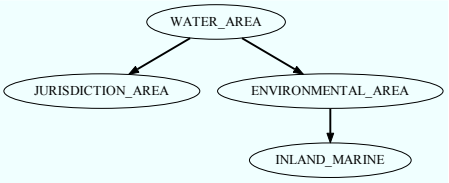

Figure 4.3: *Classification to Taxonomy* pattern example *Classification to Taxonomy* pattern [PG08]

mantic Web Best Practices and Deployment Working Group[5]. These patterns mainly address limitations of the OWL/OWL2 language, such as with the *N-ary Relations* pattern [NRHW06], or discusses different ways to realize a modeling problem in regards to complexity and ramifications as described in [NUW04] and [Rec05]. Some of the more concrete problem-patterns address *time* [HP06] and *part-of relationships* [RWNW05].

Egana-Aranguren et al. [EA09] have introduced ontology design patterns in regards to bio-ontologies. A catalog of 15 patterns can be found online[6]. Their categorization of patterns defines three main groups:

[5]http://www.w3.org/2001/sw/BestPractices/OEP/
[6]http://www.gong.manchester.ac.uk/odp/html/

Extension Pattern: An *extension* pattern by-passes the limitation of a modeling language (such as OWL) in order to achieve a certain design goal. An example for such a pattern is the *N-ary Relations* [NRHW06] pattern.

Good Practice Pattern: The goal of the *good practice* patterns is to create a more robust, cleaner and easier to maintain ontology. There exists many examples that fall into this general category, such as *Value Partition*[7] and *Selector*[8].

Domain Modeling Pattern: Patterns, which propose a solution for a specific modeling problem that is only applicable to a certain domain, are grouped into the *domain modeling* pattern category. An example for such a pattern is the *Cell Cycle Sequence*[9].

Both, the Semantic Web Best Practices and Deployment Working [NUW04] [Rec05] as well as Egana-Aranguren et al. [EA09], propose meaningful ontology design patterns that can be praised as a good example of reusable design that is widely applicable. The domain modeling patterns are thereby closely related and can often be combined with the work in this thesis. However, their suggestions do not focus on the support and effects of reasoners on ontology design patterns.

Common to most of the existing ontology pattern approaches is that they lack a methodology or an ontology life cycle for the use of design patterns. Although the introduction of design patterns establishes design reuse, it does not eliminate the need for a design methodology that can provide methodological and technological guidance while minimizing reuse efforts. An ontology design methodology that incorporates the

[7] http://www.gong.manchester.ac.uk/odp/html/Value_Partition.html
[8] http://www.gong.manchester.ac.uk/odp/html/Selector.html
[9] http://www.gong.manchester.ac.uk/odp/html/Sequence.html

design patterns, which are introduced in this chapter, as well as design patterns from other sources such as the ones listed above, is introduced in Chapter 3.

4.3 Structural Reasoning Patterns

Structural reasoning patterns are a novel combination of structural patterns and reasoning patterns, two pattern categories previously defined in [Gan05]. The main idea of structural reasoning patterns is to provide reusable definitions for logical constructs that cannot be expressed trivially using axioms and require some form of reasoning to work. In contrast to content patterns, they are domain independent. Each pattern is described as an input pattern in DL and a textual description of the problem to be solved. Where applicable, a non-ontological input is described using a UML diagram. A set of transformations is then applied to present a valuable solution (again in DL). Additionally, patterns are motivated by an example for a concrete application domain that can be refactored.

4.3.1 Limited Transitivity Pattern

In large highly inter-connected ontologies, it is not uncommon to find transitive properties that cannot be allowed to "ripple" through all possible individuals or concepts. Also, a full transitivity is sometimes not necessary, such as when the intent of the property is to represent a degree of membership that declines with the distance to the originator. In such scenarios, the transitive property can be replaced by the *Limited Transitivity* pattern that specifies a fixed number of levels, which are realized through the use of a property chain axiom. Given are the following sample definitions, using a transitive property

$similarTo$:

$$Tra(similarTo)$$

$$similarTo(Ind_1, Ind_2)$$

$$similarTo(Ind_2, Ind_3)$$

$$similarTo(Ind_3, Ind_4)$$

$$similarTo(Ind_4, Ind_5)$$

The ontology can now be transformed by replacing $similarTo$ with the following non-transitive property $similarToNonTran$ by adding, in this example three, property chain axioms (representing the similarity level/degree):

$$similarToLvl1 \sqsubseteq similarToNonTran$$

$$similarToLvl2 \sqsubseteq similarToNonTran$$

$$similarToLvl1 \circ similarToLvl1 \sqsubseteq similarToLvl2$$

$$similarToLvl3 \sqsubseteq similarToNonTran$$

$$similarToLvl1 \circ similarToLvl2 \sqsubseteq similarToLvl3$$

$$similarTo(Ind_1, Ind_2)$$

$$similarTo(Ind_2, Ind_3)$$

$$similarTo(Ind_3, Ind_4)$$

$$similarTo(Ind_4, Ind_5)$$

The resulting inferred $similarToNonTran$ relations for each individual are identical to the original transitive $similarTo$ relations (except for the missing relation between Ind_1 and Ind_5 that is out of the specified levels of indirection). An example of the *Limited Transitivity* pattern is shown in Figure 4.4.

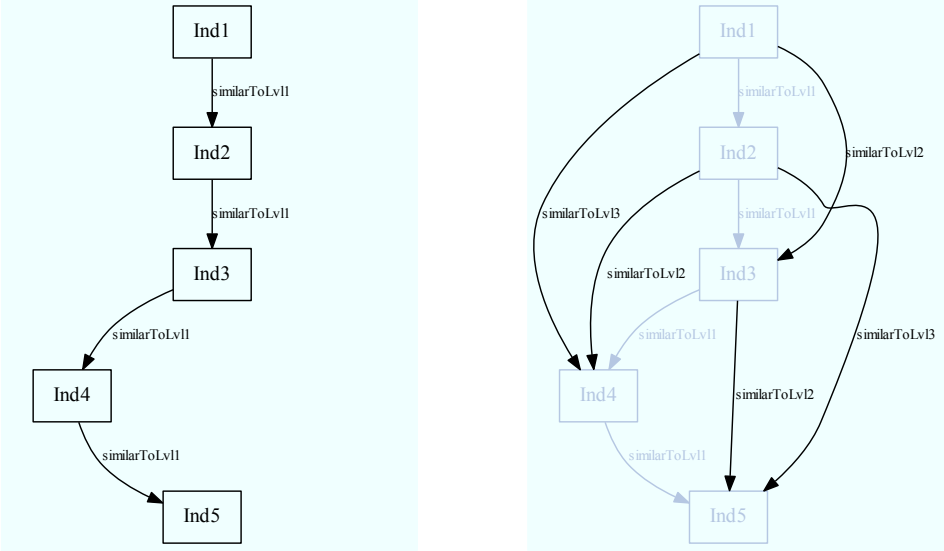

Figure 4.4: *Limited Transitivity* pattern example

The *Limited Transitivity* pattern can also be used in conjunction with a transitive property (instead of replacing it) in order to identify the level of indirection in a transitive chain. It can, for example, be used to identify components at a certain level in a $partOf$ relationship. A laptop might consist of "CPU", "LCD", etc. on the first level of the $partOf$ relationship, while one might find "Transistor" or "Silver" at higher levels. This information can be crucial in answering certain queries (e.g. "to identify the immediate sub-components to be ordered").

4.3.2 Restriction Generalization Pattern

A rather general and therefore often reoccurring pattern in ontology development/refactoring is the transformation of a value restriction to a class restriction, in order to allow for a more generic case. An individual might be selected at first during ontology development (based on the used methodology), if it is assumed there are no further elements to justify a class. Nevertheless, for extendibility reasons it can be beneficial to convert such a restriction using the *Restriction Generalization* pattern. Thereby, the individual is taken as one example of a (general) class. The restriction is then applied on the class instead of the value.

As a concrete example the following axioms declare two simple value restrictions on the class $REGULATOR$:

$$\frac{\begin{array}{l} REGULATOR \equiv\ \exists makesRegulationsFor.\{Country_1\} \\ REGULATOR \sqsubseteq\ \exists getsPaidBy.\{Country_1\} \end{array}}{hasValue(Country, ExProp)}$$

These axioms can be transformed into a class restriction by creating a new class where the individual is the main "exemplifying" member. Any properties of the individual become restrictions of the new class. New members that satisfy the conditions of the new class can then be added. The transformed axioms are:

$$\begin{array}{l} REGULATOR \equiv\ \exists makesRegulationsFor.COUNTRY_1 \\ REGULATOR \sqsubseteq\ \exists getsPaidBy.\{Country_1\} \end{array}$$

$$\frac{COUNTRY_1 \sqsubseteq \exists hasValue.\{ExProp\}}{COUNTRY_1(Country_1)}$$

In the above example, $Country_1$ is the main individual of concept $COUNTRY_1$. With the transformation complete, one could now easily add additional members, for example $Province_1$ as a province, to $COUNTRY_1$. This also allows for a partial overlap of the transformed individuals and one could further imagine an individual $Province_2$ which is part of both concepts $COUNTRY_1$ and $COUNTRY_2$.

Many real world scenarios require the use of cardinality restrictions in order to model the reality accurately. Unfortunately the OWL2 EL profile does not allow for the use of cardinality restrictions. Nevertheless, the *Restriction Generalization* pattern can be used to model a minimum cardinality of 1 for a certain individual. For this purpose, the individual is transformed into a concept, as shown above. Any restriction using the concept is then semantically equivalent to a minimum cardinality of 1. This modeling approach allows one to express that a regulator makes "at least 1" regulation for $COUNTRY_1$. Otherwise one would have to define:

$$REGULATOR \equiv \ \geq 1 \ makesRegulationsFor.COUNTRY_1$$

The *Restriction Generalization* pattern is related to "Representing Classes As Property Values" (approach 4) [Rec05] similar to the relationship of the *Adapter* and *Bridge* pattern [GHJV94]; two inherently different problems are solved using a similar implementation.

4.3.3 Property-Class Commonality Pattern

During ontology design and refactoring, it is not uncommon to come across properties which are modeled as classes instead of object properties. This is usually not a problem and can be even desired in situations when one wants to combine different attributes. Nevertheless, in such cases combining classes and object properties can be challenging. For example, the concept $SCIENTIFICBOOK$ encodes the property $isScientific$ but this property does not necessarily explicitly exist; it is only implicitly represented through the class membership of an individual. Combining the concept with an object property $hasCitation$, to create the derived commonality relation $hasScientificCitation$, is therefore non-trivial. The ontological input for the pattern is shown in Figure 4.5.

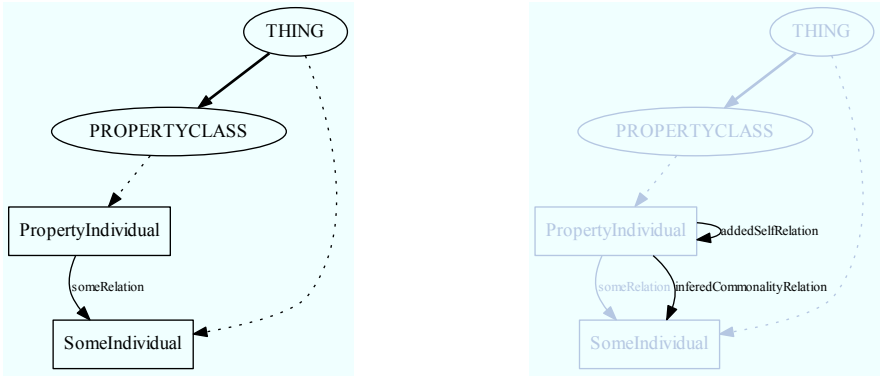

Figure 4.5: *Property-Class Commonality* pattern example

In order to infer a related $inferredCommonalityRelation$ that depends on the membership of an individual in $PROPERTYCLASS$ in combination with the existence of the relation $someRelation$, first a relation must be added to every member of $PROPERTYCLASS$ using the OWL2 $hasSelf$ constraint. Once the relation is estab-

lished, one can infer the commonality relation through a simple property chain axiom. The definition following DL axioms describe the pattern:

$$PROPERTYCLASS \sqsubseteq \exists addedSelfRelation.Self$$

$$addedSelfRelation \circ someRelation \sqsubseteq inferredCommonalityRelation$$

An example for applying the pattern to concept $SCIENTIFICBOOK$ and the desired relation $citesScientific$ is given in Figure 4.6. The $isScientific$ relation thereby constitutes the $addedSelfRelation$.

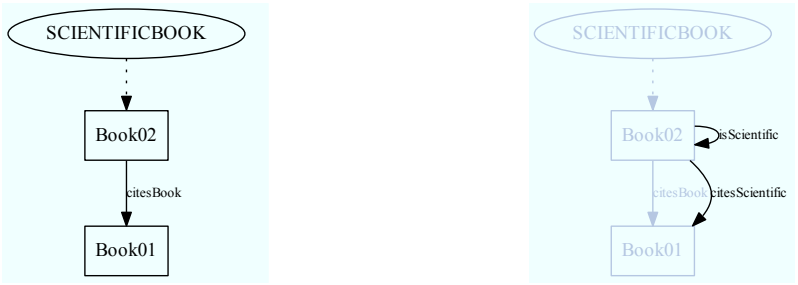

Figure 4.6: *Property-Class Commonality* pattern example

Although the OWL2 EL profile allows for all axioms used in this pattern, most EL reasoners do not support the *Property-Class Commonality* pattern due to the missing implementation of $self$ restrictions or general disallowance of any nominals.

4.3.4 Representative Individual Pattern

Ontology design differs from object-oriented design in respect to what constitutes classes (concepts) and instances (individuals). While object-oriented design dictates a class for each object in the modeled domain, ontology design is more flexible. For example, a

"Car" can be a concept or individual in an ontology, even when there exists multiple instances ("Car1", "Car2",...) of the object. In order to strengthen this understanding of ontology design, the ***Representative Individual*** pattern suggests the use of one individual, to represent a group of data. This can be beneficial in a number of scenarios in which there is no need to model an object as a concept. For example, one could define an individual $AllCountries$ to represent all possible countries instead of using a disjoint class. This can have an impact on reasoning results and has to be evaluated carefully, but is often a suitable alternative.

The example for all countries could be modeled as a relation $publishedIn$ to the concept $COUNTRY$ (shown with $Magazine_1$):

$$COUNTRY(CountryA)$$

$$COUNTRY(CountryB)$$

$$publishedIn(Magazine_1, COUNTRY)$$

It can also be modeled as a a relation to all its individuals (shown in $Magazine_2$):

$$publishedIn(Magazine_2, CountryA)$$

$$publishedIn(Magazine_2, CountryB)$$

Following the ***Representative Individual*** pattern, this scenario can be converted into a single individual $AllCountries$ such as (whereby $AllCountries$ can be an individual of

$COUNTRY$ to minimize the impact on reasoning results):

$$COUNTRY(AllCountries)$$

$$publishedIn(Magazine_X, AllCountries)$$

Another use of the pattern is the representation of facts that cannot be modeled in the knowledge representation language, or which are not part of the chosen language profile. For example, "at least 5 elements" could not be modeled in the OWL2 EL profile (as it has no support for cardinality restrictions). It is, however, possible to add a relation to an individual $AtLeast5Elements$ from all individuals, with the corresponding amount of elements. The idea of encoding semantics in an individual has to be practiced with care, as undesired side-effects can arise. If possible within the language or profile, a "correct" modeling is always the preferred solution.

4.3.5 Subclass Disjunction-Like Pattern

Intersections and disjunctions of concepts are a common restriction that are often used to form new concepts. However, in the OWL2 EL profile, disjunctions are not allowed due to performance restrictions, which has led to ontology design problems. Nevertheless, it is still possible to model a union of individuals (e.g. all individuals of a concept A and B to be part of a concept C). For example, one could define $EMPLOYED$ as a disjunction of the concepts $PROFESSOR$ and $ASSISTANT$, in a university domain. For this purpose, the *Subclass Disjunction-Like* pattern simply suggests the use of a $subClassOf$ relation to replace a $unionOf$. For the given example, the following definitions can be

stated:

$$\frac{EMPLOYED \equiv\ PROFESSOR \sqcup ASSISTANT}{\begin{array}{c} PROFESSOR(Professor_1) \\ ASSISTANT(Assistant_1) \end{array}}$$

This can be refactored using the *Subclass Disjunction-Like* pattern through the definitions:

$$PROFESSOR \sqsubseteq\ EMPLOYED$$

$$ASSISTANT \sqsubseteq\ EMPLOYED$$

This results in $Professor_1$ and $Assistant_1$ classified as $EMPLOYED$, and is fully compatible with the OWL2 EL profile. It is easily provable that the new definition of $EMPLOYED$ subsumes the original definition. However, the disjunctive information about $PROFESSOR$ and $ASSISTANT$ is lost. Nevertheless, in case only a union of individuals (as indicated by $Professor_1$ and $Assistant_1$ in the example) is required, this can be an acceptable solution.

4.3.6 Subproperty Disjunction-Like Pattern

Similar to the *Subclass Disjunction-Like* pattern, subproperty relations can be used to model a union of relations. For example, two properties $propA$ and $propB$ can be made

a subproperty of a concept $propAOrB$:

$$propA \sqsubseteq propAOrB$$

$$propB \sqsubseteq propAOrB$$

This is especially useful for the new OWL2 $hasKey$ axiom. In OWL2 $hasKey$ conjunction can be specified easily, $hasKey$ disjunction, however, can only be added through individual keys. This individual declaration differs slightly from what most programmers expect under the short circuit evaluation[10]: While it is clear that for a key conjunction all keys must be present (and identical) in order to determine whether two individuals are in fact identical, short circuit evaluation dictates that for a disjunction, only *one* of the keys must be present and identical. This is not consistent with separate $hasKey$ axioms implementation (as each key is required to be present).

