Diploma Thesis

June 2, 2009

SEON

Designing Software Engineering Ontologies

Jan Bielik
from Zurich, Switzerland (03-714-870)

supervised by
Prof. Dr. Harald Gall
Dr. Gerald Reif

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Diploma Thesis

SEON

Designing Software Engineering Ontologies

Jan Bielik

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Diploma Thesis

Author: Jan Bielik, jan_bielik@gmx.ch

Project period: December 2, 2008 - June 2, 2009

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank my supervising assistant Gerald Reif for his tireless support and his invaluable input that led to this diploma thesis. Moreover, I wish to thank Michael Würsch for the clarifying discussions we had about the vocabulary of our software engineering ontologies and for his help with the minor configuration problems I have encountered while using EVOLIZER. I would also like to thank Prof. Harald Gall for giving me the opportunity to write this interesting thesis.

Next, I would like to thank my aunt Marie-Josée de Saint Robert and my friend Jivan Brugger for proofreading my diploma thesis. The biggest acknowledgment of my gratitude is dedicated to my parents, Eve and Peter Bielik, for their endless support during my entire life and for making all this possible. Finally, I would like to thank my whole family, especially my brother Marc Bielik, and all my friends for their moral support and for believing in me in good times and in bad times.

Abstract

An ontology is an explicit and formal specification of a conceptualization

T. R. Gruber's definition, later refined by R. Studer

In the field of informatics an ontology provides a commonly shared vocabulary for a domain of discourse, which is used in order to retain the intended meaning of data.

The aim of this thesis is to describe a structured set of software engineering ontologies that represents the data found in revision control systems, issue tracking systems and the static source code information underlying object-oriented programming languages. It presents an approach to an object-oriented metrics library, which identifies design disharmonies in object-oriented software projects from annotations created with the software engineering ontologies presented in this thesis.

For this purpose, we present first the state of the art in ontology development. We stress the needs for comprehensive methodologies and present in detail the available ones, which are often applied and which rely on the most intuitive use. These methodologies should enable a straightforward development of ontologies that are commonly shared and widely used in their specific domain of discourse. Next, we analyze existing ontologies in the field of software engineering with regard to their suitability for integration in our software engineering ontologies.

In a second stage, we outline the various requirements that our software engineering ontologies should fulfill and describe the system implementations and the object-oriented programming languages that are investigated for their representations in OWL. We present SEON, our structured set of software engineering ontologies. These ontologies describe the data of revision control systems, issue tracking systems and the static source code information of object-oriented programming languages. Our software engineering ontologies are subsequently used to annotate these kinds of information for further software analyses.

Finally, we present our object-oriented metrics library, which evaluates the design of object-oriented software projects on the basis of data annotated with our software engineering ontologies and points out possible weaknesses in the design of these software projects.

Zusammenfassung

Eine Ontologie ist eine explizite und formale Spezifikation einer Konzeptualisierung

T. R. Gruber's Definition, später präzisiert durch R. Studer

In der Informatik stellt eine Ontologie ein gemeinsam benutztes Vokabular für ein Anwendungsgebiet zur Verfügung. Dieses Vokabular wird benutzt um die beabsichtigte Bedeutung von Daten festzuhalten.

In dieser Diplomarbeit wird zunächst eine strukturierte Sammlung von Ontologien der Softwaretechnik beschrieben. Mit diesen Ontologien werden die Daten von Revision Control Systemen, Issue Tracking Systemen und die statischen Quellcodeinformationen von objekt-orientierten Programmiersprachen dargestellt. Weiter wird ein Ansatz zu einer objekt-orientierten Metrikbibliothek vorgestellt, welche Disharmonien im Design eines objekt-orientierten Softwareprojektes identifiziert. Diese Metrikbibliothek benutzt dabei Daten, die mit unseren Ontologien der Softwaretechnik markiert wurden.

Zu diesem Zweck wird zuerst der aktuelle Stand bei der Entwicklung von Ontologien beschrieben. Betont werden dabei die Anforderungen an umfassende Methodologien und aufgeführt werden vor allem diejenigen, welche oft benutzt werden und in ihrer Anwendung einfach sind. Diese Methodologien sollten die Entwicklung von Ontologien ermöglichen, die eine breite Verwendung in ihrem Anwendungsgebiet finden. Als nächstes werden bereits vorhandene Ontologien im Bereich der Softwaretechnik analysiert. Dies im Hinblick auf ihre Eignung zur Integration in unsere eigenen Ontologien der Softwaretechnik.

Des Weiteren zeigen wir die verschiedenen Anforderungen auf, die unsere Ontologien der Softwaretechnik erfüllen sollten. Dargestellt sind zudem die Systeme und die objekt-orientierten Programmiersprachen, welche für die Erstellung unserer Ontologien in OWL analysiert wurden. Wir präsentieren SEON, unsere strukturierte Sammlung von Ontologien der Softwaretechnik. Diese Ontologien beschreiben die Daten von Revision Control Systemen, Issue Tracking Systemen und die statischen Quellcodeinformationen von objekt-orientierten Programmiersprachen. Wir benutzen diese Ontologien anschliessend zur Markierung derartiger Daten für weitere Softwareanalysen.

Abschliessend stellen wir unsere objekt-orientierte Metrikbibliothek vor, welche das Design von objekt-orientierten Softwareprojekten evaluiert. Mit den Ontologien der Softwaretechnik werden Daten markiert. Die Metrikbibliothek zeigt dann aufgrund dieser Daten mögliche Schwachstellen im Design dieser Softwareprojekte auf.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

In the field of software engineering various tools have been developed to support a software in different activities throughout its life cycle. In many cases the interoperability between these tools is hampered because of proprietary data formats that are used by these tools to store data. This lack of interoperability can yet be overcome with the use of Semantic Web technologies. The Semantic Web is an extension of the current Web in terms of the additional meta-data which is added to the contents already currently present. This meta-data retains the intended meaning of the information from the original author, thus the reference to semantics in the Semantic Web. Since such meta-data is more easily computer-processable it allows a better cooperation between computers and people [3]. Data enriched with this semantic meta-data can consequently be exchanged, shared and reused without loss of the original meaning. In order to have a shared understanding of a particular domain of discourse an ontology has to be defined at first. An ontology is a formal description of the important concepts (classes of objects) identified in the domain of discourse and their relationships to one another. As a result it provides a common vocabulary for a specific domain of discourse which can be used to express the meta-data needed to capture the knowledge of the exchanged, shared or reused data [1].

The benefits of using Semantic Web technologies are not limited to the improvement of the existing Web. Other areas can benefit from these technologies to enhance the interoperability between tools and applications even when they use proprietary data formats and are not designed to cooperate with each other. In the domain of software engineering, for example, numerous tools are used to assist the development, the operation and the maintenance of a software system. Typically the set of tools used in a software system is provided by many software developers; hence the collaboration between these tools is not necessarily incorporated in the tools in the first place.

The aforementioned Semantic Web technologies can improve this cooperation and lead to innovative software solutions that can enhance assistance during the whole software life cycle of a specific project. For this purpose, data from different tools and supporting systems of a project are extracted using an appropriate ontology and are associated with their corresponding meta-data. In this way the information and the intended meaning are preserved for advanced processing, sophisticated investigations and further applications.

Over the past few years, several ontologies have been designed to represent data in the software engineering domain. In many cases these ontologies just covered a specific part of the entire domain. A structured and integrated view of the domain is yet needed to provide an adequate project overview, in the various phases of the software life cycle. The aim of this thesis is to de-

velop a structured set of software engineering ontologies for the representation of data needed throughout the entire life cycle of a software project. For this purpose, a specific domain of discourse has been determined in the field of software engineering.

These software engineering ontologies can then be used to annotate data in the domain of discourse in order to enable further software analyses. Literature in the field of software engineering mentions the crucial importance of reusing source code in the development process of a software application (e.g., [9]). Consequently a software developer has to evaluate whether he implements the desired functionality from scratch or reuses the source code of previous projects; in other words, he has to assess what is best suited for a given set of problems. As the reuse of existing source code is often associated with the time-consuming activity of properly understanding the underlying behaviors, many software engineers prefer to choose the reimplementation. However, with the availability of ontologies and other arising Semantic Web technologies the alternative of reusing source code could soon become the favored choice of software developers. Sophisticated queries, which modern IDEs are currently not able to answer with their incorporated functionalities, could from now on be answered and thus help understand the source code of another person faster and with greater ease. In addition, software metrics based on the above mentioned technologies could evaluate the quality of (reused) source code. These metrics could pinpoint problematic constructs in the source code, which once resolved, would improve the design of the software system. For this purpose, an object-oriented metrics library is implemented in this thesis, which evaluates the design of object-oriented software projects.

1.2 Domain of Discourse

A software project constantly evolves in order to satisfy the requirements of the users and to enhance the functionality of the provided system. This is the subject matter of this thesis. Therefore the domain of discourse comprises the software engineering domain in general, whereas the main focus has been laid out to the following three subareas of this domain.

Revision Control Systems: Whenever multiple software programmers are simultaneously working on the same project these kinds of systems ensure that detailed information about local changes are properly merged into a central repository so that they keep track of the different versions of the software project. Hence, the historical evolution of a project managed with these kinds of systems can be traced back to any point in time in the software life cycle.

Issue Tracking Systems: During the development, the operation and the maintenance of a software project typically a wide variety of issues will be encountered that have not been considered in the first place. This category of systems enables the software developer to report these issues and to assign them a particular priority according to their impact on the functionality of the software system. By indicating this so called bug tracking information all software developers involved in the project can have an overview about the reported issues and in doing so they should ideally solve the issues belonging to their area of responsibility.

Static Source Code Information: Software analyses use static source code information in order to improve the quality of software projects. For object-oriented programming languages, this information typically includes among others the classes, methods and variables declared in software projects and the relationships between these source code entities. Object-oriented metrics analyze this information to indicate weaknesses in the design of object-oriented software projects.

In the course of this thesis, the surveyed systems for the representation of revision control data and issue tracking data are all open source software systems. The investigated systems have been chosen based on their high degree of popularity within the range of possible alternatives.

1.3 Goals

The main goals of this thesis include the development of a structured set of software engineering ontologies for the representation of revision control data, issue tracking data as well as static source code information and the implementation of an object-oriented metrics library in order to evaluate the design of object-oriented software projects.

Therefore, we investigate the state of the art in ontology development in the first part of this thesis. For this purpose, we briefly describe the needs for comprehensive methodologies in order to develop commonly shared and reused ontologies. A more detailed documentation on methodologies, which are widely used for the development of ontologies and which rely on the most intuitive use, completes this presentation. The existing ontologies within the domain of discourse are analyzed with regard to their suitability for integration in our structured set of ontologies.

In the second part, we outline the different requirements for our software engineering ontologies and describe the system implementations and the object-oriented programming languages that are investigated in order to specify our ontologies. We develop SEON, our structured set of software engineering ontologies. Thereby, we define an ontology pyramid for every subarea in the domain of discourse. The top of the pyramid comprises the general concepts of the subdomain whereas the bottom of the pyramid additionally contains system specific or language dependent concepts.

Last, we implement an object-oriented metrics library, which makes use of data annotated with our software engineering ontologies in order to evaluate the design of object-oriented software projects. Thereby, possible weaknesses in the design of these software projects are indicated by our metrics library.

1.4 Overview

The remainder of this thesis is structured as follows: Chapter 2 introduces the Semantic Web technologies used throughout this thesis.

In Chapter 3 we review the state of the art in ontology development. We summarize widely used methodologies for the development of ontologies and analyze whether existing ontologies in the field of software engineering are suited for reuse in our software engineering ontologies or not.

Chapter 4 describes the structure and the vocabulary of our software engineering ontologies. We outline in it the appropriate use of the classes, the properties and the individuals that are specified in our structured set of ontologies.

Object-oriented software metrics and the implementation of our object-oriented metrics library are outlined in Chapter 5. We overview the possible design disharmonies of software projects and the simple object-oriented metrics that constitute these design disharmonies. Further, we present the appropriate use of our object-oriented metrics library and the information provided after the desired investigations. An evaluation of a medium-sized open source software project using our metrics library closes this chapter.

Chapter 6 concludes this thesis. We provide a summary of our contributions within this thesis and describe the limitations of our software engineering ontologies as well as our object-oriented

metrics library. An outline of future work concerning this thesis and appropriate considerations that need to be kept in mind complete this final chapter.

Chapter 2

Semantic Web Technologies

This chapter introduces the Semantic Web technologies used throughout this thesis. Section 2.1 describes the Resource Description Framework (RDF), a data model used to represent meta-data about resources. In Section 2.2 we outline the requirements and the needs for an ontology language standard and introduce the Web Ontology Language (OWL), which aims to fill this gap. The three sublanguages of OWL are also presented in this section. The brief description of the Jena framework in Section 2.3 concludes this chapter.

2.1 RDF

Nowadays, the Extensible Markup Language (XML) is the universally used meta-language in order to specify markup. It facilitates the exchange of data between applications by imposing a well-defined structure on the interchanged data. Thus, it improves interoperability between applications. However, the intended meaning (the semantics) of these data cannot be incorporated in the XML data being exchanged. This means that every application needs to interpret the received data on its own in order to infer the appropriate meaning of these data. For example, there is no intended meaning associated to the nesting of the tags included in an XML document [1].

As the Semantic Web intends explicitly to state and safeguard the intended meaning associated with the data provided in the Web, a more sophisticated meta-language is needed. To achieve this purpose the research community specified numerous standards and W3C recommendations as well as development frameworks, APIs and databases. At that point in time RDF appeared on the stage.

RDF is basically a data model comparable to the entity-relationship model. It is used to make statements about resources in the Web though it is specifically intended to represent the meta-data associated with these resources. In this respect, the described resources do not have to be retrievable in the Web, it is sufficient for them to be identifiable in it. As a result, these resources do not need to be located only in the Web; they can also be located in the real world.

An important aspect concerning RDF is that no assumptions are made about a particular domain of application. Consequently, the specification of RDF is regarded as being domain-independent. The terminology used in a domain of discourse is defined in a schema language and can be worked out by the users themselves. Possible schema languages comprise among others the RDF Schema (RDFS) and OWL which is outlined in Section 2.2. The elaborated vocabulary can then be used to make statements about resources in RDF and incorporates at the same time the semantic information of the domain of discourse. Hence, the intended meaning is provided in a computer-processable way and is meant to be used whenever this kind of information needs to be

further processed by machines rather than being directly displayed on a screen. Therefore, RDF is the standard data model for the representation of semantics in a computer-processable form. The W3C recommendation of RDF attempts to additionally propagate the specification and the utilization of this advanced technology to establish a widely spread user basis [12]. In this way, the functionality and the interoperability of the Web can be enhanced to support data exchange without the loss of the intended meaning from the original author. Applications can share and reuse semantically enriched data by others even if their collaboration has not been considered at the design time.

RDF does not rely on XML, but makes use of an XML-based syntax called RDF/XML to store and exchange the information specified in RDF. By writing down the information in this RDF/XML syntax, it makes it available to machines in a computer-processable way.

The basic building block of RDF is an object - attribute - value triple, which is called a statement. The objects (resources) are described by making a set of statements about them. Hence, their characteristic attributes and the appropriate values of these attributes are specified within these statements. The common terminology used to identify the various parts of a statement comprises the following denominations. First, the subject of a statement designates the resource that the statement is made about. Second, the predicate of a statement denotes a specific property or characteristic of the subject that is defined with the aid of the statement. Lastly, the object of a statement determines the actual value of the predicate, which is assigned to a dedicated subject resource [12]. In order to properly identify each part of a statement, a computer-processable identifier is needed for every individual statement fragment.

According to Antoniou and van Harmelen [1] there are three fundamental concepts in RDF which include resources, properties and statements. The resources are the things that are further described with this kind of meta-data. RDF distinguishes the different resources from one another by means of using Web identifiers called Uniform Resource Identifiers (URIs). Thereby, the assessed URI uniquely identifies a resource among all the others. This does not necessarily imply that the described resources can be directly accessed through their appropriate URIs. Hence, objects from the real world as well as abstract concepts that are physically not existent can also be identified with such a URI. The properties characterize the relationships between resources or between a resource and a specific value. Since these properties represent a special kind of resources, their inherent URI can uniquely identify them even if they can physically not be retrieved in the Web. A significant advantage of using URIs for the identification of resources is attributed to the fact that several distinct pieces of information about the same resource can be interconnected across the Web. This spans an extensive net that contains various information sources about a single resource. Finally, the statements link a property to the desired resource with a specific value. Each of the aforementioned object - attribute - value triples consequently consist of a resource (the subject of the statement), a property (the predicate of the statement) and a value (the object of the statement). The value can either represent a resource or a literal which denotes an atomic value that cannot be described to a greater extent using additional statements. Therefore, literals cannot be used as the subject or as the predicate of a statement.

As already mentioned earlier in this section, RDF is a data model comparable to the entity-relationship model. However, its representation varies in numerous aspects which accounts for its description below. The RDF statements are represented in a directed graph with labeled nodes and labeled arcs. Each resource is modeled as an elliptical node in the graph whereas each literal is drawn as a rectangular node. The arcs interlink the nodes involved of a statement and are directed from the subject to the object of the statement. Thus, they represent the properties (the predicate of the statement) that are specified within the statements. The labels receive their values from the URI of the resource or from the URI of the property respectively. If the label belongs to a literal, it simply assumes the corresponding value of this literal. Every additional statement about a designated resource is modeled accordingly with an additional arc directed to the object of the

new statement. The described resource (the subject of the statement) does not have to be drawn again in the graph model, as it is already contained in the graph model. The illustrated graph model is the actual representation of RDF whereas the previously mentioned XML syntax only demonstrates a possible and convenient way for representing RDF statements.

Another alternative for a more clearly arranged presentation of the different parts of a statement is called triple. In this straightforward notation each statement is written as a triple on a single line in the precise order of subject, predicate and object. This representation contains exactly the same information as the graph model but additionally improves the readability of this type of meta-data.

The vocabulary that is used to express the RDF statements represented in the graph model is actually composed of a set of URIs which is defined for a certain purpose. Basically, this set of URIs can freely be mixed with numerous definitions provided by others and is therefore not restricted to a specific URI prefix. The advantage of using URIs for the identification of resources is obvious. With the use of URIs, the resources can precisely be differentiated in an unambiguous manner even if they share the same name. Consequently, additional information can be declared across the whole Web about an exactly identified resource. In this respect, URIs also support the development and the reuse of a commonly shared vocabulary in a dedicated domain of discourse. Using this common understanding of a domain, an application can then interpret the received information accordingly without losing the semantics of the data. A major disadvantage that follows from the use of URIs is the frequently appearing problem of using different and unrelated URIs to describe the same concept in a domain. Applications designed to assist the users of this specific domain would have to incorporate both definitions in order to satisfy the needs of all these users.

The aforementioned generic literals are atomic values (strings) that cannot be described to a greater extent using supplementary statements. Nevertheless, literals can be assigned with a particular data type (e.g., integer or date) to explicitly state which kind of values are represented by the literals and how they have to be interpreted by an appropriate application. Specifying this additional voluntary information improves the possibilities to process these values in an application for the purpose of providing other features.

The potentially most significant drawback of RDF is the limitation to only allowing binary predicates to be used to make statements about resources and how they relate to one another. According to this, each RDF predicate connects exactly two resources or a resource and a literal in the graph model. However, in some situations the relationships between resources are more complex and cannot be described adequately by using multiple binary predicates. Moreover, RDF cannot impose cardinality restrictions upon a predicate for a specific resource [1].

The vocabulary used within a domain is defined in an ontology definition language such as RDF Schema or OWL which is covered in the following section.

2.2 OWL

RDF and its corresponding schema language, RDF Schema (RDFS), have intentionally very limited expressiveness. As already mentioned in the last section, RDF only allows binary predicates to be used to make statements about resources and how they relate to one another. Similarly, RDF Schema is limited to subclass relationships in the class hierarchy and to subproperty relationships in the property classification. The properties specified using RDFS can additionally be restricted to an appropriate domain and range definition. However, this expressiveness is not sufficient for some typical use cases of the Semantic Web [1].

Therefore, the Semantic Web needs another language in order to display its full potential. OWL aims to fill this gap and intends to be the widely accepted ontology language standard for

the Semantic Web. The requirements that such an ontology language standard has to fulfill are briefly outlined in the following listing [1].

- Well-defined syntax: Since the meta-data information expressed with such an ontology language is processed by computers a well-defined syntax is indispensable. The XML syntax fulfills this requirement which comes along with the drawback of having a lower human readability. Due to the fact that most ontologies are defined using an ontology development tool rather than being directly written in an ontology language, this drawback is not a decisive point for choosing another syntax.
- Formal semantics: With the use of a formal semantics, the intended meaning of information is stated in an unambiguous way. The interpretation of this meta-data is not dependent from the subjective perception of a specific person and consequently enables other people or machines to extract the same knowledge from this meta-data.
- Efficient reasoning support: The aforementioned formal semantics is a precondition for reasoning about the knowledge of a domain of discourse. This reasoning entails among other things the following actions: to check the consistency of an ontology and of the knowledge included in it, to check the correctness of the relationships between classes and to automatically classify instances into classes. The ability to reason about the knowledge of a domain of discourse is especially useful when several authors are developing an ontology, when existing ontologies are reused from various sources, or when large-sized ontologies are designed.
- Sufficient expressive power: The limitations concerning the expressiveness of RDF and particularly RDF Schema have been outlined at the beginning of this section. Roughly, RDF Schema is limited to subclass relationships and subproperty relationships in order to organize the vocabulary into the corresponding class hierarchy and the appropriate property classification. The domain and range restrictions of the included properties can only be specified to apply for all classes within the ontology. An ontology language standard has to provide additional restriction possibilities to define the local scope of a property. In this way, the range definition of a property can be restricted to apply for only some classes in the vocabulary. In addition, an ontology language standard has to provide various description primitives in order to state that classes are disjoint from one another and to specify boolean combinations of classes (e.g., union, intersection or complement). The special characteristics of properties (e.g., inverse, transitive or functional) also need to be covered within an ontology language standard as well as cardinality restrictions for these properties.

Since there is a trade-off between the expressive power and the efficient reasoning support of an ontology language, three sublanguages of OWL have been defined to comply with the different requirements listed above in their own way. These sublanguages are subsequently described in more detail.

- OWL Full: As its name reveals it, OWL Full represents the whole OWL language with all its language primitives. These language primitives are combined with the ones from RDF and RDF Schema to define the structure of the meta-data vocabulary. This includes the ability to apply language primitives to each other, thus changing the predefined meaning of these RDF and OWL primitives.
- OWL DL: This sublanguage represents the description logic sublanguage of OWL. OWL DL imposes restrictions on the usage of language primitives from OWL, RDF Schema and RDF by means of not allowing these language primitives to be applied to each other in order to change their predefined meaning.

- OWL Lite: The OWL Lite sublanguage specifies additional restrictions which limit the OWL DL language primitives to a dedicated subset of these language primitives. For example, the definition of enumerated classes, arbitrary cardinalities for properties and disjoint statements between classes are not supported in OWL Lite.

Before starting with the definition of the vocabulary, an ontology developer has to decide which sublanguage of OWL best suits his needs. Whether an ontology developer chooses the OWL Lite sublanguage or the OWL DL sublanguage depends on the expressiveness that is needed to define the vocabulary of the ontology. In case all language primitives from OWL are intended to be used, OWL DL should be chosen as the appropriate ontology language. Whether an ontology developer chooses the OWL DL sublanguage or the OWL Full sublanguage depends on two aspects. First, it depends on the extent of the reasoning support that is needed for the ontology. OWL DL provides efficient reasoning support for its ontologies whereas in OWL Full no efficient or complete reasoning is possible. Second, it depends on the meta-modeling facilities of RDF Schema that are needed to define additional meta-information for the ontology (e.g., attach properties to classes or define classes of classes). In case such meta-information needs to be stated, OWL Full should be chosen as the appropriate ontology language [1].

The fundamental concepts of OWL include classes, individuals, data type properties and object properties. Classes are groups of individuals that share the same properties. Individuals are instances of classes that are related to one another through the use of appropriate object properties [13]. Data type properties are relationships between instances of classes and either RDF literals [12] or XML Schema data types [4]. Object properties are relationships between two instances of the same class or two different classes [13].

It is important to be aware of the fact that OWL still uses RDF and RDF Schema in many respects. The syntax and the instance declaration are, among others, the most significant similarities between them. In addition, some OWL language primitives are specializations of their RDF counterparts. For example, the `owl:DatatypeProperty` and the `owl:ObjectProperty` are specializations of the `rdf:Property` construct.

Whenever an existing OWL ontology is reused, the corresponding OWL sublanguage that has been used to define this ontology should be identified. This can be achieved by manually checking the ontology definition. For example, in case an interconnection between classes (not instances) is found in the ontology definition this means that the checked ontology must be specified in RDF Schema or OWL Full. Another possibility is to make use of one of the various OWL validators, like the WonderWeb OWL Ontology Validator[1], in order to identify the corresponding sublanguage of OWL.

Our structured set of software engineering ontologies (SEON), which is described in Chapter 4, is defined in the OWL DL sublanguage of OWL, because of the efficient reasoning support provided by this sublanguage.

