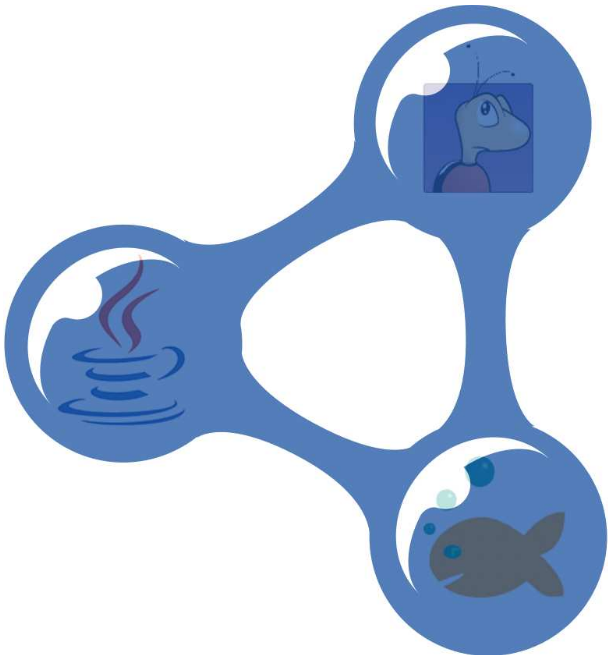
University of Zurich
Department of Informatics

Dynamic and Distributed Information Systems

Diploma Thesis March 13, 2007

Mining Software Repositories — A Semantic Web Approach



Jonas Tappolet
of Zurich ZH, Switzerland

Student-ID: 02-720-241
jonas@tappolet.ch

Advisor: **Christoph Kiefer**

Prof. Abraham Bernstein, PhD
Department of Informatics
University of Zurich
http://www.ifi.unizh.ch/ddis

Acknowledgements

I would like to thank Christoph Kiefer and Prof. Abraham Bernstein for giving me the opportunity to write this thesis and for their valuable inputs and supervision. Special thanks to Julia Schmoker for her moral support and proofreading, and, last but not least my parents, Alfred and Bernadette Tappolet for their support and making it possible that I actually could write this thesis. The logo displayed on the titlepage is inspired by the official RDF, Java, CVS and Bugzilla logos.

Abstract

Modern software development has become a complex task. Software systems grow larger and are densely interconnected to other systems making excessive use of large communication frameworks. To cope with this complexity, software developers and project managers need the assistance of tools which extract information about flaws in code as well as general information about the state of a project. In this thesis, we first introduce a data exchange format based on OWL/RDF, the Semantic Web's format of choice today, able to store data and meta data from the source code, versioning system (i.e. CVS) and bug tracking system (i.e. Bugzilla). In a next step, we present a tool to retrieve the data from the online software repositories and to store it in OWL/RDF. This tool is implemented as a plug-in for the Eclipse IDE and is able to harvest data from projects managed by Eclipse. Finally, we evaluated our data format and tools by applying a set of software metric calculations, pattern detections and similarity measures by using *iSPARQL* and *SimPack*. The results of the conducted experiments are promising, and gave a first proof of our approach.

Zusammenfassung

Softwareentwicklung wurde über die Zeit immer komplexer. Die Softwaresysteme werden grösser und sind dichter miteinander verwoben, wobei eine vielzahl von Kommunikationsframeworks und Schnittstellen zum Einsatz kommen. Damit Softwareentwickler und Projektverantwortliche den Überblick in diesen komplexen Systemen behalten können, brauchen sie die Hilfe von Werkzeugen, welche Informationen über Schwächen im Code sowie den Zustand eines Softwareprojekts im Allgemeinen liefern können. In der vorliegenden Arbeit stellen wir zuerst ein Datenaustauschformat vor welches auf der OWL/RDF Syntax des Semantic Web's basiert. Dieses Format kann Daten speichern, welche aus dem Quellcode, Versionierungssystem (CVS) und Bugtrackingsystem (Bugzilla) extrahiert wurden. In einem weiteren Schritt haben wir ein Tool implementiert, welches Daten aus den erwähnten Systemen extrahiert, und diese im OWL/RDF Format speichert. Bei diesem Tool handelt es sich um ein Eclipse Plug-in welches auf die Daten der Projekte zugreifen kann, die von Eclipse verwaltet werden. In einem letzten Schritt evaluieren wir unser Datenformat und usere Tools indem wir einige Software-Metrik Berechnungen, Design-Muster Erkennungen und Ähnlichkeitsberechnungen mithilfe von *iSPARQL* und *SimPack* durchführen. Die Resultate dieser Experimente sind vielversprechend und bestätigen unseren Ansatz.

Table of Contents

List of Listings

List of Tables

List of Figures

1 Introduction

1.1 Motivation

In the past years, software development has become a more and more complex task. Software systems grow larger and have a myriad of interfaces to other systems and make excessive use of large frameworks. An effect of this increased complexity is a raising number of members in a developer team to cope with the larger effort of software engineering. This makes software projects more cost intensive and can be of vital importance for a company. The more significant software development becomes, the more people are interested in a successful end of the project. These people can either be the software developers themselves, the project leaders or even the company's management. It is not feasible for a management member to look into the source code to receive an overview of the status of a project. Different stakeholders need different information about a software project. Where a developer needs detailed information about the location of code flaws, a project leader may wants to know about the upcoming workload and a manager needs to know the general state of a project. To answer all these questions, the underlying information first needs to be fetched from the different repositories used for software development, to later extract the needed data to receive meaningful information about a software project.

1.2 Goals of this Work

To achieve the above mentioned needs, data from the three major repositories of software development, *source code*, *versioning system* and *bug tracking system*, shall be extracted. This data should be stored in the Semantic Web's [Berners-Lee et al., 2001] preferred format OWL/RDF. To extract the data from the repositories, a tool is needed to automate the retrieval of information from the mentioned repositories. In a last step, the tools and models are evaluated to show the abilities as well as the limitations of the presented approach. This thesis is structured as follows: Chapter 2 is about the ontologies, defining the general structure of our data. Chapter 3 gives a brief introduction to our plug-in, which accesses the online software repositories and feeds our knowledge base. Finally, in Chapter 4 we conduct several experiments to demonstrate the abilities of our approach.

1.3 Area of Application

The intended approach can be a valuable component in software development. A first area of application can be the usage as a standalone tool. This tool could run within the daily development work to extract data and run a bunch of measures to annotate a project by generating a report. Another application could simply be the generation of the model data to annotate a project semantically. A rising number of open source applications and frameworks are available in the Internet on platforms like *sourceforge*[1]. These project are often well documented using *Javadoc* as a documentation format crafted for human readability. It would be an advantage if a development tool could access a foreign component's structure and functionality. This would enable an IDE[2] to automate integration checks, determine dependencies etc. Due to the use of a Semantic Web data format (OWL/RDF), the location where the information resides is secondary. As today, many software projects host the human readable software documentation on its project website, the machine-readable models could be hosted there as well to be accessed from a remote CASE[3] tool.

1.4 Related Work

There are two different kinds of related work. First, there are approaches of methods mining software repositories. These approaches do not necessarily need to use Semantic Web techniques. Second, there are Semantic Web enabled works, of mining software repositories and supporting software development. A first approach to mention is *Coogle* (Code Google) [Sager et al., 2006] which is a predecessor of this thesis. Coogle does not use Semantic Web techniques but directly does its calculations within the Eclipse framework. It can calculate the similarity between two releases of a project applying tree compare algorithms. As Coolge operates on the in-memory abstract syntax tree of Eclipse, it has a limited extendability for future measuring methods. Most related to the here presented approach is probably the work of Hyland-Wood. They introduce an ontology model for software based on Java [Hyland-Wood et al., 2006]. However, they do not include bug data and versioning data.

D'Ambros & Lanza [D'Ambros and Lanza, 2006] present a visualization technique to uncover the relationships between data from a versioning and bug tracking system of a software project. To achieve this goal, they are also using a version of the Release History Database (RHDB) introduced by Fischer in [Fischer et al., 2003]. Both, Mäntylä [Mäntylä et al., 2003] and Shatnawi & and Li [Shatnawi and Li, 2006] carry out an investigation of bad code smells in object-oriented software source code. While the first study additionally presents a taxonomy (in our sense an ontology) of smells and examines its correlations, both studies provide empirical evidence that some code smells can be linked with errors in software design.

Happel [Happel et al., 2006] present their KOntoR approach that aims at storing and querying meta-data about software artifacts to foster software reuse. The software components are stored in a repository and they present various ontologies for providing background knowledge about the components, such as the programming language and licensing models. It is certainly reasonable to integrate their models with ours in the future to result in an even larger fact base used to analyze large software systems.

Finally, Dietrich & Elgar [Dietrich and Elgar, 2005] present an approach to automatically detect design patterns in Java programs based on an OWL design patterns ontology. Again, we

[1] `http://www.sourceforge.net/`
[2] Integrated Development Environment
[3] Computer Aided Software Engineering

think it would make sense to use their approach and ontology model to collect even more information about software projects. This would allow us to conduct further evaluations to measure the quality of software.

1.5 Technology

To be able to realize the above mentioned goals, different tools are used in this thesis.

1.5.1 OWL

OWL stands for Web Ontology Language. The swapped O and W in the acronym might be in imitation of the OWL in Alan Alexander Milnes *Winnie Puh*[4]. In OWL, the format of the data is plain XML. As XML has not the ability to express semantics, owl bridges this gap by defining XML tags allowing to express semantic linkage of data. The owl language offers three different flavors [W3C, 2004a]:

- **OWL Full** The full specification of the language
- **OWL DL** A subset of OWL Full, making some restrictions to allow automated reasoning
- **OWL Lite** A subset of OWL DL as a simple-to-use, simple-to-implement version of OWL.

For this thesis, OWL DL will be used, due to the ability of automated reasoning.

A Brief Description of the OWL Concepts

The OWL format has four major concepts to store information and its associations.

- **Classes** are abstract definitions of a single concept. Classes define possible associations and properties they can have. A class itself does not store concrete data, it only acts as a container concept.
- **Individuals** (also called instances) are the concrete realization of a class. They only can have associations and store data in the defined manner of their respective class.
- **Object properties** define the associations between two classes (abstract) or two individuals (concrete). Object properties are directed associations and always belong to a specific domain (i.e. the starting point of an association) and a range (i.e. the endpoint). Domain and range can both be a list of multiple Classes
- **Datatype properties** can be, likewise object properties, considered as associations. Unlike object properties, the range is not a list of classes but rather a predefined data type. Typically the data types of XML Schema [W3C, 2004b] are used.

[4]see `http://en.wikipedia.org/wiki/Web_Ontology_Language/` for more speculation about this acronym

1.5.2 Java

During this thesis, different steps require the implementation of different programs. The evaluation framework will be needed to implement specific functionality to collect, process and store the queried data. Eclipse [5] is used as framework for the harvesting tools to generate the data needed to create the models. The latter, Eclipse, itself is implemented in Java [6] causing interacting tools to be implemented in this language too. Another Java based framework used in this thesis is Jena. The availability of these tools and frameworks let us select Java as implementation language for our here presented programs.

1.5.3 Jena

Whenever coping with semantic web, and especially OWL/RDF data, most likely the Jena[7] framework is involved somehow. Initially developed by the HP Labs[8], it has become a virtual standard for processing OWL/RDF data. Jena features a complete interface to create, manipulate and query semantic web data i.e. OWL/RDF files.