Consider the following definition of two individuals and a concept CL:

$$CL(Indivdual_1)$$

$$CL(Indivdual_2)$$

$$propA(Indivdual_1, Object_X)$$

$$propB(Indivdual_2, Object_X)$$

The usual way to model a disjunctive key would be to add two individual keys for

[10]http://en.wikipedia.org/wiki/Short-circuit_evaluation/

$propA$ and $propB$:

$$Key(CL, propA)$$

$$Key(CL, propB)$$

Ideally, keys could be represented using the OWL2 $unionOf$ axiom, but this is currently not supported in the OWL2 standard. In order to overcome this limitation, the *Subproperty Disjunction-Like* pattern promotes the use of a common superclass property (instead of the two individual keys) that combines the two properties, which can then be used as the key for the class CL:

$$propA \sqsubseteq propAOrB$$

$$propB \sqsubseteq propAOrB$$

$$Key(CL, propAOrB)$$

With these definitions in place, $Indivdual_1$ and $Indivdual_2$ can be identified as denoting the same entity ($sameAs$) as expected from a disjunctive key over the two properties with short circuit evaluation.

4.3.7 Hierarchy Creation Pattern

When parsing large amounts of data efficiently, it is impractical to keep too many elements in memory. Ideally, elements are directly processed and written to disk. In case of hierarchical structures within the parsed data, the creation of links between the individuals is of vast importance. However, the linking of individuals corresponding to a

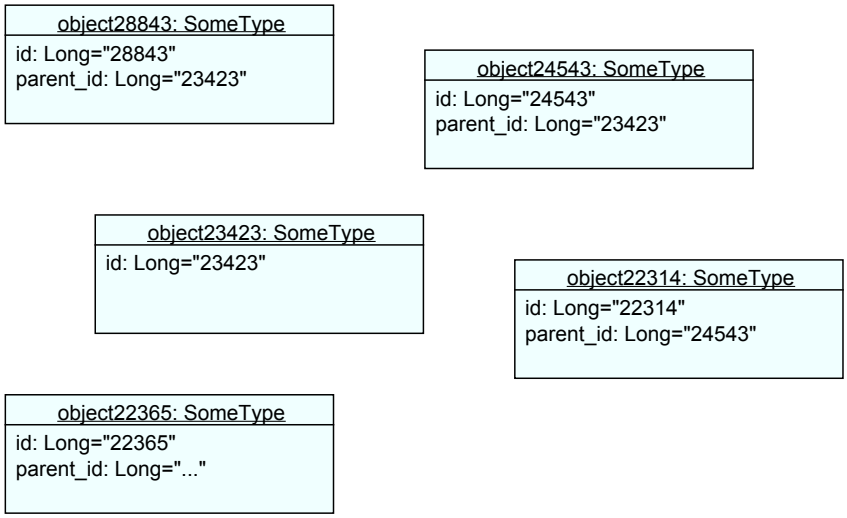

Figure 4.7: *Hierarchy Creation* pattern input

hierarchical structure (e.g. a tree), its root node and a container-class, is not trivial, as each parsed element only contains the link to a child or parent node (but not a link to all other individuals). This is shown in Figure 4.7 which visualizes independently parsed child and root nodes. An application domain for the pattern is NLP.

In order to structure parsed elements and assign a hierarchy class to them, one can first assign a random class to the root node (node without parent) and then define a concept $CHILDOFTREE$ to pull any child associations into the class. The following axioms describe the idea for the tree example (in order of parsing):

$$\frac{\begin{array}{rl} TREE \sqsubseteq & \exists hasRoo.\{Object23423\} \\ CHILDOFTREE \equiv & \exists childOf.TREE \\ CHILDOFTREE \sqsubseteq & TREE \end{array}}{TREE(Object23423)}$$

$$childOf(Object22314, Object24543)$$

$$childOf(Object28843, Object23423)$$

$$childOf(Object24543, Object23423)$$

The reasoner successively classifies objects as being a $CHILDOFTREE$ until the complete structure is added. The resulting graph, shown in Figure 4.8, contains all objects which are part of the structure (the same class $TREE$).

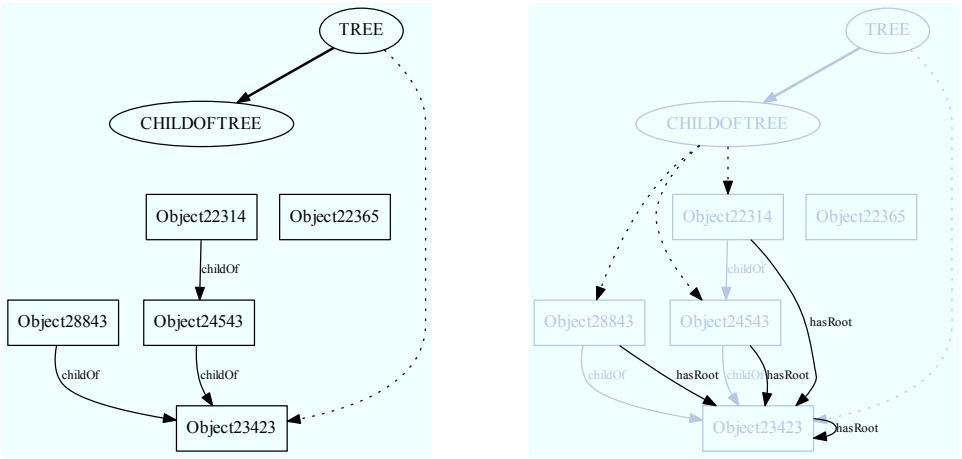

Figure 4.8: *Hierarchy Creation* pattern example

The above pattern is especially important, since non-ontological data sources tend to carry incomplete information, which can originate from poor modeling of an application or a low quality of data being parsed. In hierarchical data sources modeled using the *Hierarchy Creation* pattern, one can easily state knowledge that is applicable throughout the hierarchical structure. The pattern example models a single tree hierarchy but multiple hierarchies could be created in parallel in a similar way (e.g. by having multiple

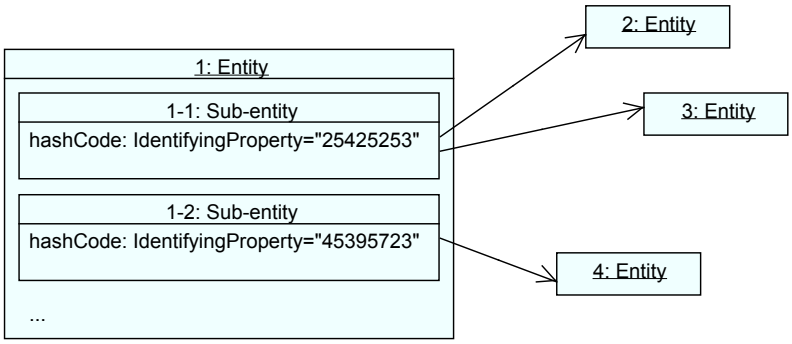

Figure 4.9: *Unbound Key* pattern input

$TREE$ and $CHILDOFTREE$ concepts).

4.3.8 Unbound Key Pattern

The OWL2 $hasKey$ axiom allows the identification of identical individuals in a concept with regards to a specified key. The axiom can, for example, be used to state that a social insurance number (SIN) identifies a person uniquely. In the case of combined keys, a key conjunction can be specified. The OWA however, leads to the requirement that all specified keys must match for two individuals to become $sameAs$. An ontology design problem therefore arises if individuals with a varying number of keys have to be found as being identical. For example, to identify a PC by its components, one also has to consider missing components: two computers without CPUs should still be identified as identical if all other components are the same. In this case, the *Unbound Key* pattern suggests the introduction of an $Unbound$ individual for each of the keys that are unspecified.

The non-ontological input for the pattern is a set of objects with distinct properties that are interlinked using a varying amount of relations (shown in Figure 4.9). This common structure can be found in various domains that can be modeled as a graph.

Given below are the definitions for the example above including the key properties $hasRAM$ and $hasCPU$:

$$Key(PC, hasRAM, hasCPU)$$

$$PC(Comp_1)$$

$$hasRAM(Comp_1, Ram93742)$$

$$PC(Comp_2)$$

$$hasRAM(Comp_2, Ram93742)$$

In order for the reasoner to identify $Comp_1$ and $Comp_2$ as $sameAs$, the object property $hasCPU$ has to be set. As suggested by the ***Unbound Key*** pattern, a new individual is introduced for this purpose:

$$hasCPU(Comp_1, UnboundCpu)$$

$$hasCPU(Comp_2, UnboundCpu)$$

This models the requirement of varying keys successfully and allows the use of the OWL2 $hasKey$ axiom to identify similarity for such individuals.

4.3.9 Equivalence Similarity Pattern

The ability of reasoners to classify (build a taxonomy for) large amounts of data presents itself as a suitable candidate to be applied in the problem space of identifying similarity between objects. The *Equivalence Similarity* pattern is an alternative to identifying similarity using the OWL2 $hasKey$ axiom (as shown in the previous section). This is

of importance, as most reasoners to this date do not support the $hasKey$ axiom or have not yet fully implemented it, according to the OWL2 specification. The pattern uses a $oneOf$ restriction in order to classify all individuals with certain properties p_1, p_2,... p_n as being the same. The following statements define two individuals Obj_1 and Obj_2 with two identical properties that should be identified as $sameAs$:

$$p_1(Obj_1, Prop_A)$$

$$p_2(Obj_1, Prop_B)$$

$$p_1(Obj_2, Prop_A)$$

$$p_2(Obj_2, Prop_B)$$

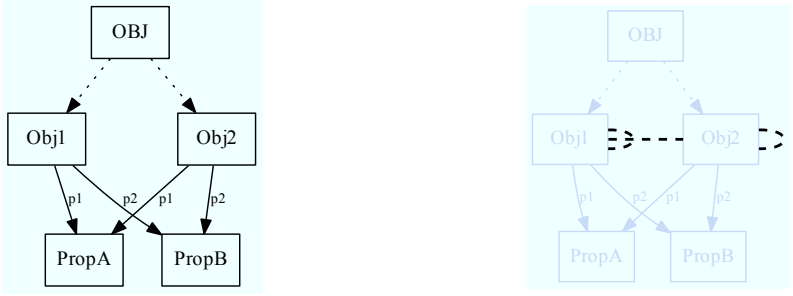

Figure 4.10: *Equivalence Similarity* pattern example

The information about which objects are the same is not known a priori; Determining it (based on the properties of individuals), is the responsibility of the reasoner. In order for the pattern to classify the two objects as being $sameAs$, the following definitions

need to be added:

$$
\begin{aligned}
OBJ_1 &\equiv \exists p_1.\{Prop_A\} \sqcap \exists p_2.\{Prop_B\} \\
OBJ_1 &\sqsubseteq \{Obj_1\} \\
OBJ_2 &\equiv \exists p_1.\{Prop_A\} \sqcap \exists p_2.\{Prop_B\} \\
OBJ_2 &\sqsubseteq \{Obj_2\}
\end{aligned}
$$

With this definition in place, the two individuals Obj_1 and Obj_2 are identified as denoting the same object ($sameAs$). In addition to identifying two individuals as the same, it can be desired to only add a relation between similar individuals. It is sometimes also not allowed to identify two individuals as being $sameAs$, as this can have implications on other parts of the ontology design. In order to accommodate this requirement, the pattern can simply be modified by adding a value restriction that infers a relation instead of restricting every individual of the concept to be the same):

$$
\begin{aligned}
OBJ_1 &\equiv \exists p_1.\{Prop_A\} \sqcap \exists p_2.\{Prop_B\} \\
OBJ_1 &\sqsubseteq \exists identicalTo.\{Obj_1\} \\
OBJ_2 &\equiv \exists p_1.\{Prop_A\} \sqcap \exists p_2.\{Prop_B\} \\
OBJ_2 &\sqsubseteq \exists identicalTo.\{Obj_2\}
\end{aligned}
$$

An example of the pattern is shown in Figure 4.10. In this case, Obj_1 and Obj_2 are classified as being $sameAs$.

Chapter 5

Ontology Application Model

As much as design patterns are best-practice solutions for the development of an ontology, "application models" are proven solutions for a specific application problem space. As discussed in the previous chapters, the SE-ONTO methodology promotes the development of ontologies for, and with an application. However, the methodology is independent from the type of the developed application, and no application model is provided. Similar to content management systems or Wikis (which can be considered application models for the web), there needs to be application models for ontology-driven applications in order to ease development and lower the barrier for novice developers. Furthermore, there is a need to incorporate non-ontological data sources as a starting point for the development of applications. This is motivated through the fact that many data sources, which are of interest to be shared and processed, are already available in some (semi-) structured form. Therefore, this chapter introduces an application model for the SE-ONTO methodology that transforms existing non-ontological data sources incrementally into an ontology (through a mapping task). In contrast to other approaches, which

only convert data into an ontology once, the approach repeatedly reads and updates an ontology with data (which remains primarily stored in a non-ontological source).

For the success of any modern technology, available tools are an important aspect that should not be overlooked. In order for OWL and ontologies to become an integrated part of application development, better tool support is needed. The vision that knowledge repositories jointly and seamlessly work together with data storages to provide semantic rich application is far from being solved. The use of application frameworks that hide some of the complexity of developing such applications is a corner stone for achieving this goal. Complexity-reduction through frameworks is inspired by the recent development and popularity of web-application frameworks such as the Ruby on Rails framework[1], which advocates tools and conventions to simplify development. Besides application frameworks, tools can also help developers to better comprehend the used methodology, by becoming an integrated part of development tools and the used Integrated Development Environment (IDE). Popular programming frameworks, such as Spring Roo[2], offer tight integration with an IDE that allows a simplified and streamlined development process. One of the major challenges for any application model, therefore, is the need to integrate various knowledge resources within an IDE in a consistent and homogeneous representation that is beneficial to developers.

[1] http://rubyonrails.org/
[2] http://www.springsource.org/roo/

In this chapter, the SE-ADVISOR application model and IDE support are presented. SE-ADVISOR has the following goals:

- Provide an application model for ontology-driven applications
- Provide an implementation of a framework for an ontology-driven application model
- Provide support for SE-ONTO and its application model within an IDE

It must be pointed out that the SE-ADVISOR application model, while being applicable to a wide variety of tasks, is only one of many ways ontologies can be incorporated into applications. Furthermore, SE-ONTO is a generic methodology that is not limited to the specifics of the SE-ADVISOR application model.

5.1 SE-ADVISOR Application Model

The SE-ADVISOR application model assumes the development of an ontology-driven application, in which an ontology is not the primary data source, but rather gets incrementally populated from existing data sources through the use of programming logic (e.g. through a Java program). It is therefore fundamentally different from other ontology-driven applications where the user is responsible for adding concepts/individuals/... manually to an ontology (e.g. through an ontology editor). The model further assumes that the population process is decoupled from the application usage (e.g. time- or event-triggered). Data is incrementally added to the ontology. A reasoner is used to materialize any inferences (taking into consideration the nature of OWA) before the current ontology is provided to the application to be queried. No information is ever written back to the

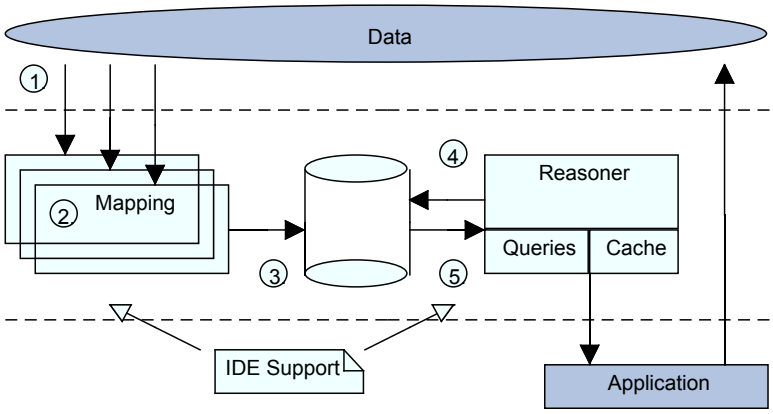

Figure 5.1: SE-ADVISOR application model

ontology in order to discourage a direct mapping between object-oriented application design and ontology design. Instead, any data updates are written to the original source, and find their way back into the ontology through the ontology population.

By separating the ontology design and population from the application (and creating a query interface for accessing information from the ontology), a clear separation of concerns is achieved. Consequently, the ontology designer can act independently from the application developer. The developer must only interact with a query interface and does not need to know about semantic web technologies. The designer incorporates responsibilities (originating from user requirements) in the ontology design and provides results to be queried.

The application model can be broken down into the following individual steps: (1) An event triggers the data retrieval. (2) This data is then transformed into axioms through a mapping. (3) The axioms are added to the current ontology representation (persistent or in-memory storage). (4) An event triggers the reasoner to load and infer information. (5) The information is materialized (inference results are made explicit) back to the ontology

representation. This application model, in which the ontology is constantly modified and incrementally extended, is shown in Figure 5.1. The data mapping is different from "populating" an ontology that usually just refers to adding individuals and relations (ABox). Instead, the data mapping can add also concepts or other axioms, and is done in a programmatic way. A Globally Unique Identifier (GUID) module is responsible for creating unique identifiers for each data element. The mapping thereby is not limited to a direct relation between data elements and ontology concepts/individuals. It can define one or more individuals or concepts for every object in an application or group objects together. For example, an object oriented design of an inventory system will dictate that an object exists for every single item, while the same data is represented as a single individual in the ontology.

The introduced application model is supported by a framework which is the SE-ADVISOR application server. The server can execute tasks through time-triggered events and in a context-aware manner. "Context-awareness" originated as a term, which sought to deal with otherwise static linking changes in computer system environments. Schilit et al. [SAW94] introduced the term context-awareness in the ubiquitous computing domain. Two important aspects of context-awareness are the "location of use" and "used resources".