2.3 Jena Framework

Nowadays, many Semantic Web applications make use of the various possibilities that are offered by Jena[2] to create, modify and query RDF meta-data or OWL meta-data. Therefore, Jena has evolved into a quasi standard for the development of Semantic Web applications. This open source Java framework that has originated from the HP Labs Semantic Web Research[3] provides a programmatic environment for RDF and OWL.

[1] http://www.mygrid.org.uk/OWL/Validator
[2] http://jena.sourceforge.net/
[3] http://www.hpl.hp.com/semweb/

The implementation of the object-oriented software metrics, which is outlined in Section 5.2, makes use of the Jena framework to process the meta-data that has been created with our software engineering ontologies.

Chapter 3

Ontology Development

This chapter describes the state of the art in ontology development. In Section 3.1 we briefly outline the importance of comprehensive methodologies in order to develop commonly shared and often reused ontologies. In addition, we present in detail the methodologies that rely on the most intuitive use from our point of view. These methodologies are likely to be applied on a widespread basis, because of their convenient use for new ontology developers. Section 3.2 discusses existing ontologies in the field of software engineering. We analyze these ontologies with regard to their suitability for integration in our structured set of software engineering ontologies. For this purpose, the intended use cases of these ontologies are described along with the significant concepts that are modeled in these ontologies. We conclude each of these ontology descriptions with several remarks on their strengths and weaknesses.

3.1 Methodologies

The Semantic Web is an extension of the current Web in terms of the additional meta-data, which is added to the contents currently present and that retains the meaning of the information from the original author [3]. Thereby, ontologies provide the commonly shared vocabulary for a specific domain of discourse, which is used to express the desired meta-data about the designated information. Since the vocabulary used for these descriptions depends on its intended purpose, everyone should have the ability to define new ontologies, to extend existing ontologies or to simply reuse existing ontologies. Therefore, various methodologies have been proposed to support the appropriate development of ontologies. These methodologies outline the possible procedures along with their recommended steps in order to define the required ontologies. Each step ideally includes precise descriptions and useful remarks about the various techniques that can be used within the designated phase of the design procedure of an ontology. In addition, the encountered modeling decisions and their possible solutions are illustrated as well as the benefits, the drawbacks and the inherent implications concerning these different solutions. Fernández et al. [7] and Mizoguchi [14] have compared the various methodologies that are currently used for the adequate development of ontologies. Although these comparisons present the differences between the methodologies, they are focused on several aspects that the corresponding authors wanted to point out and are not providing a proper summary of the suggested procedures of these methodologies.

For this reason, the following sections summarize three methodologies for the development of ontologies that are most relevant from our point of view.

3.1.1 Uschold & King Methodology

In their paper [17] Uschold and King propose a skeletal methodology for developing ontologies. Their intention was not to elaborate a comprehensive and ready to use methodology but rather to set the stage for further specification of a methodology for building ontologies. For this purpose they identified several stages that should be covered in any future methodology. Each of these stages is described in their paper in terms of what has already been done in that area and what a future definition of a methodology should ideally include, incorporating the recommended steps in a particular stage as well as between the individual stages (e.g., sequencing, nesting or inputs/outputs).

The envisioned composition of the aforementioned skeletal methodology comprises the following stages:

- Identify Purpose
- Building the Ontology
  - Ontology Capture
  - Ontology Coding
  - Integrating Existing Ontologies
- Evaluation
- Documentation

In this way the skeletal methodology outlines the estimated requirements needed for a successful ontology development methodology.

Besides the existing literature available at that time, their paper is also based on the experiences the authors had in a collaborative project. Thereby, the main part was to build a significant ontology. In this thesis, the desirable content of each stage of the skeletal methodology and the corresponding suggestions as outlined by Uschold and King are briefly presented.

Identify Purpose: This stage of a future methodology should help identify the purpose of an ontology. This includes a proper identification of the intended uses and the envisaged users of the ontology. A survey identifying these potential purposes could provide support to ontology developers who have to clarify their own purposes (e.g., the final ontology will be used by software developers to exchange data between distinct applications). Hence it would serve as a reference document for the numerous possibilities at a rather general level. Another suggestion is to define competency questions for the domain of discourse covered by the ontology. Competency questions are the kinds of questions that should be answered later on by a knowledge base using the created ontology. Since competency questions are defined using specific terms of an ontology they also describe the scope and the expected terminology of the future ontology.

Building the Ontology: This part of an ontology development methodology should typically comprise a procedure for the ontology capture phase, a meaningful list for the ontology coding related options and a decision-making process to identify whether integrating an existing ontology is beneficial or not.

Ontology Capture: The approach suggested by Uschold and King in their paper [17] is devoted to the ontology capture stage of a concrete methodology. This stage is divided into various activities. First, the key concepts and their relationships to one another have to be identified. Afterwards unambiguous text definitions and appropriate terms have to be specified for all key concepts and all their relationships. Achieving a common agreement in all these activities has been proven to be a very cumbersome task yet indispensable for the future use of the ontology.

All the definitions specified in the aforementioned activities should be represented in a language that is situated between a natural language and a formal language in respect to the language's formality. This means that the chosen language should be more formal than a natural language but less formal than a formal language. Such an intermediate representation should then typically include all the assumptions made in the ontology capture phase, their corresponding justifications and carefully worded definitions. A comparable procedure is applied before a knowledge base is coded and hence some expertise found in that field could probably be reused and adapted to the ontology capture process.

An essential part of modeling and thus also of building an ontology is the accurate allocation of all important concepts of a domain in adequate hierarchies. These concepts represent the different categories of objects (often referred to as classes of objects) found in the domain of discourse. The generalization and the specialization of these concepts play an important role in the elaboration of a proper hierarchy. Due to these remarks the general issues of this so-called categorization are briefly described by Uschold and King. For this purpose they introduce a theory of categorization from the field of cognitive psychology [10]. According to this theory the essence of categorization is to identify the basic level concepts. The characteristics of these basic level categories relevant for the development of an ontology comprise several aspects. On the one hand, they are located in the middle of a general-to-specific hierarchy. On the other hand, they are cognitively basic in terms of their usage in communications (since they are the most commonly used term) and their perceptive identification (since individuals are fastest recognized as being members of that basic level category). After the successful identification of the basic level categories the generalization defines the superordinate categories and the specialization details the subordinate categories. Table 3.1 exemplifies the discussed basic level categories and their appropriate superordinate and subordinate categories.

Table 3.1: Examples of basic level categories and their adequate superordinate and subordinate categories [10]

Superordinate	Animal	Furniture
Basic Level	Dog	Chair
Subordinate	Retriever	Rocker

Based on their experience in the field of ontology development, Uschold and King's suggestion is to define a document in natural language which includes all the terms of the ontology and their descriptions. Later on this document could serve as a requirements specification for the ontology coding phase or as documentation for non-technical but nevertheless interested people.

This stage is concluded with an overview of the general approaches available to specify the terms of a categorization that can also be applied in the ontology capture procedure. The top-down approach which was initially proposed describes only a few general terms and specializes them to more detailed ones. The inherent disadvantage of this approach is that important terms could be left out at the beginning and would therefore be missing in the final categorization. The bottom-up approach starts with the definition of a large number of detailed terms and generalizes them to more general ones. The drawback with this approach is that irrelevant terms could be included in the categorization making it less clear to understand. The middle-out approach is

comparable to the basic level categorization mentioned before as it begins with the terms situated in the middle of the categorization followed by the ones resulting from the generalization and specialization processes. Since this approach presents no considerable drawbacks it is suggested to be used in the ontology capture phase.

Ontology Coding: The ontology coding stage of a detailed methodology should describe how the conceptualization captured in the previous phase should be represented in an appropriate formal language. This includes providing sufficient assistance in choosing the suitable representation language for the ontology and it also comprises some guidance in the actual creation of the code. Considering the fact that ontologies have substantial similarities to knowledge bases it may be advisable to look at the existing methodological guidelines in the field of knowledge management. A methodology for building ontologies should consequently clarify the applicability of these guidelines for ontologies. Sometimes the ontology capture phase and the ontology coding phase are merged into one step. Therefore a comprehensive methodology should also illustrate under what circumstances these phases may simultaneously be executed.

Integrating Existing Ontologies: Providing useful guidelines and tools for this stage of a methodology definition is likely to be a very challenging task. The underlying problem is that in order to integrate an existing ontology the developers have to reach common agreement. And since developers usually come from different areas of work, this agreeing process requires a lot of efforts, which may turn out to be time-consuming and unbearable for this purpose. By explicitly stating all the assumptions made in the ontology these efforts could significantly be reduced. Additionally the expected benefits of a possible integration should be taken into account as these could compensate for the efforts mentioned before.

Evaluation: In order to make a statement about the adequate conformity of the created ontology, an evaluation procedure should be suggested in an exhaustive methodology. The compliance with the previously defined competency questions and requirements specifications could provide a considerable starting point for this objective. A promising approach could also be the application of knowledge base specific evaluation mechanisms to ontologies, which requires some adaptation.

Documentation: The importance of a useful documentation is well-known in the field of software engineering. And as an ontology is typically shared by different users, the necessity of an explanatory documentation becomes even more important since the users may have a slightly different understanding of the domain of discourse. In this case, a missing documentation often leads to ambiguous and incomprehensible terms in an ontology. According to the previous reasons and many others it is crucial to write down all the assumptions made during the whole development process of an ontology. Ineffective knowledge sharing due to unfeasible documentations of present ontologies and knowledge bases is still a widely spread problem.

Any future methodology describing how to develop an ontology should provide appropriate techniques, methods, principles and guidelines for each of these stages.

3.1.2 Noy & McGuinness Methodology

As the name *Ontology Development 101: A Guide to Creating Your First Ontology* of this paper [15] reveals it, the methodology presented in this section provides a guide to the establishment of a first ontology. On the documentation page of the official Protégé website[1], this methodology is

[1] http://protege.stanford.edu/

even mentioned as a tutorial for general ontology development guidelines including numerous helpful hints and illustrative examples using Protégé. Therefore the methodology suggested by Noy and McGuinness is widely used in combination with Protégé which is a popular open-source software for the development of ontologies.

Throughout this methodology the modeling decisions that an ontology developer is likely to encounter are presented as well as their possible solutions, thereby pointing out the benefits, the drawbacks and the inherent implications concerning the different solutions. While an ontology developer is specifying an ontology, he should keep in mind various fundamental rules. First of all, for a given domain of discourse there is not exactly one single manner to model that domain correctly. Viable alternatives are largely available in any situation and the right choice crucially depends on the intended usage of the ontology and its potential extensions in the future. Many modeling decisions are affected by the former consideration. Another fundamental rule is that during the ontology development process several steps are repeatedly performed to ensure the incorporation of the desired concepts proposed in subsequent reviews. Thus the whole ontology development procedure can be seen as an iterative process. Furthermore, the concepts included in the ontology should be close to objects in the real world. These objects can either be physical or logical. In this respect, the concepts of an ontology should reflect the reality in an appropriate way since they are a representative model of the reality.

The simple knowledge engineering methodology proposed for this purpose is divided into the following seven steps.

- Step 1: Determine the domain and scope of the ontology

  This step presents a series of questions that should be answered to determine the domain and the scope of the ontology. An ontology developer should typically start to identify the domain of discourse of the planned ontology. Afterwards, the intended usage and possible application cases should be clarified. The different types of questions that the ontology should be able to answer are defined thereafter. Moreover the awareness of the future maintainers and users of the ontology helps to overcome a possible gap between their different understandings of the ontology itself. If the users are not able to understand the maintainer's ontology a mapping to their perspective has to be provided. However this is rarely necessary since the involved parties usually agree on a commonly understandable vocabulary that is used in the ontology. The aforementioned aspects may change during the ontology design process, but at any given point in time they help to delimitate the scope of the ontology.

  To specify the scope of an ontology more precisely competency questions should be defined. As already mentioned in Section 3.1.1, competency questions are the kinds of questions that should be answered by a knowledge base using the developed ontology. These competency questions do not have to be exhaustive; they should rather sketch the most important concepts of the ontology and thus serve as a starting point for the development of the ontology. Additionally a primary evaluation could make use of these competency questions to determine whether the inspected ontology is sufficiently explicit to answer these competency questions or if more detailed information is required in a specific area of the ontology.

- Step 2: Consider reusing existing ontologies

  This possibility should especially be considered if a shared system already interacts with other applications by means of an existing ontology. Adherence to a particular ontology and therefore to a specific vocabulary means reusing the existing ontology and including possible extensions in it. The literature and the Web offer various libraries of reusable ontologies which typically contain ontologies of several fields of interest. For example, the Ontolingua

ontology library[2] or the DAML ontology library[3] could be used to search for existing ontologies in a predefined domain of discourse. Furthermore, publicly available commercial ontologies like UNSPSC[4], RosettaNet[5] or DMOZ[6] could potentially be used. Besides available ontologies, publicly accessible knowledge bases of the specific domain could already exist. The data stored in these knowledge bases could be imported and consequently the well-established information of the domain could be reused accordingly.

- Step 3: Enumerate important terms in the ontology

  All important terms that should be considered in the ontology should be assembled in an initial list. This helps the ontology developer to get a rough overview of the possible constructs, which are used in the next steps of the ontology design process. This list should be created without taking into account the potential overlaps between terms that represent concepts, the relations among these terms or the properties belonging to these terms and whether the terms model a concept, a relation between concepts or a certain property. Nevertheless separating this step from the following two steps is a challenging undertaking, because the three are heavily interconnected. Typically some concepts are extracted from the initial list and their appropriate properties are designated from the remaining terms.

- Step 4: Define the classes and the class hierarchy

  Section 3.1.1 outlines the various approaches to specify the terms of a categorization. These approaches are also applicable for the definition procedure of a hierarchical class taxonomy and are thus presented in this step of the suggested methodology. The top-down approach describes the most general concepts at the beginning and subsequently introduces additional specializations of these concepts. The bottom-up approach however starts with the most specific concepts and subsequently groups them into more generalized concepts. Ultimately the combination approach defines the cognitively most intuitive concepts in place and inserts the required specialization and generalization for these concepts. Whichever approach is best suited for an ontology creator strongly depends on personal views of the domain of discourse. The combination approach is recommendable for many ontology developers due to the fact that these concepts are cognitively simple to gather. All the suggested approaches have in common that the extracted terms from the list solely represent concepts in the examined domain. These concepts can be seen as classes of objects having an independent existence. Terms describing these classes or their objects are selected from the list in the next step. When the classes are arranged in the class hierarchy the condition that an instance of a subclass is also an instance of the corresponding superclass needs to be valid for all the instances of the subclass. Otherwise the subclass needs to be rearranged into a convenient place in the hierarchical taxonomy.

- Step 5: Define the properties of classes - slots

  After the extraction of the terms that represent concepts of the domain of discourse has taken place, the remaining terms usually state the different slots describing these concepts. These slots are often referred to as the properties of the classes and can be classified into three distinct types. Intrinsic properties denominate the characteristics of the concepts that are essential and typically inseparable from the concept itself. The inherent properties of the concepts are thus part of the intrinsic properties. Extrinsic properties of the concepts are the

[2] http://www.ksl.stanford.edu/software/ontolingua/
[3] http://www.daml.org/ontologies/
[4] http://www.unspsc.org/
[5] http://www.rosettanet.org/
[6] http://www.dmoz.org/

kinds of properties that are additionally attached to the concepts through external perception. Moreover, the relationships between instances of classes characterize the third type of slots. A difficult assignment is the proper decision to which class the considered property should be attached. Since subclasses inherit all the properties of their superclasses, they should be attached to the most general class that is capable of having these properties. Ensuring that all classes and their subclasses are indeed able to have the properties belonging to them should be an integral part of this important decision.

- Step 6: Define the facets of the slots

  The facets of the slots delineate the restrictions that are applied on the slots of a class. These restrictions can either concern the value type, the allowed values or the number of values of the designated slot. The value type describes what type of values can be assigned to the slot (e.g., a string, a number, a class, etc.). If a detailed list of all values is available, the slot value is accordingly restricted to all these allowed values. The class to which a slot is attached is referred to as the domain of the slot. The possible values of a slot that could also denote instances of classes are called the range of a slot. The slot cardinality specifies the number of values that a slot can have. Minimum and maximum cardinalities limit the number of values to a chosen range. On the other side, single and multiple cardinalities restrict the number of values to at most one value or any number of values.

- Step 7: Create instances

  In this final step instances of the classes enclosed in the hierarchical taxonomy are created and the required slots of these class individuals are filled out with appropriate values.

With the instantiation of the classes included in the class hierarchy, the end result of the suggested methodology is rather a knowledge base than an ontology. Generally this final step is not comprised in the ontology development process. Since the definition of the class hierarchy with the appropriate classes and their slots represents the most important steps in the design procedure of an ontology, they require careful scrutiny.

The pitfalls encountered during the specification of the classes and their related positions in the class hierarchy are discussed in the following paragraphs.

Presumably the most significant aspect that should be verified continuously throughout the whole life cycle of an ontology is the correctness of the class hierarchy. A common mistake thereby is to include the singular as well as the plural representation of the same concept in the class hierarchy, specifying the former as a subclass of the latter. To avoid this modeling error the ontology developers and maintainers should settle right from the beginning for one of these representations and consequently model all the concepts in such a way. As the represented concepts may probably change over time within the domain of discourse in some decisive characteristics, the hierarchical taxonomy of the classes should also evolve in an appropriate fashion. This involves a proper distinction between the classes and their associated names in the ontology. Inside a domain of discourse a specific concept is represented as a class which in turn is denoted by a name. Miscellaneous ontologies may well have different names for the same concepts in a domain because they are using other terminologies. Ontology developers should therefore keep in mind that synonymous names for several classes do not necessarily represent different concepts.

The next aspect that should be analyzed in a class hierarchy is the group of direct subclasses belonging to a single superclass. These direct subclasses should all have the same level of generality to conform to the requirements of a well-defined class hierarchy. Additionally the number of immediate descendants should be inspected in the hierarchical classification. On the one hand, a single subclass usually indicates a modeling mistake, since the additionally represented subclass often yields all the instances of the superclass and hence does not add any new information to the classification. On the other hand, more than a dozen subclasses constitute reliable evidence that

some intermediary classes are missing in the hierarchy. But if there is no such intermediate categorization in the real world, the class hierarchy should be left unchanged, because in the end the ontology should be an accurate representation of the reality without including many imaginary classes.

A challenging task during the construction of the taxonomy is the decision whether to introduce a new class or a new property to represent the distinction between the different kinds of objects. Bringing in an additional property results in adding different property values to objects that may only slightly vary. The underlying difficulties to navigate very deep class hierarchies (with only a few properties per class) or to understand entirely flat class hierarchies (with excessively many properties per class) have a determining influence on the previously mentioned decision. The general rules presented next could help alleviate the decision-making process. First of all, a new class should be modeled whenever a subclass has additional properties in comparison to the related superclass. If a subclass of objects has different restrictions or participates in other relationships than its ancestor, a supplementary class should be integrated into the classification. The important differentiation between diverse kinds of objects in the domain also suggests the insertion of a new class in the class hierarchy. But if the distinction criteria may frequently change over time they should rather be inserted in the model as another property. The key is to find a balance between the introduction of new classes for different kinds of objects and new properties describing these classes.

The ultimate consideration in the design procedure of the hierarchical taxonomy is the representation of a particular concept as a class or as an instance. There is a fine line between these possibilities and usually it is the intended application of the ontology that tips the scales. In other words, the instances that are going to be stored in a future knowledge base are the most specific concepts used to answer the competency questions defined in the first step. The exception that proves the rule is if the modeled concepts have a naturally predetermined hierarchy. Thereby the hierarchy should be represented in this way even if the classes cannot be instantiated because the structure of the hierarchy already reveals all the desirable information.

In order to avoid that a knowledge base be overloaded with redundant information, only one value of inverse properties should be stored in the knowledge base. Since the inverse properties depend on each other, a system could easily infer the missing value from the given one in the knowledge base.

The understandability of an ontology can be improved with the definition of a naming convention for the classes and the slots and by strictly adhering to it. Additionally this helps the ontology developer avoid common modeling traps. A widely established naming convention which goes beyond the usage in the field of ontology development is the capitalization of class names and the use of lower case letters to denote slot names. Readability of the ontology document can be improved considerably through consistency. If the chosen names consist of more than one word a consistent delimiter should be used between the words. For example, an underscore or a hyphen could be introduced to interconnect the words and in addition every new word could be capitalized. Given that some systems do not handle blanks very well, spaces should not be used to delimit multiple fragments of a name. The continuous denomination of the concepts in the singular or the plural form of the term helps avoid the previously mentioned modeling mistake. For a better readability of the various relations between classes, prefixes and suffixes could be added to the slot names. And since the constructed ontology is typically going to be used by numerous people, the vocabulary should (ideally) only include the commonly known abbreviations of the domain of discourse.

The final remarks that every ontology developer should keep in mind start with a fundamental comment on the completeness of an ontology. This comment contends that an ontology should not model all possible information available about the domain, but rather the information that is important for the intended applications. The design decisions taken for this purpose should be

rigorously documented in order to explicitly retain all the assumptions made about the domain. This comprises the decisions to leave out domain specific information that is useless for the envisaged applications and to add supplementary information that is anyhow necessary for future uses.

Another important aspect to remember is that the outcome of an ontology development process is not deterministic for any given domain. The potential use cases and the designers' personal views and understanding of the domain of discourse strongly influence the decisions that need to be taken during the ontology design procedure. As a result, a specified domain may usually have more than one correct ontology definition.

Ultimately, the best way to evaluate a developed ontology with respect to its quality, its completeness and its satisfaction is by using it for the intended purposes.

3.1.3 Fernández et al. Methodology

The methodology proposed by Fernández et al. [8] intends to clarify the development of an ontology from scratch. This includes the proper identification of the activities that should be performed and in which order they should be executed. Furthermore, a set of guidelines is provided for each phase of the ontology development methodology with useful remarks about the various techniques that can be used.

At the time the authors published their paper, designing an ontology has been compared to producing a piece of art rather than performing an engineering task. The absence of systematic methodologies which describe the activities as well as their intended life cycles are possible explanations for this modified perception. Moreover, appropriate tools that support existing techniques and well-defined design criteria have been missing in the field of ontology development.

When a knowledge base system is to be built the most challenging undertaking that can probably be encountered is the specification of the requirements. These requirements describe in detail how the envisioned system is intended to behave. The difficulty of specifying the behaviors of a future knowledge base system lies with the fact that the envisaged use cases often change over the years. Therefore, this kind of systems is typically created with the use of evolving prototypes, to take the varying requirements into account. Since ontologies are defined to be shared and reused among large communities, they are usually developed independently from the desired behavior of an application. The majority of the core vocabulary of an ontology can consequently be specified without considering these behaviors. The established methodologies in the field of knowledge engineering cannot entirely be applied to ontologies because of the aforementioned difference.

Ontology Development Process

This section identifies the activities needed for the development of an ontology.

- Before a developer starts building an ontology, the main tasks involved in this project should be made explicit. This includes their proper arrangement, the identification of their estimated duration and the resources needed to complete them in time.

- The purpose and the scope of an ontology should also be designated beforehand. The reasons why this ontology is being built and the intended use cases together with their end users are to be found in this activity. The results are recorded in a so-called ontology requirements specification document.

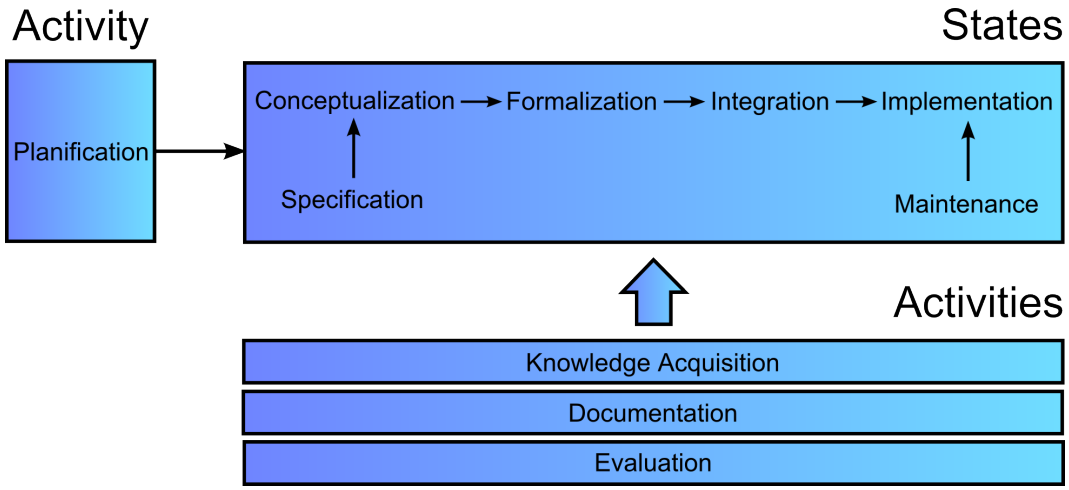

Figure 3.1: Activities and states in the life cycle of an ontology [8]

- Knowledge extraction techniques used for the development of knowledge base systems should be applied to aggregate knowledge about the domain. The processes and techniques adopted for this purpose should be described accurately afterwards.
- After having acquired enough knowledge about the domain of discourse, a conceptual model is specified which describes the problems and its solutions.
- The conceptual model is then transferred into a formal or semi-compatible model with the use of an appropriate representation system.
- Reusing an existing ontology should be considered as often as possible and hence suitable candidates should certainly be integrated into the ontology that is created.
- On the basis of the previously specified formal or semi-compatible model, the vocabulary is implemented in a formal language that is best suited to represent it.
- A crucial precondition for reusing an ontology is the existence of a detailed documentation. Thus, the activity of documenting an ontology should be integrated throughout the whole ontology development process. Without a proper documentation a specific ontology is not going to be shared and reused on a broad base.
- The evaluation of an ontology should always be carried out before the ontology is made available to others. This can be accomplished by using a frame of reference on the basis of which technical judgments are made.
- As end users of the ontology might ask for the modification or integration of a definition in the ontology, maintenance is an important activity and the ontology should be easy to maintain.