1.5.4 ARQ / SPARQL

SPARQL [Prudh́ommeaux and Seaborne, 2006] is the W3C[9] standard query language for semantic web OWL/RDF data. To retrieve data using SPARQL, a triple template is defined in the query. The core idea is to leave the subject or object of a triple blank (variable), and the query engine will try to find all the triples matching this template. Query 1.1 shows a example of a query retrieving all persons from a foaf[10] ontology whose birthday is the 14th of May. The triple ?someone foaf:birthday <05−14> consists of only two fixed elements, the predicate and the object. The subject of the triple (?someone) is a variable and the query engine will match every triple in the ontology fitting this template.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
        ?someone foaf:birthday <05-14> .
        ?someone foaf:name ?name .
}
```

Listing 1.1: Query to retrieve the persons whose birthday is the 14th of May

Where SPARQL is only a definition for the grammar of the query language, ARQ is a concrete implementation of a query engine by Jena. ARQ implements SPARQL, but also offers extensions to access special functions from within a query.

[5] http://www.eclipse.org/
[6] http://java.sun.com/
[7] http://jena.sourceforge.net/
[8] http://hpl.hp.com/
[9] http://www.w3c.org/
[10] foaf — Friend of a Friend, http://www.foaf-project.org/

2
The Ontology Models

Our goal – extract information from software repositories – needs the definition of a structure this data can be stored in. We already introduced OWL/RDF, however, this is only the data format, and does not define or make restrictions on the data's structure. When we extract the elements of a software project, we create an instance file of this project. In this file, the concrete entities such as methods with their names and signatures or classes are stored. To define the possible set of entities and associations in an instance file, we need to define these in an ontology model. This model is necessary for the semantics of an instance file and to check its consistency. From the description logic's point of view, the instance file would refer to the *ABox* and the ontology model to the *TBox* [Baader et al., 2003]. In this chapter, three such ontology models are presented for each repository, source code, bug tracking system and version control system.

2.1 Namespaces

Every OWL/RDF ontology needs to define its own namespace. When defining a distinct namespace, it can be uniquely referred to by any other ontology in the world. This mechanism is a main pillar of the Semantic Web. However, Semantic Web has no central registry for such namespaces that would guarantee unambiguity. The idea of namespaces is to rely on the existing naming system of the Internet, i.e. DNS (Domain Name System). A creator of a OWL/RDF ontology should use an Internet domain name he is in charge of for the ontologies. This guarantees that no other ontology uses the same namespace. To have multiple ontologies defined in the same domain name, a whole URL can be used to finer define a namespace. Although a URL usually hosts a web page, this is not necessary for a namespace. It is even not needed for the URL to exist, however, a creator of an ontology is advised to host the source of the ontology on the namespace's URL. For the three ontologies presented in the remaining part of this chapter, the following namespaces are used.

- **Software Ontology Model** — `http://www.ifi.unizh.ch/ddis/evoont/2007/02/som#`
- **Bug Ontology Model** — `http://www.ifi.unizh.ch/ddis/evoont/2007/02/bom#`
- **Version Ontology Model** — `http://www.ifi.unizh.ch/ddis/evoont/2007/02/vom#`

2.2 An Ontology Model to Represent Object-Oriented Software Source Code

There are two different approaches of creating a model of object oriented code. The first is a detailed model covering all operations, and every execution path can be traced back in the model. This kind of model stores most of the information but, therefore, is time and memory consuming even for the simplest tasks. This depth of information is needed, if the model is used for extensive testing like control flow tests. The second model nature only defines the structure of the code and the linkage between the structural entities. This results in smaller models (small filesize) allowing straightforward handling. The downside of this incomplexity is a loss of information and, hence, a reduced set of measuring and reasoning methods available. The model developed in this thesis uses the second approach due to the simplified handling and, as a major advantage of the OWL/RDF format, the door is still left open for further extensions to the model where needed.

2.2.1 The FAMIX Meta Model

The FAMIX meta model [Demeyer et al., 1999] was initially developed as an information exchange format for the FAMOOS (Framework-based Approach for Mastering Object-Oriented Software Evolution) project at the University of Berne. It describes the core structure of object oriented (OO) software without being fixed to one language (i.e. Java, C++ etc.). This general nature of the model makes it extensible for future languages.

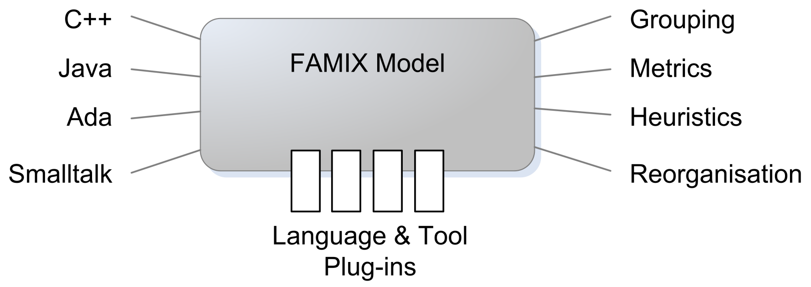


Figure 2.1: The FAMIX exchange model. Figure adapted from [Demeyer et al., 1999]

Figure 2.1 shows the original FAMIX meta model, designed to integrate different object oriented languages into one model, to be able to apply tools and measures in one manner. The FAMIX meta model also defines an interchange format to transport generated models through plain ASCII[1] streams. This characteristic is omitted completely in this work as it is done by the OWL format which has an implicit exchangeabilty due to its XML nature. The components of the FAMIX model are shown in Figure 2.2.

The model defines the entities, a typical OO language offers. The most essential ones are Classes, Methods and Attributes. FAMIX uses CDIF as interchange format which stands for *CASE Data Interchange Format* and, as the name states, it is primarily thought to be used by CASE[2] tools. The format itself is a plain text format, and both sides of a communication channel need to know the exact FAMIX specification to properly encode and decode the data. This makes FAMIX

[1]American Standard Code for Information Interchange
[2]Computer Aided Software Engineering

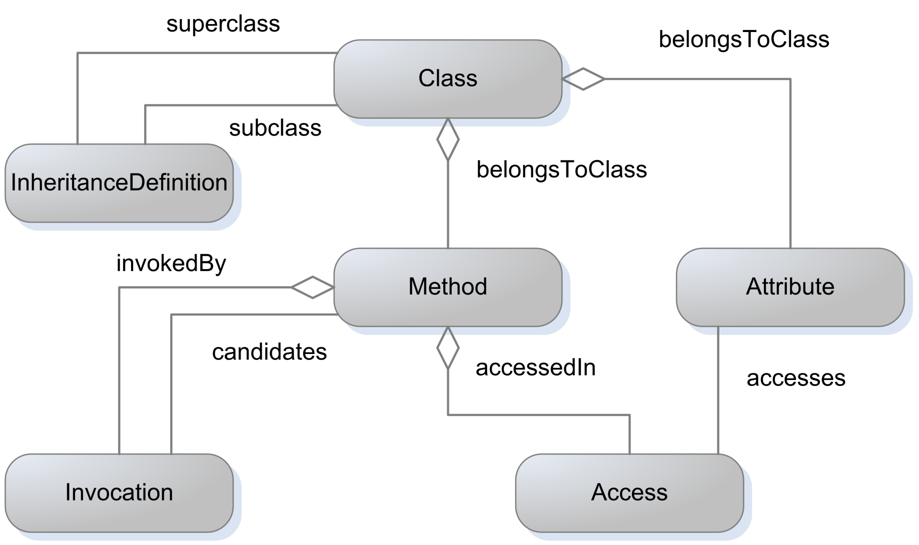


Figure 2.2: The FAMIX model for object-oriented code representation. Figure adapted from [Demeyer et al., 1999]

fragile to changes on the meta model specification itself, as a tool might no more understand or falsely decodes data streams. Another disadvantage of FAMIX and CDIF is their small domain of application and, therefore, not very wide spreading.

2.2.2 The Source Code Ontology

The approach of this thesis is to adopt the model part from FAMIX, extending it where reasonable, but using OWL as representation and exchange format. This has different advantages:

- **Interchangeability** OWL is a W3C standard based on XML and RDF. There is a myriad of tools available capable of manipulating, visualizing or reasoning with the data with OWL/RDF.
- **Non-ambiguity** A characteristic inherited from XML. With the ability to use worldwide unique namespaces (URI - Uniform Resource Identifier) a resource can be identified and linked to, from any other file or place without the risk of naming conflicts.
- **Machine-readability** OWL masters the balancing act between a human-readable and machine-processable format. An OWL file can be read and edited by a human without using any other tool than a simple text editor, but still be processed and understood[3] by a computer system.
- **Extendibility** An OWL ontology can simply be extended by attaching new ontologies. These ontologies can use the existing ontology and define new, more specific entities. Instances files, using the initial ontology are not affected by this extension and still will process properly.

These all are desirable attributes for a representation and exchange format of software meta data. Therefore, the FAMIX model was transformed to an OWL ontology.

[3]In the sense of being able to extract and categorize a piece of information

Ontology Hierarchy

The model consists of four core components. These are *Entity*, *Context*, *BehaviouralEntity* an *StructuralEntity*. As show in Figure 2.3 Entity is the superclass of all model elements.

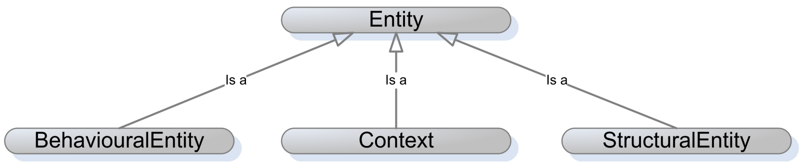


Figure 2.3: The abstract core components of the ontology

BehaviouralEntity is a superclass for all elements that describe an activity in the software. This is typically an encapsulated block of code solving a specific problem. In Java this is performed by a method, other languages know the concept of functions which is slightly different to methods.

StructuralEntity is a superclass for elements storing information. Typically these are all types of variables but also constants.

Context is the superclass for container entities. These entities are typically not involved to any information storage or activity but act as assortative concepts with their own name and namespace. A characteristic example for a context entity is a (OO-)class. It generally does not perform any actions or store information but acts as a container for methods and attributes which do the mentioned actions.

OWL Classes

Table 2.1 shows the OWL classes representing entities in the source code.

Name	**Description**
Class	a source class. This is a generic entity for either classes, abstract classes and interfaces
Directory	A filesystem directory.
File	A filesystem file. A file is a container for classes.
Package	A package serves the grouping and structural organisation of classes and defines an own namespace.
Attribute	An attribute is a variable with class scope.
FormalParameter	A formal parameter is a special kind of a local variable. It is the parameter passed to a method.
GlobalVariable	A global variable is a variable with global, system wide visibility (This concept does not exist in Java)
ImplicitVariable	An implicit variable does not have a name. It is accessed by special keywords whose value changes according to the context where they are called. Examples are *super* and *this*
LocalVariable	A variable whose lifetime (scope) is restricted to a method call.
Function	A function is an operation with a system wide scope. This concept does not exist in Java.
Method	A container for the execution of code. A methods needs to be defined inside a class and, therefore, can have no global scope.

Table 2.1: Classes of the source ontology model

In Table 2.2 the datatype properties of the source code ontology model are listed. Some of the listed properties are functional, expressing that there can be at most one value per individual for this property. The domain classes are the classes this property is applicable for.