Listing 5.1: Task interface for SE-ADVISOR application server

```
public void execute(JobExecutionContext context) throws JobExecutionException;
public void persist(OWLOntology ont) throws OWLOntologyStorageException;
```

In the SE-ADVISOR application server, mappings are provided as Java libraries that conform to a task interface, as shown in Listing 5.1. The task context includes information about the location of data that has changed since the last mapping process ("location

of use") and requires an update, as well as a split of data to be processed in case multiple mapping tasks are running in parallel. It also provides a reference to the ontology storage ("used resources"). It is the server's responsibility to `execute` tasks, store and reschedule them when needed. The server can write (`persist`) the ontology produced by a mapping task to the ontology storage. As a secondary role, the server also facilitates the storage of queries (e.g. SPARQL queries). The query storage assigns a unique ID to every query that can be called by the application. Query results are cached until the next task is triggered, in order to improve query performance. This operation is safe, as modifications to the ontology can only be made through the defined tasks.

Taking into consideration the distributed work environment in which most systems are deployed, a client-server architecture has been selected for the SE-ADVISOR application model. In this architecture clients communicate over a network with the application server. Communication between the clients and server is realized as a Representational State Transfer (REST) web-service. The server application is deployed on a Tomcat[3] server and secured through the use of Hypertext Transfer Protocol Secure (HTTPS).

The SE-ADVISOR differs from other semantic web frameworks, such as [BKvH02], as it defines an explicit data-flow. Mapping and reasoning tasks are executed based on time- or event-triggers, which limits the application model to problems that do not require "real-time" reasoning. Furthermore, data modified by the application is not written directly to the ontology but instead, is updated in its original source (and written back into the ontology through a mapping task). This requires that the mapping task is able to identify what parts of the data elements have changed since the last run (a "versioned"

[3] http://tomcat.apache.org/tomcat-7.0-doc/

data source).

5.2 SE-ADVISOR IDE Support

The research community has realized that software engineers are tired of switching between environments in order to deal with the different sources of required information [Sch02]. There is an ongoing effort in open-source development environments, to provide extensible plug-in architectures that integrate tools and artifacts. The idea of providing tool support with the SE-ADVISOR application model is rooted in the hypothesis that such an environment has the potential to increase the productivity of software engineers, and therefore, to reduce the overhead associated with learning a new methodology. At the same time, guidance provided by such an environment should benefit the overall quality of the produced software.

Most of the existing work in IDE support for application development has focused on agile methodologies. The Mylyn (former "Mylar") project[4] integrates users, task and artifacts on an abstract level. The artifacts considered are bug-trackers and revision-control systems, which are connected by providing a common editing user interface. Spring Roo[5] and the Spring Source Tool Suite[6] provide an IDE for rapid web-development. Due to its extensibility, Eclipse[7] has quickly become the most popular Java IDE used in the research community (and industry), reflecting the current state of the art of available tool support. A recent survey [Ecl09] among 1500 developers showed that 60% of all Java developers use Eclipse as their primary IDE. The SE-ADVISOR support tools have been

[4] http://www.eclipse.org/mylyn/
[5] http://www.springsource.org/roo/
[6] http://www.springsource.com/developer/sts/
[7] http://www.eclipse.org/

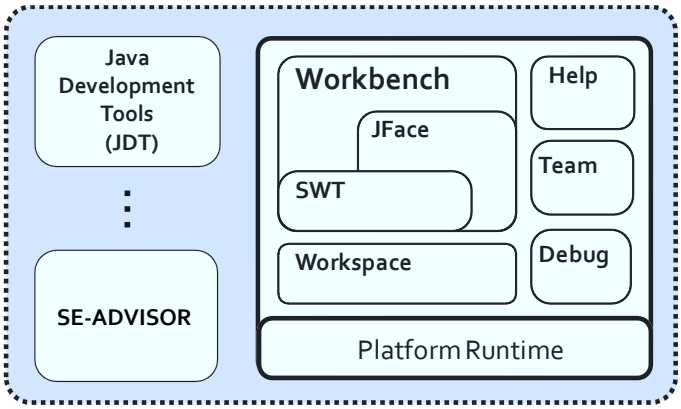

Figure 5.2: SE-ADVISOR in the Eclipse ecosystem

implemented as an Eclipse plug-in for the following reasons:

Accessibility: Eclipse is an open-source project with a large, established and still growing community. Its development is supported by multiple companies (such as IBM and Borland) and it is used in countless commercial and non-commercial projects, as well as in most universities.

Extensibility: Although Eclipse is known as an IDE, the Eclipse platform can be seen as a micro-kernel, offering the possibility to load and combine plug-ins. The Eclipse IDE itself is only a set of plug-ins extending the framework with a Java editor, compiler, etc. Eclipse supports both, the extension of the IDE, as well as the creation of stand-alone programs. The Eclipse foundation has the goal to actively support the rich ecosystem of Eclipse plug-ins that has evolved.

By integrating the SE-ADVISOR application model and SE-ONTO methodology in an IDE, the initial burden associated with using a new technology and following a new methodology is lowered. The following three goals have been addressed:

- Process guidance within the IDE

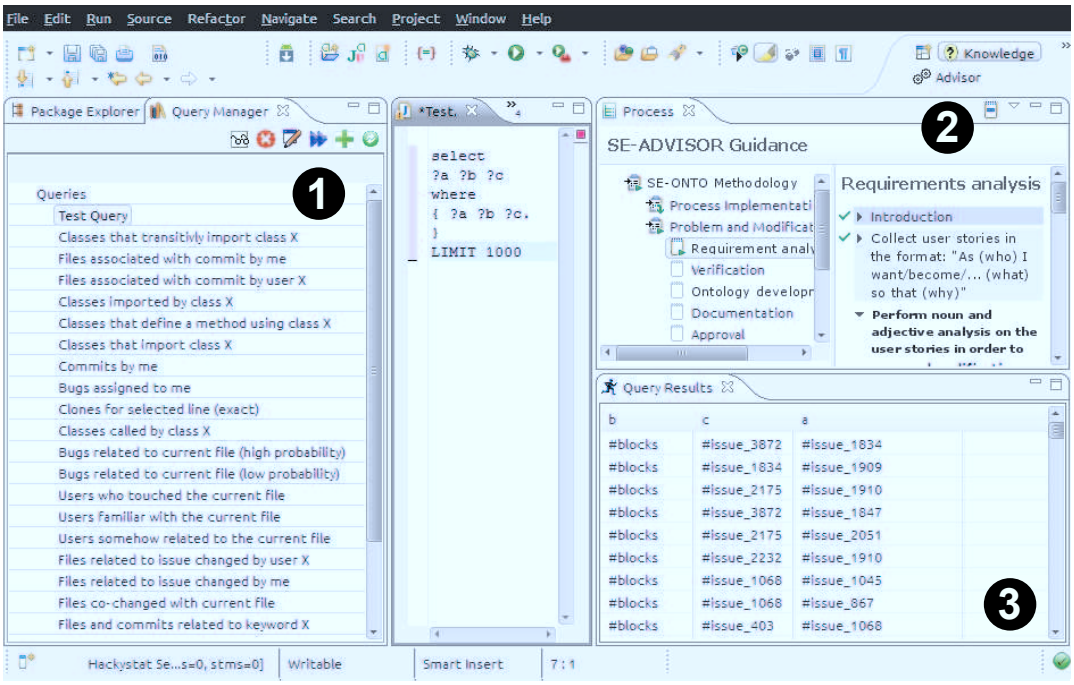

Figure 5.3: SE-ADVISOR query management and process guidance

- OWL2 profile checking for ontology design
- Java templates for ontology mapping tasks
- Query management

Figure 5.2 shows the integration of the SE-ADVISOR support tools within the Eclipse ecosystem. SE-ADVISOR builds upon the Eclipse platform and the Standard Widget Toolkit (SWT) and interacts with the Java Development Tools (JDT) to extend the Java editor's functionality. Three plug-ins have been developed: (1) A query storage plug-in that can display queries stored on the SE-ADVISOR application server. (2) A process plug-in that establishes the process context, by guiding the user through the SE-ONTO activities, and (3) a query execution plug-in that can run parameterized SPARQL queries and display results from the server.

The different plug-ins are shown in Figure 5.3. Configuration and choice are replaced by established best practice setups and conventions. The SE-ADVISOR tool support provides an ontology development framework, with a set of selected components, to encourage a fast project kick-off. Figure 5.4 shows a new project dialog, which automatically creates a Java project, with connector classes and interface definitions, for the SE-ADVISOR application server. Further, the following utility libraries are provided for each new project:

- CXF[8] - A REST web-service communication framework used to connect to the SE-ADVISOR application server. REST is a software architecture for stateless client-server communication and is used as a simple and intuitive method to realize well-defined create, read, update and delete (CRUD) operations through the standard HTTP protocol.

- OWLAPI[9] - An interface library used to interact with OWL ontologies. The Application Programming Interface (API) supports creating, loading, changing and saving ontologies whilst maintaining compliance with OWL2.

The provided libraries allow for a fast development of mappings which can be deployed to the SE-ADVISOR application server. Each mapping is compiled and packaged as a Java library (*jar* file), and transferred to the server where it is dynamically loaded by the server (a plug-in architecture, similar to the one of Eclipse). In regards to the ontology design, the SE-ONTO methodology is displayed as an interactive guide through the process plug-in. The creation of ontology files within Eclipse is thereby outside the

[8]http://cxf.apache.org/
[9]http://owlapi.sourceforge.net/

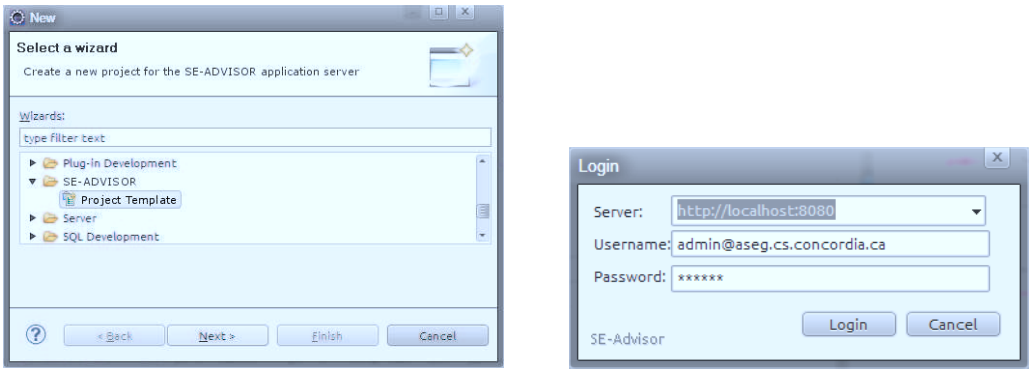

Figure 5.4: SE-ADVISOR project template and login

scope of the SE-ADVISOR. Ontology files can, however, be managed within Eclipse, using plug-ins from the open source NeOn toolkit[10], which allows editing and visualizing OWL files.

SE-ADVISOR also makes use of the new OWL2 profiles in order to constrain a developer to a certain complexity. This has been suggested by Horridge et al. [HB09] and is easily realized using the OWL API (OWLAPI) library (also available online[11]). Once the OWL2 profile is set in the SE-ADVISOR configuration, it is enforced upon check-in of the ontology.

[10] http://neon-toolkit.org/
[11] http://owl.cs.manchester.ac.uk/validator/

Chapter 6

Case Studies

In this chapter, the main contributions of this thesis, the ontology design methodology (Chapter 3), reasoning design patterns (Chapter 4) and the application model (Chapter 5), are evaluated through performance tests and multiple case studies. For each of these studies, one or more reasoning patterns are used, which have been implemented using the SE-ONTO methodology. Where applicable, the SE-ADVISOR application model is used to guide the application design. The purpose of this approach is two-fold: On the one hand, ontology design patterns are evaluated in terms of their performance using an array of semantic web reasoners (Section 6.1). On the other hand, the SE-ONTO methodology has been applied to concrete projects in order to validate its ability to produce problem-solving ontologies (Section 6.2 and following).

In regards to the case studies, the following settings have been used: An initial case study (see Section 6.2) has been carried out in a graduate software maintenance course. The main goal of this case study was the validation of the SE-ONTO methodology in regards to its feasibility. Further, two case studies have been carried out by the Ambi-

ent Software Evolution Group (ASEG) research lab which consists of 3 Ph.D. (including the author) and 5 master's students who are familiar with ontology development. The first of those two case studies targets the analysis of bug quality and the problem of bug triage for bug-trackers (see Section 6.3). It had the goal to test the SE-ADVISOR application model and identify whether the SE-ONTO design patterns can be incorporated into ontology design and serve as re-usable building blocks. The other implements a clone detection system for source code (see Section 6.4) and had the goal to investigate the influence of decisions by different knowledge engineers (using SE-ONTO) on the final ontology design, as well as the comparison of ontology-driven application development to traditionally implemented applications. It must be noted that the outcome of these two case studies also resulted in contributions in their respective scientific domains [SRC08], [SR10], [SRC11], [Sch11].

6.1 Reasoning Pattern Performance

The performance of the introduced ontology design reasoning patterns (as defined in Chapter 4) is an important factor in the modeling of ontologies. During the design of an ontology, specific patterns can be selected due to their effect on an application's runtime performance, or a specific semantic web reasoner can be selected for the application because of a used pattern. Furthermore, the patterns also provide a baseline for evaluating different semantic web reasoners. A simple Java program can create ontologies with increasing complexity. Each ontology comes in two versions: One pattern implementation and one comparable baseline version (which models the same problem without the pattern in an ad-hoc manner). These two versions can then be compared in order to de-

Name	License Type	Language	Algorithm	Expressiveness
JFact	LGPL	Java	Tableau	DL
Fact++	LGPL	C++	Tableau	DL
HermiT	LGPL	Java	Hyper-Tableau	DL
Pellet	AGPL/Commercial	Java	Tableau	EL/DL
REL/TrOWL	AGPL/Commercial	Java	CEL	EL

Table 6.1: Used reasoners and their respective details

termine the performance impact of the pattern on the ontology design. At each point, the same amount of individuals is produced in order to allow for an evaluation of the actual impact of the pattern on the reasoning time. The runtime behavior of patterns and their ad-hoc (baseline) implementation is expected to be similar. Derivations from this behavior indicate a pattern that introduces an unnecessary complexity overhead and therefore an undesired side effect. This performance evaluation, together with the analysis of the re-usability and applicability of patterns in the following sections, determines the validity of the introduced patterns.

For the performance evaluation, several state of the art semantic web reasoners have been selected. A 2010 survey by Mishra et al. [MK10] has identified the following reasoners as relevant "modern" reasoners: DLP, FaCT++, RacerPro, Pellet, CEL, Cerebra Engine, QuOnto, KAON2 and HermiT. The semantic web reasoners' landscape is moving at a fast pace, and in 2011, the development of the following reasoners has stalled or has been discontinued: Cerebra Engine (no longer available as the company has been acquired by *Software AG*[1]), DLP (last development activity in 2001[2]) and KAON2 (last updated in 2008[3]). Active development was an important selection criterion for the eval-

[1] http://www.softwareag.com/
[2] http://ect.bell-labs.com/who/pfps/dlp/
[3] http://kaon2.semanticweb.org/

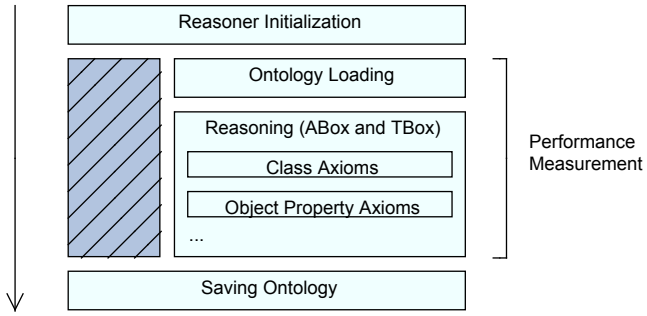

Figure 6.1: Performance measurements for evaluating patterns

uation, as OWL2 constructs are being used in most patterns and support for this latest version of the ontology language was required. QuOnto[4] only supports the OWL2 QL profile and therefore is not applicable to the patterns that are specified as part of the OWL2 EL profile. It therefore has been excluded from the tests. The remaining reasoners have been integrated in an evaluation environment based on the OWLAPI 3.0. RacerPro currently does not support OWLAPI 3.0 bindings and the available OWLAPI 2.2 bindings are insufficient due to the lack of support for the materialization of inference results. Therefore, it has also not been included in the evaluation.

Three implementations for CEL, namely the original LISP implementation[5], the Java reimplementation[6] and the newer (optimized) implementation REL/TrOWL[7], as well as two implementations for FaCT++ (the original C++ implementation[8] and a Java port[9]) have been considered, and were evaluated in an initial step. As stated by Weithoner et al.

[4]http://www.dis.uniroma1.it/quonto/
[5]http://code.google.com/p/cel/
[6]http://jcel.sourceforge.net/
[7]http://trowl.eu/
[8]http://code.google.com/p/factplusplus/
[9]http://jfact.sourceforge.net/

[WLLB06], the basic requirements for benchmarking an ABox/TBox are that "all reasoning results from any benchmark should always be checked for soundness and completeness [and] we suggest to start every ABox benchmarking session with a rigorous check of the overall reasoning capabilities of the involved reasoners". This check has led to discarding the two CEL implementations for an incomplete implementation of the OWL2 EL profile, in particular because they do not support ABox reasoning or advanced OWL2 features, such as property chains. Only REL/TrOWL is conforming to the OWL2 EL profile and therefore has been selected as an representative CEL-based reasoner. The remaining reasoners with acceptable results for OWL2 EL/DL (shown in Table 6.1) are: REL/TrOWL (Version 0.7), JFact (Version 0.7), FaCT++ (Version 1.5.2), HermiT[10] (Version 1.3.4) and Pellet[11] (Version 2.2.2). Except for REL/TrOWL which uses the CEL algorithm, all reasoners use a tableau-based reasoning algorithm (with HermiT using a specific hypertableau implementation).