Ontology Life Cycle

The previously studied activities can be arranged in a sequence that demonstrates the different phases through which an ontology moves during its life cycle.

This sequence is illustrated in Figure 3.1 and shows that the knowledge acquisition phase, the documentation phase and the evaluation phase are put into practice during the whole life cycle of an ontology. Given that the ontology developer is not an expert in the domain of discourse, most of the knowledge acquisition is made in conjunction with the elaboration of requirements specification. During the remaining ontology development process the knowledge acquisition is continuously reduced. A comparable statement can be made about the evaluation intensity. Most of the evaluation is done in the early phases of the ontology development process to avoid subsequent error transmission from one phase to another. From carrying out the documentation phase over the whole life cycle of an ontology follows the intention to produce a detailed documentation in that way. In addition, Figure 3.1 indicates when a specific activity should be executed and which activity is typically performed next.

As mentioned at the beginning of this description, the ontology development process is not identical to the one of a knowledge base system with respect to the requirements specification. A knowledge engineer faces a major challenge with the appropriate requirements specification because future behaviors of the knowledge base system have to be included. Therefore the life cycle of an ontology is rather related to the one of a software project. However, the waterfall life cycle is not suited for an adequate representation because activities are strictly sequential. This means that the development process cannot move from one activity to another until the previous one is completely finished. The vocabulary provided by an ontology would consequently have to be static, as no terms could be added, removed or modified from the ontology at a later phase. Clearly this restriction is not desirable since ontologies usually evolve in accordance with their domain and the requirements specifications are mostly incomplete in the earliest stages of the ontology development process. Although the incremental life cycle permits a partial specification of the requirements, the evolutionary aspect is not satisfying the needs of an ontology. Thereby, an ontology could only grow by additional layers and when a new version is scheduled. The evolving prototypes fulfill the needed characteristics to represent the life cycle for building ontologies. The vocabulary of an ontology evolves with the requests of the end users and their needs whereas definitions are added, removed or modified at any time to refine the specification of the terms.

METHONTOLOGY: A Methodology to Build Ontologies from Scratch

The previous sections have set the stage for the comprehensive methodology presented by Fernández et al. to build ontologies from scratch. The remaining of this description will provide detailed insights into the various techniques that are used in the different phases of the ontology development process.

Specification: The outcome of the specification phase is an informal, semi-formal or formal ontology specification document. Generally it is written in natural language and comprises the definition of a set of competency questions or intermediate representations. This document should include at least three kinds of information. First of all, the purpose of the ontology should be stated with detailed information about the intended uses, possible scenarios of use and the envisaged end users. Secondly, the formality of the implemented ontology should be specified which depends on the formality that will be used to write the final code for the terms and their meaning. Available values to describe this formality are highly informal, semi-informal, semi-formal and rigorously formal. Finally, the scope of the ontology is laid down with the definition of the terms, their characteristics and their granularity which will all be represented in the ontology in an appropriate manner.

To determine the formality of the whole ontology specification document further information has to be taken into consideration. A document is assessed as informal if it is entirely defined in natural language. Whenever competency questions are used in the ontology specification docu-

ment it causes the formality to be assessed as formal. Using the middle-out approach to specify the document results in lowering formality to semi-formal. The last-mentioned approach helps verify the completeness of the relevant terms and ensures that the irrelevant ones are excluded from the definition. As a consequence the most important concepts of the domain are modeled in the first place. After reaching an agreement on these essential concepts, the necessary specialization and generalization is performed. This leads to a more stable core of decisive concepts that requires less reworking and thus reduces the overall effort significantly.

The ontology specification document is characterized by the completeness and the conciseness of the considered terms which are only included if they are relevant enough and if their meaning makes sense in the domain of discourse.

Knowledge Acquisition: Knowledge acquisition is an independent activity that is performed during the whole life cycle of an ontology. The most part of the knowledge acquisition process is done simultaneously with the requirements specification. As ontology development progresses to the next phases, the intensity of knowledge acquisition decreases continuously.

There are several techniques that can be applied to extract the required knowledge from the numerous sources containing it. For example, brainstorming, structured and unstructured interviews, formal and informal analysis of texts or knowledge acquisition tools can be used for this purpose. On the other hand, the various sources of knowledge lie in books, handbooks, figures, tables, domain experts and other similar ontologies. If the purpose of an ontology is obvious, brainstorming, informal interviews with experts and the examination of resembling ontologies gather enough information to specify an initial glossary with the potentially relevant terms in the domain. The formal and informal analyses of texts are subsequently used to refine the glossary of terms and their meaning. Additionally, the structured and unstructured interviews conducted with the domain experts reveal which terms have respectively to be included in or removed from the glossary. These interviews are also helpful for the development of an initial classification of the concepts which is compared to the figures found in books.

Conceptualization: This phase of an ontology development process defines an enhanced glossary of terms with respect to the investigated terms of the specification phase. This glossary should contain all potentially important terms of the domain and their associated meaning. At this stage the terms include concepts, instances, verbs and properties. In the following procedure the terms are grouped by concepts and verbs that are closely related to the other members of the group. For each of these groups either concepts classification trees or verbs diagrams are built as depicted in Figure 3.2.

The concepts are initially described in a data dictionary which includes information about all potentially important domain concepts along with their meaning, their attributes and their instances. The tables of instance attributes comprise additional information about the attributes of instances and their values whereas the tables of class attributes denote the same kind of information but on the class level. Moreover, the tables of constants define domain specific information that always takes the identical value. Instances are characterized in the tables of instances and the attributes classification trees graphically show how the attributes and constants are related to the root attributes. These details are typically inferred by the use of a sequence of formulas or rules that should also be integrated in these classification trees.

The verbs represent actions in the domain of discourse and are described using a verbs dictionary and tables of conditions. The former gathers information about the meaning of each verb in a declarative way and the latter states the preconditions that need to be fulfilled to execute a specific action.

Finally, the tables of formulas and the tables of rules aggregate the needed data about formulas and rules for concepts as well as for verbs.

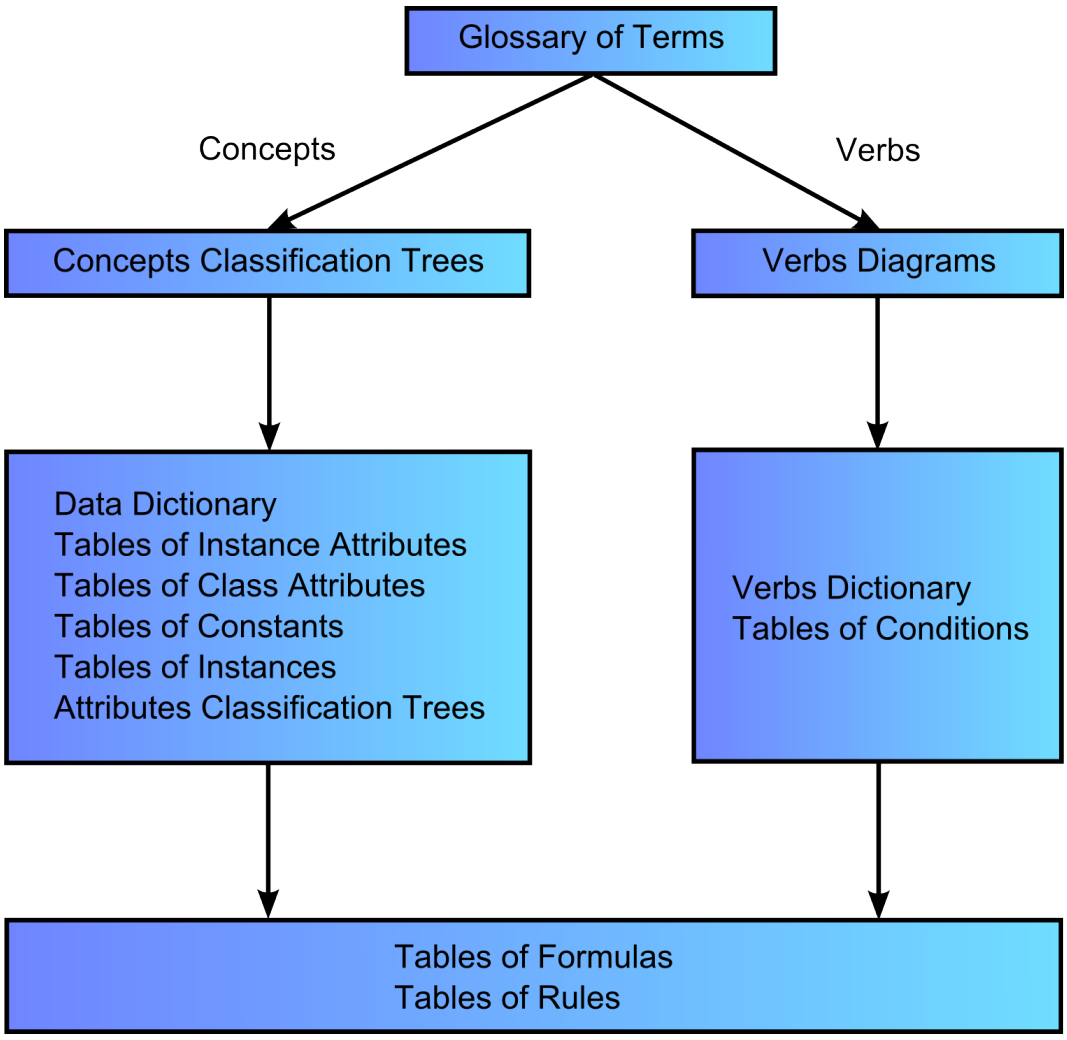

Figure 3.2: Conceptualization phase in the ontology development process [8]

Integration: To accelerate the development of an ontology, the definitions already available should be considered for integration into the vocabulary. To begin with, the existing (meta-) ontologies are reviewed as potential candidates for this purpose. According to Fernández et al. an existing definition of an ontology should be reused if it is based on the same set of essential terms than the intended vocabulary. Hence, the specification and implementation of a new ontology in a formal language should start at this time unless an appropriate definition can be found. Nevertheless, the possibility of including individual terms from other libraries of ontologies should be considered whenever identical definitions with the same semantics and implementations can be retrieved. If the definitions and implementations provided by these (meta-) ontologies differ from the intended one, a corresponding translation into the target language needs to be tracked down.

A so-called integration document is the proposed outcome of the integration phase. This document contains information about which meta-ontology is used and for every term in the conceptual model it includes additional details about the envisaged name in the ontology, the name of the existing ontology that specifies the definition for this term and the corresponding name of the term in the other ontology.

Implementation: The proper implementation of an ontology requires an ontology development environment that adequately supports the (meta-) ontology and all selected ontologies of the integration phase. The outcome of this phase is an ontology that is specified in a formal language.

Evaluation: As previously mentioned, the evaluation of an ontology is a technical judgement made with respect to a frame of references which typically includes the requirements specification document. Furthermore, the ontology development environment used and the available documentation can be evaluated during this activity.

The evaluation activity includes the verification activity and the validation activity. On the one hand, the verification process examines the correctness of an ontology, the software environment used and the established documentation. On the other hand, the validation process investigates whether the evaluated units correspond to the ones that they are supposed to represent. Reconsider that the evaluation activity is performed during each phase and in-between the phases of the entire life cycle of the ontology.

The evaluation documents resulting from this phase describe how an ontology has been evaluated and what corresponding techniques have been used in the process. Moreover, the errors found for each activity as well as the sources of knowledge used in the evaluation are listed in these documents.

Documentation: Figure 3.1 illustrates the fact that the documentation activity is performed throughout the whole life cycle of an ontology. Since a detailed documentation is a crucial precondition for the reuse of an ontology, this methodology incorporates the documentation activity into the entire ontology development process and produces appropriate documents during each phase of this process. For example, the outcome of the specification phase is a requirements specification document and after the knowledge acquisition phase a resulting knowledge acquisition document can be inspected. During the conceptualization phase a conceptual model document is created, which describes the application domain with a set of intermediate representations. The formalization, the integration, the implementation and the evaluation stages produce each one a corresponding document, namely a formalization document, an integration document, an implementation document and an evaluation document, which complete the comprehensive documentation of the ontology.

We have specified our software engineering ontologies according to the ontology development procedure described in the methodology suggested by Noy and McGuinness. We have chosen

this methodology because it is heavily based on the ontology definition with Protégé, which is a popular open source software tool for the development of ontologies and which we used to define our software engineering ontologies.

3.2 Existing Ontologies

The ontology development methodologies described in the previous sections underlined the importance of reusing existing ontologies. It is crucial to agree on a common vocabulary for a designated domain of discourse in order to share the desired data and its intended meaning beyond application boundaries. If several different ontologies are generally used in the same domain, application developers would have to support a proper interpretation of all these domain specific ontologies in order to satisfy the requirements of all data providers.

For this reason we have initially studied various existing ontologies that are widely spread in the field of software engineering. The subsequently described ontologies turned out to be relevant for our needs and for the purpose of developing a series of software engineering ontologies. Each of the outlined ontologies covered a certain part of our domain of discourse and belonged to the most popular ontologies used in that area.

We have divided up the descriptions of these ontologies with respect to the following aspects. First of all, the selected ontology is briefly elucidated with a listing of its fundamental characteristics. These include writing out the abbreviated name of the corresponding ontology, giving a concise summary, stating the website URL as well as the namespace URI and itemizing the most important concepts within the provided vocabulary. Second, the motivation behind the definition of the selected ontology is described along with the intended use cases of this ontology and the schema language that has been used to specify it. Afterwards, the significant classes and properties of the vocabulary are discussed in more detail. This outlines the coverage and the boundaries of the investigated ontology. Finally, we conclude each ontology description by stating its strengths and weaknesses from our point of view.

3.2.1 DOAP

Name:	Description of a Project
Summary:	The DOAP ontology specifies concepts to describe (open source) software projects, the different versions of these projects and the repositories where these versions are stored
Website:	`http://trac.usefulinc.com/doap`
URI:	`http://usefulinc.com/ns/doap#`
Important Concepts:	`Project`, `Repository` and `Version`

The rationale for the development of this ontology can be found in the difficulty for an open source software developer to keep all software registry entries up to date with any given project information. Similarly, an open source software user often faces the difficulty to locate open source software suited for the desired purposes. Although several software registries (like Freshmeat[7], SourceForge[8] and GNOME[9]) are available, the main problem for a user remains in the fact that these entries are not possibly all kept up to date. Because the release cycles of open source software tend to be short and the software registry entries need to be updated manually there is no guarantee that the published information is valid at the time it is considered.

[7] http://freshmeat.net/
[8] http://sourceforge.net/
[9] http://www.gnome.org/

The DOAP ontology is intended to be used whenever such open source project data needs to be described and subsequently shared between a number of involved parties. The majority of the DOAP ontology specification is defined in RDF Schema (RDFS). However, a small number of OWL specifiers are used to refine this definition.

Coverage & Boundaries: As outlined in the initial goals of this ontology, the main use of the DOAP vocabulary is to describe (open source) software projects, the participants involved and resources associated with them. These descriptions can then be used to facilitate the import of projects into software registries as well as to ease the exchange of data between these registries. Since an international use is envisaged all comments about the classes and properties included in the DOAP ontology are written in English, French and Spanish.

As expected the main concept represented in the DOAP ontology is the `Project` class which is uniquely identified through its website URL. Due to the fact that the website of a project may change, there is an additional property to denote the previous URLs in order to ensure the further use and validity of older project descriptions. The `Version` class designates the current release of a software project whereas the `Repository` class embraces the information regarding public access source code repositories which are typically read-only repositories. This removes the need to model access restrictions for writeable repositories. The provided subclasses of the `Repository` class, namely `CVSRepository`, `SVNRepository`, `BKRepository` and `ArchRepository` allow a preliminary distinction between the most popular revision control systems. The numerous people involved in a software project (e.g., a developer, a maintainer or a tester) are described with the FOAF vocabulary, which is outlined in Section 3.2.2.

Strengths & Weaknesses: A relevant advantage of the DOAP ontology is the high interoperability with other widespread meta-data projects. The successful use of the FOAF vocabulary to describe the people participating in a software project exemplifies this strength. Additionally, the DOAP vocabulary is designed to enable future extensions for specialized purposes. Not assigning cardinality restrictions to the properties results in project descriptions, which possibly contain multiple instances of a specific property. However, domain and range constraints are defined for their properties to clarify the potential classes involved in the relationships.

The major weakness that comes along with this ontology is the fact that properties having a literal as their range definition are not restricted to a specific data type. Applications based on the DOAP ontology have therefore a limited possibility to continue processing these literals, because their value can represent almost everything (e.g., a date, a name or a number). Another disadvantage is that the internal planning data of a project, such as milestones or task assignments, are not considered within the scope of this ontology.

3.2.2 FOAF

Name:	Friend of a Friend
Summary:	The FOAF ontology specifies concepts to describe people, the relationships between these people and the objects related to them
Website:	`http://www.foaf-project.org/`
URI:	`http://xmlns.com/foaf/0.1/`
Important Concepts:	`Agent` (including `Person`, `Group` and `Organization`), `Document`, `OnlineAccount` and `Project`

Since people interconnect most of the other objects in real life as well as in the Web, the creation of a commonly shared vocabulary to describe these people opens up a whole new field of possible applications enhancing the current use of the Web. Defining such computer-processable data about people and interlinking this data among each other leads to a more machine-friendly

version of today's hyperlinked Web. Machines can then make use of the cross-references among the documents to go through and accumulate new information about other people and the objects related to them.

Another purpose for the use of the FOAF ontology could be as a useful component of an information system supporting online communities. This kind of system helps manage online communities by providing them additional information that could possibly be of interest. For example, people with similar interests or new members seeking for assistance could be presented to the interested users of an online community.

The FOAF ontology specification has been designed to be applicable with a wide variety of generic services and tools that have been developed for the Semantic Web.

Coverage & Boundaries: The FOAF vocabulary is intended to be used to describe people, their relationships to other people and the things they create and do. The thus gathered information can be shared and reused online by means of transferring it between different websites. Moreover, numerous FOAF descriptions can be combined and merged to create a unified database of information about people and how they are associated to other objects. People are pictured in images, they attend conferences and they create documents: all these are feasible relationships to objects in the FOAF ontology. However, the exploration of cross-referenced FOAF documents and the concurrent aggregation of the data included in them spans a large spectrum of the Web for computer-processable information sources about people.

Before the represented concepts are elucidated in this section, the basic evolution of FOAF vocabulary terms is illustrated. Usually the terms of an ontology evolve with the publication of a new vocabulary specification. The modifications typically affect a small number of negligible terms that have either been omitted or introduced into the vocabulary. On the contrary, the terms of the FOAF ontology mature with respect to their stability in the ontology. Newly adopted terms are labeled with a stability status of unstable to express the probability that these terms are presumably going to be changed in a future specification. As they stabilize in their usage and documentation, their status advances to testing and finally reaches a stable state.

In the domain of discourse specified for the FOAF ontology, the most significant concept is represented by the `Person` class. Each instance of this class is uniquely identified through a particular e-mail address. Regarding the classification within the hierarchical taxonomy, the `Person` class is modeled as a subclass of the `Agent` class which additionally contains an `Organization` subclass and a `Group` subclass. There are other classes defined at the same level of generality as the `Agent` class, namely the `Document` class, the `OnlineAccount` class and the `Project` class.

Strengths & Weaknesses: Cross-referencing FOAF documents is a powerful mechanism that enables machines to go through and simultaneously aggregate descriptions about a large number of people. However, the FOAF documents missing these cross-references are not included in this procedure. To enable machines to find the latter kind of FOAF documents with ease, additional markup has to be embedded in the HTML source of the website that hosts the FOAF documents. This possibility is referred to as the auto-discovery of FOAF documents. An important advantage of using RDF for the definition of an ontology is the inherent ability to extend the provided vocabulary for specific purposes. This extensibility is especially needed whenever such complex concepts as people and their boundless relationships to other objects are modeled in an ontology. According to this, the FOAF ontology only represents the fundamental aspects concerning people leaving the detailed definitions to ontology developers that require them.

Although the core of the FOAF ontology is stable, the majority of the properties and some classes are still in the unstable state or in the testing state. Even the essential naming constructs are involved among these properties. Furthermore, the FOAF vocabulary contains various properties

which are either obsolete, like the Geek Code[10], or only included to reflect the history of the internet, such as .plan files of UNIX machines. The weakness of the other FOAF properties is to be found in the generic range definition which in principle imposes almost no restrictions. The concluding remark that the FOAF ontology is defined in OWL Full entails additional constraints for the possible reasoning on an ontology.

3.2.3 EvoOnt

Name:	A Software Evolution Ontology
Summary:	The EvoOnt ontologies specify concepts to describe revision control data, issue tracking data and object-oriented source code information
Website:	`http://www.ifi.uzh.ch/ddis/evo/`

URI:	`http://www.ifi.uzh.ch/ddis/evoont/2008/11/vom/`
	`http://www.ifi.uzh.ch/ddis/evoont/2008/11/som/`
	`http://www.ifi.uzh.ch/ddis/evoont/2008/11/bom/`

Important Concepts:	VOM: `Path` and `Release` BOM: `Issue` (including `Defect` and `Enhancement`), `Activity`, `Attachment`, `Comment`, `Component`, `ComputerSystem`, `Milestone`, `Priority`, `Product`, `Resolution` and `Severity` SOM: `Entity` (including `BehaviouralEntity`, `Context` and `StructuralEntity`), `BehaviouralEntity` (including `Function` and `Method`), `Context` (including `Class` and `Namespace`), `StructuralEntity` (including `Attribute`, `FormalParameter`, `GlobalVariable`, `ImplicitVariable` and `LocalVariable`)

Over the years, software developers have experienced various changes in the development of software projects, which affect many different aspects. Software projects have become more and more complex, which is why nowadays large-scale software projects are considered as the norm rather than the exception. In addition, software developers had to get used to working in distributed teams so as to shorten the release cycles of software projects. In order to assist software developers throughout the entire life cycle of software projects, various tools have been developed in the last decades. These tools include revision control systems and issue tracking systems, which are often referred to as bug tracking systems due to the fact that they primarily store the reported bugs of software projects. Hence, different information about software projects is distributed among several tools and needs to be brought together whenever a meaningful overview of this project information is needed for further software analyses.

The EvoOnt ontologies are intended to be used whenever such software project information is extracted from the various repositories of these tools and consequently needs to be described appropriately. Therefore, the EvoOnt ontologies represent the significant concepts that are found in revision control systems, issue tracking systems and object-oriented programming languages. The extracted information that is annotated with these ontologies is subsequently used for software analyses.

The EvoOnt ontologies are defined in OWL. Since OWL has evolved into a quasi standard for the definition of ontologies, plenty of tools enable among other things uncomplicated and direct specifications, modifications and visualizations of OWL ontologies.

[10] http://www.geekcode.com/

Coverage & Boundaries: As previously mentioned, the main use of the EvoOnt vocabulary is to describe the extracted data of revision control systems, issue tracking systems and object-oriented programming languages. These descriptions can then be used to exchange data between such tool repositories and to analyze different aspects of object-oriented software projects.

In this connection, the Version Ontology Model (VOM) defines the classes and properties that are required to represent the significant concepts within revision control systems and the relationships between these concepts. The reference system that has been used for this definition is the CVS implementation.

The Bug Ontology Model (BOM) specifies the classes and properties, which model the concepts and their appropriate relations within issue tracking systems. Thereby, the Bugzilla implementation has been used as the reference system to represent the corresponding classes and properties.

Finally, the Software Ontology Model (SOM) includes the classes and properties that are required to describe the fundamental concepts within object-oriented programming languages. For this purpose, the object-oriented source code information is modeled on the basis of the FAMIX meta-model [6]. This meta-model defines the structure of object-oriented source code and the links between these structural entities. EvoOnt has adapted the FAMIX meta-model for the SOM ontology and has extended it with reasonable classes and properties.

Strengths & Weaknesses: EvoOnt reuses various existing ontologies to define its entire vocabulary in order to represent the data that is extracted from revision control systems, issue tracking systems and object-oriented programming languages. Another eye-catching strength of the EvoOnt definitions is to be found in the consistent domain and range restrictions of the properties which clarify their intended use. Additionally, adherence to the commonly known naming conventions improves the readability and the understandability of the EvoOnt vocabulary.