Name	Type	FP	Domain	Description
comments	string		Entity	The comment written by a developer or tool providing further information about the functionality of this entity
accessControlQualifier	string	✓	Class, Method, Attribute, GlobalVariable, LocalVariable, Function	The visibility of this entity. This would refer to Java's *private*, *public*, *package* and *protected* keywords.
sourceAnchor	string	✓	Entity	Identifies the location in the source where the information is extracted. The exact format of the qualifier is dependent on the source of the information. Currently this property is not being used.
isInterface	boolean	✓	Class	true if this class is an interface.
isAbstract	boolean	✓	Class, Method	true if this entity is declared abstract and has to be implemented by a inheritance entity.
isStatic	boolean	✓	Entity	true if the entity is declared static and can be accessed without being bound to an object.
isFinal	boolean	✓	Entity	true if the entity is immutable. A final variable is also known as a constant.
isInit	boolean	✓	BehaviouralEntity	true if the entity symbols a global starting point for the software. Every language implements this concept in a slightly different way (Java: main method).
isConstructor	boolean	✓	Method	true if the method has the special functionality to be called during object generation. A Function entity cannot have this property because a function is not bound to an object.
name	string	✓	Entity	The name property stores the entity's name. This name may not be unique in the model.

uniqueName	string	✓	Entity	The uniqueName property stores the full name of an entity and must be unique within the model's namespace. This is typically the full package name followed by the name of the entity.
position	int	✓	FormalParameter	The position property identifies the order of FormalParameters. Every FormalParameter has a position in its declaring BehaviouralEntity signature. This position is necessary because a method or function can be declared muliple times (overloading) by only changing the order of its parameters.

Table 2.2: Datatype properties of the source ontology model; FP = Functional Property

Figure 2.4 shows a graphical representation (including classes and object properties) of the software ontology model.

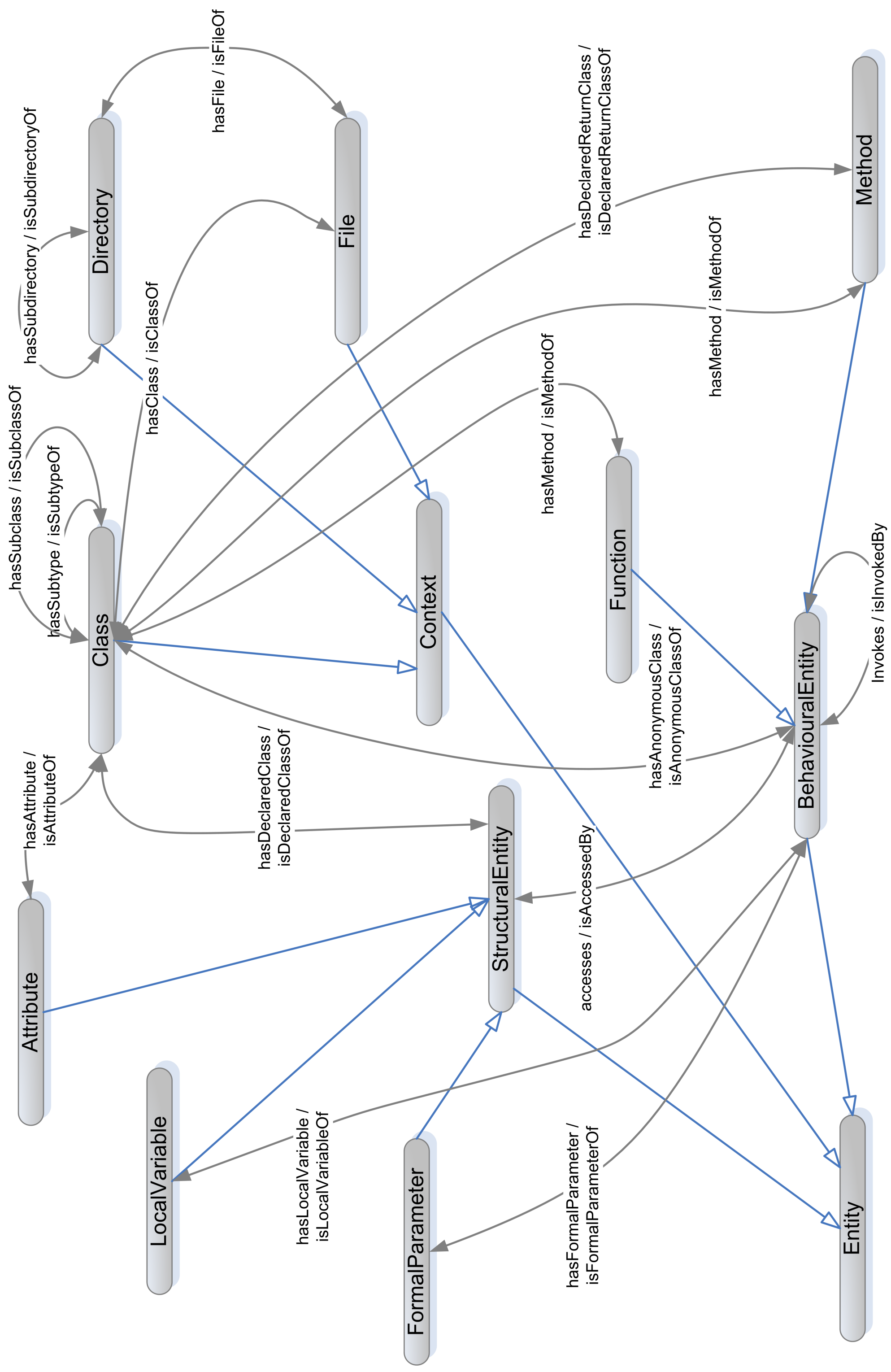

Directory
File
Method
Class
Context
Function
BehaviouralEntity
Attribute
StructuralEntity
LocalVariable
FormalParameter
Entity
hasFile / isFileOf
hasSubdirectory / isSubdirectoryOf
hasClass / isClassOf
hasDeclaredReturnClass / isDeclaredReturnClassOf
hasMethod / isMethodOf
hasMethod / isMethodOf
hasSubclass / isSubclassOf
hasSubtype / isSubtypeOf
hasAttribute / isAttributeOf
hasDeclaredClass / isDeclaredClassOf
hasAnonymousClass / isAnonymousClassOf
Invokes / isInvokedBy
accesses / isAccessedBy
hasLocalVariable / isLocalVariableOf
hasFormalParameter / isFormalParameterOf

Figure 2.4: The software ontology model

2.3 An Ontology Model to Represent Software Defects

Our next ontology model is to represent the entities of a defect tracking system, also called bug tracking system. In contrast to the source code ontology, there is no generic model of a bug tracking system. There may be some common entities such as the bug (or issue) itself and descriptions etc. Due to the lack of a common concept, we had to decide to use a specific implementation as archetype for the meta model. We chose the Bugzilla[4] system for several reasons.

- Bugzilla is a very wide spread bug tracking system. Most larger open source software uses it.
- It has a reasonable long history (about 10 years) of development making the software as well as the concept mature.
- Due to the early beginning of development, many other, commercial and non-commercial, defect tracking systems have adapted Bugzilla's concept of bug tracking. This will reduce the effort to port the meta model to other systems.
- Bugzilla offers a XML interface, easing the automated access.

A very central element of Bugzilla, as well as our meta model is an *Issue*. Whenever a user reports a bug or request a feature, an *Issue* element is created. Many other elements such as the reporter's name, the affected computer system or the severity are linked to an *Issue*. Table 2.3 shows an overview over all elements (classes) in the bug ontology model.

Name	Description
Activity	Describes an activity in the bug tracking system such as an assignment of an Issue to a Person or changes to the state of an Issue.
Attachment	To every bug report, several attachments can be added. This could be screenshots or error logs. This class is only intended to store meta data, not the attached file itself.
Comment	A comment describes a discussion concerning a bug. Developers or reporters can provide additional information about a bug by adding a comment.
Component	A component always belongs to a product. It corresponds to a certain, functional part of a software system.
ComputerSystem	This indicates the computing environment where the bug was found.
Issue	The Issue is the bug report itself. It is a central entity, linked to several other components like ComputerSystem or Component.
Milestone	A future version by which the bug is to be fixed. This is also known under the name Target or Target Milestone.
Person	A person is a generic entity for every human interacting with the bug tracking system. A person can hold different roles like developer, reporter, administrator etc.
Product	A product is a complete software product. It consists of different components and is usually the form a software product is released.

Table 2.3: Classes of the bug ontology model (bom)

[4]`http://www.bugzilla.org`

In Table 2.4 we present the datatype properties of the classes from Table 2.3.

Name	Type	FP	Domain	Description
keyword	string		Issue	An issue can be tagged with different keywords to ease the search for a specific bug.
status	string	✓	Issue	The state of an issue. Possible values are: ASSIGNED, CLOSED, NEW, REOPENED, RESOLVED, UNCONFIRMED and VERIFIED.
priority	string	✓	Issue	The priority of the issues' fixing. Values can be P1, P2, P3, P4 and P5. This is not a global priority but a help for an assignee to priorise his or her bugs.
lastModified	dateTime	✓	Issue	The date of the last change made to this bug.
fileName	string	✓	Attachment	The filename of the attachment.
platform	string	✓	ComputerSystem	A computer system's platform. For example: Macintosh or PC.
email	string		Person	The email address used to inform a person.
text	string	✓	Comment	The text part of a comment element.
what	string	✓	Activity	Describes the element affected by this activity.
date	dateTime	✓	Comment	The date when this comment was composed.
os	string	✓	ComputerSystem	The operating system represented by this ComputerSystem. For example: Windows Vista or Mac OS X.
name	string	✓	Product, Component, Milestone, Person, Attachment	The name of several components. Pretty self explaining.
name	string	✓	Product, Component, Milestone, Person, Attachment	The name of several components. Pretty self explaining.
dateOpened	dateTime	✓	Issue	The reporting date of this Issue.
description	string	✓	Issue	A description of the Issue, how it can be reproduced or under which conditions it usually occurs.
bugURL	string	✓	Issue	A track-back to the Bugzilla system where this Issue can be found.
number	int	✓	Issue, Comment	The number of an Issue describes its unique identifier. The number of a comment can be used to reconstruct the discussion by ordering the comments.

type	string	✓	Attachment	The filetype of this attachment. For example: gif, txt.
performed	string	✓	Activity	The date and time when this activity took place.
version	string	✓	Product, Version	Describes a specific version of this product or version.
target	dateTime	✓	Milestone	The due date of this milestone.
resolution	string	✓	Issue	There can be different reasons why a bug is closed and therefore inactive. Possible values: DUPLICATE, FIXED, INVALID, MOVED, WONTFIX, WORKSFORME, LATER and REMIND
removed	string	✓	Activity	The part that was removed during this activity.
added	string	✓	Activity	The part that was added during this activity.

Table 2.4: Datatype properties of the bug ontology model; FP = Functional Property

Figure 2.5 shows the connection between the classes of the bug ontology model (object properties).

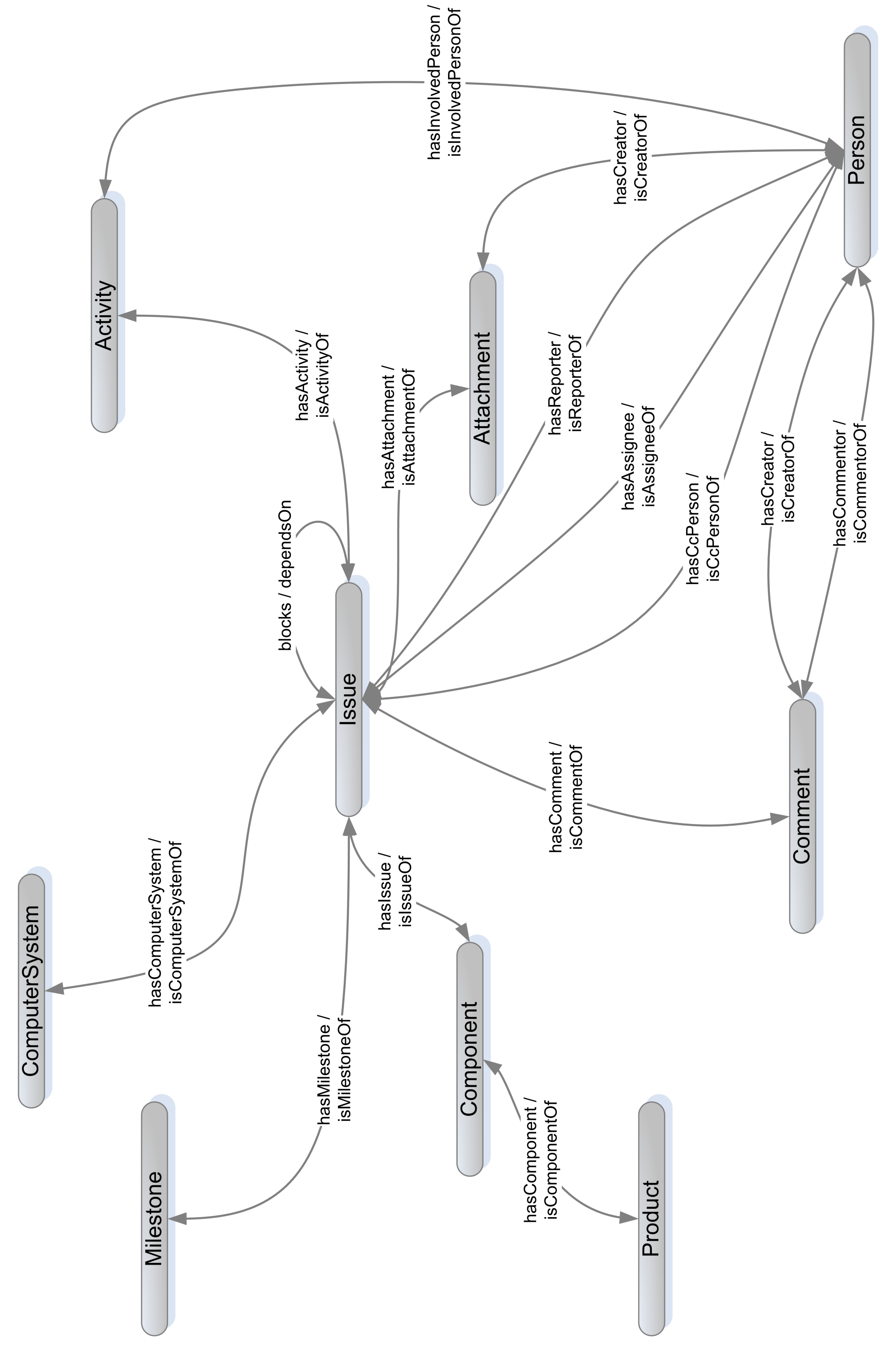


Figure 2.5: The bug ontology model

2.4 An Ontology Model to Represent Software Versions

Different versions of a software are maintained by a Version Control System (VCS). One of the most common concepts are Files, Releases and Revisions. A file is checked into such a VCS where it will be stored and given a revision number. On every change to this file, the revision number is increased to mark this new version. The older revision of this file is still available and can be rolled back to. The revisions mark the very own history of a file. The history and versions of the whole software project, i.e. a number of files, can be expressed by creating releases. A release combines a couple of file revisions to a version with a specific name. This name usually denotes the external version number of the software, but can also be a date string denoting the point of the reach of a milestone. We reduced the ontology model to the core of such a VCS. We use only the three classes `Revision`, `Release` and `File`. There are some more facets such as branches or transactions which are omitted in the actual ontology model because the used CVS parser does not provide this information, but might be part of a future extension. For this thesis, this reduced model contains all the information needed. Table 2.5 describes the classes of the version ontology model.

Name	Description
File	A file from the filesystem.
Revision	A revision denotes a version of a file.
Release	A release is a tag, multiple elements can be annotated with.

Table 2.5: Classes of the version ontology model

Table 2.6 describes the datatype properties in the version ontology model.

Name	Type	FP	Domain	Description
name	string	✓	Release, File	The name of a Release or File.
state	string	✓	Revision	The state describes the number of lines added or removed. An example is `Exp;  lines: +1 -0` .
creationTime	dateTime	✓	Revision	The date and time this revision was created.
fullPath	string	✓	File	The full path of this file. It contains all the folder names including the root folder.
author	string	✓	Revision	The name of the user who created this revision.
number	string	✓	Revision	The number of this revision.
commitMessage	string	✓	Revision	The message entered during the creation of this revision.

Table 2.6: Datatype properties of the version ontology model; FP = Functional Property

Figure 2.6 shows the connection between the classes of the version ontology model (object properties).