All experiments have been performed on a Windows machine with the maximum memory allowed (by the virtual machine), set to 2GB. Values ought to be seen holding an inaccuracy of $\pm$10ms due to the limitation of the system clock and scheduling of the operating system. Results exceeding the memory limit are set to `MEM`. Incomplete results (i.e. not all object properties were inferred) are set to `INC`. Results that exceed the time limit of 600s are marked with $>$600s. Each test has been repeated three times whereby the best value, rounded to one digit after the comma, is selected. As with other evaluations, it must be noted that results always depend on the underlying implementations, and can only represent an approximate guidance into how patterns affect the performance of

[10] http://hermit-reasoner.com/
[11] http://clarkparsia.com/pellet/

	Limited Transitivity					
	10		100		1000	
Reasoner	Baseline	Pattern	Baseline	Pattern	Baseline	Pattern
JFact	1.8s	2.2s	45.2s	226.5s	>600.0s	>600.0s
FaCT++	0.5s	0.5s	11.3s	53.9s	>600.0s	>600.0s
HermiT	1.4s	1.5s	5.5s	5.8s	33.3s	34.2s
Pellet	0.7s	0.7s	1.6s	2.1s	6.9s	7.6s
REL/TrOWL	0.4s	0.4s	1.1s	1.6s	5.2s	7.6s

Table 6.2: *Limited Transitivity* pattern performance

different reasoners; no restrictions can be set on what specific steps are executed internally by a reasoner. While the saved end-result corresponds to a complete materialized ontology (including all inferences), reasoners might, for example, optimize/index the ontology to improve query performance. Figure 6.1 shows the different steps taken for the evaluation: reasoner initialization, ontology loading, reasoning (classifyng the TBox, realizing the ABox, etc) and saving the ontology with the materialized inferences to disk for inspection. For the evaluation, loading and reasoning times have been combined, and adjusted by the initialization time for the different reasoners.

6.1.1 Limited Transitivity Pattern

The *Limited Transitivity* pattern replaces transitivity by a combination of property chains. While it is clear from the example in Section 4.3 that this can limit the amount of transitive relations to be inferred and therefore can speed up an application, it is unclear how the pattern compares against the same amount of relations created through transitive relationships. Therefore, in this evaluation, the amount of individuals is gradually increased for both transitive relations (in the baseline ontology) and the property chains in the *Limited Transitivity* pattern ontology.

	Restriction Generalization					
	10		100		1000	
Reasoner	Baseline	Pattern	Baseline	Pattern	Baseline	Pattern
JFact	0.5s	0.7s	3.2s	3.9s	330.0s	530.1s
FaCT++	0.2s	0.2s	0.3s	0.3s	73.3s	73.7s
HermiT	0.2s	0.3s	1.0s	1.1s	21.2s	65.0s
Pellet	0.6s	0.7s	1.3s	163.0s	22.8s	>600.0s
REL/TrOWL	0.3s	0.3s	0.6s	0.7s	3.6s	8.2s

Table 6.3: *Restriction Generalization* pattern performance

As shown in Table 6.2, the pattern performs well on REL/TrOWL as well as Pellet (which seems to be optimized for this type of inference). In contrast, JFact and FaCT++ do not scale well for this property chain and therefore the pattern should only be used with reasoners that are optimized for the OWL2 EL profile. It is, however, a valid ontology design pattern that can perform well.

6.1.2 Restriction Generalization Pattern

In this test, the baseline ontology contains a $hasValue$ restriction. The *Restriction Generalization* pattern transform such a restriction into a $someValuesFrom$ restriction with a new concept. As shown in Table 6.3, Pellet does not scale well to larger ontologies (number of concepts and individuals). In contrast, FaCT++ and HermiT perform much better, handling significantly larger ontologies.

As the *Restriction Generalization* pattern remains in the OWL2 EL profile, it can be handled by the CEL algorithm which shows a remarkable performance improvement for larger ontologies. REL/TrOWL is an order of magnitude faster than any tableau-based algorithms. The introduction of a new concept and the $someValuesFrom$ restriction slows down reasoning results depending on the used reasoner. Applying the *Restric-*

	Property-Class Commonality					
	10		100		1000	
Reasoner	Baseline	Pattern	Baseline	Pattern	Baseline	Pattern
JFact	0.5s	0.6s	1.7s	2.0s	190.0s	289.3s
FaCT++	0.3s	0.3s	0.6s	0.7s	50.s8	74.3s
HermiT	0.4s	0.5s	0.6s	0.6s	7.3s	9.4s
Pellet	0.5s	0.5s	0.6s	0.6s	2.8s	4.2s
REL/TrOWL	0.3s	INC	0.6s	INC	2.9s	INC

Table 6.4: *Property-Class Commonality* pattern performance

tion Generalization pattern therefore has to be weighed against restricting reasoning to the OWL2 EL profile (as opposed to the OWL2 DL profile). When using an OWL2 DL reasoner the pattern should only be applied to a low number of individuals.

6.1.3 Property-Class Commonality Pattern

The *Property-Class Commonality* pattern adds a $hasSelf$ restriction to an ontology, which allows the inference of a new object property. In order to evaluate the impact of the added restriction, the pattern is compared against an ontology where the new object property is already present (and no inference is needed). It is therefore expected to see a linear increase in reasoning time for the baseline version of the ontology. As $hasSelf$ is not implemented in REL/TrOWL and therefore yields incomplete results, only tableau-based reasoners were compared in this evaluation.

In general, the run-time complexity of the pattern (shown in Table 6.4) is increasing as expected, except for the JFact and FaCT++ reasoners, which perform poorly for larger loads of object properties (with and without the $hasSelf$ restriction and reasoning). This indicates, that the FaCT family of reasoners might not be well-suited when one has to deal with a large amount of interlinked data. The pattern performs well on both tableau-

	Hierarchy Creation					
	10		100		1000	
Reasoner	Baseline	Pattern	Baseline	Pattern	Baseline	Pattern
JFact	0.3s	0.4s	0.8s	1.3s	4.2s	25.8s
FaCT++	0.2s	0.2s	0.3s	0.4s	1.2s	6.4s
HermiT	0.4s	0.4s	0.7s	0.7s	1.4s	1.5s
Pellet	0.2s	0.3s	0.4s	0.6s	0.9s	4.4s
REL/TrOWL	0.2s	0.2s	0.3s	0.3s	0.8s	0.8s

Table 6.5: *Hierarchy Creation* pattern performance

as well as CEL-based reasoners and therefore is a valid ontology design pattern.

6.1.4 Hierarchy Creation Pattern

Reasoning in the *Hierarchy Creation* pattern involves two steps: (1) Identifying that an individual is part of a hierarchy (concept) through a $someValuesFrom$ restriction. (2) Identifying recursively that another individual is part of the same hierarchy.

In contrast to the *Restriction Generalization* pattern, a single $someValuesFrom$ and $hasValue$ restriction is "executed" multiple times on different individuals (instead of introducing multiple restrictions). It is therefore of interest to evaluate the impact of this "recursive" classification on the reasoning performance. The pattern is compared against a baseline ontology in which individuals are asserted to be part of a hierarchy (and no reasoning is required).

Results of the performance analysis are shown in Table 6.5. The pattern does not have any major negative effect on the run-time performance of HermiT and REL/TrOWL. Although the CEL-based REL/TrOWL implementation is slightly faster, it is overall on par with the hypertableau-based HermiT. JFact, FaCT++ and Pellet do not perform as well and the pattern should be applied with care when selecting those reasoners.

6.2 Software Maintenance Case Study

This initial case study was carried out in a graduate software maintenance course setting (with a total of 18 students) that had been given an introduction into Knowledge Engineering, and the development of ontology-driven applications, using the SE-ONTO methodology. The students were largely unfamiliar with ontologies at the time of the course and only a short introduction into ontologies and ontology design patterns was possible. The following was described to the students:

- SE-ONTO requirements analysis
    - Noun and adjectives analysis
    - Entity analysis diagram
    - Entity refinement
- SE-ONTO ontology development
    - Ontologies as a form of knowledge modeling (subject:predicate:object)
    - Outline of patterns and inference results through reasoning

The SE-ADVISOR application model was unfinished and thus not part of the evaluation. The aim of the case study was to expose students to the requirements analysis phase of the SE-ONTO methodology, and gather initial data on the feasibility of the SE-ONTO ontology development phase. In order to evaluate SE-ONTO, the following quality criteria, which are inspired by general Software Engineering quality criteria (as discussed in [ISO01] and [Hoy01]), were selected:

- Thoroughness - Are different problems solvable using the methodology?

- Reliability - Do different users of the methodology reach the same (or similar) goal?
- Effectiveness - Can the methodology support the knowledge engineer in building an ontology (connecting the dots between requirements and design)?
- Comprehensibility - Are the individual steps in the methodology well explained and complete?

For this experiment, students were split into groups that had to perform the following assignment: "Develop a system that supports the IEEE maintenance process". Groups were initially provided with a set of user stories. This is in agreement with the SE-ONTO methodology that starts with the collection of user stories with a customer. The following is an excerpt from the list of user stories:

> "As a user, I want to know the type of maintenance being performed. There exists corrective maintenance, adaptive maintenance as well as perfective and emergency maintenance. ... As an administrator, I want to see all maintainers working on a project. ... As a user, I want to see all methods affected by a commit in the revision system. A method is part of a class. ... As a user, I want to identify all methods that invoke another method (through 3 levels). ..."

The case study included an anonymous questionnaire, which was submitted online by each of the students, in order to address the evaluation of the defined quality criteria. The participants had to answer questions in the following categories: thoroughness (T), reliability (R), effectiveness (E) and comprehensibility (C). As some students did not

follow the methodology, the results from the questionnaire were split: Questions marked with * were answered by those students that were using the SE-ONTO methodology. Question marked with Q_a could be answered with "fully disagree" (FD), "somehow disagree" (SD), "somehow agree" (SA), "fully agree" (FA) and "cannot say" (NN). Question marked with Q_b could be answered with "yes" or "no". The following questions were asked:

- $T0_a$. Do you think there is a need for a detailed methodology in building an ontology-driven application?
- $T1_a$*. Do you think the provided patterns are sufficient to model the application problem?
- $T2_a$*. Do you think that additional ontology design patterns could be specified?
- $R0_b$. Do you think there exist other solutions that are better suited to the specified problem? If yes, what kind of solutions?
- $R1_b$*. Did you at any point feel you could have made a different choice in following the steps of the methodology?
- $E0_a$. Did you find the application was built in a fast and straight-forward way?
- $E1_a$*. Did you find the methodology helpful in transforming the requirements into an application?
- $C0_a$*. Did you find the individual steps in the methodology well explained and complete?

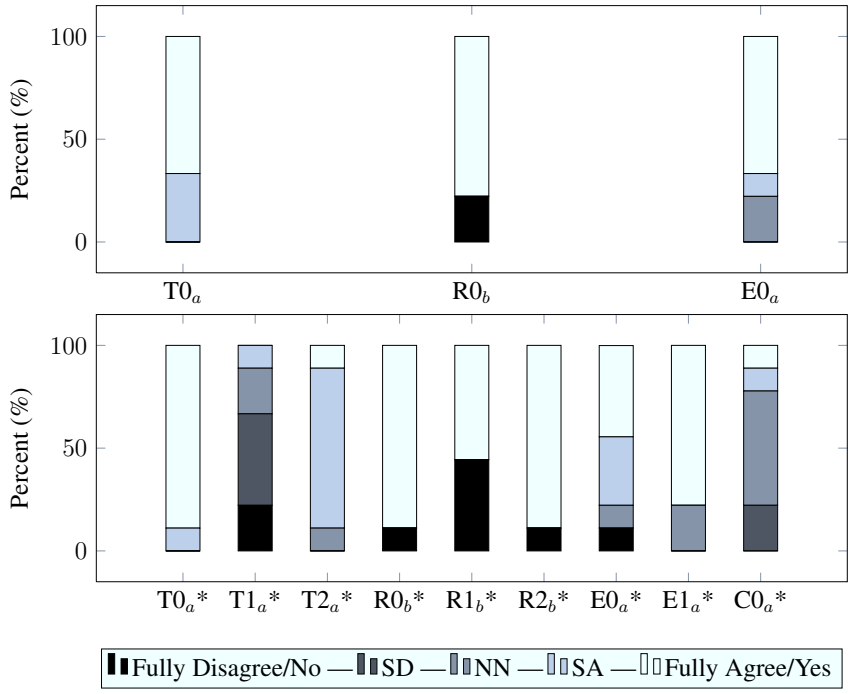

Figure 6.2: Software maintenance case study results

The collected data from the questionnaires is shown in Figure 6.2. Results from the study show a consent on the need for a detailed methodology (T0), which was answered "fully agree" by 89%*/67% of the participants. The answers to T1 show that the provided patterns were not regarded as sufficient (44% answered "somehow disagree"), which is explainable by the preliminary nature of the study that did not include many details on patterns. However, most participants agreed on the possibility that more patterns could (and should) be specified (T2). R0 was specifically encouraging as most participants suggested an ontological approach as a good solution for the specified problem (89%*/78% answered "yes"). R2 indicates that SE-ONTO is well-defined and has clear instructions; although students noted that "the selection of patterns is left up to the user, which can result in difficulties". This is also reflected in C0 that most people (56%) answered with "cannot say". E0 shows a slightly lower satisfaction of users that employed the SE-ONTO

methodology (44% versus 67%), which can be attributed to the overhead of trying to apply an unfamiliar methodology. It is nevertheless a value that, together with question E1 (78%), can be seen as a positive indication for the effectiveness of SE-ONTO.

Although participants could only evaluate part of SE-ONTO, there was an overall consensus that a methodological approach (like the SE-ONTO methodology) is needed for application-specific ontology design. In the following sections, more complex application problems are tackled, and the ontology development process of the SE-ONTO methodology is further evaluated.

6.3 Bug Quality and Triage Case Study

In this section, a case study, which was performed in conjunction with the Defence Research and Development Canada (DRDC), is presented. The DRDC thereby acted as a customer for a system, which had the goal of identifying bug quality and facilitate bug triage. The system was developed using the SE-ONTO methodology by two students of the ASEG research lab. The application proposed by the DRDC was selected as a case study since it fits the SE-ADVISOR application model (no real-time reasoning is used and non-ontological data can be read incrementally). Results from the case study have been published in the International Workshop on Semantic Web Enabled Software Engineering [SRC08] and as an invited book chapter [SR10].

The following sections provide a short introduction into the research field of bug-trackers, bug quality and triage before describing the ontology design and implementation aspects of the case study. The case study description finishes with a validation of the achieved results.

6.3.1 Background

A bug-tracker represents a repository for reporting and retrieving error reports. Bug-trackers store error reports in a structured form and offer advanced means to search within them. While the original purpose of bug-tracking-systems has been to manage bug reports, their usage meanwhile has shifted to include all kinds of data such as: feature requests, improvements and general tasks [ZK02]. Due to their more general usage, bug trackers are nowadays more appropriately referred to as issue-trackers. Key questions of the research community in the area of issue-trackers are:

- How do we know what issue/bug to fix first?
- Who should we assign an issue/bug to (bug triage)?
- What information can be mined from issues/bugs?

Existing work on analyzing bug reports has shown that many bug reports contain invalid or duplicate information [AHM05]. For the remaining ones, a significant portion tends to be of low quality [ZBPS09]. As a result, many of these bug reports end up being treated in an untimely or delayed manner [BPZK08]. Providing an automated or semi-automated approach to evaluate the quality of bug reports can provide an immediate added benefit to organizations, which often have to deal with a large number of bug reports. More recently, there has been work on bug reporting systems that can extract information from written reports through the means of text mining [BJS$^+$08]. Text mining (also referred to as "knowledge mining") corresponds to the process of deriving non-trivial, high quality information from unstructured text that is typically derived through the division of patterns and trends through means such as statistical pattern learn-

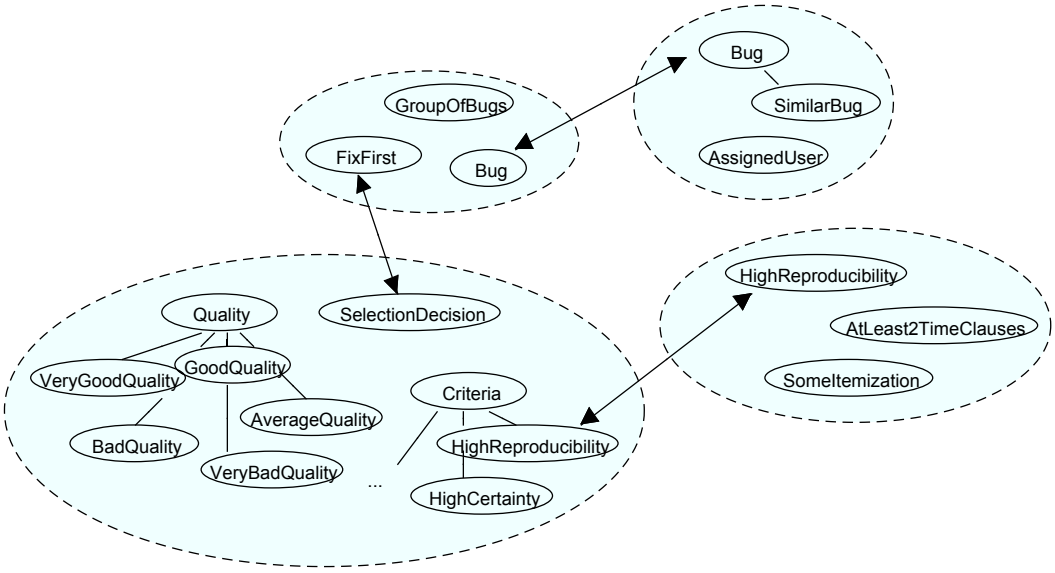

Figure 6.3: Entity analysis diagram for bug quality case study

ing [FS06]. Unlike Information Retrieval (IR) systems [BYRN99], text mining does not simply return documents pertaining to a query, but rather attempts to obtain semantic information from the documents using techniques from NLP and AI [MJ08]. Text entered by users remains a primary source to locate and eliminate errors in a system [ZZWD05]. It is therefore of interest to analyze such text in order to identify bug report quality. Quality criteria (such as focus, reproducibility, etc.) can then be used to guide a maintainer in selecting the next bug to fix. Additionally, bugs that share similar named-entities can be identified as being similar to each other, in order to facilitate the detection of duplicate entries. A bug that is similar to an already fixed one can be suggested to be assigned to the same maintainer (bug triage).