A considerable weakness of the EvoOnt ontologies is that the Version Ontology Model (VOM) and the Bug Ontology Model (BOM) are defined for two specific systems, namely the CVS revision control system and the Bugzilla issue tracking system. Ontology developers, that model other revision control systems or issue tracking systems with an appropriate ontology have to evaluate each class and property of the corresponding EvoOnt ontology and decide whether it can be reused in their system representation or not, because it is not suited to be incorporated in their ontology. Only after this cumbersome procedure, ontology developers have to define the missing classes and properties, which they need in order to model the revision control system or issue tracking system in an appropriate way. From our point of view, an additional modeling weakness within the Bug Ontology Model (BOM) is to be found in the degree of detail that is represented with several subclasses. For instance, the preconfigured choices to describe the priority, the resolution and the severity of an issue within the Bugzilla implementation are modeled as separate subclasses of the `Priority` class, the `Resolution` class and the `Severity` class. Upon closer examination, one can notice that defining several instances of these subclasses does not make much sense as these instances are always representing exactly the same choice within the Bugzilla implementation. For example, instances of the `P1` subclass always represent the highest priority that can be assigned to a specific issue. A more adequate ontology design should define separate instances of the concerned superclasses in order to replace this kind of subclasses.

3.2.4 NEPOMUK Ontologies

Name:	Networked Environment for Personalized, Ontology-based Management of Unified Knowledge
Summary:	The NEPOMUK ontologies specify concepts to describe the Social Semantic Desktop
Website:	`http://www.semanticdesktop.org/ontologies/`
URI:	`http://www.semanticdesktop.org/ontologies/2007/11/01/pimo#`
Important Concepts:	PIMO: `PersonalInformationModel`, `Thing`, `Agent` (including `Person`, `Organization` and `PersonGroup`), `Collection`, `Location`, `LogicalMediaType`, `ProcessConcept` (including among others `Event`, `Meeting` and `Project`) and `Topic`

Nowadays, the way people share their knowledge across social and organizational networks heavily depends on the tools they use to create and share their data. Whenever these tools make use of a proprietary data format, people have to cope with the inherent difficulties of properly interpreting the received data. The exchanged data is typically stored in files or groupware systems, which include text documents, contact information, appointments or Enterprise Resource Planning (ERP) systems. The diversity of file formats and groupware systems makes it difficult to exchange appropriate information or to collaborate.

The NEPOMUK project intends to realize a comprehensive solution to overcome these difficulties. For this purpose, the personal computer will be extended into a collaborative environment which generally improves collaboration between people. In addition, this collaborative environment will provide a novel approach for personal data management with regard to organizing and providing information, which has been created by a single person or a group of persons.

This comprehensive solution is called the Social Semantic Desktop and is intended to be realized with appropriate methods, data structures and a large set of tools. In the context of the NEPOMUK project, the desktop of a person, which is also referred to as the personal knowledge workspace of a person, will be extended in terms of giving data a well-defined meaning. As a result, this semantically enriched data can be processed by computers and lead to innovative software solutions that make use of this information. The Social Semantic Desktop will additionally support the interconnection and the information exchange between these desktops and their users.

The NEPOMUK ontologies are intended to be used to describe essential concepts, which realize the vision of the previously mentioned Social Semantic Desktop. Thereby, the Personal Information Model (PIMO) ontology represents the concepts that are needed to model all the participating people and things within the personal knowledge workspaces of these people.

PIMO is defined in RDFS and the NEPOMUK Representational Language (NRL), which extends the RDF model with a semantic model and provides the desired support for named graphs in semantic statements as well as other necessary features to realize the Social Semantic Desktop [16].

Coverage & Boundaries: As outlined in the previous section, the main use of the NEPOMUK vocabulary is to describe the concepts that are present in the Social Semantic Desktop vision. Although the NEPOMUK ontologies were initially designed to conform to the requirements of the NEPOMUK project, they can be beneficial to the Semantic Web community in general.

The Personal Information Model (PIMO) ontology defines the classes and properties that are required to express the Personal Information Models of individuals [16]. These Personal Information Models describe the concepts in the environment of a user with reference to the data that is relevant to the user for either personal use or knowledge management. In this respect, the atten-

tion is mainly focused on the data used within a Social Semantic Desktop or another personalized Semantic Web application.

Strengths & Weaknesses: The NEPOMUK project has various established partners within international IT organizations and Semantic Web research facilities, which support the realization of the Social Semantic Desktop vision with the included ontologies. Therefore, these ontologies will potentially play an important role within the Semantic Web in the near future.

Although the properties of the PIMO ontology are carefully specified with regard to the appropriate domain and range restrictions as well as minimum and maximum cardinality definitions, their intended meaning and use are sometimes difficult to grasp. This can be traced back to the fact that these relationships are not present between the concepts in their daily use, but in exchange they are fundamental for the realization of the Social Semantic Desktop vision. For example, it is difficult to grasp the intended meaning and the appropriate use of the `groundingOccurrence` property or the `referencingOccurrence` property (not to mention the implied difficulty to actually distinguish these properties), because these relations are not typically used to characterize their concepts in the daily use.

Chapter 4

Software Engineering Ontologies

This chapter describes the structure and the vocabulary of our software engineering ontologies. During the development of our ontologies we conformed to the different requirements identified in Section 4.1. The general structure of our ontologies is outlined in Section 4.2 along with the investigated systems and object-oriented programming languages that have been used to define our set of software engineering ontologies. In Section 4.3 we introduce the vocabularies of our version ontologies (namely the Version ontology, the CVS ontology and the SVN ontology), which model the data found in revision control systems. Section 4.4 discusses the vocabularies of our bug ontologies (namely the Bug ontology, the Bugzilla ontology and the Trac ontology) representing the data of issue tracking systems. Section 4.5 concludes this chapter with the description of the vocabularies included in our code ontologies (namely the Code ontology, the Java ontology and the CSharp ontology), which specify the static source code information of object-oriented programming languages.

The URI prefixes used throughout these documentations are illustrated in Table 4.1. These prefixes indicate in an unambiguous way in which ontology a specific term is defined. However, the terms defined in the discussed ontology are not extended with their corresponding prefix in order to improve the readability of these definitions.

4.1 Requirements

Before starting with the development of our software engineering ontologies, we have identified various requirements that should be fulfilled by these ontologies.

First, the structure of our software engineering ontologies should be designed to be extensible in a straightforward way. In this respect, supplementary ontologies describing new systems or new object-oriented programming languages should be added to this structure as a completely new ontology without influencing or changing the definition of the previously specified ontologies. Therefore, the ontologies defined in each subarea of the domain of discourse are arranged in a pyramid structure. The system specific ontologies or language dependent ontologies at the bottom of the pyramids can make use of the more general ontologies at the top of the pyramids by importing their classes and properties and refining them according to their needs.

Second, the vocabulary provided by our software engineering ontologies should be intuitive in its appropriate usage. As a result, unambiguous and comprehensible terms should be used to specify the concepts and their relationships to one another in the domain of discourse.

Table 4.1: URI prefixes used in the documentation of the software engineering ontologies

URI Prefix	URI
doap	http://usefulinc.com/ns/doap#
foaf	http://xmlns.com/foaf/0.1/
version	http://www.evolizer.org/ontology/Version.owl#
cvs	http://www.evolizer.org/ontology/CVS.owl#
svn	http://www.evolizer.org/ontology/SVN.owl#
bug	http://www.evolizer.org/ontology/Bug.owl#
bugzilla	http://www.evolizer.org/ontology/Bugzilla.owl#
trac	http://www.evolizer.org/ontology/Trac.owl#
code	http://www.evolizer.org/ontology/Code.owl#
java	http://www.evolizer.org/ontology/Java.owl#
csharp	http://www.evolizer.org/ontology/CSharp.owl#

The ontology development methodologies described in the previous chapter stressed the importance of a detailed and explanatory documentation so that ontologies be shared and reused on a widespread basis. Therefore, providing such a detailed and explanatory documentation is an essential part of the requirements that our software engineering ontologies should fulfill.

In addition, the understandability of an ontology can be improved with a proper description of the design decisions taken throughout the development process of this ontology. In the field of architectural decision engineering, Zimmermann et al. introduce a collaboration system which facilitates the knowledge exchange of architectural decisions beyond project boundaries [19]. Their collaboration system has already shown to be very useful whenever architectural decisions have to be shared and reused for the development of large-scale enterprise applications. Therefore, capturing and sharing the design decisions made during the development of our software engineering ontologies are also included in the requirements of our ontology documentations. These documentations explicitly state all our assumptions about the domain of discourse and provide a starting point for future discussions and improvements concerning our software engineering ontologies.

Finally, the static source code information in our domain of discourse is used in order to improve the software quality of an object-oriented project with the indication of the corresponding metrics. Hence, appropriate classes and properties have to be incorporated in our software engineering ontologies to enable evaluating and improving the design of object-oriented software projects.

4.2 Investigated Systems & Object-Oriented Languages

For the purpose of our software engineering ontologies, we have investigated the commonly known and widespread revision control systems, issue tracking systems and object-oriented programming languages. In doing so, we have limited our selection procedure to open source software systems and object-oriented programming languages that are not commercially sold.

In each subarea of the domain of discourse, we have identified two adequate systems or object-oriented programming languages that were subsequently analyzed. At first, we have determined the similarities between the two systems or object-oriented programming languages (of one subarea of the domain of discourse) and have then defined appropriate classes and properties for

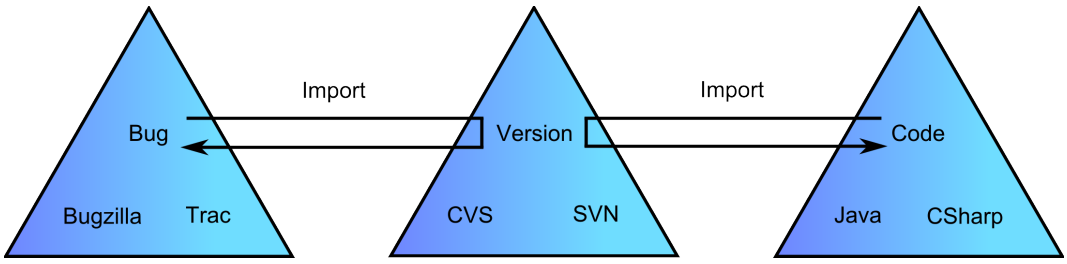

Figure 4.1: SEON: Structured set of software engineering ontologies

their similar concepts and the comparable relationships between them. Therefore, these classes and properties represent the common vocabulary used in that subarea of the domain of discourse. The system specific or language dependent concepts are incorporated in the corresponding ontologies of the revision control systems, issue tracking systems and object-oriented programming languages.

Among the revision control systems we have chosen the CVS implementation [2] and the SVN implementation [5] as our information sources to develop the desired ontologies. In the subarea of issue tracking systems, the Bugzilla system and the Trac system were considered as providing the required information for our ontologies. As for object-oriented programming languages, we have chosen to investigate the Java specification and the C# specification to represent the static source code information we need.

Figure 4.1 illustrates the basic structure of our software engineering ontologies. The three subareas of the domain of discourse are represented as individual ontology pyramids whereas the shared classes and properties are defined in the ontologies located at the top of the pyramids. The system specific ontologies or language dependent ontologies import the common classes and properties of the ontology located above them in the same pyramid. Due to these imports the ontologies are in a position to reuse all the classes and properties that are specified in the imported ontology.

4.3 Version Ontology Pyramid

The version ontology pyramid is at the core of our software engineering ontologies because it interconnects the three subareas of the domain of discourse. The significant concepts of the investigated revision control systems and the fundamental relationships between them are modeled in these version ontologies. On the one hand, the similarities between these systems are specified in the Version ontology. On the other hand, system specific concepts are defined in the CVS ontology and in the SVN ontology.

4.3.1 Version Ontology

The Version ontology reuses several classes and properties from two already existing ontologies, namely DOAP which is briefly described in Section 3.2.1 and FOAF which in turn is outlined in Section 3.2.2. A detailed documentation of the reused classes and properties can be retrieved on the corresponding website of these ontologies. The class hierarchy and the corresponding property classification of the Version ontology are illustrated in Figure 4.2 and provide an initial overview of the classes and properties that are included in this ontology.

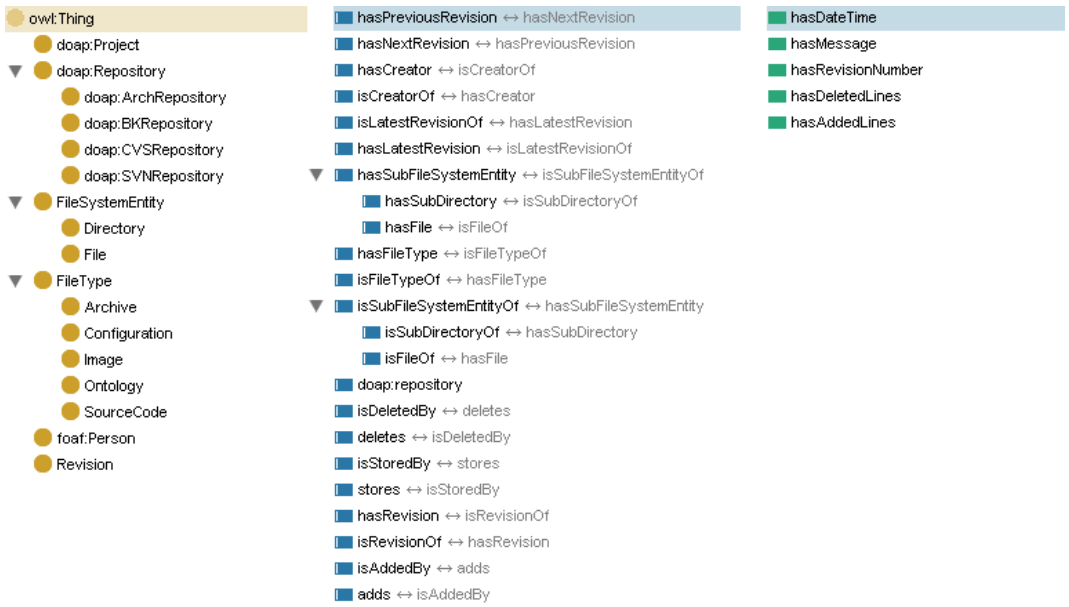

Figure 4.2: Class hierarchy and property classification of the Version ontology; classes (on the left), object properties (in the middle) and data type properties (on the right)

A project and its corresponding repository in a revision control system are modeled with the `doap:Project` class and the `doap:Repository` class as can be seen in Figure 4.3. This illustration also depicts the subclasses of `doap:Repository` (`doap:CVSRepository`, `doap:SVNRepository`, `doap:BKRepository` and `doap:ArchRepository`) which allow a simple distinction between the most popular revision control systems. The `doap:repository` property specifies the `doap:Repository` for the `doap:Project` in question. We have additionally modeled various classes and properties to describe the content stored in the repository and the revision specific data that can be extracted from these systems.

Classes

To facilitate the comprehension of our part of the Version ontology, Figure 4.4 depicts the contained classes and their essential relationships among each other. The following paragraphs describe these classes in more detail.

FileSystemEntity: A `FileSystemEntity` is an abstract class that is either a `Directory` or a `File`. Consequently, this class has no directly instantiated individuals as it is completely covered by its subclasses. This approach has been chosen since a `Directory` and a `File` have many similar properties regarding their interrelation with the `Revision` class. For example, each revision adds or deletes a number of directories and files.

The `FileSystemEntity` class has therefore the following two subclasses:

- **Directory:** This subclass models a directory in the repository of a project.
- **File:** This subclass models a file in the repository of a project.

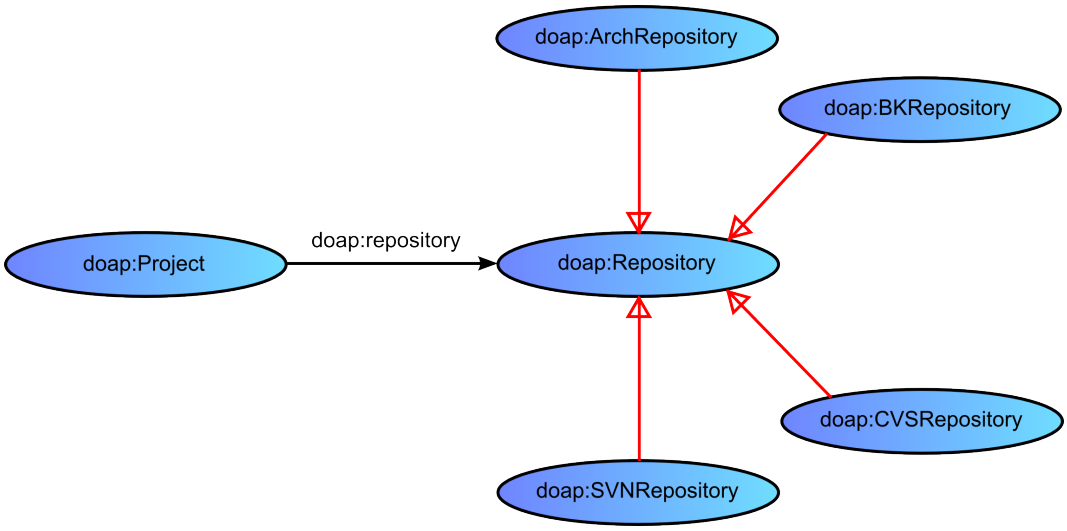

Figure 4.3: Classes and properties of the DOAP ontology used in the Version ontology

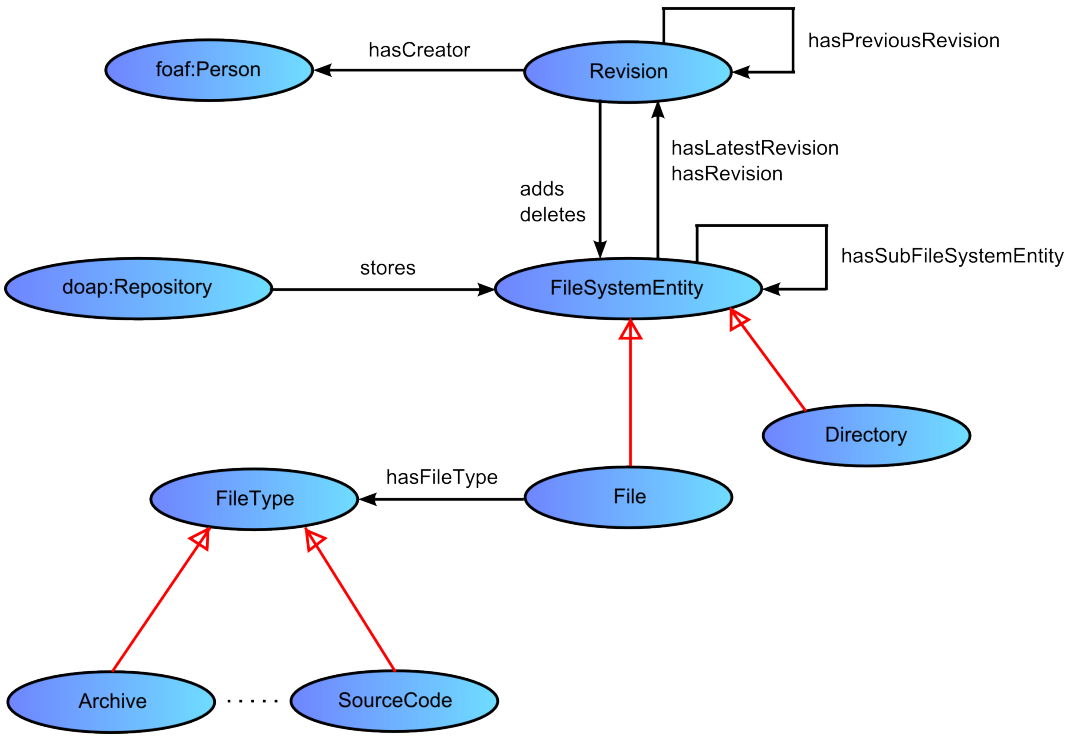

Figure 4.4: Classes and properties of the Version ontology

FileType: Each file in the repository has a specific file type which can be expressed using the `FileType` class and its corresponding subclasses. This information is explicitly included, because binary files are not handled very well by all of the inspected systems. Additionally, the file type information is used to refine the definition of a designated file and its content.

- **Archive:** This subclass models the archive file formats of a file.
- **Configuration:** This subclass models the configuration file formats of a file.
- **Image:** This subclass models the image file formats of a file.
- **Ontology:** This subclass models the ontology file formats of a file.
- **SourceCode:** This subclass models the source code file formats of a file.

We have already specified appropriate individuals for the `FileType` class and for each of its subclasses. These are outlined in the Individuals section of this ontology description.

Revision: The revision control systems keep track of a file's evolution with increasing revision numbers. This whole aspect is modeled with the `Revision` class in our Version ontology. As a revision is typically created by a specific user of the revision control system, we represent this fact with a property interconnecting the `Revision` class and the `foaf:Person` class.

Properties

For the sake of clarity, we did not include any data type properties nor all of the subproperties (and subclasses) in Figure 4.4. The subsequent descriptions elucidate the various properties in one direction and merely mention the inverse properties.

adds: Whenever a `Revision` adds a `FileSystemEntity`, the `adds` property is used to model this event. The inverse property is called `isAddedBy`.

deletes: Whenever a `Revision` deletes a `FileSystemEntity`, the `deletes` property is used to model this event. The inverse property is called `isDeletedBy`.

hasCreator: This property describes the relationship between a `Revision` and its creator which is modeled as a `foaf:Person`. The system specific username of a `foaf:Person` is represented in the corresponding ontology. The inverse property is called `isCreatorOf`.

hasFileType: A `File` description is refined with the indication of the appropriate `FileType`. The `hasFileType` property represents this relation between a `File` and a particular `FileType`. The inverse property is called `isFileTypeOf`.

hasLatestRevision: This property connects a `FileSystemEntity` to its latest possible `Revision` in the repository. The inverse property is called `isLatestRevisionOf`.

hasPreviousRevision: The `hasPreviousRevision` property denotes the relation between a `Revision` and its previous `Revision`. The inverse property is called `hasNextRevision`.

hasRevision: The `hasRevision` property is used to model the relationship between a `FileSystemEntity` and one of its `Revision`. The inverse property is called `isRevisionOf`.

hasSubFileSystemEntity: The `hasSubFileSystemEntity` property is an abstract property that represents the relationship between a `FileSystemEntity` and one of its included `FileSystemEntity`s. The two subproperties `hasSubDirectory` and `hasFile` are the ones actually used to indicate the structure of the directories and files in the repository of a software project. The inverse property is called `isSubFileSystemEntityOf`.

- **hasSubDirectory:** This subproperty models the relation between a `Directory` and one of its directly included subdirectories (also modeled as a `Directory`). The inverse property is called `isSubDirectoryOf`.
- **hasFile:** This subproperty models the relation between a `Directory` and one of the `File`s it includes. The inverse property is called `isFileOf`.

stores: This property indicates the fact that a `Repository` stores a `FileSystemEntity` of a certain project. The inverse property is called `isStoredBy`.

hasAddedLines: The log information stored by the investigated revision control systems and the diff command output in these revision control systems inspired the specification of the `hasAddedLines` property. It is deliberately not restricted to a specific data type, since it can represent either the absolute number of added lines or a range of new lines that are included in this dedicated `Revision`.

hasDateTime: This property denotes the date and the time when a `Revision` has been created.

hasDeletedLines: The log information stored by the investigated revision control systems and the diff command output in these revision control systems inspired the specification of the `hasDeletedLines` property. It is deliberately not restricted to a specific data type, since it can represent either the absolute number of deleted lines or a range of obsolete lines that are excluded in this dedicated `Revision`.

hasMessage: When a new `Revision` is committed the respective user can attach a comment or a message describing the changes he made. The `hasMessage` property is used to represent this fact with a string.

hasRevisionNumber: The `hasRevisionNumber` property specifies the revision number of a `Revision` as a string. This approach has been chosen because a revision number is not necessarily indicated as an integer in some common revision control systems.

Individuals

For the commonly known and widespread file types we have already predefined several individuals in our ontology that are listed in Table 4.2. These individuals are not intended to be exhaustive and can be broadened with respect to own needs.

4.3.2 CVS Ontology

The CVS ontology imports the formerly described Version ontology and can therefore reuse all the definitions specified in the previous section. Moreover, Figure 4.5 shows the additional classes and properties that are included in the CVS ontology.

The subsequent sections outline these classes and properties in more detail.

Table 4.2: Predefined individuals of the FileType class and each of its subclasses

FileType	Archive	Configuration	Image	Ontology	SourceCode
CSS	BZIP2	Ant	BMP	OWL	C
HTML	EAR	Config	GIF		COBOL
PDF	GZIP	ConfigureScript	JPEG		CPlusPlus
Text	JAR	Make	PNG		CSharp
XML	RAR	Maven	TIFF		ECMAScript
	TAR	Properties			Java
	WAR				JavaScript
	ZIP				Perl
					PHP
					Python

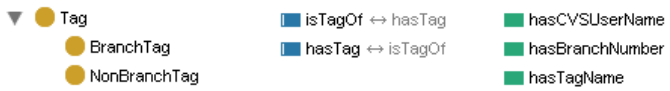

Figure 4.5: Class hierarchy and property classification of the CVS ontology; classes (on the left), object properties (in the middle) and data type properties (on the right)

Classes

We start our presentation of the CVS ontology with an overview of the classes and their relationships. This is illustrated in Figure 4.6.

Tag: The CVS revision control system stores additional data about tags, which represent a snapshot of a project in the selected revision. The `Tag` class is completely covered up by its subclasses `BranchTag` and `NonBranchTag`. Consequently, this class is not intended to have any instances on its own.