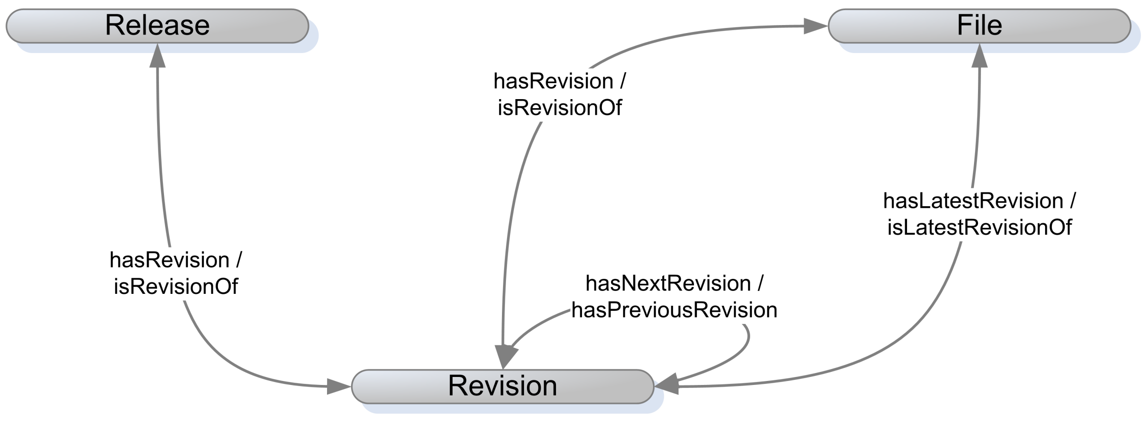

Release
File
hasRevision / isRevisionOf
hasRevision / isRevisionOf
hasLatestRevision / isLatestRevisionOf
hasNextRevision / hasPreviousRevision
Revision

Figure 2.6: The version ontology model

2.5 Interconnection of the Ontology Models

Every of the three presented meta models contains the entities of its respective repository, source code, versioning system and bug tracking system. These repositories, are usually interconnected to each other. The source code is checked into the versioning system and thereby may be referenced by a bug number. To unleash the full power of these three meta models, they should also be interconnected to each other to have the semantics between the repositories in the ontology models. Figure 2.7 shows the connection between each of the three models.

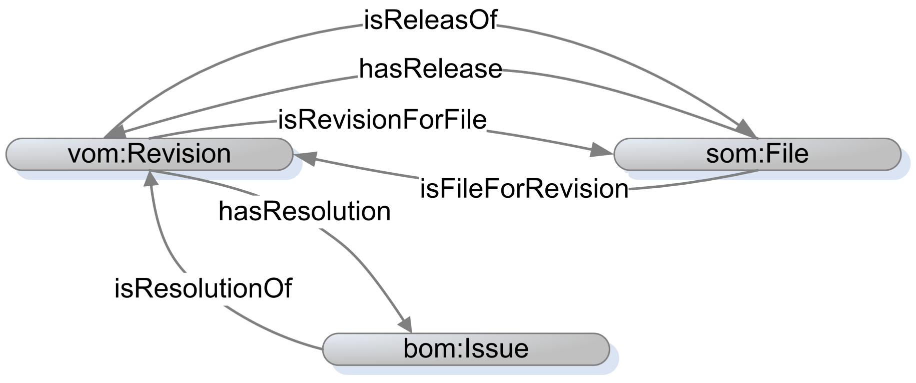


Figure 2.7: The connection between the three meta models

The *hasRelease* and *isReleaseOf* associations are specific to our approach. There is no such association between a file and a release in the versioning system or the source code. This connection was included, to later separate two releases of the software. Figure 2.8 shows an example with three individual files. The first contains the versioning data, the second and the third the source code data each for one release. In the case shown in the figure, there were no changes in the source file. So the two source files are linked to the same revision. If we want to list all files from a specific revision, we would first select the revisions of a release to next list the file for this revision. The problem with the case shown in Figure 2.8 is, that we would receive two files for this revision. This is not desired and would tamper the results. Therefore, we needed a way to uniquely assign a file from the source code to a release. With the *hasRelease* association we can determine to which release a file belongs, even if they are linked to the same revision.

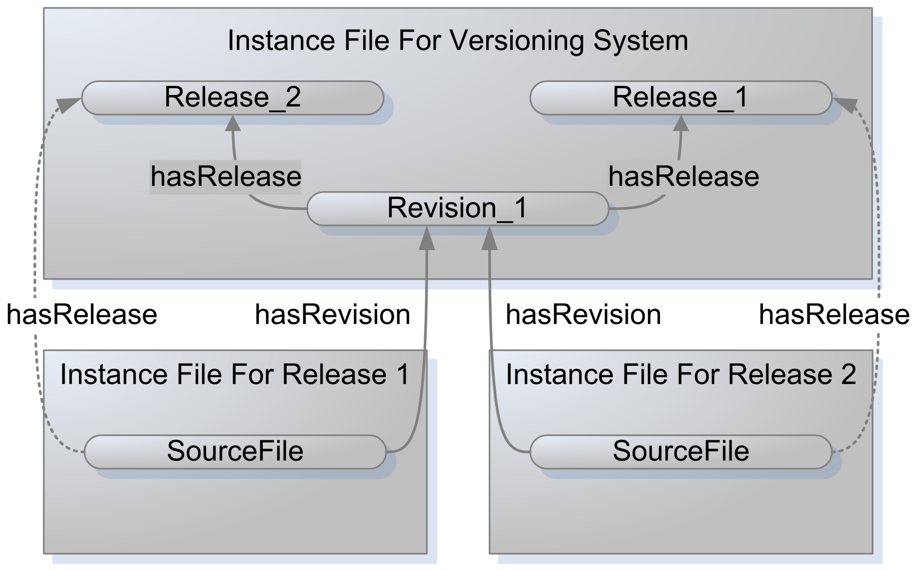


Figure 2.8: The need of the *hasRelease* association

2.5.1 Different Concepts of Filesystem Entities

There are two classes named *File*, once in the software ontology model and once in the version ontology model. Although these classes have the same name, they describe different concepts. A *File* in the version ontology model is only a virtual name that can have lots of versions (revisions). If a file is requested from the verisoning system, a user has to provide the file name and a revision number of the file. The versioning system will then query its internal database to retrieve a file denoting the state of the given revision number. Such a revision describes the *File* entity of the software ontology model. In other words, every version (revision) a versioning system *File* has, leads to a source code *File*.

3 The Plug-in

To collect the data about the evolution of a software project in its three characteristics – source code, defects and version control – these three repositories have to be transformed to OWL ontologies. As mentioned in Section 1.5, Eclipse is used as execution environment for these tools. Eclipse, as a integrated development environment, has direct access to two of the three repositories namely the source code and the versioning system.

3.1 Decision of Data Depth

If a change to the source code is checked into a version control system (VCS), a completely new version of the software is generated implicitly; meaning only one file changed, but as a nature of source code, this file is linked to numerous other files (to specify: classes) whose functionality might have also changed because they are now linked to the changed file. *Subversion*[1], as a representative of a VCS, exposes this fact by incrementing the revision numbers of all files even though only one file changed. These implicit versions of a software may not reflect a real change to the functionality, however, an outsider would assume that an increased version number bases on a changed functionality. To mark a specific version of a software project, a VCS usually features the possibility of creating releases. A release can be seen as a human annotation to an implicit, given by the revisions, version of the software. For example these annotations could base on the reach of a milestone. However, for the generation of OWL data we have to decide whether to create models from the implicit or explicit versions.

3.1.1 Implicit Version

Generating a new software instances file whenever a change[2] occurs is the most precise way to write down the history of a software project. By using this approach, no information about a change will be omitted. The downside of this method is the huge amount of data that is possibly generated without having much new information in it. As OWL, and XML in general, is a relatively disk space intensive format this would result in cumbersome and slowly processable models.

[1]`http://subversion.tigris.org`

[2]Change is used as generic term. Tools may name this a commit or check-in action

3.1.2 Explicit Version

As an alternative, the explicit versions can be considered solely. Compared to the foregoing approach this results in smaller sized models but also in a coarse-grained depth of information. Considering only the explicit versions of a software project does not omit the single changes made within the implicit versions, in fact it bundles them as one change made between two explicit versions. As such a version is explicitly created by a human, we can infer that a noticeable change has happened to sway someone to create this version. In this thesis the explicit version approach will be used. The advantage of the gained manageability of the models outweighs the loss of information. However, this is an implementation decision and can, if the need of finer grained data arises, be extended to satisfy the first suggestion.

3.2 Implementation

As a core goal of this thesis, a way of creating a source code model shall be implemented. This task should be done automatically, as an average software project has tenthousands of entities and associations, a manual creation of such a model would take an unreasonable long time. Therefore, we created a plug-in for the Eclipse platform to automate the data retrieval from the three examined repositories. Figure 3.1 shows the structure and of the components of this plug-in and its interaction. In the remaining part of this chapter, a brief description of the components and its functionality will be given.

3.2.1 User Interaction

At a first stage, the plug-in needs to interact with the user to collect necessary data for the parser. The Eclipse platform provides a well defined programming interface to create dialogs featuring the same look and feel of the rest of the development environment. There are different ways, a plug-in can extend the user interface. Eclipse already has the ability to export projects or part of project to different formats like .jar or .zip. These export functionalities are located in a central place of the IDE and is accessible through the context menu of the resource tree. This central exporter registry can be extended by a custom implemented exporter. A user will then have the ability to select this exporter from the list of all exporters. Our exporter is registered in the section *OWL* under the name *OWL Files*. If the plug-in is activated, so a user wants to export a project to OWL/RDF, a first dialog page shows up to ask the user for the project to export. After the selection of the project, a request to the Eclipse framework is sent to retrieve a complete list of the releases of the project. This list is presented to the user for the selection of the releases for which the instances files should be generated. A last dialog page asks the user for the location of the *Bugzilla* system of the project. The plug-in will need this to retrieve bug information if a referenced issue number is found within a commit message of a CVS revision. This URL can also be left blank, if the creation of the bug instances file is not desired. The last information needed by the plug-in is the location where the generated models (files) should be stored to. This location must be a folder somewhere in the filesystem. Optionally, the output format can be selected. Table 3.1 shows the available formats and their description.

The above described steps is the complete information needed to generate the models. The user can, as a final step, start the export activity.

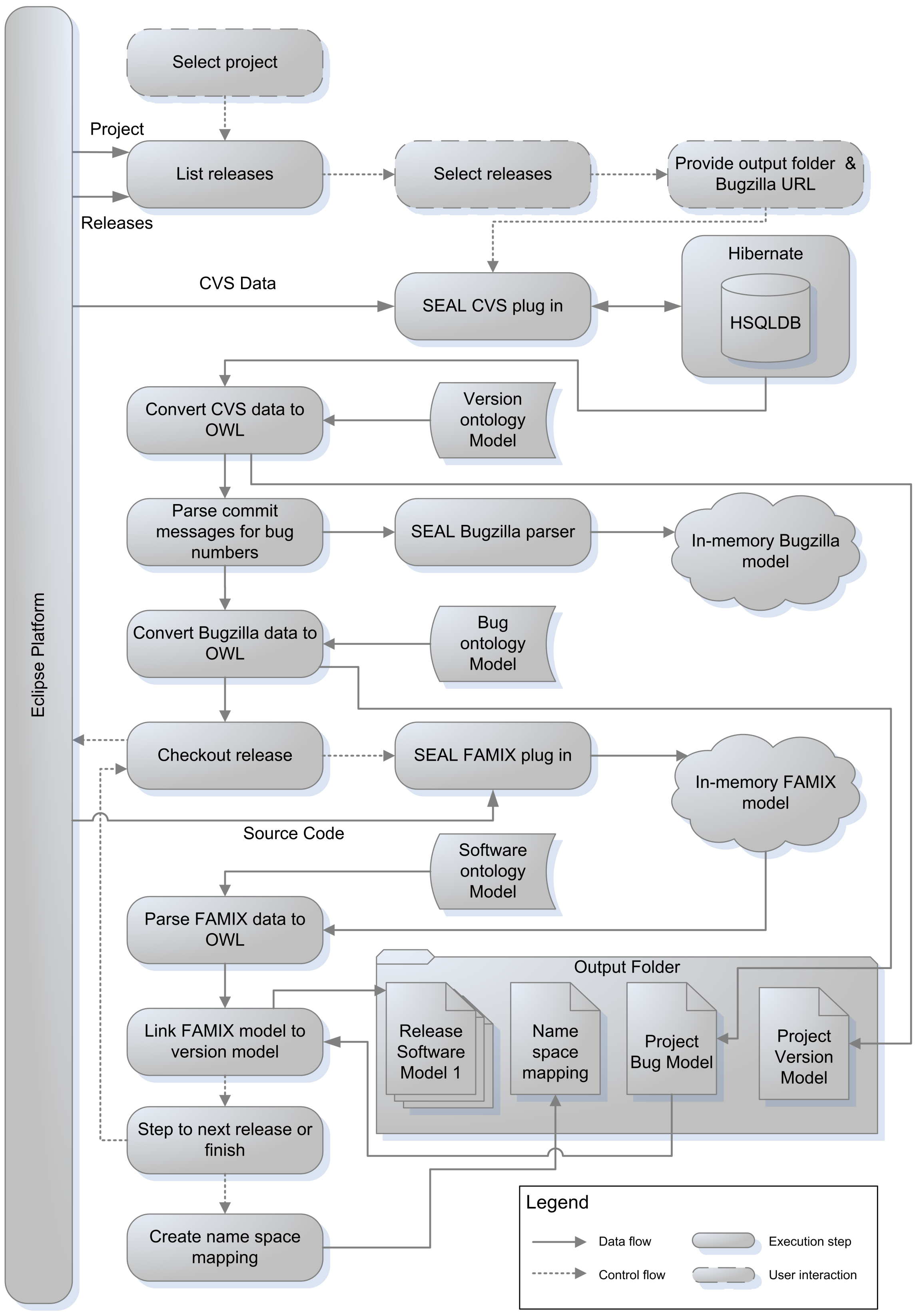


Figure 3.1: The components of the implemented plug-in

Name	Purpose
RDF/XML-ABBREV	Optimized for human readability. Default format for our plug-in.
RDF/XML	Optimized for automated processing. Not very well readable for humans.
N-Triples	Format tailored specially for OWL/RDF. Very compact and expressive, but no XML syntax.

Table 3.1: The different export formats

3.2.2 CVS Parser

To create a CVS in-memory model we use the CVS parser[Fischer et al., 2003] plug-in implemented by the S.E.A.L.[3] group at the University of Zurich. The CVS parser does not generate an in-memory model but directly stores the elements in a RDBMS[4]. This storage is maintained by Hibernate[5], a object-relational mapper. To prevent the plug-in from needing such a database system from the host system, we use HSQLDB[6], a small database management system featuring the generation of in-memory databases. The CVS parser writes its data into this in-memory database, our exporter queries this database to retrieve the entities. To create the OWL/RDF models, we first read the version meta model into memory by Jena. This is needed to be used as stencil to create the individuals stored in the model. Figure 3.2 shows the process of creating an individual for the model.

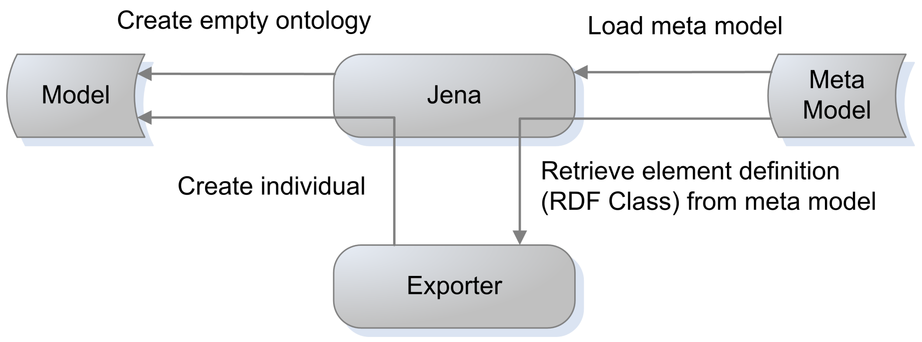


Figure 3.2: Typical steps when creating individuals

During the parsing activity of the plug-in, whenever a revision is encountered this revision's commit message is analyzed for a reference to a bug number. The next section will describe how bugs are identified, retrieved, parsed and exported.

3.2.3 Bugzilla Parser

In the section above, the process of parsing CVS data was described. The extraction of bug numbers from the commit messages of a revision is done by applying a regular expression. The regular expression checks for three types of bug references. The three reference patterns are:

[3]Software Evolution and Architecture Lab
[4]Relational DataBase Management System
[5]`http://www.hibernate.org`
[6]`http://www.hsqldb.org`

- # character followed by a series of numbers
- *bug* followed by a series of numbers
- *issue* followed by a series of numbers

These three patterns are combined in one regular expression presented below.