6.3.2 Ontology Design

The ontology design phase of the case study was implemented by two members of the ASEG research lab familiar with ontology development. In the following, the ontology design process (using the SE-ONTO methodology and spanning across four iterations) is described in detail. The SE-ADVISOR application model and server were used to develop, test and deploy the application.

Iteration 0

In a first step, user stories were selected as being realized through application logic or through ontology design (as the ontology's responsibility). Most of the NLP related tasks, such as named-entity detection and identification of textual quality, were selected to be realized in application logic. The following are excerpts of the user stories created for the application together with the DRDC, which show some of the relevant user stories that were selected as the ontology's responsibility:

> "... (#2) As a user, I need to see what groups of bugs exist, in order to fix one first. The program must be able to identify groups of bugs (parent/child relationship)... (#5) As a user, I can display bugs in four qualities (very good, good, average, poor and very poor) to make a selection decision. A very good quality bug has certain criteria such as high certainty, high focus, high reproducibility and high observability. ... (#6) As a user, I want to identify bugs with high reproducibility. A bug with high reproducibility uses at least two time-clauses or has some itemization. ... (#7) As a user, I should be assigned a bug, if I have fixed a similar one. A similar bug is a maximum of

two levels of similarity away. ...”

Figure 6.3 shows the created entity analysis diagram (for the selected user stories) that followed the noun and adjectives analysis. Each user story was further analyzed and refined, and new vocabulary was added to the diagram when needed. The refinement, partly shown in Listing 6.1, helped to clarify that there exists multiple instances for certain entities (e.g. `Bug` or `TimeClause`). The refined user stories also formed the basis for the unit tests that were performed later during the development. The refinement tends to produce verifiable rules which also could be maintained in an expert system shell but this was omitted to not complicate the case study.

Listing 6.1: Bug quality case study entity refinement

```
If Bug1 has AtLeast2TimeClauses and SomeItemization
then it has HighReproduceability
If Bug3 has TimeClause1 and TimeClause2 and Bug2 has the same
then they are SimilarBug
If Bug4 has HighReproduceability and HighCertainty and HighFocus and HighObervability
then it is VeryGoodQuality
If Bug1 is a child of Bug3
then they are part of the same GroupOfBugs3
...
```

Iteration 1

In the first iteration, the user story #7 (with `SimilarBug`, `AssignedUser` and `Bug`) was selected to be formalized. The refined user story already indicated that bugs should be represented as individuals during the ontology design. The goal for the user story was identified as “detecting similar bugs”. Based on the SE-ONTO methodology, a design pattern was selected to model this application goal. In this case, the *Equivalence*

Similarity pattern was selected, as it allows to add a similarity relation between different individuals. This resulted in the following definitions (shown here for two bugs):

$$\begin{aligned} BUG_1 \equiv\ & HAS2PARTS \sqcap \exists hasPart.\{TimeClause_1\} \sqcap \\ & \exists hasPart.\{Itemization_1\} \\ BUG_1 \sqsubseteq\ & \exists similarTo.\{Bug_1\} \\ BUG_2 \equiv\ & HAS2PARTS \sqcap \exists hasPart.\{TimeClause_1\} \sqcap \\ & \exists hasPart.\{Itemization_1\} \\ BUG_2 \sqsubseteq\ & \exists similarTo.\{Bug_2\} \end{aligned}$$

$$HAS2PARTS(Bug_1)$$

$$HAS2PARTS(Bug_2)$$

Once similar bugs were connected through this relation, the `AssignedUser` could be modeled as an object property that is "moved" along the relation using a property chain axiom:

$$similarTo \circ assignedTo \sqsubseteq assignedTo$$

$$assignedTo(Bug_2, User152452)$$

The resulting ontology design modeled most parts of the user story, except the part of task descriptions which mentions "two levels of similarity". As suggested by SE-ONTO, another design pattern was searched and applied. In this case, the *Limited Transitivity* pattern was used (as it can model levels using the existing similarity relation). The

$similarTo$ relation was refactored into $similarToLvl0$ and the following definitions were added to automatically infer the level of similarity through the reasoner:

$$similarToLvl0 \sqsubseteq similarTo$$

$$similarToLvl1 \sqsubseteq similarTo$$

$$similarToLvl0 \circ similarToLvl0 \sqsubseteq similarToLvl1$$

The newly refactored model successfully captured all the requirements from the user story and completed the design iteration. The formalized statements were cast into a mapping task and deployed to the SE-ADVISOR application server together with test cases.

Iteration 2

As part of the next iteration, user story #2 (with entities `GroupOfBugs`, `Bug` and `FixFirst`) was selected. Again, SE-ONTO suggested the identification of an applicable pattern. The description "must be able to identify groups of bugs (parent/child relationship)" was interpreted as a hierarchy and consequently the *Hierarchy Creation* pattern was selected. As a concept for each possible (bug) group was already present, the following definitions from the pattern were merged with the ontology (adding a new concept per bug that acts as the hierarchy root):

$$BUG_1CHILD \equiv \exists childOf.BUG_1$$

$$BUG_1CHILD \sqsubseteq \exists fixFirst.\{Bug_1\}$$

$$BUG_2CHILD \equiv \exists childOf.BUG_2$$

$$BUG_2CHILD \sqsubseteq \exists fixFirst.\{Bug_2\}$$

With this definition, every child of a bug is also part of the same `GroupOfBugs` when $childOf$ relations are added (and also inherits the $similarTo$ relation based on previous definitions). This facilitated the modeling of the "group requirement" of the user story. Statements were again cast into a mapping task and deployed to the SE-ADVISOR application server together with test cases.

Iteration 3

For the third iteration, user story #6 (with the entities `HighReproducability` as well as `SomeItemization` and `AtLeast2TimeClauses`, etc.) was formalized. The term `SomeItemization` can be designed by refactoring the ontology to differentiate the $hasPart$ property. The object property was split into multiple subproperties that are called $hasItemization$, $hasTimeClause$, etc. The term `AtLeast2TimeClauses` could be modeled in the ontology using OWL DL, but based on the possible large number of bugs added to the system (and considering the performance analysis from Section 6.1), it was decided to stay within the OWL2 EL profile. Therefore, the *Representative Individual* pattern was selected and the responsibility of identifying a bug as having at least 2 time clauses (`AtLeast2TimeClauses`) was transferred to the application logic. The application logic has to add X individuals and relations in the form $hasAtLeast\ 2TimeClauses$, $3TimeClauses$, etc., depending on the number of time clauses detected, yet bound by a user-definable maximum. The OWL2 EL profile also disallows object unions. Therefore, the *Subclass Disjunction-Like* pattern had to be applied to model the sentence "A bug with high reproducibility uses at least two time-clauses or has some

itemization". Although, no concept disjunction is possible within the OWL EL profile, the *Subclass Disjunction-Like* pattern allows for the classification of two individuals under the same "union concept" (in the following $HIGHREPRODUCABILITY$) as shown in the following exemplifying statements that group the NLP properties $hasItemization$ and $hasTimeClause$ together:

$$\begin{aligned} hasItemization &\sqsubseteq hasPart \\ hasTimeClause &\sqsubseteq hasPart \\ SOMEITEMIZATION &\equiv \exists hasItemization.\top \\ SOMEITEMIZATION &\sqsubseteq HIGHREPRODUCABILITY \\ ATLEAST2TIMECLAUSES &\equiv \exists hasAtLeast.\{2TimeClauses\} \\ ATLEAST2TIMECLAUSES &\sqsubseteq HIGHREPRODUCABILITY \end{aligned}$$

These definitions, together with additional definitions for certainty, focus and observability (as described in the following section), capture all knowledge described by the user story and thus concluded the third design iteration.

Iteration 4

During a final design iteration, the definitions for `VeryGoodQuality`, as well as `GoodQuality`, etc. from user story #5 were added as disjoint concepts of their respective criteria concepts:

$$\begin{aligned} VERYGOODQUALITY \equiv\ & HIGHREPRODUCABILITY \sqcap \\ & HIGHCERTAINTY \sqcap ... \end{aligned}$$

The fourth iterations successfully transformed the supplied user stories to a complete ontology design by applying the SE-ONTO methodology. By passing all unit tests on the application server, the SE-ONTO ontology design and development phase concluded.

Summary

In summary, the different applied iterations of the SE-ONTO methodology incrementally transformed user stories into an application-centric ontology that can be consumed through the SE-ADVISOR application server. As part of the presented case study, the following ontological reasoning patterns were applied:

1. *Equivalence Similarity* pattern
2. *Limited Transitivity* pattern
3. *Hierarchy Creation* pattern
4. *Representative Individual* pattern
5. *Subclass Disjunction-Like* pattern

Based on selected queries (e.g. "Select all bug individuals that are of 'VeryGoodQuality' "), application logic can communicate with the ontology to display information about bugs.

6.3.3 Application Logic

As discussed in the previous section, parts of the requirements extracted from the user stories are captured by application logic rather than through the ontology design. For the bug report case study, the responsibility of the SE-ADVISOR mapping task is to provide the following concepts, individuals and properties for the ontology:

- An individual for each bug with a unique name

    - $childOf$ relation to any parent

    - $hasItemization$ (and others) for NLP quality attributes

    - $hasAtLeast$ relation for each time clause,...

    - $assignedTo$ relation for the assigned maintainer

- A concept for each bug with a unique name

    - $fixFirst$ relation

In order for the application to access different bug-trackers, code from the Eclipse Mylyn[12] open-source project was used. While most of the required information can be simply extracted from bug-tracker database fields, the detection of quality attributes from the textual description of a bug is more complicated; in order to analyze bugs for bug report quality, the textual description of bugs has to be mined using NLP. NLP systems are often implemented using component-based frameworks, such as Apache's UIMA[13] (Unstructured Information Management Architecture) or GATE[14] (General Architecture for Text Engineering), where the latter has been selected in this implementation. As the developed approach is a novel technique and therefore a contribution in itself, details are presented here.

Existing work on analyzing bug reports has shown that bug reports provide a number of distinctive characteristics, which allow developers to judge their quality. The quality of a bug report thereby largely depends on its helpfulness in identifying and understanding

[12]http://www.eclipse.org/mylyn/
[13]http://uima.apache.org/
[14]http://gate.ac.uk/

the reported problem. A survey performed by Bettenburg et al. in [BJS+07] shows that the most important properties developers are looking for in a bug report are: the steps to reproduce the problem, followed by stack traces, test cases, screenshots, code examples and a comparison of observed versus expected behavior. Additionally, bug report guidelines have been formulated to describe the characteristics of a high quality bug report[15]:

- Be as precise as possible
- Explain it so others can reproduce it
- Only describe one bug per report
- Clearly separate fact from speculation

From the previously mentioned characteristics and suggestions published by Bettenburg et al. ([BJS+08] and [BPZK08]), the following NLP activities for extracting quality attributes from bug trackers have been defined:

Certainty: The level of speculation found in a bug description must be low. A high certainty indicates a clear understanding of the problem, and often also implies that the reporter can provide suggestions on how to solve the problem. Kilicoglu et al. show in [KB08], [KB10] that hedges can be found with high accuracy using syntactic patterns and a simple weighting scheme. The used gazetteer lists have been provided by the authors and were utilized to identify hedges in this case study.

Focus: The bug description must not contain any off-topic discussions, complaints or personal statements. Only one bug is described per report. Therefore, in this case

[15] http://www.chiark.greenend.org.uk/ sgtatham/

study, the focus of bug reports was assessed by identifying emotional statements (such as “exciting”), as well as topic splitting breaks (such as “by the way” or “on top of that”) through a gazetteer list.

Reproducibility: The bug report description must include steps to reproduce a bug, or the context under which a problem occurred. By manually evaluating 500 bug reports, time clauses used in bug descriptions could be identified as a reliable hint for paragraphs describing the context where a problem occurred. For example: “When I clicked the button” or “While starting the application”. Such statements were annotated using a Part of Speech (POS) tagger and Java Annotation Patterns Engine (JAPE) grammar. To identify the listing of reproduction steps, the standard GATE sentence splitter was modified to recognize itemizations (characters “+” “-” “*”) as well as enumerations (in the form of “1.” “(1)” “[1]”).

Observability: The bug report must contain a clearly observed (positive or negative) behavior. Evidence of the problem, such as screenshots, stack traces, or code samples, must be provided. To identify observations in bug descriptions, word frequencies have been compared with the expected numbers from non-bug related sources. For words appearing distinctively more often than expected, a categorization in positive and negative sentiment was performed for the case study implementation. Table 6.6 shows a sample of identified words and their sentiments. A gazetteer list was used to annotate both positive and negative observations.

Type	Examples	Total
Neg. Noun	attempt, crash, defect, failure, ...	22
Neg. Verb	disappear, fail, hang, ignore, ..	32
Neg. Adj.	broken, faulty, illegal, invalid, ..	34
Pos. Verb	allow, appear, display, found, ...	24
Pos. Adj.	correct, easy, good, helpful, ...	16

Table 6.6: Sentiment analysis examples

6.3.4 Validation

In order to validate the modeled properties in this case study, the resulting application was tested with an open-source bug-tracker. ArgoUML[16], a leading UML editor, that has since its inception in 1998 undergone several release cycles and is still under active development, had been selected for this purpose. Its bug database counts over 5100 open/closed defects and enhancements. The validation of the case study was two-fold: First, it was shown that the proposed NLP quality characteristics can be used to classify bugs. For this purpose, manually annotated bugs were compared against automatically classified bugs. Second, the performance of the case study had to be validated as sufficient for a bug-tracker. This was measured through a performance comparison using different semantic web reasoners.

Seven experienced Java developers (Master's and Ph.D. students who have previously worked with ArgoUML at the source code level) had been asked to participate in this study and to complete a questionnaire assessing the quality of bugs. For each of the selected bugs, the users performed an evaluation of the bug report quality using a scale ranging from 1 to 5 (with 1 corresponding to "very high quality" and 5 to "very low quality"). The evaluation was performed as part of a course assignment, with students

[16] http://argouml.tigris.org/

	100	1000	5000
FaCT++	35.2s	758.6s	>1000.0s
HermiT	5.4s	120.5s	542.0s
Pellet	23.9s	731.0s	>1000.0s
REL/TrOWL	3.5s	12.3s	19.3s

Table 6.7: Bug-tracker quality case study performance validation

being given one week to complete the assignment.

For the first part of the evaluation, 178 bugs were manually classified from having "very good quality" to having "very bad quality" in terms of their textual description of the bug. The results were then compared against the classification provided by the implementation. The classification performance thereby reached 81%, which is high enough to justify the use of the implementation as a supporting tool in bug-triage.

For a performance analysis of the implementation (shown in Table 6.7) the number of processed bugs was scaled from 100 to all bugs in the bug-tracker (around 5000), which successfully demonstrated the scalability of the implementation. Both HermiT and REL/TrOWL were able to materialize inferences for the whole bug-tracker in under 10 minutes. The application developed for the case study is therefore valid in terms of its performance.

The case study demonstrated the feasibility of using the SE-ONTO methodology for the development of ontology-driven applications. The successful use of the defined ontology design patterns and reasoning services are indicators that the proposed guidelines and work products of the methodology are sufficient to develop such applications. The resulting ontology incorporates part of the business logic of the application which can be easily modified (to change the runtime behavior of the application). This is a clear advantage in comparison to traditional software development in which such business

logic is "hard-coded" into the application. For example, a change in what constitutes a "high quality bug" can be performed easily without recompiling or re-deploying the application. Further it is also possible to define new criteria that lead to such a high quality bug without modifying the original application code. In the following section, an additional case study evaluates whether an application developed using SE-ONTO and the SE-ADVISOR application model can compete with applications developed without ontologies.

6.4 Clone Detection Case Study

The clone detection case study, described in this section, had the goal to "develop a clone detection approach that is capable of discovering inter-project source code clones in open-source projects". The study was performed together with (and for) the DRDC, who provided the user stories and acted as a customer for the evaluation of the SE-ONTO methodology. The DRDC's goal was an application that was capable of reading and parsing source code files (incrementally) from open-source repositories, and was able to link information in a knowledge repository to identify source code clones.

The project was taken on by the ASEG research lab, as it fitted the SE-ADVISOR application model; source code (structured non-ontological information) is transformed into an knowledge repository (mapping) where it is processed to detect clones (through the use of reasoning services) and can be queried. Moreover, it is an incremental process that can make use of the OWA to not infer any information about the source code that has not yet been parsed. The goal of the case study was the comparison of an ontology-driven application developed using the SE-ONTO methodology with existing

non-ontological applications (clone detections tools). The assumption that an ontology-driven application can be developed with less effort/complexity (measured in lines of code) using the SE-ONTO methodology was evaluated. Additionally, the case study investigated whether the SE-ONTO methodology leads to identical ontologies between different knowledge engineers. For this purpose, the case study was developed by two students of the ASEG research lab in parallel. Results from the case study have been published in the International Workshop on Software Clones at ICSE'11 [Sch11] and the IEEE Computer Software and Applications Conference (COMPSAC'11) [SRC11].

The following sections provide a short introduction into the research field of clone detection before describing the ontology design and implementation aspects of the case study. The case study concludes with a validation of the achieved results.

6.4.1 Background

A *code clone* is a source code fragment that is identical or similar to another one [Kap09]. Clone detection techniques can generally be grouped by their representation of source code that is used to match code fragments [RC07]. String-based clone detection tools compare files, without taking into consideration their underlying semantics, and therefore have the advantage of working on any kind of file; strings are loosely matched in order to account for changing variable names or missing code. Token-based approaches transform text into language specific tokens that can be matched using distance measures. Similarly, Abstract Syntax Tree- (AST) based methods include the semantics of the underlying code by fully parsing its structure according to the language specifications and generating an AST. Then, sub-tree matching is performed to identify potential clones. At a byte-code (or machine-code) level, one can identify clones by comparing compiler optimized

instructions.