- **BranchTag:** This subclass models the tags that additionally stand for the creation of a new branch. Hence, a branch tag is always located at the root revision of a novel branch.
- **NonBranchTag:** This subclass models the tags that stand for an ordinary snapshot taken at a specific revision.

Properties

The following paragraphs describe on the one hand the additional possibilities to relate the aforementioned classes to other classes of the Version ontology and on the other hand how their definitions are refined with adequate information.

hasTag: The `hasTag` object property relates a `version:Revision` with a `Tag` that has been created along with this `version:Revision`. The inverse property is called `isTagOf`.

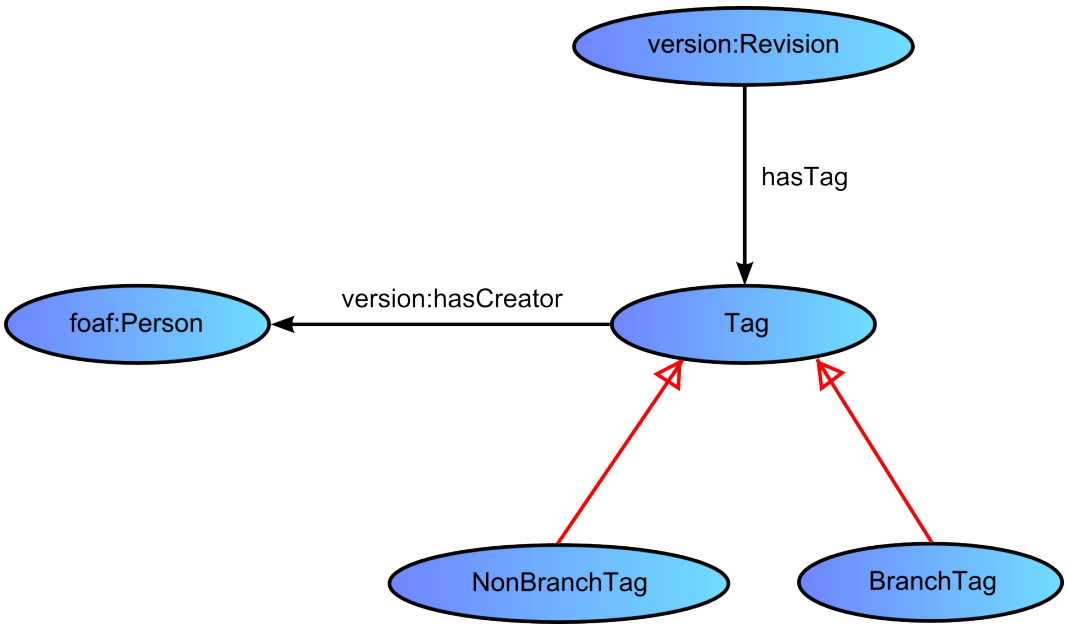

Figure 4.6: Classes and properties of the CVS ontology

hasBranchNumber: Each `BranchTag` sets up a unique branch number that is typically used to access the branch in a desired revision. Since a branch number is composed of a series of numbers separated by dots, it is restricted to string values and not to integer values.

hasCVSUserName: The `hasCVSUserName` property assesses a CVS user name to a `foaf:Person` as a new string.

hasTagName: This data type property defines the unique name of a `Tag` as a string. This name is often used to retrieve a specific tag (snapshot) in the CVS system.

4.3.3 SVN Ontology

Like the CVS ontology, the SVN ontology imports the Version ontology and reuses these classes and properties to model additional SVN system specific information. For this purpose, we have defined a couple of new properties which are displayed in Figure 4.7.

Properties

To better grasp the use of the aforementioned properties, Figure 4.8 shows which classes are related through their appropriate usage.

copies: The SVN revision control system provides a very powerful and multifunctional copying mechanism that is used for various fundamental purposes. For example, whenever a user intends to create a new tag or a new branch in this system, a copy of the desired file system entities is relocated to another location in the repository. It is important to be aware of the fact that the SVN system does not distinguish whether a new tag or a new branch is created. The SVN

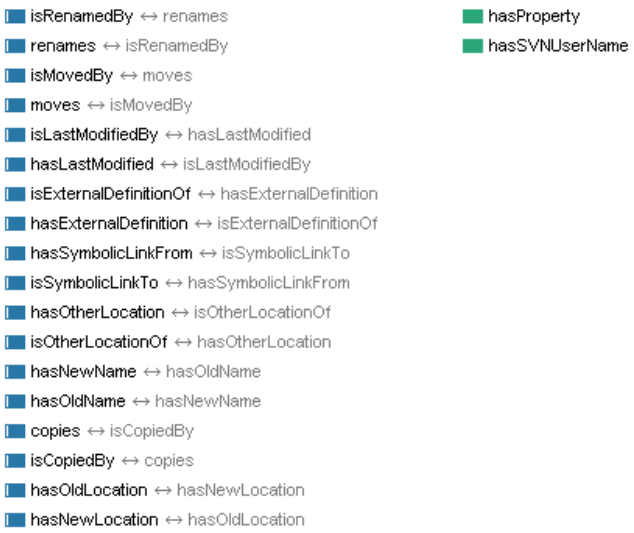

Figure 4.7: Property classification of the SVN ontology; object properties (on the left) and data type properties (on the right)

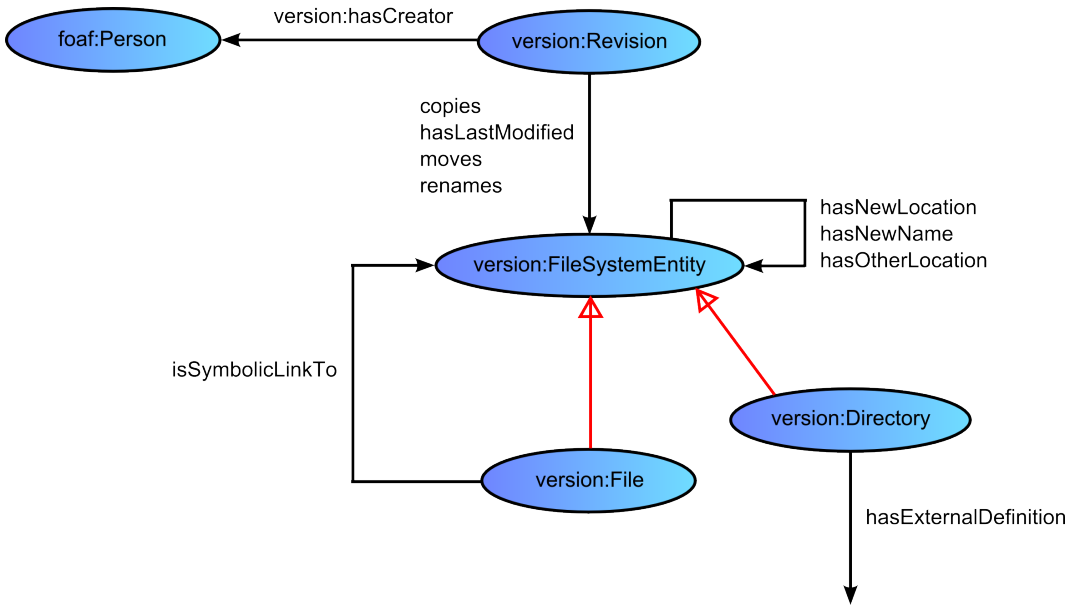

Figure 4.8: Properties of the SVN ontology

system merely records that a copy of the selected file system entities has been placed at a different location. The intended meaning of this copy, whether it is a new tag or a new branch, is entirely left to the user's own interpretation.

As a result, the `copies` property relates a `version:Revision` with the `version:FileSystemEntity` that is copied to a different storage location. To appropriately indicate this other location, the `hasOtherLocation` property is intended to be used in combination with the `copies` property. The inverse property is called `isCopiedBy`.

hasExternalDefinition: If the content of a directory is defined in another location within the same repository or at an external location, this information is registered with the `hasExternalDefinition` property. The range of this property has deliberately not been restricted to a `version:FileSystemEntity`, because the external definition does not necessarily have to be on another repository. Nevertheless, in most cases it is appropriate to expect that this property connects a `version:Directory` with a `version:FileSystemEntity` located in another repository. The inverse property is called `isExternalDefinitionOf`.

hasLastModified: The SVN revision control system keeps global revision numbers for entire tree structures of the repository's file system structure. Hence, the additional information when a specific `version:File` or `version:Directory` has lastly been modified needs to be modeled with another property. The `hasLastModified` relationship states for a given `version:Revision` which `version:FileSystemEntity` has lastly been modified in this `version:Revision`. The inverse property is called `isLastModifiedBy`.

hasNewLocation: This property is used to record the fact that a `version:FileSystemEntity` has been moved to a new location. Thereby, it connects the old location of the `version:FileSystemEntity` with the new location of this `version:FileSystemEntity`. It is only intended to be used in conjunction with the `moves` property outlined in this section. The inverse property is called `hasOldLocation`.

hasNewName: The `hasNewName` property is only used in combination with the `renames` property outlined in this section. It relates the `version:FileSystemEntity` having the obsolete name with the `version:FileSystemEntity` having the up-to-date name. The inverse property is called `hasOldName`.

hasOtherLocation: This property is used to relate the original location of a `version:FileSystemEntity` with the location of a copy of this `version:FileSystemEntity`. It is only intended to be used in conjunction with the `copies` property outlined in this section. The inverse property is called `isOtherLocationOf`.

isSymbolicLinkTo: If a file merely represents a symbolic link to another file system entity in the repository, the `isSymbolicLinkTo` property can be used to capture this relationship between a `version:File` and the corresponding `version:FileSystemEntity`. The inverse property is called `hasSymbolicLinkFrom`.

moves: Besides the possibilities of adding and deleting a file system entity in a specific revision, the SVN implementation provides a mechanism to move a file system entity to a new location and to keep track of this repositioning event. The `moves` property models this relationship between a `version:Revision` and a `version:FileSystemEntity` that has been

moved in this version:Revision. To include the information of the new location in our ontology, this property is planned to be used along with the hasNewLocation property defined for the version:FileSystemEntity. The inverse property is called isMovedBy.

renames: Renaming a file system entity is not supported by many revision control systems in a satisfying manner. The most often implemented workaround in these systems is that the file system entity with the obsolete name is deleted and a new file system entity with the up-to-date name is added at the same location. However, the SVN system provides an appropriate mechanism to rename a file system entity and to keep track of this action. First, the renames property connects the version:Revision with the version:FileSystemEntity that is renamed by this version:Revision. The subsequent use of the hasNewName property retains a record of the renaming action associated with the renames property. The inverse property is called isRenamedBy.

hasProperty: Aside from versioning the directories and files of a software project, the SVN revision control system provides the possibility to add, modify or remove versioned meta-data on each version:FileSystemEntity and on each version:Revision. This meta-data is referred to as properties in the daily use of SVN and is specified as a string value in our ontology.

hasSVNUserName: The hasSVNUserName property assesses a SVN user name to a foaf:Person as a new string.

4.4 Bug Ontology Pyramid

The bug ontology pyramid models the concepts and their appropriate relationships among each other for issue tracking systems of the domain of discourse. The investigated systems displayed many similar concepts and dependencies which are defined in the Bug ontology. System specific aspects are included in the Bugzilla ontology and in the Trac ontology.

4.4.1 Bug Ontology

The Bug ontology imports the previously outlined Version ontology and is therefore capable of reusing all classes, properties and individuals included in the Version ontology definition. Figure 4.9 provides an initial overview of the classes and properties that are specified in the Bug ontology. Thus it appears that the Bug ontology is a flat and broadly based ontology with no hierarchical classifications. This can be traced back to the fact that the investigated issue tracking systems describe the reported issues with various characteristics which cannot be classified into an appropriate hierarchy because of their fundamental difference to one another.

The Bug ontology is interconnected with the Version ontology through several properties of the Bug ontology. These properties are illustrated in Figure 4.10 whereas their corresponding definitions are specified in the Properties section of this ontology description.

Classes

With the use of an issue tracking system, developers are able to report in a central location issues encountered with a software project in order to keep track of these issues with ease. The Issue class in our Bug ontology represents the reported issues in an issue tracking system and is the

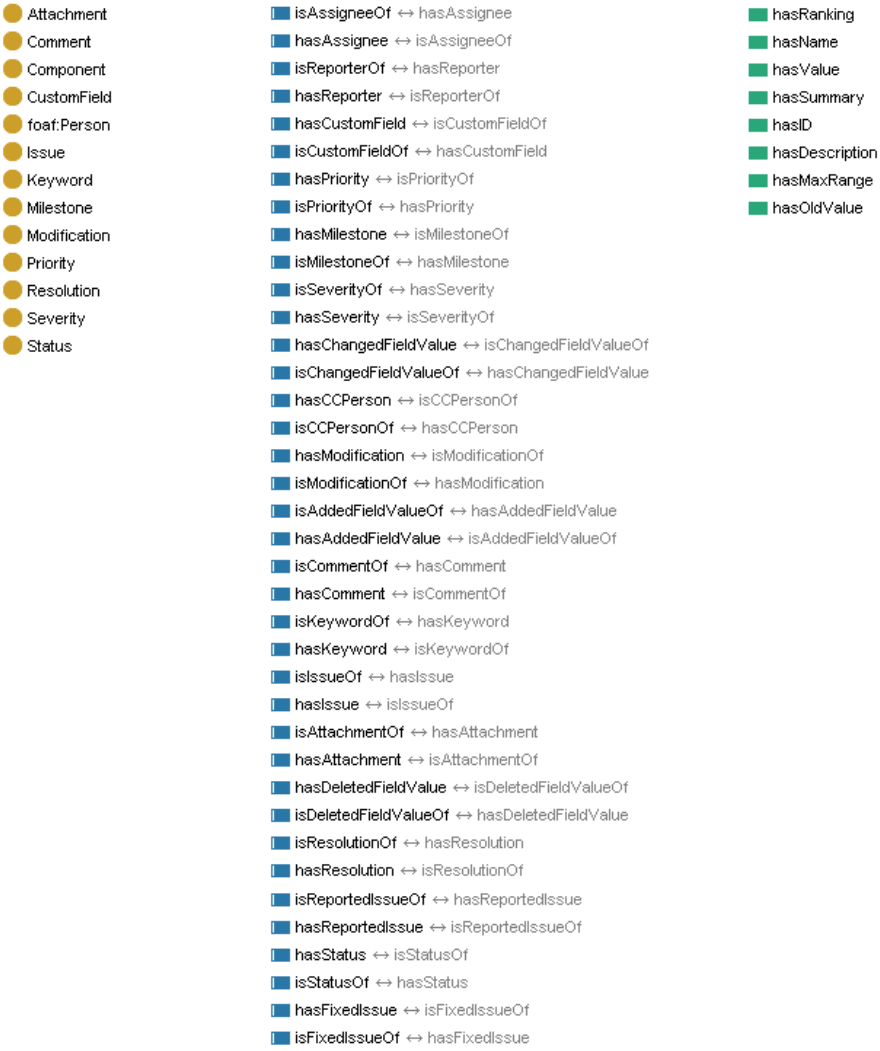

Figure 4.9: Class hierarchy and property classification of the Bug ontology; classes (on the left), object properties (in the middle) and data type properties (on the right)

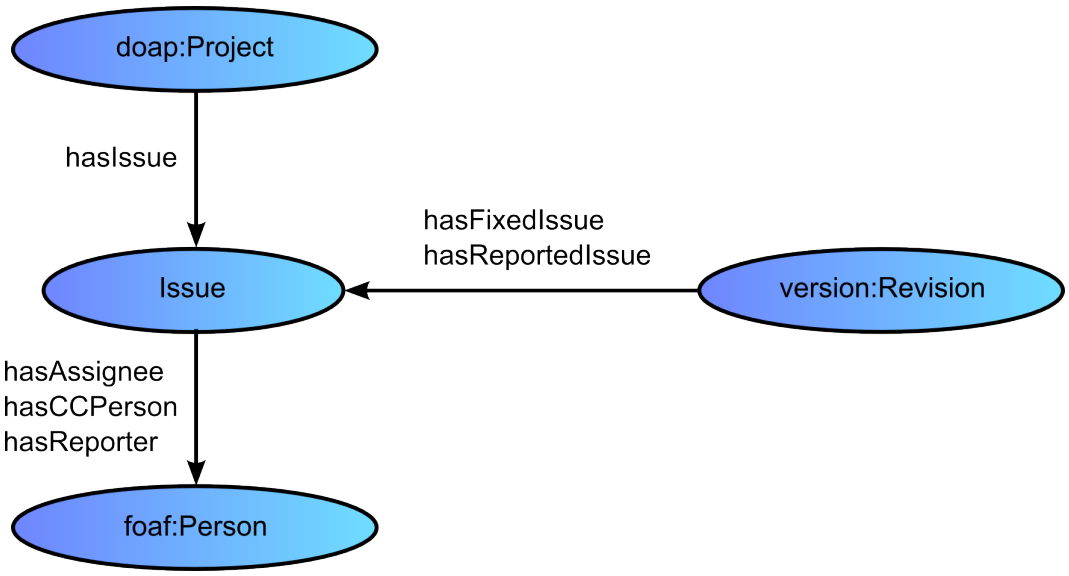

Figure 4.10: Connection between the Bug ontology and the Version ontology

most significant class in this ontology because it is detailed with numerous relationships to other classes. This fact is depicted in Figure 4.11.

The following paragraphs describe these classes in more detail.

Attachment: The `Attachment` class models the attachments that have been added to the description of an issue in the investigated issue tracking systems.

Comment: This class represents the comments that have been appended to the description of an issue or of a modification in order to explain the different field values chosen in either of them.

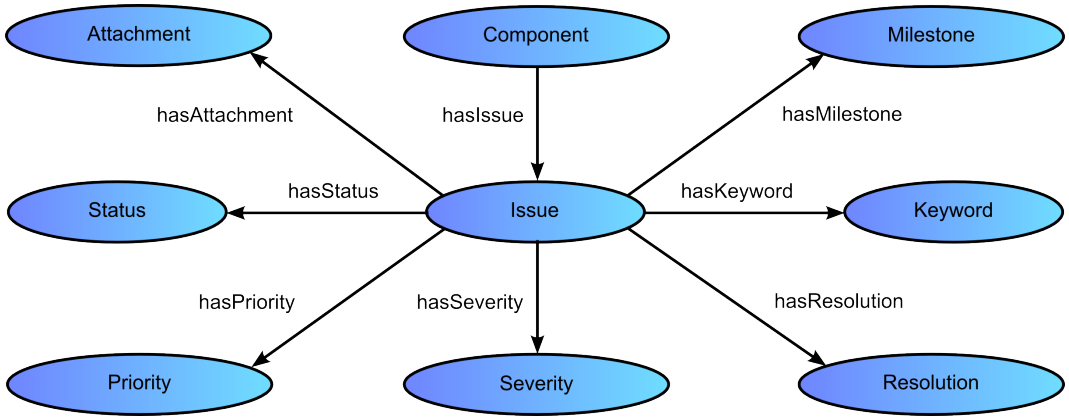

Figure 4.11: Classes and properties of the Bug ontology used to describe reported issues

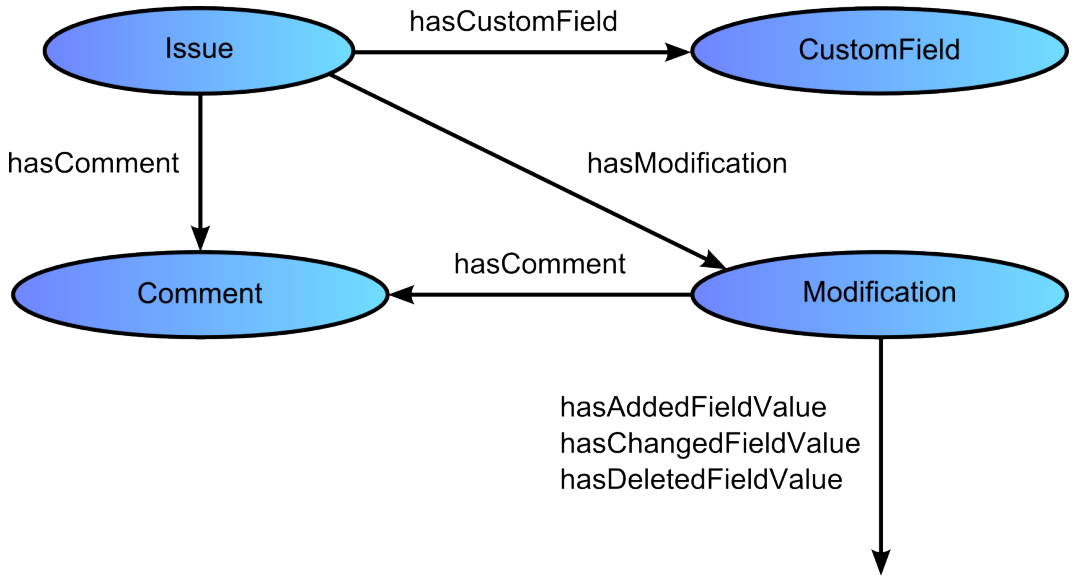

Figure 4.12: Classes and properties of the Bug ontology used to describe modifications and additional characteristics of reported issues

Component: A software project is typically divided up into several components, which in turn have their own developers and specialists. These components are represented with the `Component` class in our Bug ontology.

CustomField: In addition to the preconfigured characteristics of an issue, the investigated issue tracking systems also allow users to define additional custom-designed fields to further describe an issue with the required information. The `CustomField` class models these additional fields in our Bug ontology.

Issue: As previously mentioned, issues reported in an issue tracking system are at the core of our Bug ontology. These issues are represented through appropriate instances of the `Issue` class. Most of the other classes in this ontology are used to detail the description of the `Issue` class.

Keyword: The categorization of reported issues can be refined with the assignment of appropriate keywords. Through their proper usage, keywords can also assist in retrieving the desired issues. These keywords are incorporated in our Bug ontology with the `Keyword` class.

Milestone: In the context of issue tracking systems, a milestone often refers to a future release of a software project. An issue that is assigned to a specific milestone is intended to be fixed in the corresponding release. The `Milestone` class models this aspect in our Bug ontology.

Modification: The investigated issue tracking systems keep hold of the modifications that have been brought to the description of an issue. These modifications contain information about the different field values that have been added, changed or deleted from the description of an issue. Figure 4.12 shows among other aspects how these modifications are included in our Bug ontology.

Priority: The reporter of an issue decides on its priority on the basis of his subjective priority perception. The various priority indications that are available in issue tracking systems are modeled with the `Priority` class in our Bug ontology.

Resolution: The `Resolution` class specifies how an issue has been resolved. The investigated issue tracking systems provide identical resolution possibilities, which are included in our Bug ontology as corresponding instances of the `Resolution` class. Table 4.3 outlines these individuals later on in this ontology description.

Severity: The reporting user of an issue identifies its severity based on the impact of this issue on the software application. The `Severity` class represents the numerous severity indications that are available in issue tracking systems.

Status: The `Status` class models the different states in the workflow of an issue. The preconfigured workflow states in the investigated issue tracking systems show several similarities, which are incorporated into our Bug ontology with appropriate individuals of the `Status` class. These are listed in Table 4.3 later on in this ontology description.