```
(?:bug|issue|\#)+\D*(\d+)
```

If the pattern matches a bug, the number is passed to the Bugzilla parser. Again we used a parser created by the S.E.A.L. group which parses the XML output of an issue from Bugzilla. The parser takes a bug number as argument, and creates a Java object representation of the bug (issue). The parser implementation features a *Model* component to have all the elements in one container. Everytime an issue is retrieved, all elements in the *Model* are checked for associations between the existing and the new issues. Associations can be the *blocks / depends on* property of an issue. When the Bugzilla and CVS parsing has finished, the creation of the source code models will be done.

3.2.4 Source Code Parser

To parse the source code and generate a first model of the software project, another parser developed by the S.E.A.L. group is used [Fischer et al., 2003]. This FAMIX parser also runs within the eclipse environment and makes excessive use of the provided functionality of eclipse. The core element used is the Abstract Syntax Tree (AST) eclipse uses as its own, internal model for the source code. This tree is traversed by the FAMIX parser creating a in-memory model based on Java objects. Figure 3.3 shows the UML notation of this model. This parser can only handle Java source code.

Figure 3.3 shows already some canges made to the original FAMIX meta model. For example the new concept *Context* is already implemented. The task of the plug-in is to export this in-memory model to RDF/OWL.

The SEAL FAMIX plug-in features a `Model` component, where all entities and associations are stored in. After the parser has finished its work, our exporter first iterates over the entities in the `Model`. Entities are the aforementioned components like `Class`, `Method` or `Attribute`. In a next step the associations are handled. Associations are constructs like inheritance definitions or interface implementations. The entities have to be defined first in the generated OWL model because an association can be only added if the *from* and the *to* entities are already defined. To keep the coupling low between the FAMIX model and the Software Ontology Model elements, we created a mapping file holding the associations between the RDF/OWL ontology model entities and the in-memory FAMIX model elements. With this mapping file, the meta model's entity names could be changed without the need of adapting the plug-in's code. When the export of a source code release has finished, the Eclipse framework is instructed to checkout the next release the user had selected. After this release is checked out, the exporter will be started again to create a model from the newly checked out version until all releases in the list are exported.

3.2.5 Persistent Storage

To persistently store the models to the filesystem the provided output folder and output format is used. As the models were generated only in-memory, and need to be stored to the filesystem. Jena provides writers to execute these steps. The filenames are generated using the patterns presented below.

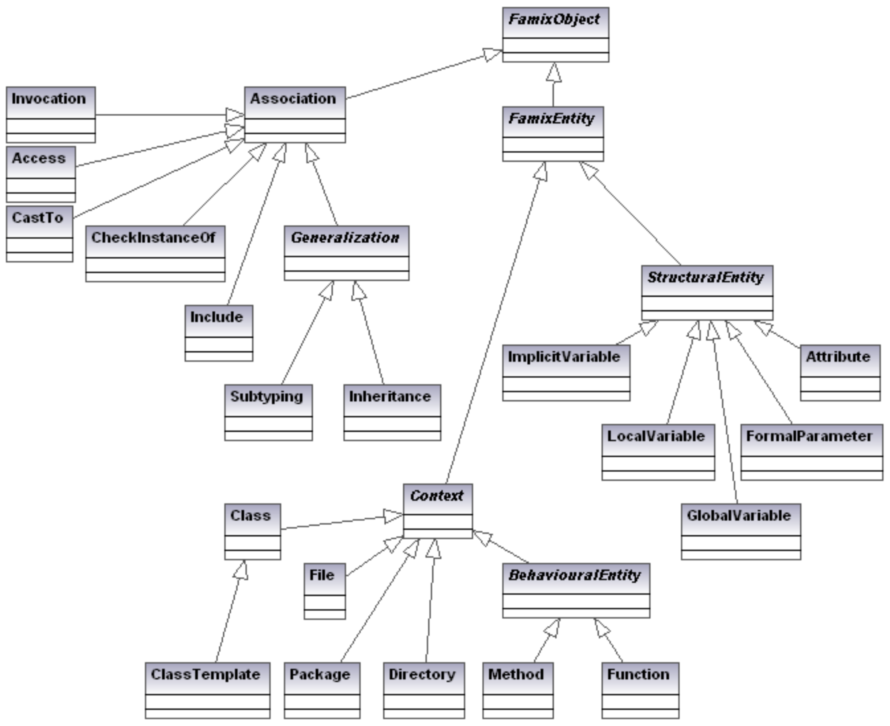


Figure 3.3: The structure of the in-memory model

- <**projectname**>_**vcs.**{**owl**|**n3**} for the version model of a project
- <**projectname**>_**bug.**{**owl**|**n3**} for the bug model of a project
- <**projectname**>_<**releasename**>**.**{**owl**|**n3**} for a release of a project

According to every file, a separate namespace is used to separate the different versions and projects from each other. Below, the pattern by which the namespaces are generated is shown.

- **http://www.ifi.unizh.ch/ddis/evoont/2007/02/som/parsed/**<**projectname**>_<**releasename**>**#** Every software ontology model can be uniquely identified by this namespace. <projectname> is the name of the project in Eclipse. <releasename> is the name of the exported release.
- **http://www.ifi.unizh.ch/ddis/evoont/2007/02/vom/parsed/**<**projectname**>**#** <projectname> is the name of the project in Eclipse.
- **http://www.ifi.unizh.ch/ddis/evoont/2007/02/bom/parsed/**<**projectname**>**#** <projectname> is the name of the project in Eclipse.

To be able to handle the files by just knowing their namespace, we created a mapping file to link a namespace to a filename. This mapping file uses the standard Java property file XML syntax. It is named *nsmapping.xml* and also stored in the output folder.

4

Evaluation

The effort made to create the application described in Chapter 3 serves as a base to query and reason about the evolution of a software project. In this section an evaluation framework will be presented as well as different facettes of mining data in the format shown in Chapter 2.

4.1 Pre-processing Steps

To mine data from the generated models, no further steps are required. However, for convenience we created a framework to automatize recurring tasks and some solutions for common problems.

4.1.1 Evaluation Framework

Whenever querying a model, more or less the same steps need to be performed. At first, the data files have to be read back into the Jena framework. Jena holds these loaded models in-memory, where they can be accessed through the Jena OntModel API. Multiple input files can be merged into one in-memory OntModel. With this ability, the meta model and the model can be merged, enabling a reasoner to know about the implicit associations and types. After the model is combined, a query can be formulated and executed on this knowledge base. The results can be programmatically consumed by iterating over the set of returned entries. The steps described are recurring, differing only in the query itself and maybe the set of merged models. To do these tasks, the framework should take the variable part of such a query execution as arguments. As a central component in this framework we created the class DataModel. This class offers a simple way of merging the created models by only specifying the folder where the respective files reside. The class abstracts the filesystem completely by taking only namespaces as arguments, but not filenames. A simple creation of a models could look as shown in Listing 4.1