The following lists examples of implementations (ordered by granularity and representation type):

- String (e.g. *Simian* [Har03], *Duploc* [DRD99])
- Token (e.g. *JPlag* [PMP00], *CCFinder* [KKI02])
- AST (e.g. *CloneDr* - approach by Baxter et al. [BYM+98])
- Byte-code (e.g. *JCD* [DG10])

Various studies exist comparing different clone detection methods. Burd and Bailey [BB02] evaluated five clone detection techniques for use in software maintenance activities. Their findings suggest that *CCFinder* has one of the highest recognition rates of token-based tools. Koschke [KFF06] analyzed various clone detection tools and found that AST-based methods, such as *CloneDr*, have the highest precision, while token-based approaches offer a better recall. In [WJL03], it is argued that clone detection studies suffer from a lack of objectivity when annotating what constitutes a clone, since human reviewers are used. This finding is also noted by Kapser in [Kap09] who gives an excellent overview of currently used techniques, and provides and empirical evaluation of code clone patterns. In terms of large-size code clone analysis, a distributed *CCFinder* has been implemented by Liverie et al. [LHMI07].

A crucial factor when trying to find duplicate code in large amounts of data is the granularity of the target source code elements. The challenge is to find a compromise between expressivity and size, to ensure that good precision and recall values are maintained (and to ensure that parallel processing is still possible). For the purpose of the

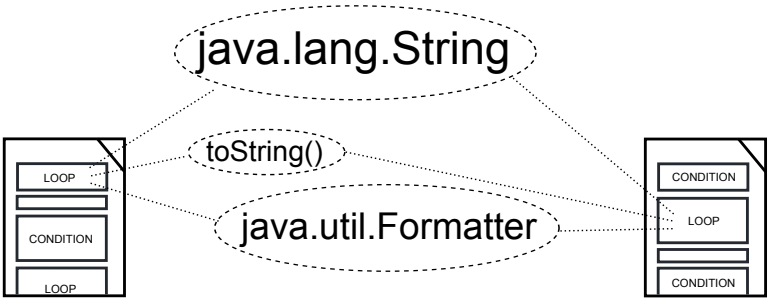

Figure 6.4: Clone detection approach

case study, functions/methods were identified as the most coarse grained element of interest, and control-blocks (conditions and loops) as the most fine grained elements that are compared against each other. The clone detection approach itself is based on the idea of comparing used data types and called methods of a block, as a compact but distinguishing factor between code (shown in Figure 6.4). While this approach is similar to AST-based implementations and also relies on building an AST, the information used to identify what constitutes a clone is different. Instead of loosely matching AST identifiers, operations and expressions over a complete class, the approach in this case study only compares the control-block signature (the set of used data types and method calls as well as the control-block type) to signatures in other classes. Although this method is simpler than a full exhaustive search for code clones, the list of clones not covered by this procedure remains manageable: (1) Clones larger than a method; these can only be indirectly identified by determining that multiple blocks/methods are identical. (2) Clones smaller than a control-block and (3) non-code clones.

The comparison of blocks in this case study is based on used data types and called methods, and therefore has an immediate benefit — the automatic invariance to certain code changes that are typical for code clones (type 1 and 2 [RC07]):

- Code order and the use of parentheses
- Renaming of identifiers
- Change of arithmetic operations and literals
- Formatting (spaces, etc.) and comments

Summarizing the detection approach, the following holds: If a block calls the same methods and uses the same data types as another block, it is a clone of the other block. Within control-blocks, switch and if statements are mapped together as "conditions", and for and while statements as "loops", as they can be expressed interchangeably. This approach suggested an ontology design in which cloned methods and blocks are identified as equivalent concepts (or identified as being the same individual).

6.4.2 Ontology Design

The ontology design phase of the case study was implemented by two students of the ASEG research lab familiar with ontology development. The ontology was designed in parallel in order to investigate the effect of selecting user stories and design patterns in the SE-ONTO methodology. The two different iteration cycles are marked as S_1 for the first, and S_2 for the second student. The students were asked to keep a log of their design decisions for future comparison of the rational behind them.

Iteration 0 / $S_{1,2}$

As a first step, user stories were collected and refined with the DRDC. The following excerpts show some of the user stories created for the application:

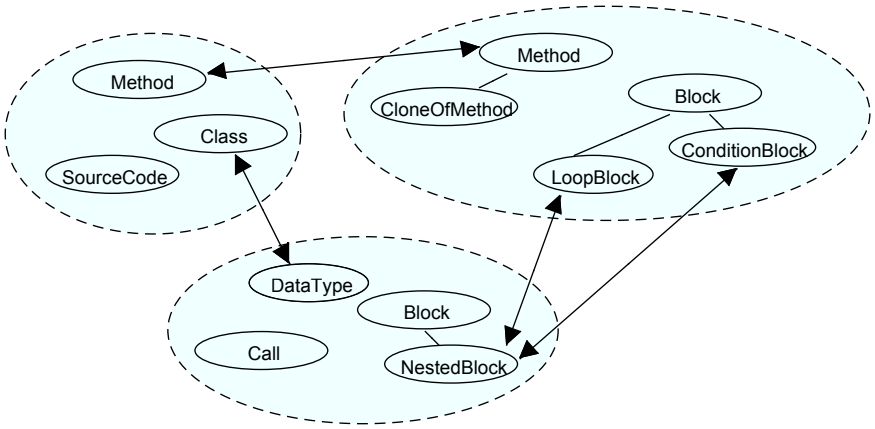

Figure 6.5: Entity analysis diagram for the clone detection case study

> "(#1) As a user, I need to see if a method is similar to (a clone of) another method. A method consists of multiple blocks which can be conditions or loops. The order of blocks does not play a role. ... (#4) As a user, I want to see if a block is the same as another block. Each block can have multiple uses of data types and calls (of other methods) as well as nested blocks. (#5) As a user, I must be able to see the corresponding class and source code of a method. ..."

As shown in Figure 6.5 the created entity analysis diagram was used to clarify synonyms such as `DataType` and `Class`. For each user story, examples were created and discussed in conjunction with the entity analysis diagram. These discussions lead to a further refinement of the entity relation diagram, adding new entities and attributes to it (e.g. multiple instances for `Block` and `Method`). The refinement, as shown in Listing 6.2, prepared the user stories for the next iterations of the SE-ONTO methodology where the formalization of entities into a knowledge representation language was performed in parallel by the two students.

Listing 6.2: Clone detection case study entity refinement

```
If LoopBlock_namespace_Test1 uses DataType_java_lang_String and
LoopBlock_namespace_Test1 uses DataType_java_lang_String
then CloneOfMethod
If Method_java_lang_Object_ToString has ConditionBlock_java_lang_Object_ToString_1 and
Method_Test_ToString has ConditionBlock_Test_ToString_1 and
ConditionBlock_java_lang_Object_ToString_1 is the same as ConditionBlock_Test_ToString_1
then Method_java_lang_Object_ToString is the same as Method_Test_ToString
...
```

Iteration 1 / S_1

S_1 started with user story #4 that covers the entities `Block` and `NestedBlock`, as well as `Datatype` and `Call`. The user story mentions "[one] block is the same as another block". Therefore, the term `Block`, with its variable number of calls, conditions and loops, was modeled using the *Unbound Key* pattern (that supports such an equality requirement). An upper limit of five conditions and loops as well as ten calls was assumed sufficient to identify a block (making the pattern applicable). The following key definition was added:

$$Key(BLOCK, uses_0, uses_1, ...condition_0, condition_1, ..., loop_0, ..., calls_0, ...)$$

A new `Block` could then be defined using the following statements (example shows a partial definition for individual $Block_1$ which uses one data type and has a condition and loop block):

$$BLOCK(Block_1)$$

$$uses_0(Block_1, DataType_1)$$

$$condition_0(Block_1, Block_2)$$

$$loop_0(Block_1, Block_3)$$

Any unused slots for data types uses, conditions or loops are set using the special individual $Unset$ as suggested by the ***Unbound Key*** pattern. For example:

$$loop_1(Block_1, Unset)$$

This modeled the user story successfully and concluded the iteration. The formalized statements were cast into a mapping task and deployed to the SE-ADVISOR application server together with the test cases.

Iteration 2 / S_1

For the next iteration, user story #1 with `Method`, `CloneOfMethod`, etc. was selected. Based on the sentences "[As a]...if a method is similar to (a clone of) another method. ... The order of blocks does not play a role.", from the corresponding user story, the ***Equivalence Similarity*** pattern was chosen to allow the grouping of different individuals without considering the order of elements. The following example definitions show two methods $Method_1$ and $Method_2$ that are clones of each other:

$$\begin{aligned} METHOD_1 \equiv\ & HAS2PARTS \sqcap \exists hasPart.\{Block_1\} \sqcap \\ & \exists hasPart.\{Block_4\} \\ METHOD_1 \sqsubseteq\ & \{Method_1\} \end{aligned}$$

$$\frac{}{HAS2PARTS(Method_1)}$$

And:

$$\frac{\begin{array}{rl} METHOD_2 \equiv & HAS2PARTS \sqcap \exists hasPart.\{Block_1\} \sqcap \\ & \exists hasPart.\{Block_4\} \\ METHOD_2 \sqsubseteq & \{Method_2\} \end{array}}{HAS2PARTS(Method_2)}$$

Once the formalized statements were added to a mapping task, it was discovered that a `Method` can also have calls and data type usages, and these were therefore allowed as a range of the $hasPart$ object property. Ultimately, the mapping task for SE-ADVISOR was created and deployed to the application server.

Iteration 3 / S_1

For the final iteration, user story #5 was selected. As a relation between `DataType` and `Method` was already present, the only entity remaining at this point was the entity `SourceCode`, which was added as a data property of `Method`. This successfully transformed all supplied user stories to an ontology design, by applying the SE-ONTO methodology.

Iteration 1 / S_2

For the first iteration, S_2 selected the smaller user story #5 with `Method`, `Class` and `SourceCode`. No applicable ontology design patterns were identified but individuals

for `Method` and `Class` were added (as identified by the refinement). `SourceCode` was not added to the ontology design because "[SE-ONTO sets] aside any properties that cannot be identified as affecting the data-model". Consequently, the only object properties added, were $uses$ and $calls$ (to connect `Method` and `Class` as shown in the following example definition):

$$uses(Method_1, Class_1)$$

This concluded the first design iteration and the formalized statements that were cast into a mapping task and deployed to the SE-ADVISOR application server.

Iteration 2 / S_2

For the next iteration, the user story #4 with `DataType`, `Block` and `NestedBlock` was selected. `Block` and `Method` were identified as having similar properties, and therefore modeled as the same individual $MethBlock$. No pattern was identified for the user story. Instead, a concept was created for each individual in order to let the reasoner find identical definitions by concept equivalence. The following definitions were added (example showing two methods that are a clone of each other):

$$\begin{aligned} METHBLOCK_1 &\equiv \exists calls.METHBLOCK_3 \sqcap \exists uses.\{Class_1\} \\ METHBLOCK_2 &\equiv \exists calls.METHBLOCK_3 \sqcap \exists uses.\{Class_1\} \end{aligned}$$

$$\begin{aligned} &METHBLOCK_1(MethBlock_1) \\ &METHBLOCK_2(MethBlock_2) \end{aligned}$$

A mapping for the SE-ADVISOR application server was created together with tests, which revealed that individuals with, for example, three $calls$ relations could become part of concepts, which required two $calls$ relations. For this reason, the ***Representative Individual*** pattern was used to add the number of "calls" and "uses" to each `Method`/`Block`. The following example shows the refined definition of $MethBlock_1$:

$$METHBLOCK_1 \equiv \exists calls.METHBLOCK_3 \sqcap \exists uses.\{Class_1\} \sqcap \exists hasExactly.\{2CallsAndUses\}$$

$$METHBLOCK_1(MethBlock_1)$$

$$hasExactly(MethBlock_1, 2CallsAndUses)$$

The newly refactored model successfully captured all the requirements from the user story and finalized the design iteration. The formalized statements were cast into a mapping task, and deployed to the SE-ADVISOR application server together with the test cases.

Iteration 3 / S_2

In the last iteration, the entities `LoopBlock` and `ConditionBlock` from user story #1 were picked up, and the object property $calls$ was refactored to contain two call types, one for conditions ($conditionCalls$) and one for loops ($loopCalls$). In addition, the previously left out data property for `SourceCode` was made part of `Method`. The deployment of the mapping task and the passing of all tests finalized the ontology design phase.

Summary

Both S_1 and S_2 correctly transformed the supplied user stories into an application-centric ontology design using the SE-ONTO methodology. Nevertheless, the order in which the user stories were transfered varied, and consequently two different ontologies were created. S_1 applied the *Unbound Key* and *Equivalence Similarity* ontology design patterns. Based on the designed ontology, `Method` and `Block` individuals were found to be the same ($sameAs$) when they are a clone of each other, and an application can identify clones by querying the ontology for such matching individuals. S_2 modeled `Method` and `Block` as the same concept/individual and applied the *Representative Individual* pattern during the ontology design. In the resulting ontology, equivalent `Method` concepts denote a clone of the method. An application can identify clones by querying the ontology for such equivalent concepts.

6.4.3 Application Logic

As with the bug quality and triage case study (Section 6.3), some parts of the user stories that are generated in the requirements analysis phase must be implemented by application logic. The responsibility of the SE-ADVISOR mapping task is to provide creation rules for all required concepts, individuals and properties for each formalization.

- A concept and individual for each method/block ($S_{1,2}$)

  - Relations for the keys ($condition$, etc.) for each block (S_1)

  - $conditionCalls$ and $loopCalls$ relations for blocks (S_2)

  - $hasExactly$ relations and individuals for the number of uses/calls (S_2)

  - $hasPart$ for each method (S_1)

- An individual for each data type/class ($S_{1,2}$)

  - $uses$ relation for data types ($S_{1,2}$)

In order to create the concepts, individuals and properties stated above, source code files are processed one-after-another by a mapping task at the SE-ADVISOR application server. Each file is parsed using a Java parser (JAPA[17]), an AST is built and method calls, fully qualified data type names and control-blocks (loops and conditions) are extracted. Although the approach relies on identifying fully qualified type names of variables, it is not mandatory that the source code is compiled; instead, an AST is constructed on a file-per-file basis. This is a key aspect of the approach in terms of horizontal scalability (multiple mapping tasks can work in parallel). In case data type ambiguities occur (e.g. a protected variable of a super class), identifier names are used as opposed to the fully qualified type names.

6.4.4 Validation

One intent of the clone detection case study has been the study of the development of two ontologies, by different individuals using the same methodology, in particular with regards to the effect of selecting user stories and design patterns. From the two distinctive ontologies, it can be deduced that the SE-ONTO methodology cannot guarantee a uniform output; there exists a certain degree of flexibility and freedom in ontology design. This conclusion is in agreement with the work of de Bruijn [Bru03]. Ontology design, much like application design, can never be fully formalized and part of it remains a creative activity. Notwithstanding, the SE-ONTO methodology has led to two

[17] http://code.google.com/p/javaparser/

functional ontologies that follow "minimal encoding bias" and "clarity", two significant ontology design qualities [Gru95]. As an observation from the case study, it can be noted that not all user stories carry pertinent information for the design of the ontology, and it can be valuable to take on such user stories in combination with others. Alternatively, one can start with the larger user stories, which has the benefit of tackling harder design problems on the outset (a proven agile principle [Sub05]). While the order of chosen user stories plays a certain role in the ontology design, the selection of ontology design patterns has a far greater impact on the quality of the developed ontology. Thus, the existence of well described re-usable design solutions such as ontology design patterns is of high importance.

The main goal of the clone detection case study has been the comparison of an application developed using the SE-ONTO methodology (an ontology-driven application) with traditionally developed (non-ontological) applications. Four popular clone detection tools, each using a different internal representation model and granularity, have been selected for this purpose: (1) *Simian* is a commercial String-based approach that is popular due to its integration in Eclipse. (2) The open-source *CCFinder* tool has shown commendable recognition rates and is the best token-based approach available. (3) *JCD* (Java Clone Detector) is a recent development from the University of Waterloo that matches Java *pcode* and finally (4) *DECKARD* is a distributed implementation of an AST-based approach.

Tests have been carried out to compare the performance of the developed ontology-driven application with the non-ontological tools listed above. The performance analysis of the implementation (shown in Table 6.8) has been scaled from a few hundred lines of

	100	1000	10000
FaCT++	501.3s	>1000.0s	>1000.0s
HermiT	65.5s	113.0s	254.1s
Pellet	MEM	MEM	MEM
REL/TrOWL	3.6s	7.3s	21.6s

Table 6.8: Clone detection case study performance validation

code (LOC) to 10000 LOC to showcase the scalability of the implementation. Although FaCT++ performed poorly and Pellet was not able to reason over the ontology at all, both HermiT and REL/TrOWL were able to materialize inferences for the ontology in under 10 minutes. This is comparable to the selected "traditional" clone detection tools and thus, the application developed for the case study is valid in terms of performance.

The complexity of the developed application (and effort that went into creating it) is measured in LOC. This is, of course, only one indicator that can be influenced by many factors, such as coding style or refactoring. Nevertheless, without the actual development time of clone detection tools available, it serves as the only available approximation of the complexity (that can be used to compare the ontology-driven and non-ontological applications). The used tool for counting LOC across different languages is CLOC[18]. Formatting, blank lines and comments are automatically ignored by the tool. The results from the comparison are shown in Figure 6.6. As Simian's source code is not available publicly, it has been decompiled using JAD[19] for the comparison. The case study application (including the application logic and ontology) shows a lower complexity than other comparable tools which indicates that ontology-driven development can produce better maintainable applications.