Properties

To better understand the relationships between the previously defined classes, the subsequent property descriptions specify which classes are connected through their intended usage and how these class definitions are refined with the specification of system independent information.

hasAddedFieldValue: For any given `Modification`, this object property indicates which field value has been added to the description of an issue. The inverse property is called `isAddedFieldValueOf`.

hasAssignee: The `hasAssignee` relationship states for any given `Issue` which `foaf:Person` has been assigned with the task to fix this `Issue`. The inverse property is called `isAssigneeOf`.

hasAttachment: The `hasAttachment` property relates an `Issue` to an `Attachment` that has been added to the description of this `Issue`. The inverse property is called `isAttachmentOf`.

hasCCPerson: The `hasCCPerson` object property models the relationship between an `Issue` and a `foaf:Person` that is kept informed about the changes of this `Issue`. Additionally, this property states for a given `bugzilla:Flag` which `foaf:Person` is notified on the request of such a `bugzilla:Flag`. As a result, the domain of the `hasCCPerson` property is not restricted to a specific class definition of the Bug ontology. The inverse property is called `isCCPersonOf`.

hasChangedFieldValue: The `hasChangedFieldValue` property denotes for a given `Modification` which field value has been changed in the description of an issue. The inverse property is called `isChangedFieldValueOf`.

hasComment: This property is used to model the relationship between either an `Issue` and a `Comment` or a `Modification` and a `Comment` that has been appended to either one of them. The inverse property is called `isCommentOf`.

hasCustomField: The `hasCustomField` property represents the relation between an `Issue` and a `CustomField` that has been specified to further describe this `Issue`. The inverse property is called `isCustomFieldOf`.

hasDeletedFieldValue: This property indicates for a given `Modification` which field value has been deleted from the description of an issue. The inverse property is called `isDeletedFieldValueOf`.

hasFixedIssue: The `hasFixedIssue` property describes the relationship between a `version:Revision` and an `Issue` that has been fixed in this `version:Revision`. The inverse property is called `isFixedIssueOf`.

hasIssue: The `hasIssue` property models the relationship between either a `Component` and an `Issue` which belongs to this specific `Component` or a `doap:Project` and an `Issue` which belongs to this specific `doap:Project`. The inverse property is called `isIssueOf`.

hasKeyword: This property represents the connection between an `Issue` and a `Keyword` that has been allocated to this `Issue`. The inverse property is called `isKeywordOf`.

hasMilestone: This property describes the relationship between an `Issue` and a `Milestone`, which indicates when this `Issue` should be fixed. The inverse property is called `isMilestoneOf`.

hasModification: This object property relates an `Issue` to a `Modification` that has added, changed or deleted a field value of the description of this `Issue`. The inverse property is called `isModificationOf`.

hasPriority: The `hasPriority` property describes the relationship between an `Issue` and a `Priority` which in turn highly depends on the subjective priority perception of the reporting user. The inverse property is called `isPriorityOf`.

hasReportedIssue: The `hasReportedIssue` property represents the relationship between a `version:Revision` and an `Issue` that has been reported in this `version:Revision`. The inverse property is called `isReportedIssueOf`.

hasReporter: The `hasReporter` relationship states for any given `Issue` which `foaf:Person` has reported it in the issue tracking system. The inverse property is called `isReporterOf`.

hasResolution: This object property relates an `Issue` with a `Resolution` that specifies how this `Issue` has been resolved. The inverse property is called `isResolutionOf`.

hasSeverity: The `hasSeverity` property is used to model the relationship between an `Issue` and a `Severity` which indicates how severe this reported `Issue` is with respect to its impact on the software application. The inverse property is called `isSeverityOf`.

hasStatus: The `hasStatus` property describes the relation between an `Issue` and a `Status`. This `Status` denotes a corresponding state in the workflow of an issue and consequently indicates where this `Issue` is located. The inverse property is called `isStatusOf`.

hasDescription: The `hasDescription` property records the description of a concept in an issue tracking system as a new string. Since this property is also used in the system specific Bugzilla ontology, it is not restricted to use with the appropriate classes of the Bug ontology.

hasID: This data type property specifies the unique identification number of an `Issue` or an `Attachment` as an integer.

hasMaxRange: This data type property is used to refine the description of the `Priority` class and the `Severity` class with an additional integer value and states the number of individuals that are encountered in each of them. With the indication of a corresponding ranking among these individuals within this range, their importance can be quantified and compared to other issue tracking systems that use a different number of individuals within these classes to represent the priority or the severity of an issue.

hasName: The `hasName` property specifies the name of a particular individual that belongs to one of our classes in the Bug ontology as a new string. Since this property is also used in the system specific Bugzilla ontology, it is not restricted to use with the appropriate classes of the Bug ontology.

hasOldValue: The `hasOldValue` property retains a string of the old field value of an issue characteristic that has been changed or deleted in a modification. The domain of this property has deliberately not been restricted to a specific group of issue characteristics, because the system specific ontologies of the Bugzilla implementation and the Trac implementation define additional issue characteristics that could not be included within this group.

hasRanking: This property is used in conjunction with the `hasMaxRange` property to refine the definition of the `Priority` instances and the `Severity` instances. Thereby, it specifies the appropriate ranking of these individuals within a possible range which in turn is set up by the `hasMaxRange` property. With these two integer values the importance of an individual can be quantified and therefore compared to other issue tracking systems that use a different number of individuals within these classes to model the priority or the severity of an issue.

hasSummary: The `hasSummary` property assesses a summary to an `Issue` as a new string.

hasValue: This data type property denotes the field value of a user-defined `CustomField` as a new string.

Individuals

The investigated issue tracking systems show numerous similarities in the description of an issue. However, the predefined field values of certain issue characteristics vary substantially in the Bugzilla implementation and the Trac implementation. Nevertheless, we have specified several individuals for the `Resolution` class and the `Status` class of our Bug ontology that are identically present in the Bugzilla system and the Trac system. These individuals can be seen in Table 4.3.

Table 4.3: Predefined individuals of the Resolution class and the Status class in the issue tracking systems

Resolution	Status
Duplicate	Assigned
Fixed	Closed
Invalid	New
Wontfix	Reopened
Worksforme	

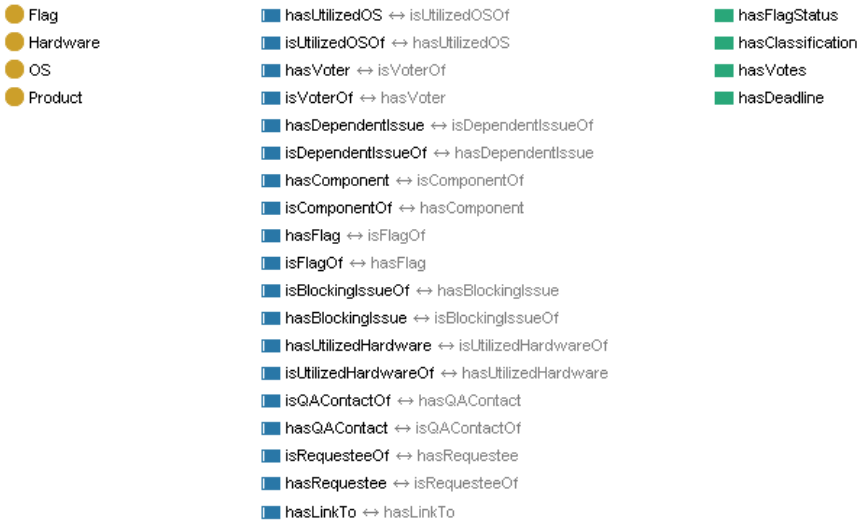

Figure 4.13: Class hierarchy and property classification of the Bugzilla ontology; classes (on the left), object properties (in the middle) and data type properties (on the right)

4.4.2 Bugzilla Ontology

Since the general concepts of the issue tracking systems investigated here are represented in the Bug ontology, the Bugzilla ontology imports these definitions along with the ones that are imported in the Bug ontology. Therefore, the Bugzilla ontology is capable of reusing all classes, properties and individuals that are modeled in the Bug ontology and the Version ontology. For the proper representation of the system specific aspects of the Bugzilla implementation, we have defined appropriate classes, properties and individuals and included them in the Bugzilla ontology. Figure 4.13 provides an initial overview of the additional definitions incorporated in this ontology.

Classes

In comparison with the Trac system, the preconfigured Bugzilla system enables additional information to be stated about a reported issue and its described details. The classes that have been added for this purpose and their appropriate relationships among each other are depicted in Figure 4.14.

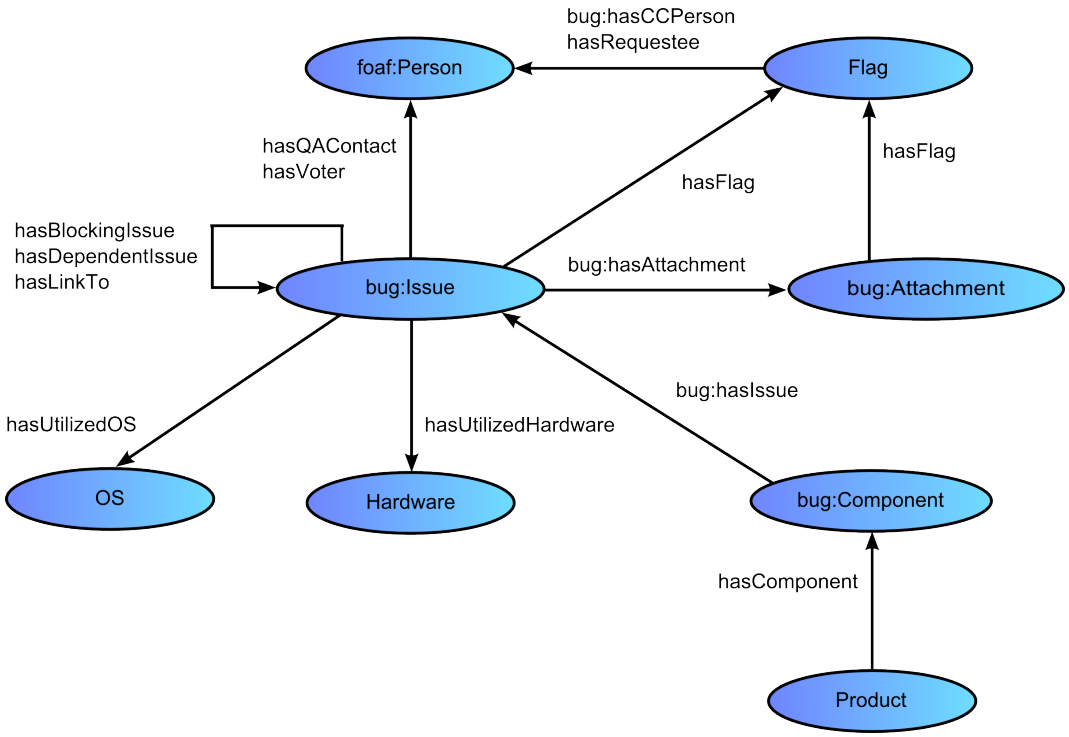

Figure 4.14: Classes and properties of the Bugzilla ontology

Flag: The Bugzilla system offers the possibility to specify user-defined flags to mark the reported issues or their attachments. Each flag is either restricted to be used with issues or attachments. The corresponding representation of these flags is built in with the `Flag` class in our Bugzilla ontology specification. The `Flag` definition is also refined with two properties of the Bug ontology. The `bug:hasName` property captures the appropriate name of a `Flag` as a string whereas the `bug:hasCCPerson` property states which `foaf:Person` is related to a specific `Flag` with respect to a list of users that are notified if such a `Flag` is added, changed or deleted.

Hardware: This class models the hardware system that has been in use when the issue occurred in the first place.

OS: This class models the operating system that has been in use when the issue occurred in the first place.

Product: In the Bugzilla implementation issues are divided up by products and components. Thereby, a typical product contains at least one or multiple components. The `Product` class represents these products in the Bugzilla ontology.

Properties

The subsequent property definitions specify the additional relationships between the aforementioned classes and other classes of the Bug ontology. Furthermore, these properties refine the descriptions of certain classes with adequate information.

hasBlockingIssue: The Bugzilla implementation allows a user to explicitly state the dependencies of a specific issue to other reported issues in the system. Therefore, this property models the relationship between two different instances of the `bug:Issue` class and indicates that an issue is blocked by another issue. The inverse property is called `isBlockingIssueOf`.

hasComponent: The `hasComponent` property relates a `Product` with a `bug:Component` that is contained in the specific `Product`. The inverse property is called `isComponentOf`.

hasDependentIssue: The Bugzilla implementation allows a user to explicitly state the dependencies of a specific issue to other reported issues in the system. Therefore, this property models the relationship between two distinct instances of the `bug:Issue` class and indicates that an issue depends on another issue. The inverse property is called `isDependentIssueOf`.

hasFlag: The `hasFlag` property is used to model the relationship between either a `bug:Issue` and a `Flag` or a `bug:Attachment` and a `Flag`. The inverse property is called `isFlagOf`.

hasLinkTo: This property models the relationship between two distinct `bug:Issue` instances that have an explicit link to one another in the Bugzilla system. By default, this information is recorded in the *See Also* field of a Bugzilla implementation. The inverse property is also called `hasLinkTo` because it is a symmetric property.

hasQAContact: The `hasQAContact` object property relates a `bug:Issue` with a `foaf:Person` that is responsible for the quality assurance of this `bug:Issue`. The inverse property is called `isQAContactOf`.

hasRequestee: This property indicates the relation between a `Flag` and a `foaf:Person` which is requested to either grant or deny this `Flag`. The inverse property is called `isRequesteeOf`.

hasUtilizedHardware: The `hasUtilizedHardware` property models the relationship between a `bug:Issue` and a `Hardware` which has been in use when the issue occurred in the first place. The inverse property is called `isUtilizedHardwareOf`.

hasUtilizedOS: The `hasUtilizedOS` property models the relationship between a `bug:Issue` and a `OS` which has been in use when the issue occurred in the first place. The inverse property is called `isUtilizedOSOf`.

hasVoter: The Bugzilla system provides a useful voting facility that enables its users to vote for issues which they think are important and that they would like to have resolved in the near future. The software developers can make use of this information to incorporate the needs of the users into the project planning. This property represents the relationship between a `bug:Issue` instance and a `foaf:Person` instance that has voted for this `bug:Issue` instance. The inverse property is called `isVoterOf`.

hasClassification: In the Bugzilla implementation the components are grouped according to the products that they belong to. On the other hand, the Bugzilla system also provides a classification mechanism for products in order to construct meaningful time series and improve the retrieval of the desired issues. The `hasClassification` property is used to represent this aspect of a `Product` with a string.

hasDeadline: The `hasDeadline` property defines the deadline of a `bug:Issue` as a date.

hasFlagStatus: This data type property describes the status of a `Flag` as a new string. The allowed values are restricted to the strings "+" (which states that the flag has been granted), "-" (which states that the flag has been denied) and "?" (which states that the flag has been requested but has not been granted or denied yet).

hasVotes: The `hasVotes` property specifies the number of votes of a `bug:Issue` as an integer. This information is explicitly modeled because a user can use more than one vote for a specific issue. Simply counting the number of `hasVoter` properties of a `bug:Issue` would not answer this purpose.

Individuals

Some characteristics of an issue have different predefined description choices in the Bugzilla implementation compared to their counterparts in the Trac implementation. Thus, these different choices that are offered by the Bugzilla system are incorporated in our Bugzilla ontology as additional individuals, which can be perceived in Table 4.4.

Table 4.4: Predefined individuals of the bug:Priority class, the bug:Resolution class, the bug:-Severity class and the bug:Status class in the Bugzilla system

bug:Priority	bug:Resolution	bug:Severity	bug:Status
P1	Later	Blocker	Resolved
P2	Moved	Critical	Unconfirmed
P3	Remind	Major	Verified
P4		Normal	
P5		Minor	
		Trivial	
		Enhancement	

4.4.3 Trac Ontology

The Trac ontology imports the Bug ontology which in turn imports the Version ontology. Consequently, the Trac ontology is able to reuse the classes, the properties and the individuals included in either one of them. Nevertheless, we have specified some additional classes, properties and individuals to refine these definitions so that they suit our needs for the Trac system. Figure 4.15 provides a hierarchical overview of the newly added classes and properties.

Classes

Since the investigated issue tracking systems are basically configured with similar possibilities to describe an issue, only a few additional classes had to be included in the Trac ontology. The specialization of the `bug:Issue` class, which is depicted in Figure 4.16, represents the only class extensions factored in this ontology.

Figure 4.15: Class hierarchy and property classification of the Trac ontology; classes (on the left) and data type properties (on the right)

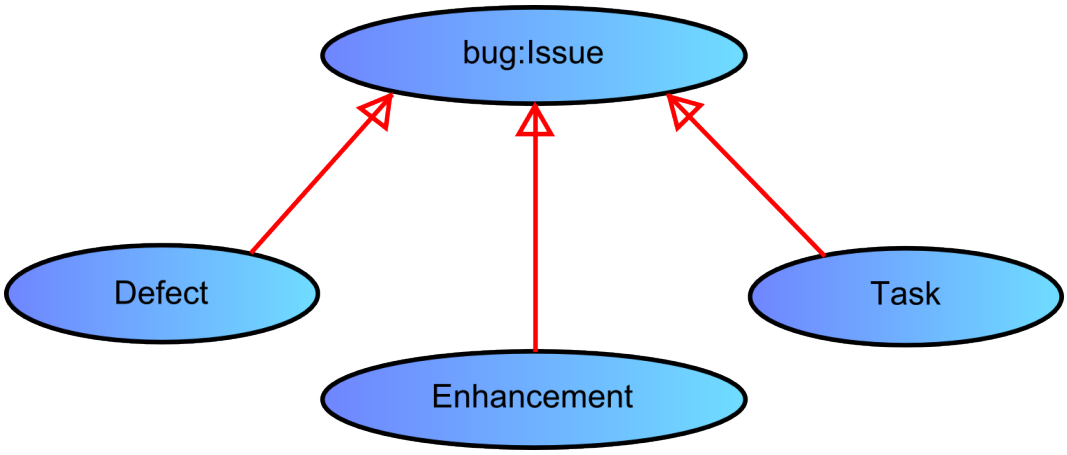

Figure 4.16: Classes of the Trac ontology

bug:Issue: The Trac system classifies the reported issues in three different categories depending on their impact on the software project. This preliminary classification is represented in our Trac ontology with the specialization of the `bug:Issue` class in the subsequent three subclasses.

- **Defect:** This subclass models the issues that are reported as an identified defect in the Trac system.
- **Enhancement:** This subclass models the issues that are reported as an enhancement request in the Trac system.
- **Task:** This subclass models the issues that are reported as an outstanding task in the Trac system.

Properties

The additional properties improve the description of a milestone in the Trac system environment with different date specifications.

hasCompletionDate: This data type property indicates the completion date of a `bug:Milestone`.

hasDueDate: This data type property indicates the due date of a `bug:Milestone`.

Individuals

In the Trac system the predefined choices to describe certain characteristics of an issue differ from the ones specified in the Bugzilla system. Hence, these different choices provided by the Trac system are incorporated in our ontology as supplementary individuals which can be seen in Table 4.5.

Table 4.5: Predefined individuals of the bug:Priority class, the bug:Severity class and the bug:Status class in the Trac system

bug:Priority	bug:Severity	bug:Status
Highest	Blocker	Accepted
High	Critical	
Normal	Major	
Low	Minor	
Lowest	Trivial	

4.5 Code Ontology Pyramid

The code ontology pyramid models the static source code information of our domain of discourse. Similarities in the investigated object-oriented programming languages are represented with appropriate classes and properties in the Code ontology. Language specific particularities are defined in the Java ontology respectively in the CSharp ontology.

4.5.1 Code Ontology

The Code ontology imports the previously described Version ontology and is therefore able to reuse all classes, properties and individuals included in the specification of the Version ontology.

The hierarchical taxonomies of the identified classes and properties of the Code ontology are displayed in Figure 4.17 and offer a preliminary overview of the classes and properties that are defined in this ontology.

The Code ontology is interconnected with the Version ontology through the `hasSourceCode` property of the Code ontology. This is illustrated in Figure 4.18. The corresponding definition of this property is outlined in the Properties section of this ontology description.

Classes

In order to extract the required static source code information, we have classified the fundamental types of an object-oriented programming language into a small-sized class hierarchy. Figure 4.19 depicts this class hierarchy along with the relationships to the superordinate classes.

However, the majority of the object-oriented source code information is represented with additional classes and properties which are illustrated in Figure 4.20. The following paragraphs describe these classes in more detail.

Constructor: The `Constructor` class represents the constructors of an object-oriented software project.

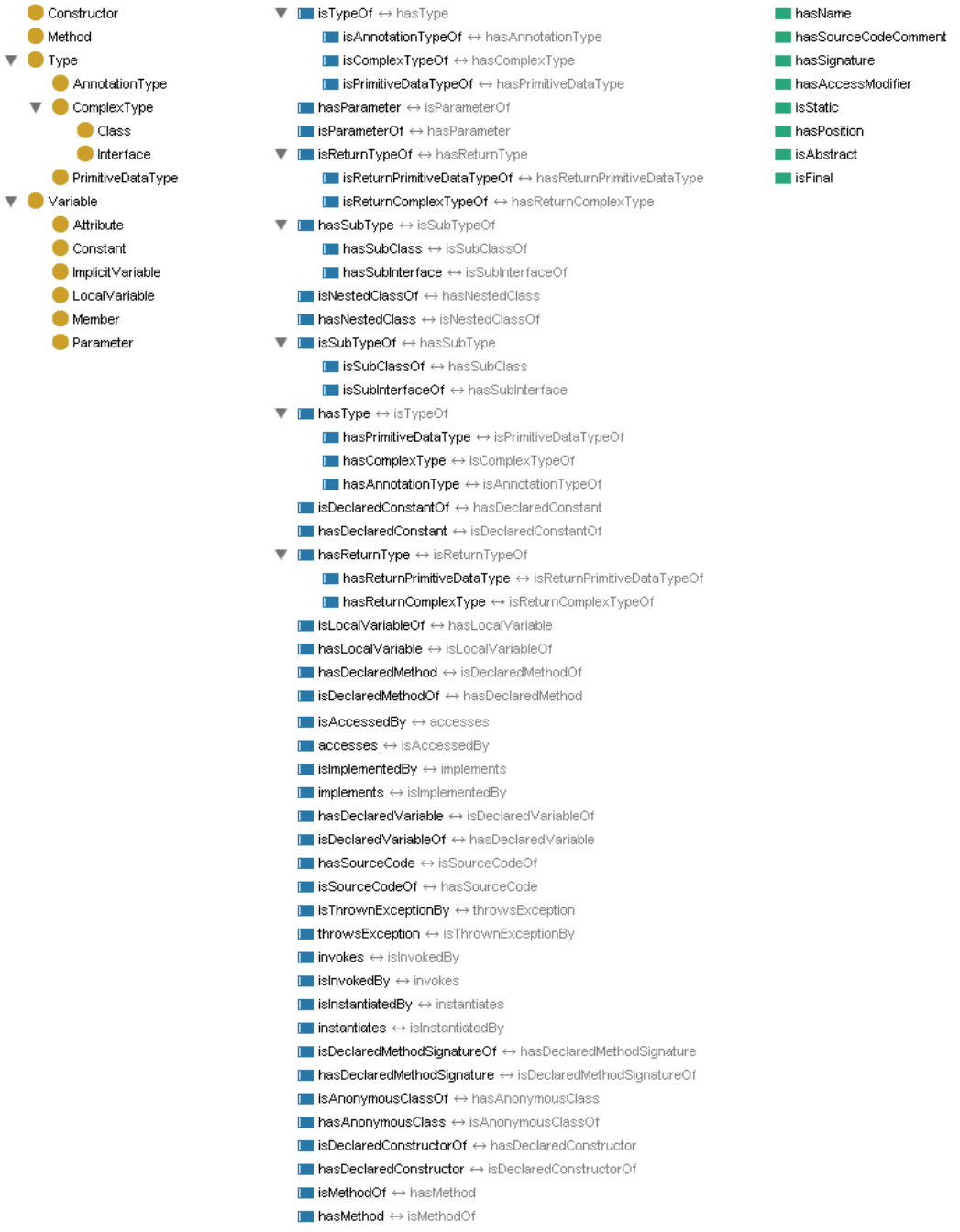

Figure 4.17: Class hierarchy and property classification of the Code ontology; classes (on the left), object properties (in the middle) and data type properties (on the right)

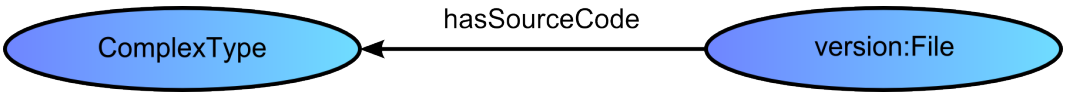

Figure 4.18: Connection between the Code ontology and the Version ontology

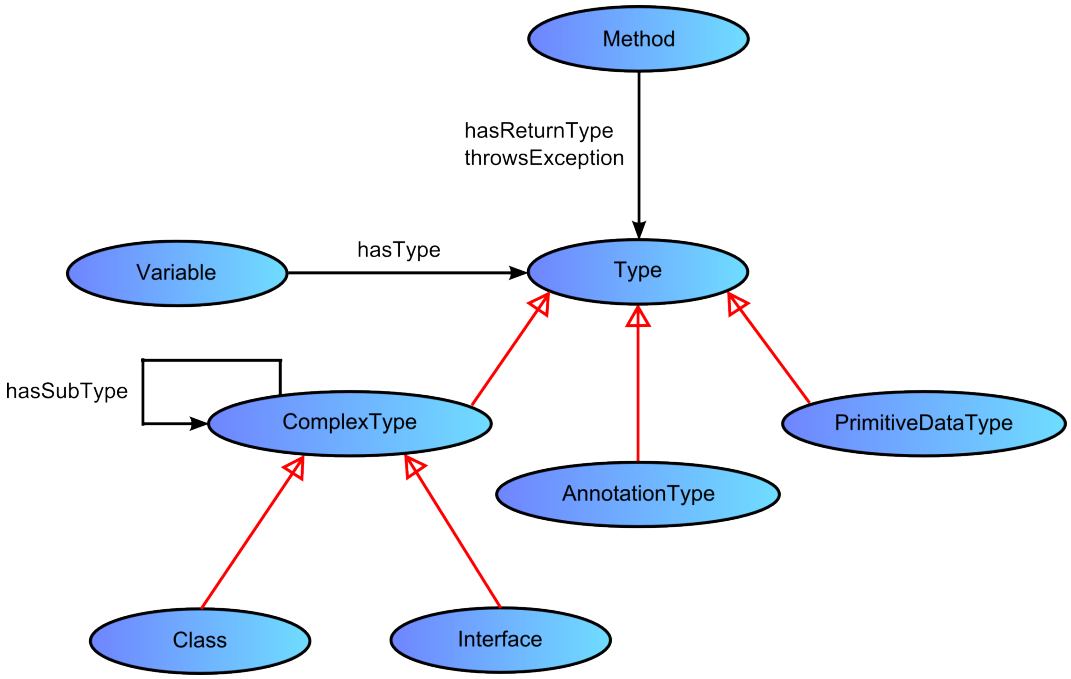

Figure 4.19: Classes and properties of the Code ontology used to describe fundamental types

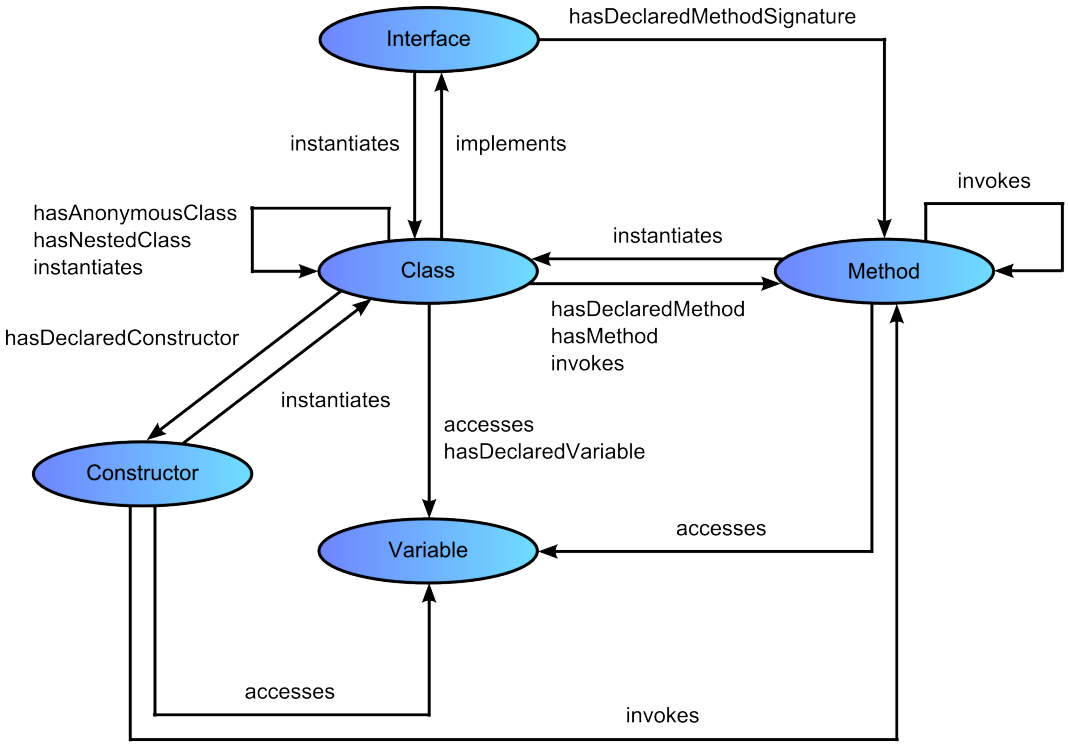

Figure 4.20: Classes and properties of the Code ontology used to describe static source code information

Method: The `Method` class represents the methods of an object-oriented software project.