```
DataModel dm = new DataModel("path/to/datafolder");
OntModel model = dm.getOntology(
        Namespace.SOM,
        Namespace.OO_DATA_NS_PREFIX+"<project_name>#"
        );
```

Listing 4.1: Usage of the class DataModel

To keep the functions encapsulated, a single query resides in its own class. This Query class holds the query itself, a description, the referenced namespaces and the level of language features needed to execute. This prevents an implementing class from being rewritten when a change to a query occurs. The query and its execution environment can be changed by swapping the Query object; The API stays the same. The mentioned namespaces are only the ones directly referenced by a query. With increasing complexity of a problem a single query is not powerful enough anymore. Therefore, a combination of multiple queries and computations is needed. We considered this fact with the framework component named *Measure*. A measure is a solution for a specific problem, can use multiple queries, does the effective processing of the results and provides a method of outputting or persistently storing the computed results. Figure 4.1 shows an overview over the components of the evaluation framework.

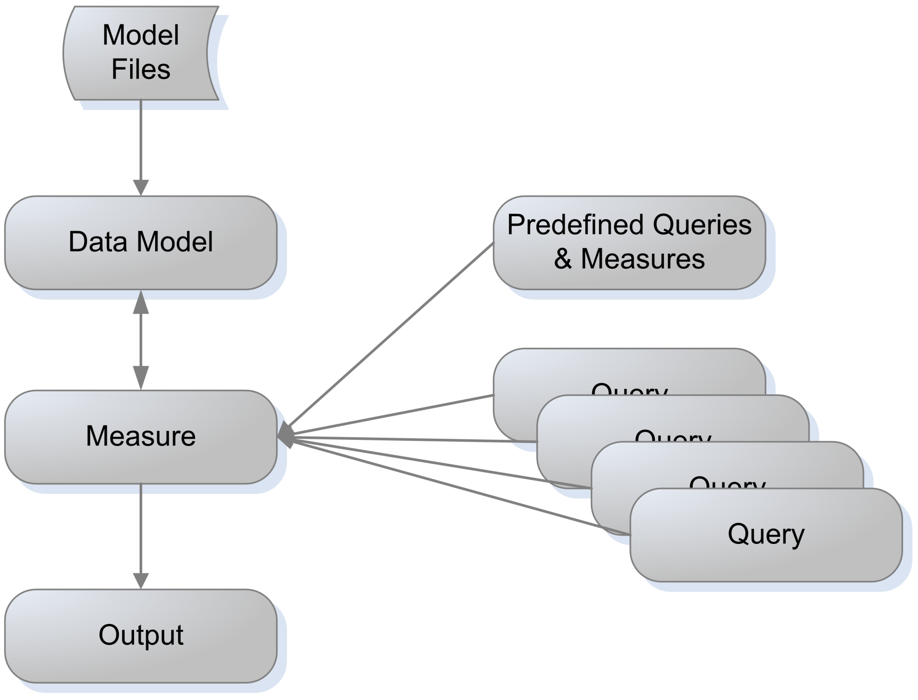


Figure 4.1: Overview over the evaluation framework

As shown above, our framework provides a best practice structure for the execution of queries and measures. Another task that should be computed by this framework are solutions for common problems. Therefore we implemented certain queries and measures being available as support for a framework client. In the succeeding part, some of these common problems and their solution are presented.

4.1.2 The Order of Releases

The releases in CVS are tags, annotating a revision. These tags only contain their name but no further information. To determine the date a release was created the name of the tag could be examined to extract a date. Often a release is named with a timestamp component like *I200605251200* what would refer to a release created on the 25th May, 2006 at 12 o'clock. However, this might work well, and as an advantage the effective date of a release can be determined. The downside of this method is the risk of losing the date information of releases not properly named with a date. The naming of releases is subject to the user creating it and, at the utmost, there may be only a convention forcing a developer to include a date in the release name. This can never be a guarantee for a release being traceable to a date. The entity in CVS having date information available is a revision. Every time a revision is created, the actual date and time is stored as a timestamp value. With the date information of the revisions, the release's date can be determined. The date of a release can be defined as the date of the latest change made to the software project before the release tag was set. This would refer to the latest date of a revision belonging to a release. This approach allows a reliable determination of the release date but might not be the exact date when the release was generated. Assuming no change has happened to a project for two weeks and a release is generated, the described method of determining the release date would return the date of last change, say two weeks before the effective release date. This is a bearable limitation because the two dates refer to the same state of a project but may mislead a user by receiving a date and a contradicting name. To compute the date, a Measure was created. At first a simple SPARQL query was written to retrieve a list of revisions and corresponding releases. The query is shown in listing 4.2.