[18]http://cloc.sourceforge.net/
[19]http://www.varaneckas.com/jad/

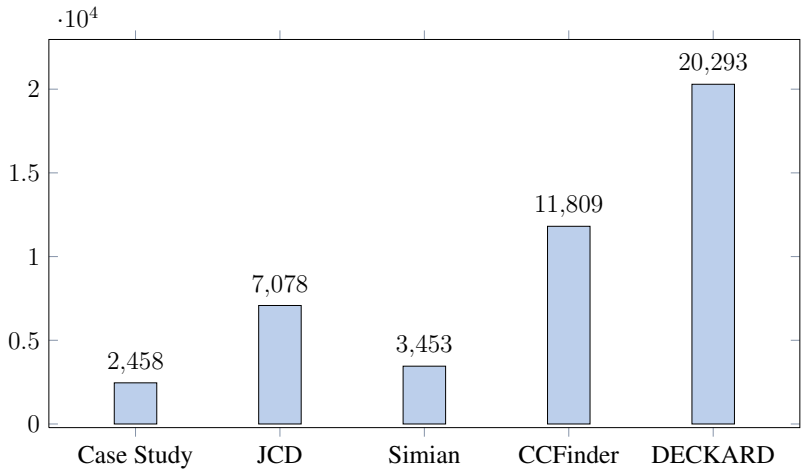

Figure 6.6: Clone detection application complexity comparison

In order to demonstrate that its lower complexity does not impact functionality negatively, the developed clone detection tool has been tested with source code from the Java Development Kit (JDK)[20] and the open-source Apache Commons[21] project (a commonly used library and building block of many applications). A random selection from the JDK 1.4 `swing` package (97 files with around 10000 LOC) and the complete `javax` and `org` packages from the JDK 1.5 (620 files with around 50000 LOC) have been used. As mentioned earlier, the number of analyzed files is low, as the goal of the validation is to demonstrate a similar functionality (not an improved clone detection performance). The recognition performance depends on identifying the fully qualified type name of all used identifiers. As asterisk import statements degrade the performance, only those files not containing asterisk imports were selected. Although frequent within the JDK (around 380 in a sampled set of 1000), most modern Java programs do not have asterisk imports, as their imported data types are now automatically managed by the IDE. This can be observed in the second analyzed project, the Apache Commons library (containing 3348

[20] http://jdk6.java.net/
[21] http://commons.apache.org/

	Ont.-Driven App.	JCD	Simian	CCFinder	DECKARD
Matches	1264	21	145	617	813
Blocks	1375	39	679	895	1263
Methods	603	0	337	473	663
Recall	0.79	0.02	0.40	0.53	0.74

Table 6.9: Clone detection validation JDK 1.4 (swing)

	Ont.-Driven App.	JCD	Simian	CCFinder	DECKARD
Matches	3919	2037	3381	2002	2034
Blocks	4066	1219	1569	3152	3572
Methods	1838	70	616	1628	1751
Recall	0.68	0.21	0.26	0.53	0.60

Table 6.10: Clone detection validation JDK 1.5 (javax, org)

Java files with about 100000 LOC) where only 6 files contained such import statements.

To compare clone detection results from the case study and other tools, a parser for the output of *JCD*, *Simian*, *CCFinder* and *DECKARD* has been developed, in order to read file names and matching line numbers of each clone. The parameters of the tools are based on the recommendations from the respective web-sites and papers. Table 6.9 and Table 6.10 show the processed blocks and methods. The number of matching blocks thereby is higher than the number of matches, as one match might cover more than one block. *CCFinder* and *DECKARD* both detect large clones that often span across multiple methods. Once these clones are broken down to matched complete methods, the number of detected clones in this case study and *CCFinder*/*DECKARD* becomes similar. The algorithm used in *JCD* does not find clone blocks bigger than size N, and does not try to expand a matching fragment until the maximum number of matching statements have been found. As a result, it performs poorly when the detection criteria are completely cloned methods. *Simian* only performs a String-based comparison of code fragments, so

	Ont.-Driven App.	Simian	CCFinder	DECKARD
Matches	16549	10250	7865	7980
Blocks	18078	6800	17092	16519
Methods	7729	2842	7374	8848
Recall	0.61	0.23	0.57	0.56

Table 6.11: Clone detection validation Apache Commons

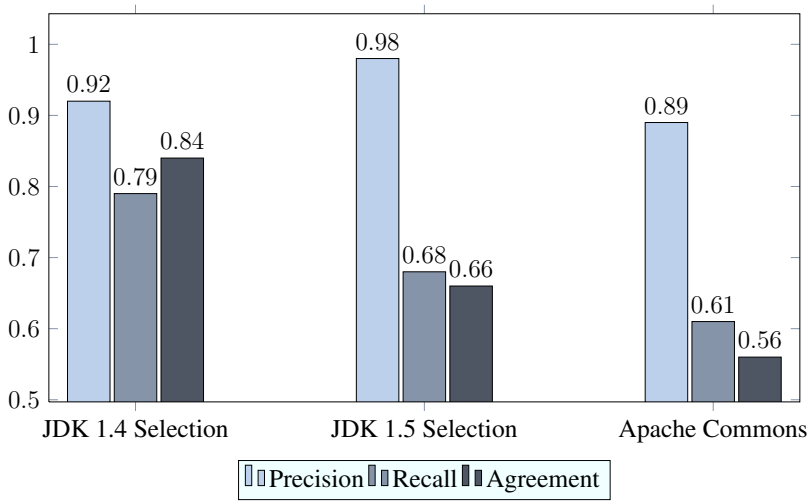

Figure 6.7: Precision and recall for the ontology-driven application

a lower number of matching blocks and methods is not surprising.

Precision and recall values have been calculated by assembling an oracled set of clones and manually annotating the source code, similar to the clone tool evaluation of Bellon [Bel98]. The oracled set consists of a union of clone blocks detected by *Simian*, *CCFinder*, *DECKARD* and *JCD*. For the manual annotation of source code, blocks from the JDK selection and 15% of random blocks from the Apache Commons were selected. The results gathered are comparable to [BKA+07] in terms of their recall. Obtained precision values are naturally higher, as an oracled set of clones is used (and not an absolute "ground truth"). It has to be noted, that for the comparison only clones detectable by the ontology-driven application were considered (the described blocks) and not all clones

detectable by other tools. This explains the high recall of the ontology-driven application that even outperforms other tools for this type of "block-clone" (see inter-tool agreement between the case study and all of the existing tools in Figure 6.7). Results of the validation indicate that the developed ontology-driven application is capable of identifying source code clones with satisfactory performance and that its functionality is comparable to that of non-ontological tools.

Chapter 7

Conclusions

The main objective of this thesis was an investigation into the use of ontology design in modern Software Engineering processes, and how developers can leverage reasoning services in order to develop ontology-driven applications. The contributions of this thesis are three-fold: (1) A methodology for the incorporation of ontology design into an agile software process-skeleton. (2) A novel set of ontology design patterns that foster the development of reasoning-enabled ontologies; and (3) an application model that takes advantage of ontologies (with reasoning capabilities) and enables its users to incrementally develop and rapidly deploy ontology-driven applications. This combination of providing a methodology, a set of reasoning design patterns and an application model, creates an essential foundation for the adaptation of ontology-driven development and for the incorporation of semantic web technologies into the product development and maintenance cycles of modern applications.

The SE-ONTO methodology proposed in this thesis is an advancement compared to other methodologies, such as Uschold and King's methodology [UK95] or the method-

ology of Gruninger and Fox [GF94], as it allows for the integration of ontology design into modern agile software processes, and for iterative ontology development through design patterns. It differs from the *On-To-Knowledge* methodology [SSS00] and the *XD* methodology [SFBD+09], which do not consider reasoning services in their ontology design and fail to provide an application model for their methodologies. In comparison to other approaches that have solely been applied to toy-ontologies or a single ontology design study, the SE-ONTO methodology was tested in two case studies addressing real-world problems:

- The implementation of an ontology-driven application for bug quality and triage.
- The development of a code clone detection system that is enabled by ontology reasoning services.

Structural reasoning design patterns capture architectural solutions that solve a particular design problem by leveraging reasoning services. As shown throughout this thesis, reasoning design patterns play a fundamental role in how ontology-driven applications are built. The presented catalog of patterns differs from the work of Presutti et al. ([PG08], [PGD+08]) who fail to show how their proposed patterns can enhance the designed ontology or application. The presented reasoning patterns can further be seen as complementary to ontology design patterns exhibited in [EA09] and [NUW04], which deal with different design problems.

The SE-ADVISOR application model introduced in this thesis, is a concrete solution for a well-defined problem space: the incorporation of (semi-) structured non-ontological data into an ontology-driven application. This approach differs from semantic web frameworks, such as [BKvH02], that do not impose such a specific data flow on the

processed data. The lack of explicit data flow in traditional techniques is assumed to be one of the reasons developers produce ontologies of low design quality, that is, ontologies with simple 1:1 mappings of existing data or ontologies that do not exploit the advantages of reasoning services. The presented application model can be seen as a best practice solution for developing ontology-driven applications where the primary data source is non-ontological.

The findings of this thesis support the use of ontologies as an empowering technology in the development of applications. Future work could introduce additional ontology reasoning patterns, which allow for the development of more extensive ontologies. It is also imperative to further investigate if certain ontology design patterns are mutually exclusive and cannot be combined. Although the work in this thesis has laid the foundations for the incorporation of ontology design into modern agile Software Engineering, there remains the need to advocate for the use of ontology-driven applications in industry. By embracing Knowledge Engineering (ontology design, reasoners and semantic web technologies) as an integral part of a student's Software Engineering curriculum, a future generation of knowledge engineers will realize this goal.

Bibliography

[ABH+00] Klaus-Dieter Althoff, A. Birk, S. Hartkopf, W. Müller, M. M. Nick, D. Surmann, and C. Tautz. Systematic Population, Utilization, and Maintenance of a Repository for Comprehensive Reuse. *Learning Software Organizations - Methodology and Applications*, 1756:25–50, 2000. → pages 27

[ABKD+02] Klaus-Dieter Althoff, U. Becker-Kornstaedt, B. Decker, A. Klotz, E. Leopold, J. Rech, and A. Voss. The indiGo project: enhancement of experience management and process learning with moderated discourses. pages 53–79, 2002. → pages 11

[Aha91] David W. Aha. Case-based learning algorithms. In *DARPA Case-Based Reasoning Workshop*, pages 147–158, 1991. → pages 13

[AHM05] John Anvik, Lyndon Hiew, and G. C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39. ACM, 2005. → pages 115

[AJWH03] A. Aurum, R. Jeffery, C. Wohlin, and M. Handzic. *Managing software engineering knowledge*. Springer, 2003. → pages 11

[AN04] Klaus-Dieter Althoff and M. M. Nick. How to Support Experience Management with Evaluation - Foundations, Evaluation Methods, and Examples for Case-Based Reasoning and Experience Factory. In *Lecture Notes of Computer Science Articial Intelligence*. Springer, 2004. → pages 13

[ASH+06] A. Ankolekar, K. Sycara, J. Herbsleb, R. Kraut, and C. A. Welty. Supporting Online Problem solving Communities With the Semantic Web. pages 575–584, 2006. → pages 28

[AWWH08] A. Aseeri, Pornpit Wongthongtham, C. Wu, and F. K. Hussain. Towards Social Network based Ontology Evolution Wiki for an Ontology Evolution. In *Proceedings of the 10th International Conference on*

Information Integration and Web-based Applications & Services, pages 500–502, Linz, 2008. → pages 28

[BA96] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Press, 1996. → pages 8

[Bas85] V. R. Basili. Quantitative evaluation of software methodology - Tech. Report 1519. Technical report, University of Maryland, 1985. → pages 11

[BB02] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 36–43, 2002. → pages 131

[BCG01] A. Berardi, D. Calvanese, and G. Giacomo. Reasoning on UML Class Diagrams using Description Logic Based Systems. In *Proceedings of the KI 2001 Workshop on Applications of Description Logics*, 2001. → pages 29

[BCM+03] Franz Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. P. Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003. → pages 14, 16, 19, 21, 22, 23

[BCR94] V. R. Basili, G. Caldiera, and H. D. Rombach. *Encyclopedia of Software Engineering*. John Wiley & Sons, New York, USA, 1994. → pages 11

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999. → pages 8

[Bec01] Kent Beck. Manifesto for Agile Software Development, 2001. URL: <http://agilemanifesto.org/>. → pages 8

[Bel98] Stefan Bellon. *Vergleich von Techniken zur Erkennung duplizierten Quellcodes*. Master thesis, Universität Stuttgart, 1998. → pages 146

[BJS+07] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiß, Rahul Premraj, Thomas Zimmermann, A. Schroter, and C. Weiss. Quality of Bug Reports in Eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 21–25, 2007. → pages 125

[BJS+08] Nicolas Bettenburg, Sascha Just, A. Schroter, C. Weiss, Rahul Premraj, Thomas Zimmermann, and A. Schröter. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318, 2008. → pages 115, 125

[BKA+07] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, September 2007. → pages 146

[BKvH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame : A generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the first Int'l Semantic Web Conference (ISWC 2002)*, pages 54–68, 2002. → pages 95, 149

[BLHL02] Tim Berners-Lee, James A. Hendler, and Ora Lassila. The Semantic Web - A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, (April):24–30, 2002. → pages 19

[BLP00] K. Breitman, J. Leite, and J. C. Prado. Scenario Evolution: A Closer View on Relationships. In *Proceedings of the 4th International Conference on Requirements Engineering (ICRE'00)*, pages 95–105. Published by the IEEE Computer Society, 2000. → pages 8

[Boe86] B. Boehm. A spiral model of software development and enhancement. *SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986. → pages 7

[BPZK08] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Extracting structural information from bug reports. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 27–30, 2008. → pages 115, 125

[Bru03] J. Bruijn. Using Ontologies - Enabling Knowledge Sharing and Reuse on the Semantic Web. Technical report, DERI, Innsbruck, Austria, 2003. URL: <http://www.debruijn.net/publications/>. → pages 141

[BS05] Joachim Baumeister and Dietmar Seipel. Smelly OwlsDesign Anomalies in Ontologies. In *The Florida AI Research Society Conference*, pages 215–220, 2005. → pages 35

[BYM+98] Ira D. Baxter, A. Yahin, L. Moura, M. Sant Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, 1998. → pages 131

[BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999. → pages 116

[Cha01] S. K. Chang. *Handbook of software engineering and knowledge engineering*. World Scientific, 2001. → pages 13

[Cre93] D. Crevier. *AI: The Tumultuous Search for Artificial Intelligence*. Basic Books, 1993. → pages 10

[CTP00] Peter Clark, John Thompson, and Bruce Porter. Knowledge patterns. In *Proceedings of KR-2000*, pages 591–600, 2000. → pages 66

[DBL10] Pete Deemer, Gabrielle Benefield, and Craig Larman. The Scrum Primer. Technical report, 2010.
URL: <http://goodagile.com/scrumprimer/>. → pages 47

[DE05] J. Dietrich and C. Elgar. A Formal Description of Design Patterns Using OWL. *2005 Australian Software Engineering Conference*, pages 243–250, 2005. → pages 28

[DG10] Ian J. Davis and Michael W. Godfrey. Clone detection by exploiting assembler. *Proceedings of the 4th International Workshop on Software Clones - IWSC '10*, (January):77–78, 2010. → pages 131

[DRD99] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 109–118. Ieee, 1999. → pages 131

[DWL00] D. Deridder, B. Wouters, and W. Lybaert. The Use of an Ontology to Support a Coupling between Software Models and Implementation. In *Proc. European Conference on Object-Oriented Programming (ECOOP'00)*, 2000. → pages 27

[EA09] M. Egana-Aranguren. *Role and Application of Ontology Design Patterns in Bioontologies*. Phd thesis, University of Manchester, 2009. → pages 70, 71, 149

[Ecl09] Eclipse Foundation. The Open Source Developer Report - 2009 Eclipse Community Survey. Technical report, 2009.
URL: <www.eclipse.org/org/press-release/>. → pages 96

[Fel98] Christiane Fellbaum. *WordNet- An Electronic Lexical Database*. The MIT Press, 1998. → pages 39

[FLGPSS99] Mariano Fernandez-Lopez, Asuncion Gomez-Perez, Juan Pazos Sierra, and Alejandro Pazos Sierra. Building a chemical ontology using methontology and the ontology design environment. *IEEE Intelligent Systems*, 14(1):37–46, 1999. → pages 38

[FNM+03] R. A. Falbo, A. C. Natali, P. G. Mian, G. Bertollo, and F. B. Ruy. ODE: Ontology-based software Development Environment. pages 1124–1135, 2003. → pages 27

[FS06] Ronen Feldman and James Sanger. *The Text Mining Handbook*. Cambridge University Press, 2006. → pages 116

[Gan05] Aldo Gangemi. Ontology design patterns for semantic web content. *The Semantic WebISWC 2005*, pages 262–276, 2005. → pages 66, 72

[Ger92] German Ministry of Defense. V-Model: Software Lifecycle Process Model. Technical report, General Preprint No. 250, 1992. → pages 7

[GF94] Michael Gruninger and M.S. Fox. The design and evaluation of ontologies for enterprise engineering. In *Workshop on Implemented Ontologies, European Workshop on Artificial Intelligence*, 1994. → pages 36, 44, 149

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. → pages 66, 76

[GL90] R. V. Guha and D. B. Lenat. Cyc: A Midterm Report. *AI Magazine*, 11(3), 1990. → pages 39

[GP94] Asuncion Gomez-Perez. From Knowledge Based Systems to Knowledge Sharing Technology: Evaluation and Assessment. Technical report, Knowledge Systems Laboratory, Stanford University, 1994. URL: <http://oa.upm.es/6498/>. → pages 33, 38

[GP99] Asuncion Gomez-Perez. Evaluation of taxonomic knowledge in ontologies and knowledge bases. In *Proceedings of the 12th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, pages 611–618. University of Calgary, Alberta, Canada, 1999. → pages 34

[GPFLC04] Asuncion Gomez-Perez, Mariano Fernandez-Lopez, and Oscar Corcho. *Ontological Engineering*. Springer, 2004. → pages 15

[Gru93] Thomas R. Gruber. A Translation Approach to Portable Ontology Specifications (KSL 92-71). Technical Report 2, 1993.