Type: This class is inserted in our Code ontology in order to hierarchically classify the various types of an object-oriented programming language. And since there is no corresponding concept in the source code information of an object-oriented software project, this class is not intended to be instantiated directly.

- **AnnotationType:** This subclass models the annotation types of an object-oriented programming language along with the annotation types that have been defined by the developers themselves.
- **ComplexType:** This subclass serves as a container for the complex types of an object-oriented programming language. Moreover, this subclass has no directly instantiated individuals on its own since a corresponding concept does not exist in the source code of an object-oriented software project. Nevertheless, the subclasses of the `ComplexType` class can be instantiated as their counterparts are present in the source code of an object-oriented software project.
  - **Class:** This subclass of the `ComplexType` class models the classes of an object-oriented software project.
  - **Interface:** This subclass of the `ComplexType` class models the interfaces of an object-oriented software project.
- **PrimitiveDataType:** This subclass models the primitive data types of an object-oriented programming language. We have incorporated these language specific primitive data types as separate individuals in the Java ontology and in the CSharp ontology.

Variable: The `Variable` class models all variable types of an object-oriented software project as separate subclasses.

- **Attribute:** This subclass models the attributes of an object-oriented software project.
- **Constant:** This subclass models the constants of an object-oriented software project.
- **ImplicitVariable:** This subclass models the implicit variables of an object-oriented software project. The actual value of such an implicit variable is context-dependent and is therefore determined on the basis of the appropriate context information.
- **LocalVariable:** This subclass models the local variables of an object-oriented software project. Local variables are defined within a method or a constructor and they cease to exist with the completion of the method or the constructor in which they are defined.
- **Member:** This subclass models the member variables of either a class or an interface in an object-oriented software project. Member variables represent all the variables that are defined within a specific class or a specific interface.
- **Parameter:** This subclass models the parameters of an object-oriented software project.

Properties

To better grasp the relations between the aforementioned classes, the following property definitions describe which classes are interlinked through their appropriate use and how these class specifications are refined with language independent information.

accesses: The numerous variables specified in the source code of a software project can be accessed from different locations within a class. On the one hand, they can be accessed within a method or a constructor. These possibilities are represented with the corresponding `accesses` relation between a `Method` and a `Variable` or a `Constructor` and a `Variable`. On the other hand, the variables that are accessed from the outside of a method or a constructor are merely modeled with a relationship between a `Class` and a `Variable` which is accessed within this `Class`. The inverse property is called `isAccessedBy`.

hasAnonymousClass: The investigated object-oriented programming languages provide the ability to define a class within another class. Usually such a class is labeled with its own name and its required definition is separated from its necessitated instantiations. However, the definition and the use of an anonymous class differ from the ones of usual classes in many ways. First of all, an anonymous class is defined without a name. Second, the specification and the instantiation of an anonymous class are combined within a single expression. Therefore, anonymous classes are often included as arguments in method invocations. The `hasAnonymousClass` property models this fact with the relationship between two distinct `Class` instances. The inverse property is called `isAnonymousClassOf`.

hasDeclaredConstant: The `hasDeclaredConstant` property relates an `Interface` to a `Constant` that is declared within this `Interface`. The inverse property is called `isDeclaredConstantOf`.

hasDeclaredConstructor: This object property outlines the relation between a `Class` and a `Constructor` which is declared within this `Class`. The inverse property is called `isDeclaredConstructorOf`.

hasDeclaredMethod: The hasDeclaredMethod property outlines the relation between a Class and a Method which is declared within this Class. The inverse property is called isDeclaredMethodOf.

hasDeclaredMethodSignature: This property specifies the relationship between an Interface and a Method which has its signature declared within this Interface. The inverse property is called isDeclaredMethodSignatureOf.

hasDeclaredVariable: This property outlines the relationship between a Class and a Variable which is declared within this Class. The inverse property is called isDeclaredVariableOf.

hasLocalVariable: The hasLocalVariable property models the relationship between either a Method and a LocalVariable or a Constructor and a LocalVariable. The inverse property is called isLocalVariableOf.

hasMethod: In an object-oriented programming language an instance of a class is referred to as an object. If the appropriate access modifier of a method is set, such an object is able to invoke the methods that are declared in its derived class. Furthermore, the inherited methods of its superclasses can also be invoked by such an object. The hasMethod property in our Code ontology models this relationship between a Class and a Method whereas in this situation the Method class represents both of the aforementioned method types. The inverse property is called isMethodOf.

hasNestedClass: The investigated programming languages allow the definition of a class within another class. Such a class is referred to as a nested class. The hasNestedClass property models this relationship between two distinct Class instances in our Code ontology. The inverse property is called isNestedClassOf.

hasParameter: The hasParameter property models the relationship between either a Method and a Parameter or a Constructor and a Parameter that is included in the parameter declaration of either of them. The inverse property is called isParameterOf.

hasReturnType: The hasReturnType property is an abstract property that represents the relationship between the Method class and the Type class which denotes the corresponding return type of this Method. The two subproperties mentioned below are the ones actually used to indicate the class object or the primitive data type that is returned through a method invocation. The inverse property is called isReturnTypeOf.

- **hasReturnComplexType:** This subproperty models the relationship between the Method class and the ComplexType class, which represents the appropriate return type of the interlinked Method instance. The inverse property is called isReturnComplexTypeOf.
- **hasReturnPrimitiveDataType:** This subproperty models the relationship between the Method class and the PrimitiveDataType class, which represents the corresponding return type of the interconnected Method instance. The inverse property is called isReturnPrimitiveDataTypeOf.

hasSourceCode: The `hasSourceCode` property interconnects the Code ontology with the Version ontology. In this connection a `version:File` is related to a `ComplexType` which contains the source code definition of a `Class` or an `Interface`. The inverse property is called `isSourceCodeOf`.

hasSubType: The `hasSubType` property is an abstract property that relates the `ComplexType` class to itself. The two subproperties stated below are the ones actually used to indicate the subclass relationship between two `Class` instances or to indicate the subinterface relationship between two `Interface` instances. The inverse property is called `isSubTypeOf`.

- **hasSubClass:** This subproperty models the relationship between two different `Class` individuals by interconnecting the superordinate `Class` instance with the subordinate `Class` instance. The inverse property is called `isSubClassOf`.
- **hasSubInterface:** This subproperty models the relationship between two distinct individuals of the `Interface` class by connecting the superordinate `Interface` instance with the subordinate `Interface` instance. The inverse property is called `isSubInterfaceOf`.

hasType: The `hasType` property is an abstract property that represents the relationship between the `Variable` class and the `Type` class, which denotes the corresponding type of this `Variable`. The subproperties mentioned below are the ones actually used to indicate what annotation type, complex type (class type or interface type) or primitive data type a specific variable has. The inverse property is called `isTypeOf`.

- **hasAnnotationType:** This subproperty models the relationship between a `Variable` and an `AnnotationType` that outlines what kind of annotation is represented with this `Variable`. The inverse property is called `isAnnotationTypeOf`.
- **hasComplexType:** This subproperty models the relationship between a `Variable` and a `ComplexType` that outlines what kind of class or interface is represented with this `Variable`. The inverse property is called `isComplexTypeOf`.
- **hasPrimitiveDataType:** This subproperty models the relationship between a `Variable` and a `PrimitiveDataType` that outlines what kind of primitive data type is represented with this `Variable`. The inverse property is called `isPrimitiveDataTypeOf`.

implements: This property represents the relationship between a `Class` and an `Interface` that is implemented by this `Class`. The inverse property is called `isImplementedBy`.

instantiates: An object of a class can be instantiated in various positions within the source code of a software project. First, an object is potentially instantiated as an attribute of another class. This possibility is modeled with the `instantiates` connection between two different `Class` instances. Second, an object can be instantiated within a method or a constructor. These occurrences are represented with the corresponding `instantiates` relationship between either a `Method` and a `Class` or a `Constructor` and a `Class`. Although the upcoming possibility is rarely used in practice for the reason that it would result in an unchangeable object, object-oriented programming languages allow the instantiation of an object within an interface. The `instantiates` relationship between an `Interface` and a `Class` incorporates this aspect in our Code ontology. The inverse property is called `isInstantiatedBy`.

invokes: In an object-oriented programming language a method can be invoked in different source code positions. On the one hand, a method is usually invoked within another method. The `invokes` property models this relationship between two `Method` individuals. On the other hand, a method is possibly also invoked within a constructor. This aspect is represented within the `invokes` relation between a `Constructor` and a `Method`. Finally, a method which is not invoked within these two constructs is simply represented as an `invokes` connection between a `Class` and a `Method` which has been invoked in this particular `Class`. The inverse property is called `isInvokedBy`.

throwsException: As illustrated in Figure 4.19, the `throwsException` property interconnects the `Method` class with the `Type` class which represents the exception type that is thrown by this `Method`. The inverse property is called `isThrownExceptionBy`.

hasAccessModifier: The `hasAccessModifier` property specifies the access modifier of a `Class`, a `Constructor`, a `Method`, a `Variable` or an `Interface` as a new string. The intended values include among others the strings “public”, “private” and “protected”.

hasName: The `hasName` property defines the name of a specific individual that belongs to one of our classes in the Code ontology as a new string. Since this property is also used in the language specific Java ontology and CSharp ontology, it is deliberately not restricted to be used with the appropriate classes of the Code ontology.

hasPosition: This property denotes the position of a `Parameter` within the parameter declaration of either a method or a constructor as an integer.

hasSignature: This data type property specifies the signature of either a `Method` or a `Constructor` as a string. In the investigated object-oriented programming languages the signature of either one of them comprises the name and the parameter declaration of the method or the constructor in question.

hasSourceCodeComment: The `hasSourceCodeComment` property represents a source code comment of the object-oriented programming languages as a string. This property is not restricted to be used with the classes of the Code ontology as the language specific classes of the Java ontology and the CSharp ontology might also use it for their comments.

isAbstract: This data type property denotes whether a `Class`, a `Method` or an `Interface` is abstract or not with the corresponding value of a boolean.

isFinal: This data type property denotes whether a `Class`, a `Method` or a `Variable` is final or not with the corresponding value of a boolean.

isStatic: This data type property denotes whether a `Class`, a `Method`, a `Variable` or an `Interface` is static or not with the corresponding value of a boolean.

Figure 4.21: Class hierarchy and property classification of the Java ontology; classes (on the left) and object properties (on the right)

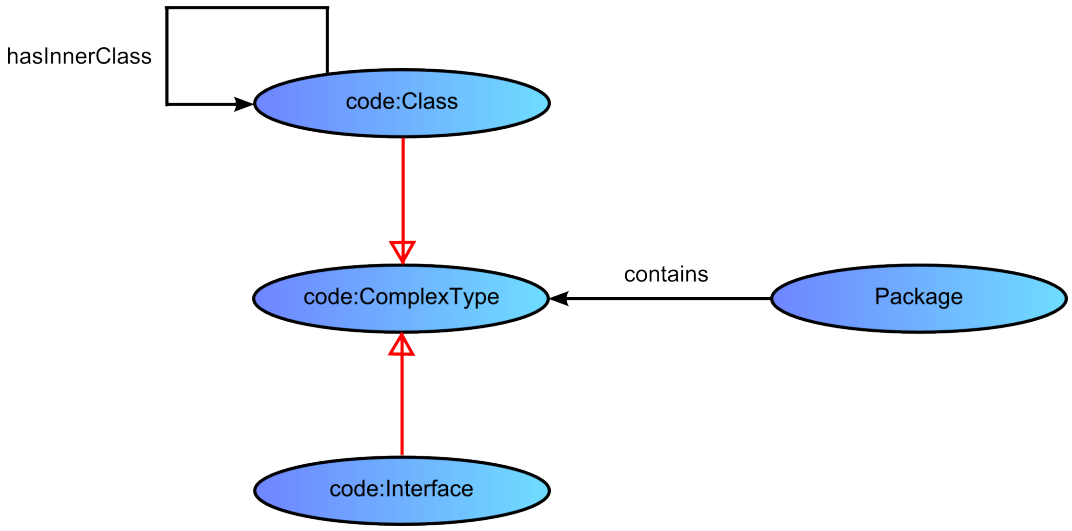

Figure 4.22: Classes and properties of the Java ontology

4.5.2 Java Ontology

The Java ontology imports the Code ontology and is therefore capable of reusing all class definitions along with the property definitions included in the Code ontology. Due to the transitive characteristic of the import statement the Java ontology is also able to reuse all definitions specified in the Version ontology. For the purpose of our static source code information ontologies, we have not included every detail of the Java programming language in our ontology as this would simply lead to a confusing and unclear specification. Nevertheless, the defined vocabulary provides a basis for further discussions and future enhancements.

To exemplify the intended extensions, we have specified additional classes and properties for the packaging mechanism of the Java programming language. These are displayed in Figure 4.21.

Class

The Java programming language provides a straightforward packaging mechanism for its programmers to bundle related classes and interfaces into packages. The many-sided advantages of such a grouping include a simplified access control and the ability to retrieve and reuse classes and interfaces with ease while avoiding naming conflicts. Figure 4.22 illustrates how this mechanism is built in within our Java ontology.

Package: The `Package` class represents a package of the Java programming language.

Properties

The subsequent property descriptions indicate the meaning of these properties in the Java domain and which classes are related through their proper use.

contains: The `contains` property relates the aforementioned `Package` class to the `code:ComplexType` class, which includes the `code:Class` class and the `code:Interface` class. These stand for the classes and interfaces that are bundled with the packaging mechanism. The inverse property is called `isContainedBy`.

hasInnerClass: The Java programming language allows the definition of a class within another class. Such a class is referred to as a nested class in the Java environment. Nested classes that are declared static are typically called static nested classes and are represented in our ontologies with the `code:hasNestedClass` property. Nested classes that are not declared static are usually called inner classes. The `hasInnerClass` property models this relationship between two different `code:Class` instances. The inverse property is called `isInnerClassOf`.

Individuals

The investigated object-oriented programming languages have predefined different primitive data types. Hence, the Java language specific primitive data types are taken into account with the definition of appropriate individuals for the `code:PrimitiveDataType` class in our Java ontology. These are listed in Table 4.6.

Table 4.6: Predefined individuals of the code:PrimitiveDataType class in the Java language

code:PrimitiveDataType
Boolean
Byte
Char
Double
Float
Int
Long
Short

4.5.3 CSharp Ontology

Like the Java ontology, the CSharp ontology imports the Code ontology and is consequently able to reuse the definitions included in the Code ontology. And since the Code ontology imports the Version ontology on its own, the CSharp ontology can make use of these classes, properties and individuals as well. To cover the static source code information of the domain of discourse, we did not include all particulars of the C# programming language in our ontology as this would only result in an overcrowded and confusing specification. Our intention is to define a fundamental vocabulary that provides a basis for future discussions and continuing improvements.

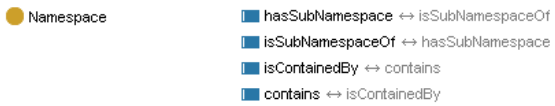

Figure 4.23: Class hierarchy and property classification of the CSharp ontology; classes (on the left) and object properties (on the right)

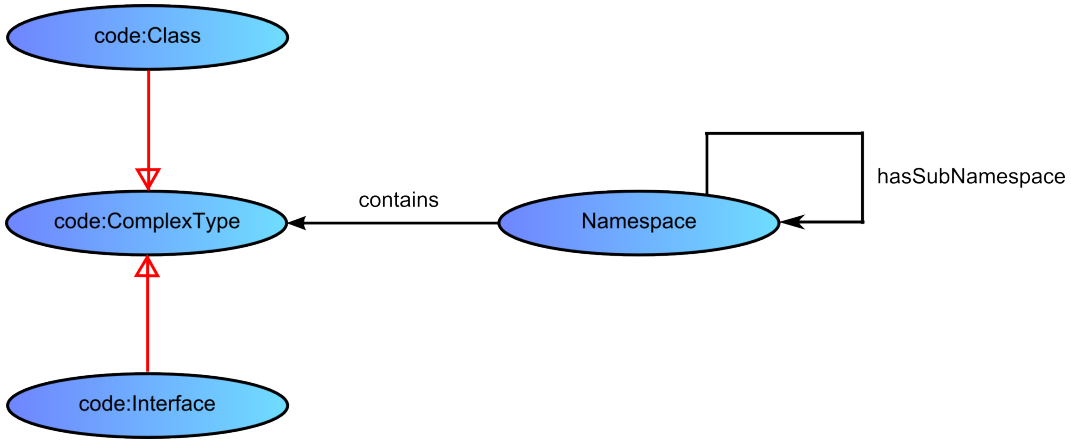

Figure 4.24: Classes and properties of the CSharp ontology

The hierarchical organization of a source code project into appropriate namespaces and subnamespaces is an essential equipment of the C# programming language. This equipment is incorporated in our CSharp ontology with additional classes and properties to exemplify future extensions. Figure 4.23 shows the classes and properties that have been added for this purpose.

Class

In the C# environment the specification of namespaces and subnamespaces helps organize the source code of a software project into a hierarchical system. A namespace is basically an abstract container which includes among others the classes and interfaces defined in a specific software project. An essential advantage of this hierarchical organization is that it avoids naming conflicts in large source code projects. Therefore, we have added a couple of novel classes and properties to represent this fundamental structuring.

Figure 4.24 relates these newly added classes and properties to better illustrate their proper use in the C# domain.

Namespace: The `Namespace` class models a corresponding namespace of the C# programming language.

Properties

The following property descriptions explain which classes are interconnected with their appropriate use and which actual situation they represent in the C# domain.

contains: The `contains` property relates the `Namespace` class to the `code:ComplexType` class which includes the `code:Class` class and the `code:Interface` class. These stand for the classes and interfaces that are organized into namespaces in the C# programming language. The inverse property is called `isContainedBy`.

hasSubNamespace: The C# programming language supports the hierarchical structuring of namespaces. Therefore, the `hasSubNamespace` property models the relationship between two instances of the `Namespace` class by interconnecting the superordinate namespace with the subordinate namespace. The inverse property is called `isSubNamespaceOf`.

Individuals

With respect to the predefined primitive data types, the main difference between the Java programming language and the C# programming language consists of additional support for unsigned primitive data types in the C# language. Unsigned primitive data types merely represent positive numbers whereas signed primitive data types represent positive numbers as well as negative numbers. Thereby, the range of possible numbers is typically divided into two almost equal ranges with the zero point located in the middle. Hence, an unsigned primitive data type is able to represent a maximum positive number that is twice as high as the one represented in the corresponding signed primitive data type.

Table 4.7 lists the possible primitive data types in the C# language as individuals of the `code:PrimitiveDataType` class. The unsigned primitive data types are represented with the `Byte`, the `UInt`, the `ULong` and the `UShort` individuals. Their signed counterparts are modeled with the `SByte`, the `Int`, the `Long` and the `Short` individuals.

Table 4.7: Predefined individuals of the code:PrimitiveDataType class in the C# language

code:PrimitiveDataType
Bool
Byte
Char
Decimal
Double
Float
Int
Long
SByte
Short
UInt
ULong
UShort

Chapter 5

Object-Oriented Metrics

This chapter describes sophisticated software metrics, which are used to evaluate the design of object-oriented software projects, and the implementation of our object-oriented metrics library. Section 5.1 provides an initial overview of the different design disharmonies of software projects and outlines the simple object-oriented metrics that constitute these design disharmonies. The Feature Envy and the Shotgun Surgery design disharmonies are described in detail, since they are implemented in our object-oriented metrics library. Section 5.2 discusses the annotation of static source code information with our ontologies using EVOLIZER and outlines the appropriate use of our object-oriented metrics library. We conclude this chapter by evaluating a medium-sized open source software project with our metrics library.

5.1 Overview

The object-oriented software metrics incorporated in our library are based on the explanations provided by Lanza and Marinescu in [11]. The purpose of these software metrics is intended to be used in order to evaluate and improve the design of object-oriented software projects.

Due to the complexity of today's object-oriented software systems, it is important to find an appropriate design for the development of an object-oriented software project. The adherence to the chosen design is typically measured with object-oriented metrics that focus on the quality of the implemented design. Simple object-oriented metrics are not suited for this purpose, because they cannot evaluate the actual design of an object-oriented software project. The object-oriented metrics described by Lanza and Marinescu in [11] aim to check design harmony in an object-oriented software project to enable more sophisticated conclusions about the implemented design. Lanza and Marinescu identified three different categories of design disharmonies, which are subsequently described in more detail.

Identity Disharmonies: This kind of design disharmonies is characterized by the fact that single entities such as classes and methods are affected by them. In addition, for proper identification of these design disharmonies, the desired entities are investigated in isolation without paying attention to their relationships and dependencies to other entities.

For the purpose of our object-oriented metrics library, we are mainly interested in the simple object-oriented metrics that constitute the various identity disharmonies.

Table 5.1 illustrates the required object-oriented metrics for the identification of the God Class and the Feature Envy design disharmonies. If a class tends to attract more and more features from other classes, a God Class design disharmony might be detected for this class. The CYCLO metric

is only used for the measurement of the WMC metric which is why the check mark of the CYCLO metric is put into round brackets. The Feature Envy design disharmony identifies the methods that access more attributes of other classes than the ones from their own class.

Table 5.1: Identity Disharmonies: Object-oriented metrics required for the identification of the God Class and the Feature Envy design disharmonies

Identity Disharmonies	ATFD	WMC	TCC	LAA	FDP	CYCLO
God Class	✓	✓	✓			(✓)
Feature Envy	✓			✓	✓	

Legend: ATFD = Access To Foreign Data, WMC = Weighted Method Count, TCC = Tight Class Cohesion, LAA = Locality of Attribute Accesses, FDP = Foreign Data Providers, CYCLO = Cyclomatic Number

Table 5.2 outlines the object-oriented metrics that are needed to identify the Brain Method and the Brain Class design disharmonies. Thereby, oversized and too complex methods are often a sign for the presence of a Brain Method design disharmony and the complex classes that host these methods are just as often affected by a Brain Class design disharmony.

Table 5.2: Identity Disharmonies: Object-oriented metrics required for the identification of the Brain Method and the Brain Class design disharmonies

Identity Disharmonies	LOC	CYCLO	MAXNESTING	NOAV	WMC	TCC
Brain Method	✓	✓	✓	✓		
Brain Class	✓	(✓)			✓	✓

Legend: LOC = Lines Of Code, CYCLO = Cyclomatic Number, MAXNESTING = Maximum Nesting Level, NOAV = Number Of Accessed Variables, WMC = Weighted Method Count, TCC = Tight Class Cohesion

Table 5.3 describes the various object-oriented metrics that build up the Data Class and the Significant Duplication design disharmonies. The classes that are identified as having a Data Class design disharmony are merely providing raw data and do not offer much functionality in terms of the methods they implement. The Significant Duplication design disharmony is primarily present in oversized and too complex classes or methods and describes the repeated occurrence of a similar chain of operations.