```
PREFIX vom: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/vom#>
SELECT ?releaseName ?revisionDate
WHERE {
        ?release a  vom:Release .
        ?release vom:name ?releaseName .
        ?revision vom:hasRelease ?release .
        ?revision vom:creationTime ?revisionDate .
}
```

Listing 4.2: Query to retrieve a list of revisions and its releases

The query is a join of releases and their respective revisions. The retrieved results are shown in table 4.1.

releaseName	revisionDate
...	...
v20030825	2003-03-10T20:25:35Z
v20030825	2003-03-10T20:25:35Z
v20030825	2001-05-17T12:21:39Z
v20030825	2003-03-10T20:25:35Z
v0_119	2001-05-17T12:21:39Z
v0_119	2001-05-28T16:47:56Z
v0_119	2001-05-28T16:47:56Z
v0_119	2001-05-30T08:38:55Z
v0_119	2001-05-28T16:47:56Z
...	...

Table 4.1: Parts from the result set of Query 4.2

The results are read one-by-one into a simple Java HashMap whose key the release name, and the value the revision date is. Before a release / date pair is inserted, we check if this release is already stored in the map. If not, it will be inserted immediately else, we check if the existing date is before the new date. If so, the new value will replace the old value. At last the map needs to be sorted by its values. For this task, we created a simple sorter class to allow ascending and descending sorting. After we evaluated this method by using the release data from Eclipse's compare plug-in, a flaw showed up if a release does not contain any files at all. This may lead to a random order within these empty releases.

4.1.3 An Extension to ARQ for Counting

SPARQL does not support aggregate functions. The Jena framework provides a language called ARQ[1], a superset of the SPARQL language. The ARQ language allows the definition of extensions implemented in Java. Such an extension is a Java class that will be, after registering, called every time the class name appears in a query. ARQ provides a special namespace to access this functionality. If a query wants to use a custom extension, a special prefix for ARQ, similar to **PREFIX** agg: <java:[name.of.package].> is needed to give the query engine a hint where to find the extensions. Extensions can be written in triple form, and, therefore, are also called *magic properties*, to better suit the syntax of a SPARQL query. A line in a query like ?number agg:count ?input . will instantiate the class named *count* in the package defined by the *agg* prefix. This class receives the elements of ?input as argument and has access to the variable ?number to store the result in. The effective counting is done by rather a grouping than a counting algorithm. Because ARQ will provide all possible solutions of a query, the count class has to group all the same elements in the ?input variable and, later count the group members and store the result in the ?number variable. Tables 4.2 and 4.3 visualize the counting process.

Class	Method
java.lang.Object	clone()
java.lang.Object	equals(Object obj)
java.lang.Object	finalize()
java.lang.Object	getClass()
java.lang.String	charAt(int index)
java.lang.String	codePointAt(int index)
java.lang.String	codePointBefore(int index)

Table 4.2: Counting example output

Table 4.2 shows a first stage of the counting algorithm. It will take the result set and will group same elements (i.e. `java.lang.String` and `java.lang.Object`) and count the number of occurrences. The next step is to take the number of members of a group and the first member of this group into the final result set. The entries for the rest of the group members are eliminated. Table 4.3 shows the final result set.

Class	Method	Count
java.lang.Object	clone()	4
java.lang.String	charAt(int index)	3

Table 4.3: Counting entities in a SPARQL query

This is a generic method of implementing a count function. It works for every query and every kind of data. But there are some limitations: An element cannot be counted solely, in fact, only the associations are considered not the elements itself. This is different to the aggregate functions known from other query languages like SQL. The SQL counterpart of Query 4.3 would return

[1]`http://jena.sourceforge.net/ARQ`

the number of elements. The here implemented count extension will not, unless an association is included as shown in Query 4.4.

```
PREFIX agg: <[name.of.package].>
PREFIX vcs: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/vom#>
SELECT ?elementCount
WHERE {
        ?element a  vcs:Release .
        ?elementCount agg:count ?element .
}
```

Listing 4.3: Query that won't retrieve the number of elements due to limitations of the implmentation

```
PREFIX agg: <[name.of.package].>
PREFIX vcs: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/vom#>
SELECT ?elementCount
WHERE {
        ?element vcs:hasRelease ?release .
        ?elementCount agg:count ?element .
}
```

Listing 4.4: Working query to retrieve the number of elements

Queries 4.3 and 4.4 show the limit of the implementation. In fact only the associations can be counted. This has the effect that the topmost entity cannot be counted because there is no association to it.

4.2 Metrics

Metrics describe a way of making software source code measurable and therefore comparable. On this base, best practice as well as design flaws can be defined and searched for. The latter, design flaws, can be considered as harmful to software because they possibly lead to defects and endanger the success of the whole project. The name *metrics* inheres the fact, a lot of counting will be necessary. Due to the open world assumption (OWA) of OWL, SPARQL does not offer the ability of aggregate functions. An expression saying "there are 100 birds" might be correct, if we query an ontology containing 100 individuals of the RDF type Bird, but, as a matter of fact, it exists a lot more birds than just 100. Due to the common nature of OWL this may be a comprehensible reason for not being able to count entities. In our specific area of application we can be more expressive by closing down the open world. There are two preconditions allowing to perform this step.

- As described in Chapter 3 we know, the complete source code of a release will be transformed to an OWL/RDF model stored in exactly one file. By reading in this file and querying it, we can imply that if an entity or relation is not present, it also is not present in the source code.
- Our scope of examination is a single software project or release of it. This scope has to be kept in mind. For example if we want to express the number of invokers of a method, we can count the *invokes* relation to this method. As public methods are exposed to the outworld, we don't know the exact number of invokers because many other software projects can reference and invoke the examined method. In this case the counted number of invokers may

be incorrect and the effective number is unknown. But at least we can calculate the exact number in our scope. Saying “The method toString() is invoked 100 times” may be incorrect but if we extend it to “The method toString() is invoked 100 times in the org.eclipse.compare plug-in” the expression is correct again. In the remainder of this thesis, the scope won’t be mentioned every time but is assumed implicitly.

Below, some selected metrics defined by [Lanza and Marinescu, 2006] will be presented. The same abbreviations will be used like proposed by [Lanza and Marinescu, 2006]. It is not the goal of these metrics to fix a border between good code and bad code. The level when a metric value becomes harmful can vary from case to case. Therefore a metric can only indicate entities in code which may need to be redesigned. The effective decision of the need of such a redesign has still to be made by a human.

4.2.1 HIT — Height of Inheritance Tree

Years ago, the use of inheritance was proclaimed as one of the core advantages of object oriented programming. Indisputable inheritance enables a programmer to encapsulate functionality and derive similar or more specialized versions of it as subclasses. But, inheritance can also cause confusion because a class can use functionality not implemented by itself, but using it from superclasses. This may be manageable if operations from a super- or a supersuperclass is used. Whenever the inheritance tree grows larger, it may be no longer clear which class is responsible for which kind of functionality (for a developer creating a subclass at the lowermost end of the tree). So, the number of nodes on the way to the uppermost superclass of a class can be an indicator for confusing, and therefore error-prone code. To retrieve the inheritance tree in the code ontlogy, the object property *hasSublcass* or its inverse *isSubclassOf* has to be found. Like described in Chapter 2 these properties are modeled as transitive properties. A general definition of transitivity is shown in the definition below.

$$P \subseteq O \times O \tag{4.1}$$

$$\forall x, y, z \in O : xPy \wedge yPz \Rightarrow xPz \tag{4.2}$$

Query 4.5 is a first stage for the HIT metric. It retrieves the classes and all of its superclasses. Table 4.4 shows parts of the results of Query 4.5. The more superclasses a class has, the higher the inheritance tree is. To better express this fact we counted the number of superclasses instead of listing them. Query 4.6 makes use of the implemented functions to count associations.

```
PREFIX agg: <[name.of.package].>
PREFIX som: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/som#>
SELECT ?subclass ?superclass
WHERE {
       ?subclass som:hasSuperclass ?superclass .
}
```

Listing 4.5: Query to list all superclasses of all classes in a project

subclass	superclass
...	...
AddFromHistoryAction	BaseCompareAction
AddFromHistoryAction	Object
AddFromHistoryAction$1	WorkspaceModifyOperation
AddFromHistoryDialog	Dialog
AddFromHistoryDialog	ResizableDialog
...	...

Table 4.4: Parts from the result set of Query 4.5

```
PREFIX agg: <ddis.evoont.evaluation.extensions.>
PREFIX som: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/som#>
SELECT ?subclass ?HIT
WHERE {
        ?subclass som:hasSuperclass ?superclass .
        ?HIT agg:countChilds ?subclass .
}
ORDER BY desc(?HIT) .
```

Listing 4.6: Query to count all the superclasses of a class

Table 4.5 shows the height of the *org.eclipse.compare* plug-in inheritance trees with 4 as the highest tree for class *class_name*. This seems to be a quite fair height of the tree. One has to consider the topmost element of the tree is the generic *java.lang.Object* class. As this class and its functionality is best-known to a Java developer, the tree has even to be considered as the queried value minus one. So an inheritance path of three classes can viewed as straightforward for understanding as well as development.

subclass	HIT
...	...
ResourceCompareInputFilteredBufferedResourceNode	4
ResourceCompareInputMyDiffNode	4
AddFromHistoryDialog$4	3
CompareEditorInput$2	3
CompareEditorInput$3	3
CompareEditorInput$4	3
...	...

Table 4.5: Parts from the result set of Query 4.6

4.2.2 NOPA — Number of Public Attributes

A class can define different types of attributes. As a private attribute serves internal purposes, a public attribute can be considered as part of the API of this class. There exist two different kinds of an access to such a public attribute. First, there is an attribute intended for read access. An object may exposes internal states to an outside entity. The other kind is an object that is configurable through its public attributes. Both ways are straightforward but are prone to defects due to changes on the API. To be able to identify these classes we need to find their public attributes. We can hypothesize, that the most vulnerable class is the one with the most of such public attributes. Therefore the list of classes with public attributes can be ordered by the occurrence of these attributes. Query 4.7 lists the classes name and its NOPA. The result set is ordered by the number of public attributes.

```
PREFIX som: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/som#>
PREFIX agg: <java:ddis.evoont.evaluation.extension.>
SELECT ?class ?NOPA
WHERE {
        ?class som:hasAttribute ?attribute .
        ?attribute som:accessControlQualifier ?acc .
        FILTER(?acc = "public") .
        ?NOPA agg:countChilds ?class .
}
ORDER BY DESC(?NOPA)
```

Listing 4.7: Query to find all public attributes and its declaring class

The returned result set is shown in table 4.6.

class	NOPA
PatchMessages	62
CompareMessages	26
ICompareContextIds	19
Differencer	10
ComparePreferencePage	10
...	...

Table 4.6: Parts from the result set of Query 4.7

When inspecting the resulting classes, it arises that most of the public attributes are final. Final fields are often used as enumeration types to identify a symbolic name, whose value is unimportant. Therefore, classes often define a list of public, final and often static attributes with an int type and a value with only internal relevance to distinguish between the attributes.