URL: <ftp://ksl.stanford.edu/pub/KSL_Reports/>. → pages 16

[Gru95] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human Computer Studies*, 43(5):907–928, November 1995. → pages 33, 36, 60, 142

[Gua97] Nicola Guarino. Semantic Matching: Formal Ontological Distinctions for Information Organization, Extraction, and Integration. *SCIE 97: International Summer School on Information Extraction*, pages 139–170, 1997. → pages 16

[Har03] Simon Harris. Simian Clone Detection Tool, 2003. URL: <http://www.harukizaemon.com/simian/>. → pages 131

[HB09] Matthew Horridge and Sean Bechhofer. The OWL API : A Java API for Working with OWL 2 Ontologies. In *6th OWL Experienced and Directions Workshop*, 2009. → pages 100

[Hep05] Martin Hepp. Representing the Hierarchy of Industrial Taxonomies in OWL : The gen / tax Approach. In *Proceedings of the ISWC Workshop Semantic Web Case Studies and Best Practices for eBusiness (SWCASE05)*, Galway, Irland, 2005. → pages 44

[Hep07] Martin Hepp. ProdLight : A Lightweight Ontology for Product Description Based on Datatype Properties. In *10th International Conference on Business Information Systems*, pages 260–272, 2007. → pages 18

[HJK+04] Maciej Hapke, Andrzej Jaszkiewicz, Krzysztof Kowalczykiewicz, D. Weiss, and P. Zielniewicz. OPHELIA - Open Platform for Distributed Software Development. In *Proceedings of Open Source International Conference*. Poznan University of Technology, 2004. → pages 9

[HK87] D. Harmon and D. King. *Expert Systems - Perspectives, Tools, Exoeriences*. Oldenbourg Verlag, Munic, Germany, 1987. → pages 13

[HKR09] Pascal Hitzler, Markus Krotzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. CRC Press Textbook, 2009. → pages 1

[HKST06] H. J. Happel, A. Korthaus, S. Seedorf, and P. Tomczyk. KOntoR: An Ontology-enabled Approach to Software Reuse. In *Proceedings of SEKE*, pages 349–354, San Francisco, 2006. → pages 27

[Hoy01] David Hoyle. *ISO 9000: Quality Systems Handbook*. Fourth edition, January 2001. → pages 32, 110

[HP06] Jerry R. Hobbs and Feng Pan. Time Ontology in OWL, 2006. URL: <http://www.w3.org/2001/sw/BestPractices/OEP/>. → pages 70

[HPPSH05] Ian Horrocks, Bijan Parsia, P. F. Patel-Schneider, and James A. Hendler. Semantic web architecture: Stack or two towers? *Principles and Practice of Semantic Web Reasoning*, pages 37–41, 2005. → pages 19

[HPSH03] Ian Horrocks, P. F. Patel-Schneider, and F. Harmelen. From SHIQ and RDF to OWL: The Making of a Web Ontology Language. *Journal of Web Semantics*, 1(1):7–26, 2003. → pages 24

[HS03] Maia Hristozova and Leon Sterling. Experiences with Ontology Development for Value-Added Publishing. In *3rd Workshop on Ontologies in Agent Systems*, page 17, 2003. → pages 41

[HSW91] W. S. Humphrey, T. R. Snyder, and R. R. Willis. Software Process Improvement at Hughes Aircraft. *IEEE Software*, 8(4):11–23, 1991. → pages 11

[HSW98] F. Houdek, K. Schneider, and E. Wieser. Establishing experience factories at Daimler-Benz: An experience report. In *Proceedings of the 20th international conference on Software engineering*, pages 443–447, 1998. → pages 11

[IEE90] IEEE Computer Society. IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.121990), 1990. → pages 31, 32

[IEE98] IEEE Computer Society. IEEE Standard for Software Project Management Plans (IEEE Std 1058-1998), 1998. → pages 31

[ISO01] ISO International Organization for Standardization. ISO Standard for Software Engineering - Product quality (ISO / IEC Std 9126), 2001. → pages 32, 110

[Jac99] P. Jackson. *Introduction to Expert Systems*. Addison Wesley Longman, Harlow, UK, 1999. → pages 14

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Longman Publishing Co. Inc., 1999. → pages 8

[JC05] D. Jin and James R. Cordy. Ontology-Based Software Analysis and Reengineering Tool Integration: The OASIS Service-Sharing Methodology. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 613 – 616, 2005. → pages 27

[Kap09] C.J. Kapser. *Toward an Understanding of Software Code Cloning as a Development Practice*. PhD thesis, University of Waterloo, 2009. URL: <http://plg.uwaterloo.ca/ migod/>. → pages 130, 131

[KB08] Halil Kilicoglu and Sabine Bergler. Recognizing Speculative Language in Biomedical Research Articles. *BMC bioinformatics*, 9 Suppl 11(1):S10, January 2008. → pages 125

[KB10] Halil Kilicoglu and Sabine Bergler. A High-Precision Approach to Detecting Hedges and Their Scopes. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning (CoNLL-2010)*, pages 70–77, 2010. → pages 125

[KCS01] G. Kadoda, M. Cartwright, and M. J. Shepperd. Issues on the effective use of CBR technology for software project prediction. In *4th International Conference on Case-Based Reasoning*, pages 276–290, Vancouver, 2001. → pages 12

[Kem87] C. F. Kemerer. An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429, 1987. → pages 13

[KFF06] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *2006 13th Working Conference on Reverse Engineering*, pages 253–262, 2006. → pages 131

[KHL+07] Akrivi Katifori, Constantin Halatsis, George Lepouras, Costas Vassilakis, and Eugenia Giannopoulou. Ontology Visualization Methods - A Durvey. *ACM Computing Surveys*, 39(4), November 2007. → pages 62, 63

[KKI02] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002. → pages 131

[LD09] Alexander De Leon and Michel Dumontier. A platform for distributing and reasoning with OWL-EL knowledge bases in a Peer-to-Peer environment. In *Proceedings of the 6th International Workshop on OWL: Experiences and Directions (OWLED 2009)*. Citeseer, 2009. → pages 26

[Leh79] M. M. Lehman. On understanding laws, evolution, and conservation in the largeprogram life cycle. *Journal of Systems and Software*, 1:213–221, 1979. → pages 8

[LHMI07] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder. *29th International Conference on Software Engineering (ICSE'07)*, pages 106–115, May 2007. → pages 131

[LN03] Thorsten Liebig and Olaf Noppens. OntoTrack: Fast browsing and easy editing of large ontologies. In *Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools (EON2003)*, 2003. → pages 62

[MÖ96] Ralf Möller. A Functional Layer for Description Logics: Knowledge Representation Meets Object Oriented Programming. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming*, San Jose, 1996. → pages 29

[Mai91] N. A. Maiden. Analogy as a paradigm for specification reuse. *Software Engineering Journal*, 6(1):3–15, 1991. → pages 13

[MJ08] James H. Martin and Daniel Jurafsky. *Speech and Language Processing*. 2nd edition, April 2008. → pages 116

[MK10] R. B. Mishra and Sandeep Kumar. Semantic web reasoners and languages. *Artificial Intelligence Review*, 35(4):339–368, December 2010. → pages 103

[MM10] Raghava Mutharaju and Frederick Maier. A MapReduce Algorithm for EL+. *23rd International Workshop on Description Logics (DL2010)*, pages 464–474, 2010. → pages 26

[MVP92] T. Mukhopadhyay, S. S. Vicinanza, and M. J. Prietula. Examining the feasibility of a case-based reasoning model for software effort estimation. pages 155–171, 1992. → pages 13

[NM01] Natalya F. Noy and Deborah L. Mcguinness. Ontology development 101: A guide to creating your first ontology (KSL-01-05). Technical report, Stanford Knowledge Systems Laboratory, 2001.
URL:
<http://www.ksl.stanford.edu/people/dlm/papers/>.
→ pages 32

[NRHW06] Natasha Noy, Alan Rector, Pat Hayes, and Chris Welty. Defining N-ary Relations on the Semantic Web, 2006.
URL: <http://www.w3.org/TR/swbp-n-aryRelations/>.
→ pages 70

[NUW04] Natasha Noy, Mike Uschold, and Chris Welty. Representing Classes As Property Values on the Semantic Web, 2004.
URL:
<http://www.w3.org/TR/swbp-classes-as-values/>.
→ pages 70, 71, 149

[Obe04] Daniel Oberle. Semantic management of middleware. *Proceedings of the 1st International Doctoral Symposium on Middleware*, pages 299–303, 2004. → pages 16

[OHPDB92] E. Ostertag, James A. Hendler, R. Prieto-Diaz, and C. Braun. Computing similarity in a reuse library system: an AI-based approach. *ACM Transactions on Software Engineering Methodology*, 1(3):205–228, 1992. → pages 13

[PG08] Valentina Presutti and Aldo Gangemi. Content ontology design patterns as practical building blocks for web ontologies. *Conceptual Modeling-ER 2008*, pages 128–141, 2008. → pages 67, 68, 69, 70, 149

[PGD+08] Valentina Presutti, Aldo Gangemi, Stefano David, Guadalupe Aguado de Cea, Mari Carmen Suárez-Figueroa, Elena Montiel Ponsoda, and Maria Poveda. NeOn D2.5.1 A library of ontology design patterns: reusable solutions for collaborative design of networked ontologies. Technical report, NeOn Project, 2008.
URL: <http://www.neon-project.org/>. → pages 67, 68, 69, 149

[PMP00] Lutz Prechelt, Guido Malpohl, and M. Philippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, Universität Karlsruhe, 2000. → pages 131

[Pol03] P. R. Polsani. Use and Abuse of Reusable Learning Objects. *Journal of Digital information*, 3(4), 2003. → pages 28

[PPRB07] M. Petrenko, Denys Poshyvanyk, V. Rajlich, and J. Buchta. Teaching Software Evolution in Open Source. *Computer*, 40(11):25–31, 2007. → pages 27

[PRPB91] D. Premkumar, B. Ronald, G. S. Peter, and B. W. Bruce. LaSSIE: A Knowledgebased Software Information System. *Communications of the ACM*, 34(5):34–49, 1991. → pages 28

[PVSFGP10] M. Poveda-Villalon, Mari Carmen Suarez-Figueroa, and Asuncion Gomez-Perez. Reusing Ontology Design Patterns in a Context Ontology Network. In *Proceedings Second Workshop on Ontology Patterns (WOP 2010) co-located at ISWC 2010*. CEUR-WS, 2010. → pages 44

[RC07] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research (No. 2007-541). Technical report, Queen's University, Kingston, Canada, 2007. → pages 130, 132

[Rec05] Alan Rector. Representing Specified Values in OWL: "value partitions" and "value sets", 2005.
URL:
<http://www.w3.org/TR/swbp-specified-values/>. → pages 70, 71, 76

[RJ00] Linda Rising and Norman S. Janoff. The Scrum Software Development Process for Small Teams. *IEEE Software*, (August), 2000. → pages 47

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003. → pages 15

[Roy70] W. W. Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*, volume 26, pages 1–9. Los Angeles, 1970. → pages 7

[RU89] H. D. Rombach and B. T. Ulery. Establishing a measurement based maintenance improvement program: lessons learned in the SEL. Technical report, 1989. → pages 11

[RWNW05] Alan Rector, Chris Welty, Natasha Noy, and Evan Wallace. Simple part-whole relations in OWL Ontologies, 2005.
URL:
<http://www.w3.org/2001/sw/BestPractices/OEP/>. → pages 70

[SA77] R. C. Schank and R. P. Abelson. *Scripts, plans, goals, and understanding: an inquiry into human knowledge structures*. Erlbaum Associates, New York, USA, 1977. → pages 12

[SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, pages 85–90, 1994. → pages 94

[SBF98] Rudi Studer, Richard Benjamins, and Dieter Fensel. Knowledge Engineering : Principles and Methods. *Data Knowledge Engineering*, 25(1-2):161–197, 1998. → pages 1

[Sch97] Ken Schwaber. Scrum development process. *OOPSLA Business Object Design and Implementation*, (April 1987):10–19, 1997. → pages 4, 8, 46

[Sch02] Albrecht Schmidt. *Ubiquitous Computing Computing in Context*. PhD thesis, Lancaster University, 2002. → pages 96

[Sch11] Philipp Schügerl. Scalable Clone Detection Using Description Logic. In *Proceedings of the 5th International Workshop on Software Clones (IWSC'11) at ICSE'11*, 2011. → pages 102, 130

[Sel92] Peter G. Selfridge. Knowledge-Based Software Engineering - Guest Editor's Introduction. *IEEE Expert*, pages 11–12, 1992. → pages 28

[SFBD+09] Mari Carmen Suarez-Figueroa, Eva Blomqvist, Mathieu DAquin, Mauricio Espinoza, Asuncion Gomez-Perez, Holger Lewen, Igor Mozetic, Raúl Palma, Maria Poveda, Margherita Sini, Boris Villazon-Terrazas, Fouad Zablith, and Martin Dzbor. NeOn D5.4.2. Revision and Extension of the NeOn Methodology for Building Contextualized Ontology Networks. Technical report, NeOn project, 2009.
URL: <http://www.neon-project.org>. → pages 41, 149

[SFdCB+08] Mari Carmen Suarez-Figueroa, Guadalupe Aguado de Cea, Carlos Buil, Klaas Dellschaft, Mariano Fernandez-Lopez, Andres Garcia, Asuncion Gomez-Perez, German Herrero, Elena Montiel-Ponsoda, Marta Sabou, Boris Villazon-Terrazas, and Zheng Yufei. NeOn D5.4.1. NeOn Methodology for Building Contextualized Ontology Networks. Technical report, NeOn project, 2008.
URL: <http://www.neon-project.org>. → pages 66

[SKRR97] Bill Swartout, Kevin Knight, Tom Russ, and Marina Rey. Toward Distributed Use of Large-Scale Ontologies. In *Symposium on Ontological Engineering of AAAI*, pages 138–148, 1997. → pages 39, 44

[SPSW01] Michael Q. Stearns, Colin Price, Kent A. Spackman, and Amy Y. Wang. SNOMED clinical terms: overview of the development process and project status. In *Proceedings of the AMIA Symposium*, pages 662–666, January 2001. → pages 18

[SR10] Philipp Schügerl and Juergen Rilling. Chapter 9 - Enriching SE Ontologies with Bug Quality. In *Semantic Web Enabled Software Engineering*, pages 139–151. 2010. → pages 102, 114

[SRC08] Philipp Schügerl, Juergen Rilling, and Philippe Charland. Enriching SE Ontologies with Bug Report Quality. In *International Workshop on Semantic Web Enabled Software Engineering (SWESE'08)*, 2008. → pages 102, 114

[SRC11] Philipp Schügerl, Juergen Rilling, and Philippe Charland. A Semantic Web-based Approach to Scalable Clone Detection. In *Proceedings of the 35th Computer Software and Applications Conference*, 2011. → pages 102, 130

[SS09] Steffen Staab and Rudi Studer. *Handbooks on Ontologies*. Springer, 2nd edition, 2009. → pages 35

[SSS00] Hans-Peter Schnurr, York Sure, and Rudi Studer. On-To-Knowledge Methodology Baseline Version. Technical report, Institute AIFB, University of Karlsruhe, 2000. → pages 40, 149

[Sub05] Venkat Subramaniam. *Practices of an Agile Developer: Working in the Real World*. Pragmatic Bookshelf, 1st edition, 2005. → pages 142

[TPO+06] P. Tetlow, J. Z. Pan, Daniel Oberle, E. Wallace, Mike Uschold, and E. Kendall. Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering, 2006. URL: <http://www.w3.org/2001/sw/BestPractices/>. → pages 27

[TZ88] W. T. Tsai and I. Zualkernan. Expert Systems for Software Engineering? *Software and Applications Conference*, pages 2611–2611, 1988. → pages 10

[UG96] Mike Uschold and Michael Gruninger. Ontologies: Principles , Methods and Applications. *Knowledge Engineering Review*, 11(2):93–136, 1996. → pages 16

[UK95] Mike Uschold and Martin King. Towards a Methodology for Building Ontologies. In *In Workshop on Basic Ontological Issues in Knowledge Sharing*, Montreal, Canada, 1995. → pages 37, 148

[WCDS05] Pornpit Wongthongtham, Elizabeth Chang, T. S. Dillon, and I. Sommerville. Software Engineering Ontologies and their Implementation. In *Proceedings of the IASTED International Conference on Software Engineering*, pages 208–213, Innsbruck, 2005. → pages 28

[Wel97] C. A. Welty. Augmenting Abstract Syntax Trees for Program Understanding. In *Proceedings of the 12th IEEE International Conference on Automated Software Engineering*, pages 126–133, 1997. → pages 28

[Wel00] C. A. Welty. Ontologies: Expert Systems all over again?, 2000. URL: <http://www.cs.vassar.edu/ weltyc/aaai-99/>. → pages 18

[WF99] C. A. Welty and D. A. Ferrucci. A Formal Ontology for Reuse of Software Architecture Documents. In *Proceedings of the 14th IEEE international conference on Automated software engineering*, page 259, 1999. → pages 28

[Win93] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, 1993. → pages 9

[WJL03] A. Walenstein, N. Jyoti, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, pages 285–294, 2003. → pages 131

[WLLB06] T. Weithoner, Thorsten Liebig, M. Luther, and S. Bohm. Whats wrong with OWL benchmarks. In *Proc.of the Second Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2006)*, pages 101–114, 2006. → pages 104

[ZBPS09] Thomas Zimmermann, Silvia Breu, Rahul Premraj, and Jonathan Sillito. Improving Bug Tracking Systems bug tracking. *Companion to the 31th International Conference on Software Engineering*, 2009. → pages 115

[ZK02] A. Zeller and J. Krinke. *Essential Open Source Toolset*. Wiley, 2002. → pages 115

[ZSG79] Marvin V. Zelkowitz, Alan C. Shaw, and John D. Gannon. *Principles of Software Engineering and Design*. Prentice Hall Professional, 1979. → pages 7, 8

[ZZWD05] Thomas Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005. → pages 116