Table 5.3: Identity Disharmonies: Object-oriented metrics required for the identification of the Data Class and the Significant Duplication design disharmonies

Identity Disharmonies	WOC	NOPA	NOAM	WMC	SEC	LB	SDC	CYCLO
Data Class	✓	✓	✓	✓				(✓)
Significant Duplication					✓	✓	✓	

Legend: WOC = Weight Of a Class, NOPA = Number Of Public Attributes, NOAM = Number Of Accessor Methods, WMC = Weighted Method Count, SEC = Size of Exact Clone, LB = Line Bias, SDC = Size of Duplication Chain, CYCLO = Cyclomatic Number

Collaboration Disharmonies: In contrast to the identity disharmonies, the collaboration disharmonies affect several entities at once. As its name reveals it, this category of design disharmonies regards the collaboration between these entities, which is needed to provide a specific functionality. Therefore, the presence of a collaboration disharmony triggers various changes and adaptations in these entities in order to improve the design of object-oriented software projects.

For the purpose of our object-oriented metrics library, we are mainly interested in the simple object-oriented metrics that constitute the various collaboration disharmonies.

Table 5.4 illustrates which object-oriented metrics are used to describe the Intensive Coupling, the Dispersed Coupling and the Shotgun Surgery design disharmonies. If a method invokes a substantial number of different methods that are declared in very few other classes, an Intensive Coupling design disharmony might be identified for this method. The Dispersed Coupling design disharmony identifies the methods that are invoking distinct methods from too many other classes. The Shotgun Surgery design disharmony combines the two previously mentioned collaboration disharmonies and applies to the methods that invoke a substantial number of different methods that are declared in many other classes.

Table 5.4: Collaboration Disharmonies: Object-oriented metrics required for the identification of the Intensive Coupling, the Dispersed Coupling and the Shotgun Surgery design disharmonies

Collaboration Disharmonies	CINT	CDISP	MAXNESTING	CM	CC
Intensive Coupling	✓	✓	✓		
Dispersed Coupling	✓	✓	✓		
Shotgun Surgery				✓	✓

Legend: CINT = Coupling Intensity, CDISP = Coupling Dispersion, MAXNESTING = Maximum Nesting Level, CM = Changing Methods, CC = Changing Classes

Classification Disharmonies: The classification disharmonies characterize the kind of design disharmonies that concern the inappropriate use of the inheritance mechanism within object-oriented software projects. Thereby, the inheritance mechanism is mainly misused as an alternative possibility for reusing source code rather than being appropriately used as the ability to define more specific objects for more general ones.

For the purpose of our object-oriented metrics library, we are mainly interested in the simple object-oriented metrics that constitute the various classification disharmonies.

Table 5.5 illustrates the various object-oriented metrics that are required for the identification of the Refused Parent Bequest design disharmony. If a class denies the inherited bequest, the Refused Parent Bequest design disharmony might be detected for this class.

Table 5.5: Classification Disharmony: Object-oriented metrics required for the identification of the Refused Parent Bequest design disharmony

Classification Disharmony	NProtM	BUR	BOvR	AMW	WMC	NOM	CYCLO
Refused Parent Bequest	✓	✓	✓	✓	✓	✓	(✓)

Legend: NProtM = Number of Protected Members, BUR = Base Class Usage Ratio, BOvR = Base Class Overriding Ratio, AMW = Average Method Weight, WMC = Weighted Method Count, NOM = Number Of Methods, CYCLO = Cyclomatic Number

Table 5.6 outlines for the Tradition Breaker design disharmony, which object-oriented metrics are needed to identify it. Whenever a class excessively increases the inherited interface of its superclass with additional services that do not fit into the same level of abstraction, a Tradition Breaker design disharmony will probably be found for this class.

Table 5.6: Classification Disharmony: Object-oriented metrics required for the identification of the Tradition Breaker design disharmony

Classification Disharmony	NAS	PNAS	AMW	WMC	NOM	CYCLO
Tradition Breaker	✓	✓	✓	✓	✓	(✓)

Legend: NAS = Number of Added Services, PNAS = Percentage of Newly Added Services, AMW = Average Method Weight, WMC = Weighted Method Count, NOM = Number Of Methods, CYCLO = Cyclomatic Number

In order to be in the position to implement the aforementioned design disharmonies, we had to identify the simple object-oriented metrics that are supported by our static source code information ontologies. The outcome of this investigation is shown in Table 5.7, which briefly illustrates the supported as well as the unsupported simple object-oriented metrics. Thereby, the ATFD metric is listed as being both supported and unsupported by our software engineering ontologies. This can be traced back to the fact that only part of this metric is supported by our ontologies, namely accessing attributes directly is represented in our ontologies whereas accessing attributes through an accessor method is not modeled in our ontologies.

Table 5.7: Supported and unsupported object-oriented metrics by our software engineering ontologies

Supported Metrics	ATFD, TCC, LAA, FDP, WOC, NOPA, NOAV, CINT, CDISP, CM, CC, NProtM, BUR, BOvR, NOM, NAS, PNAS
Unsupported Metrics	ATFD, WMC, NOAM, LOC, CYCLO, MAXNESTING, SEC, LB, SDC, AMW

Even though the majority of the simple object-oriented metrics are supported by our static source code information ontologies, the disadvantageous composition of the various design disharmonies in terms of their required object-oriented metrics only enables the complete implementation of the Feature Envy and the Shotgun Surgery design disharmonies in the first place. The following sections describe these two design disharmonies in more detail.

5.1.1 Feature Envy

In an object-oriented programming language, objects are used to group the data and the methods that make use of these data. Whenever a method accesses a substantial number of attributes that are declared in other classes, a Feature Envy design disharmony might be present. Whether the attributes are accessed directly or through an accessor method is not essential for the identification of this design disharmony, which could indicate the methods that have been placed in an inappropriate class.

The rationale for the identification of the Feature Envy design disharmony can be found in the benefits of minimizing ripple effects when methods are changed. This means that changing these methods will not lead to the propagation of other method modifications and adaptations.

To properly state that the investigated method has a Feature Envy design disharmony, the three characteristics listed below have to be satisfied by this method.

- **The investigated method accesses (directly or through an accessor method) more than a few attributes that are declared in other classes.** For this purpose, the Access To Foreign Data (ATFD) metric is determined, which counts the number of attributes that are declared in another class and are accessed within this method.
- **The investigated method accesses far more attributes that are declared in other classes than the ones that are declared in the same class to which the investigated method belongs.** For this purpose, the Locality of Attribute Accesses (LAA) metric is determined, which relates the number of accessed attributes that are declared in other classes to the total number of accessed attributes. If more than one third of the accessed attributes is declared in other classes, this could indicate a Feature Envy design disharmony.
- **The investigated method accesses attributes that are declared in very few other classes.** For this purpose, the Foreign Data Providers (FDP) metric is determined, which counts the number of other classes that declare the attributes accessed within this method.

5.1.2 Shotgun Surgery

When a method is invoked in too many locations within the source code of an object-oriented software project, the modification of this method triggers various changes and adaptations in these different locations. If these locations are spread over a large number of classes and methods, the probability that one of the required changes and adaptations will be missed out is not to be underestimated. The Shotgun Surgery design disharmony identifies the methods that are invoked in too many other locations from the outside of the class hierarchy that declares the investigated method.

In order to primarily detect the methods that are most concerned, this design disharmony takes into account the strength as well as the dispersion of coupling to the other locations. The strength of the coupling is defined by the number of methods that invoke the investigated method and are declared in another class. The dispersion of the coupling is defined by the number of classes that declare the methods which invoke the investigated method.

An early identification of the Shotgun Surgery design disharmony can lead to less maintenance efforts, because the required changes and adaptations in the other locations can be minimized. Therefore, this design disharmony only considers the method invocations outside of the class hierarchy that declares the investigated method, since these are harder to trace back for their required changes and adaptations.

To properly state that the investigated method has a Shotgun Surgery design disharmony, the two characteristics listed below have to be satisfied by this method.

- **The investigated method is invoked by too many other methods that are declared in another class as this method.** For this purpose, the Changing Methods (CM) metric is determined, which counts the number of methods that invoke this method.
- **The investigated method is invoked by other methods that are declared in many other classes.** For this purpose, the Changing Classes (CC) metric is determined, which counts the number of classes that declared the other methods invoking this method.

With the background knowledge gained in the previous two design disharmony definitions, the following section describes our implementation of an object-oriented metrics library, which evaluates the design of object-oriented software projects with regard to these two design disharmonies.

5.2 Implementation

One intended use case of our software engineering ontologies is their application in our object-oriented metrics library. For this purpose, the source code information of software projects is adequately annotated with our static source code information ontologies in order to identify design disharmonies for the investigated source code entities. Since writing these annotations by hand would be a very cumbersome and time-consuming task, we leveraged the functionality of EVOLIZER[1] to extract and annotate the desired source code information of software projects. Initially implemented as a set of data importers and preprocessors, EVOLIZER has evolved into a platform for various tools that assist software developers in their daily tasks [18]. Thereby, the incorporated implementation of the FAMIX meta-model [6] describes object-oriented source code in a language independent way. These descriptions are used whenever static source code information is required for further software analyses. The FAMIX entities that are described in this way include among others classes, methods and attributes. These FAMIX entities can directly be extracted from the source code with EVOLIZER and are annotated with our static source code information ontologies.

Listing 5.1 illustrates how these FAMIX entities are annotated with our static source code information ontologies by using the customized `@rdf` Java annotation.

```
1  @rdf("http://www.evolizer.org/ontology/Code.owl#Method")
2  public class Method extends FamixEntity {
3  ...
4
5     @rdf("http://www.evolizer.org/ontology/Code.owl#hasReturnType")
6     public Class getDeclaredReturnClass() {
7        ...
8     }
9     ...
10
11    @rdf("http://www.evolizer.org/ontology/Code.owl#hasParameter")
12    public List<FormalParameter> getFormalParameters() {
13       ...
14    }
15    ...
16
17    @rdf("http://www.evolizer.org/ontology/Code.owl#hasLocalVariable")
18    public Set<LocalVariable> getLocalVariables() {
19       ...
20    }
21    ...
22
23 }
```

Listing 5.1: Example of an annotated FAMIX entity representing a Java method in the source code

The URI prefix used in the following descriptions is the same as the one we introduced in Table 4.1 in Chapter 4. The Java class `Method`, which is instantiated for every method in the source code

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

of an object-oriented software project, is annotated with the URI `code:Method` of our Code ontology. A consequence of this annotation is that for every instance of the Java class `Method` a corresponding OWL class `code:Method` is created in the RDF graph. In more detail, a new resource is created in the RDF graph with a `rdf:type` property that has the `code:Method` as its value. The annotated methods in the Java class `Method` are represented as additional properties of the previously created resource in the RDF graph whereas the return values of the annotated methods denote the corresponding values of the properties in the RDF graph [18]. For example, the annotated `getFormalParameters()` method is represented as a `code:hasParameter` property in the RDF graph. Since the return type of the annotated method is `List<FormalParameter>`, a new `code:hasParameter` property is added to the resource for every `FormalParameter` in the `List<FormalParameter>` with a `FormalParameter` resource as its value.

This mapping between Java and OWL is applied whenever a Java class of the implemented FAMIX meta-model corresponds to a OWL class in our source code ontologies. Thereby, the methods of a Java class correspond to the properties of a OWL class. If a Java class is modeled as a property in our ontologies, the appropriate `@rdf` Java annotation differs in several ways. This is illustrated in Listing 5.2.

```
1  @rdf(
2     isPredicate=true,
3     value="http://www.evolizer.org/ontology/Code.owl#accesses"
4  )
5  public class Access {
6  ...
7
8     @subject
9     public Method getAccessedIn() {
10       ...
11    }
12    ...
13
14    @object
15    public StructuralEntity getAccesses() {
16       ...
17    }
18    ...
19
20 }
```

Listing 5.2: Example of an annotated FAMIX object representing a Java attribute access in the source code

The additional flag `isPredicate` of the `@rdf` Java annotation is set to **`true`** in order to explicitly state that the Java class is modeled as a property in our ontologies. In Listing 5.2 the Java class `Access` is modeled as the `code:accesses` property in our source code ontologies. The two other Java annotations in Listing 5.2, namely the `@subject` and the `@object` Java annotations, are used to specify the subject and the object of the RDF statements that use this property. In our example, the return type of the annotated `getAccessedIn()` method specifies the subject of the `code:accesses` property and the return type of the annotated `getAccesses()` method specifies the object of this property [18].

In this way, we can extract the FAMIX entities from the source code of an object-oriented

software project with EVOLIZER and annotate them with our software engineering ontologies.

Our implementation of the Feature Envy and the Shotgun Surgery design disharmonies makes use of these annotations to determine whether a source code entity is affected by one of these design disharmonies or not. As mentioned in Section 5.1 of this chapter, the ATFD metric is only partially supported by our software engineering ontologies. Therefore, the implementation of the Feature Envy design disharmony merely considers the attributes that are directly accessed to determine the ATFD metric. The attributes that are accessed through an accessor method are currently ignored for the calculation of the ATFD metric, but are intended to be incorporated in a later release of our object-oriented metrics library in order to provide more precise investigations.

The remainder of this section describes the appropriate use of our object-oriented metrics library, which is depicted in Listing 5.3.

```
1  // instantiating object-oriented metrics library
2  OOMetrics metrics = new OOMetrics(inputURI, baseURI);
3  ...
4
5  // adding local and remote ontology documents
6  metrics.addLocalOntDocument(docURL, locationURI);
7  metrics.addRemoteOntDocument(docURL, locationURI);
8  metrics.addRemoteOntDocument(docURL);
9  ...
10
11 // creating inferred model
12 metrics.createInfModel();
13 ...
14
15 // investigating method for Feature Envy design disharmony
16 try {
17    FeatureEnvyReport report = metrics.createFeatureEnvyReport(methodURI);
18    report.printReport();
19 } catch (NoMethodURIException e) {
20    ...
21 }
22 ...
23
24 // investigating method for Shotgun Surgery design disharmony
25 try {
26    ShotgunSurgeryReport report =
          metrics.createShotgunSurgeryReport(methodURI);
27    report.printReport();
28 } catch (NoMethodURIException e) {
29    ...
30 }
```

Listing 5.3: Object-oriented metrics library in use

A new `OOMetrics` object is instantiated with the two string parameters `inputURI` and `baseURI`. The `inputURI` string specifies the location of the input document, which stores the annotated FAMIX entities extracted from the source code with EVOLIZER. Currently, this input docu-

ment needs to be located in the local file system and consequently its URI needs to start with the well-known "file:" string as its prefix. If the input document contains relative URIs, the `baseURI` string has to be specified, which outlines the prefix that needs to be added to these relative URIs. Otherwise the `OOMetrics` object is instantiated with an empty string for the `baseURI`. The instantiation of an `OOMetrics` object creates an initial in-memory ontology model, which can already be used to identify the affected methods by the Feature Envy or the Shotgun Surgery design disharmonies.

In most cases, the input document contains only part of the possible meta-data that can be expressed with our static source code information ontologies. In order to be in the position to infer additional meta-data about the annotated source code entities, the ontology documents that define our static source code information ontologies need to be added to the `OOMetrics` object. It is important to be aware of the fact that this is not a mandatory step in order to use our object-oriented metrics library and therefore it can be omitted if all required meta-data are included in the input document. Nevertheless, adding these ontology documents with `addLocalOntDocument(docURL, locationURI)`,`addRemoteOntDocument(docURL, locationURI)` and `addRemoteOntDocument(docURL)` has the benefit of more precise identifications of the affected methods with our Feature Envy and Shotgun Surgery design disharmony implementations. The two possible string parameters `docURL` and `locationURI` specify on the one hand the URL of the ontology document in the Web and on the other hand the local URI of the same ontology document in the local file system. Therefore, the `docURL` needs to start with the "http:" prefix whereas the `locationURI` begins with the "file:" string as its prefix.

The subsequent `createInfModel()` method makes use of the initial in-memory ontology model and the added ontology documents (local and remote) in order to create the inferred model, which contains much more meta-data about the different source code entities as the initial model. Invoking this method is not a mandatory step in order to use our object-oriented metrics library and therefore it can be omitted if no additional ontology documents (local and remote) have been added in the previous step.

The `createFeatureEnvyReport(methodURI)` method and the `createShotgunSurgeryReport(methodURI)` method are used to investigate a specific method entity, which is identified with the `methodURI` string parameter. The `FeatureEnvyReport` object and the `ShotgunSurgeryReport` object contain detailed information about the corresponding design disharmony and can be printed to the systems output stream with the use of the `printReport()` method. Listing 5.4 illustrates the included information within these reports.

5.2.1 Evaluation

To evaluate the implementation of our object-oriented metrics library, we have used the latest release of the JFreeChart software project[2]. This open source chart library is implemented in Java and has more than 250,000 LOC. We have extracted a total of 36,148 FAMIX entities from the source code of JFreeChart with EVOLIZER whereas these entities contained 10,713 annotated methods, which we have analyzed with respect to being affected by the Feature Envy or the Shotgun Surgery design disharmonies. Table 5.8 illustrates the obtained results for both of these design disharmonies.

Even though our evaluation of JFreeChart did not identify a significant amount of affected methods, our object-oriented metrics library has shown to scale well for the evaluation of medium-sized software projects and can be useful to evaluate and improve the design of object-oriented software projects that do not meet the standard of the JFreeChart design.

[2]JFreeChart 1.0.13, http://www.jfree.org/jfreechart/

```
SHOTGUN SURGERY REPORT:
-----------------------

Inspected MethodURI: methodURI
RESULT: true or false

Number of Changing Methods: int
Number of Changing Classes: int

FEATURE ENVY REPORT:
--------------------

Inspected MethodURI: methodURI
RESULT: true or false

Number of Directly Accessed Foreign Attributes: int
Number of Directly Accessed Own Attributes: int
Number of Foreign Classes of Directly Accessed Foreign Attributes: int
Number of Directly Accessed Total Attributes: int
Directly Accessed Own Attributes Ratio: double
```

Listing 5.4: FeatureEnvyReport and ShotgunSurgeryReport outputs

Table 5.8: Number of affected and unaffected methods by the Feature Envy and the Shotgun Surgery design disharmonies within the JFreeChart software project

Design Disharmonies	Number of affected methods	Number of unaffected methods
Feature Envy	67 (0.6%)	10,646 (99.4%)
Shotgun Surgery	355 (3.3%)	10,358 (96.7%)
Total	422 (2.0%)	21,004 (98.0%)

Chapter 6

Conclusions

6.1 Summary of Contributions

In this thesis we have developed a structured set of software engineering ontologies for the representation of revision control data, issue tracking data and static source code information. In addition, we have implemented an object-oriented metrics library in order to evaluate and improve the design of object-oriented software projects.

First, we have reviewed the state of the art in ontology development. For this purpose, we have briefly described the needs for comprehensive methodologies in order to develop commonly shared and reused ontologies. In the subsequent methodology documentations we have presented in detail the widespread methodologies that rely on the most intuitive use from our point of view. Furthermore, we have analyzed the existing ontologies in the field of software engineering with regard to their suitability for integration in our software engineering ontologies.

Next, we have outlined the various requirements for our software engineering ontologies along with the investigated systems and object-oriented programming languages that have been considered for their appropriate specifications in OWL. We presented SEON, our structured set of software engineering ontologies, which is divided into three ontology pyramids. Each of these ontology pyramids represents a subarea in the domain of discourse and includes a general ontology defining similar concepts (and their relationships to one another) within the designated subarea as well as two additional ontologies defining the system specific or language dependent concepts (and their relationships to one another). The three subareas of the domain of discourse describe the data of revision control systems, issue tracking systems and the static source code information of object-oriented programming languages.

Finally, we have illustrated the object-oriented design disharmonies [11] with respect to the simple object-oriented metrics that constitute them and consequently are required for their adequate identification. We have presented the implementation and the proper use of our object-oriented metrics library, which is intended to be used to evaluate and improve the design of object-oriented software projects. Current evaluations have shown that our implementation scales well for medium-sized software projects and can be useful for the identification of affected methods.

6.2 Limitations

In this initial draft of our static source code information ontologies, we have currently not incorporated all language dependent concepts of the Java programming language and the C# pro-

gramming language. In the C# environment concepts like pointers, structs or delegates are omnipresent in the source code of software projects. Since we are not experts in this field, we deliberately left out the representation of these concepts in our CSharp ontology as it would not make much sense to provide an inappropriate definition of these concepts that is not going to be used for their descriptions. Experts in this field are encouraged to participate in further discussions about the included vocabularies and about the appropriate definitions of these concepts.

Furthermore, we have discussed many additional concepts and relationships that could be included in the general ontology of this subarea. Subjects of our discussions were concepts and relationships like generics, casts or instanceof tests. These concepts were intentionally left aside, because incorporating them into our initial ontology draft would make the entire ontology difficult to understand.

We conclude this section with a remark about the limitation of our object-oriented metrics library, which currently only supports the identification of the Feature Envy and the Shotgun Surgery design disharmonies. The reasons for this limitation are to be found in the present EVOLIZER implementation that cannot extract and annotate all the required information from the source code and in the current draft of our static source code information ontologies that have not incorporated several details, which are needed to calculate some essential metrics (e.g., the CYCLO metric).

6.3 Future Work & Considerations

The most suitable evaluation of an ontology with respect to conformity with its identified requirements is the application of this ontology for its intended purpose. Thereby, the life cycle of an ontology is comparable to the life cycle of a software project. In order to be shared and reused on a widespread basis, the ontology definition has to evolve constantly to satisfy the changing requirements of its users. Therefore, this initial draft of our software engineering ontologies should not be considered to be complete, but rather as an attempt to provide structured and elaborated vocabularies for the domain of discourse that can serve as a starting point for further discussions and continuous improvements.

Possible improvements concern including more fine-grained source code information in our static source code information ontologies. For instance, the `accesses` property in our Code ontology could be refined in order to distinguish whether a variable is accessed to be read or to be modified. Incorporating the statements that are found within a method could be another example of a future extension. It is important to keep in mind that an ontology definition needs to evolve constantly in order to meet the changing requirements of its users.

In addition, future work includes the development of another ontology for the representation of simple object-oriented metrics that are directly read out from the source code information of software projects. For example, the currently unsupported Lines Of Code (LOC) metric could then be described appropriately. The CYCLO metric, which outlines the number of linearly independent paths through the source code, could also be represented with this additional metrics ontology.

Finally, by using this new ontology our object-oriented metrics library could be extended to support the identification of other design disharmonies, even if the corresponding data cannot be extracted and annotated with EVOLIZER for the time being.

References

[1] G. Antoniou and F. van Harmelen. *A Semantic Web Primer*. The MIT Press, 2004.

[2] M. Bar and K. Fogel. *Open Source Development with CVS, 3rd Edition*. Paraglyph Press, 2003.

[3] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[4] P. V. Biron and A. Malhotra. *XML Schema Part 2: Datatypes Second Edition*. W3C Recommendation, 2004.

[5] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion, 2nd Edition*. O'Reilly Media, 2008.

[6] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0: The FAMOOS Information Exchange Model. Technical report, Software Composition Group, Institute of Computer Science and Applied Mathematics, University of Berne, Switzerland, 1999.

[7] M. Fernández. Overview of Methodologies For Building Ontologies. In *Proceedings of the IJCAI-99; Workshop on Ontologies and Problem-Solving Methods*, 1999.

[8] M. Fernández, A. Gómez-Pérez, and N. Juristo. METHONTOLOGY: From Ontological Art Towards Ontological Engineering. In *Proceedings of the AAAI97 Spring Symposium Series on Ontological Engineering*, pages 33–40, Stanford, USA, March 1997.

[9] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.

[10] G. Lakoff. *Women, Fire, and Dangerous Things*. University of Chicago Press, 1990.

[11] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.

[12] F. Manola and E. Miller. *RDF Primer*. W3C Recommendation, 2004.

[13] D. L. McGuinness and F. van Harmelen. *OWL Web Ontology Language Overview*. W3C Recommendation, 2004.

[14] R. Mizoguchi. Tutorial on ontological engineering part 2: Ontology development, tools and languages. *New Generation Computing*, 22(2):61–96, 2004.

[15] N. F. Noy and D. L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology. Technical report, Knowledge Systems, AI Laboratory, Department of Computer Science, Stanford University, California, United States of America, 2001.

[16] L. Sauermann, L. van Elst, and K. Möller. *Personal Information Model (PIMO)*. NEPOMUK Recommendation, 2009.

[17] M. Uschold and M. King. Towards a Methodology for Building Ontologies. In *Workshop on Basic Ontological Issues in Knowledge Sharing; held in conjunction with IJCAI-95*, 1995.

[18] M. Würsch, G. Ghezzi, G. Reif, and H. C. Gall. Supporting Developers with Natural Language Queries. Technical report, s.e.a.l. - software architecture and evolution lab, Department of Informatics, University of Zurich, Switzerland, 2009.

[19] O. Zimmermann, N. Schuster, and P. Eeles. Modeling and sharing architectural decisions, part 1: Concepts. Technical report, developerWorks, IBM Zurich Research Laboratory, Rüschlikon, Switzerland, 2008.