```
PREFIX som: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/som#>
PREFIX agg: <java:ddis.evoont.evaluation.extensions.>
SELECT ?class ?NOPA
WHERE {
        ?class som:hasAttribute ?attribute .
        ?attribute som:accessControlQualifier ?acc .
        ?attribute som:isFinal ?final .
        ?attribute som:isStatic ?static .
        FILTER(?acc = "public") .
        FILTER(?final = false) .
        FILTER(?static = false) .
        ?NOPA agg:countChilds ?class .
}
ORDER BY DESC(?NOPA)
```

Listing 4.8: Query to find all public non-static and non-final attributes and its declaring class

This usage of public attributes cannot be considered harmful because their value is irrelevant (and immutable) and has no effect to the internals of the class. To hide these attributes from the NOPA metric, Query 4.7 can be modified to Query 4.8. The result set of Query 4.8 is empty, meaning the org.eclipse.compare plug-in does not contain any public, non-final and non-static attributes. The code can, therefore, be considered as well encapsulated regarding to public attributes.

4.2.3 CM — Changing Methods

When changing the functionality of a public method, in most cases this will have an impact to the invoker of this method. For instance a sorting method that takes a list of strings as input and returns a sorted list. If a change to that method occurs, a caller might not be able to process the returned results. A change can be internal only, say there is no change to the API (the methods signature) but only to the semantics of the arguments or returned data. The mentioned method's sorting algorithm may be changed to optimize the sorting speed. As a side effect, the sorted list is no more sorted ascending but descending. Such a change can lead to defects of the invoking methods and classes because they expect the result to be sorted ascending. A method that is invoked by many other methods has a higher risk of causing a defect because a developer might not remember every invoking method. Query 4.9 retrieves a list of method and the number of invocations.

```
PREFIX som: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/som#>
PREFIX agg: <java:ddis.evoont.evaluation.extensions.>
SELECT ?class ?method ?CM
WHERE {
        ?method som:isInvokedBy ?invoker .
        ?class som:hasMethod ?method .
        ?CM agg:countChilds ?method .
        FILTER regex(?className, "compare", "i") .
}
ORDER BY DESC(?numberOfInvokers)
```

Listing 4.9: Query to retrieve the number of invokers (changing methods)

The line **FILTER regex**(?className, ”compare”, ”i”) in Query 4.9 ensures, that only methods are considered from inside the project. Without this line, methods of the Java API would also be listed in the result set. Table 4.7 shows the topmost results of Query 4.9.

class	method	CM
CompareUIPlugin	getDefault()	30
Utilities	getString(Ljava.util.ResourceBundle,Ljava.lang.String)	26
Utilities	getString(Ljava.lang.String)	24
ICompareInput	getLeft()	16
ICompareInput	data:getRight()	15
...	...	...

Table 4.7: Parts from the result set of Query 4.9

This result states that the method `getDefault()` in the class `CompareUIPlugin` is accessed from 30 different places in the project. A change to the method `getDefault()` may have effects on those 30 locations in code. With this information a developer knows to be extra cautious if the functionality of this method is modified.

4.2.4 NOP — Number of Parameters

To influence the way a method executes its code, parameters can be passed. This is a core functionality of almost every programming language. To keep a method comprehensive it should not be responsible for too much of functionality. It is a better style to spread the programs functionality to a number of methods than just a few large methods. There are different approaches to determine a methods functional weight. Query 4.10 retrieves the number of parameters a method takes. This can also be an indicator for a complex functionality, if a method needs a lot of parameters. Table 4.8 shows the five topmost results of Query 4.10.

```
PREFIX som: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/som#>
PREFIX agg: <java:ddis.evoont.evaluation.extensions.>
SELECT ?class ?method ?CM
WHERE {
        ?method oomodel:isInvokedBy ?invoker .
        ?class oomodel:hasMethod ?method .
        ?method oomodel:hasParameter ?parameter .
        ?NOP agg:countChilds ?method .
}
ORDER BY DESC(?NOP)
```

Listing 4.10: Query to retrieve the length of the parameter list (NOP)

class	method	NOP
TextMergeViewerDiff	init(...)	14
TextStreamMerger	merge(...)	9
IStreamMerger	merge(...)	9
TextMergeViewer	simpleTokenDiff(...)	7
RangeDifference	init(...)	7
...	...	...

Table 4.8: Parts from the result set of Query 4.10

4.2.5 NOR — Number of Revisions

The metrics above were queried from the code ontology model only. The version and bug ontology model contain some interesting metrics as well. For instance we can determine the activity a file underlies. The activity is reflected by the number of revisions a file has. A revision is created every time a developer made a change to the code and checked it in to the version control system, so a file with a lot of revisions is either old, or underlies an ongoing evolution. The evolution itself can be reactive or proactive. A reactive evolution is an activity due to a reported defect (bug), proactive describes a real evolution in the sense of extending the functionality or adapting the software to new requirements. Our first step is Query 4.11, retrieving the number of revisions per class.

```
PREFIX vom: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/vom#>
PREFIX agg: <java:ddis.evoont.evaluation.extensions.>
SELECT ?file ?NOR
WHERE {
        ?file vom:hasRevision ?revision .
        ?NOR agg:countChilds ?file.
}
ORDER BY DESC(?NOR)
```

Listing 4.11: Query to retrieve number of revisions per file (NOR)

The result retrieved from Query 4.11 are shown in table 4.9. The results have been filtered to show only source code files (no images, licenses etc.).

file	NOR
TextMergeViewer.java	213
CompareEditorInput.java	88
CompareUIPlugin.java	70
ContentMergeViewer.java	69
EditionSelectionDialog.java	66
Utilities.java	64
CompareEditor.java	57
Patcher.java	51
ComparePreferencePage.java	50
DiffTreeViewer.java	47
StructureDiffViewer.java	45
PatchWizard.java	44
CompareConfiguration.java	41
...	...

Table 4.9: Parts from the result set of Query 4.11

Table 4.8 shows the ten most active classes (files) in the org.eclipse.compare project. An obvious reason for the differences between the number of revisions could be the different age of the files. To eliminate this possibility the date of the first revision can be examined. Most of the file's first revision is dated to the 2nd of May 2001. Due to the same age of the files the difference in the number of revisions must therefore be caused by a higher development activity. Our next step to determine the reason for the activity is to detect if it is caused by bug fixes or "real" development.

4.2.6 NOB — Number of Bugs

The next metric we can extract from the version and bug ontology model is the number of bugs a file is linked to. This information can help to detect vulnerable files. There might be different reasons why a file is linked to a large number of bugs.

- The file takes a central role in the software project. Because of its importance, bugs are detected sooner than in less important files.

- The file is responsible for a very complex functionality that is error-prone by nature

- The code in the file is confusing, causing a developer to modify things he does not understand

In all of the three mentioned causes, a refactoring could be advisable, to break down the complexity and distribute it onto multiple, and therefor less complex structures. Query 4.12 retrieves the number of bugs, the base of the above mentioned thoughts.

```
PREFIX vom: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/vom#>
PREFIX bom: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/bom#>
PREFIX agg: <java:extensions.>
SELECT ?file ?NOB
WHERE {
        ?bug bom:hasResolution ?revision .
        ?file vom:hasRevision ?revision .
        ?NOB agg:countChilds ?file .
}
ORDER BY DESC(?NOB)
```

Listing 4.12: Query to retrieve number of bugs per file (NOB)

In Table 4.10 parts from the result set of Query 4.12 is presented.

file	NOB
TextMergeViewer.java	36
CompareEditor.java	16
Patcher.java	15
PreviewPatchPage.java	13
ResourceCompareInput.java	12
DiffTreeViewer.java	10
Utilities.java	10
CompareUIPlugin.java	9
StructureDiffViewer.java	9
CompareViewerPane.java	6
...	...

Table 4.10: Parts from the result set of Query 4.12

4.2.7 Bug and Evolution Densities

To determine if a class' evolution is based on functional extension or reactive bug fixing, we set the number of bugs and the number of revisions in relation. Therefore, we extracted the ratio of the bug fixing activity over all activities. This ratio describes the error density (ERD) of a file and can be defined as in Equation 4.3.

$$ERD_{file} = \frac{NOB}{NOR} \tag{4.3}$$

The inverse of the above error density is the ratio of functional extension. It is the evolution density (EVD) representing the performed changes that were no bug fixes.

$$EVD_{file} = 1 - \frac{NOB_{file}}{NOR_{file}} \text{ or } EVD_{file} = 1 - ERD_{file} \tag{4.4}$$

When combining the results sets 4.9 and 4.10 the ERD and EVD metric can be determined. Table 4.11 shows this combination. With the calculation of the NOB metric we identified the files `TextMergeViewer.java` (36), `CompareEditor.java` (16) and `Patcher.java` (15) as the ones with the most bugs reported. When analyzing the results in Table 4.11 the mentioned files are no more in the upper ranks. Now the file `StatusLineContributionItem.java` has to be considered as the most bug intensive file although there are only three bugs, but there are also only three revisions. In the former ERD metric, the three files `TextMergeViewer.java`, `CompareEditor.java` and `Patcher.java` had the highest ERD value but now, are only the 24th, 12th and 11th most bug-prone files in the project.

#	file	NOR	NOB	EVD	ERD
1	StatusLineContributionItem.java	3	3	0.000	1.000
2	CompareNavigator.java	3	2	0.333	0.667
3	IResourceProvider.java	4	2	0.500	0.500
4	DifferencesIterator.java	10	5	0.500	0.500
5	PatchProjectDiffNode.java	2	1	0.500	0.500
6	IStructureCreator.java	11	4	0.636	0.364
7	PreviewPatchPage.java	37	13	0.649	0.351
8	UnmatchedHunkTypedElement.java	3	1	0.667	0.333
9	WorkerJob.java	3	1	0.667	0.333
10	ResourceCompareInput.java	38	12	0.684	0.316
11	Patcher.java	51	15	0.706	0.294
12	CompareEditor.java	57	16	0.719	0.281
13	RangeDifference.java	11	3	0.727	0.273
14	ResizableDialog.java	11	3	0.727	0.273
15	WorkspacePatcher.java	11	3	0.727	0.273
16	CompareViewerPane.java	23	6	0.739	0.261
17	CheckboxDiffTreeViewer.java	4	1	0.750	0.250
18	IRangeComparator.java	8	2	0.750	0.250
19	DiffNode.java	23	5	0.783	0.217
20	DiffTreeViewer.java	47	10	0.787	0.213
21	ColorEditor.java	5	1	0.800	0.200
22	StructureDiffViewer.java	45	9	0.800	0.200
23	MergeSourceViewer.java	31	6	0.806	0.194
24	TextMergeViewer.java	213	36	0.831	0.169
...	...	...	...	...	...

Table 4.11: The bug density of the org.eclipse.compare plug-in

To visualize the data from table 4.11 Mathlab was used. Figure 4.2 shows some interresting facts. First, only about 25% of the source files contain bugs at all. Nearly 75% of the code is (measured by the reported and mentioned bugs) free of defects. Next, the concentration of the errors is exponential decreasing thus only a very few files have a high concentration of bugs.

The histogram of the number of bugs is shown in figure 4.3. The ERD metric is visualized by creating ten classes (x axis) and the number of files having an ERD value in these categories (y axis).

To calculate the ratio of bugfixes over all activity in the project (TEVD — Total EVD, TERD — Total ERD), the single values of the ERD and EVD can be summarized as in Equation 4.5 and 4.6.

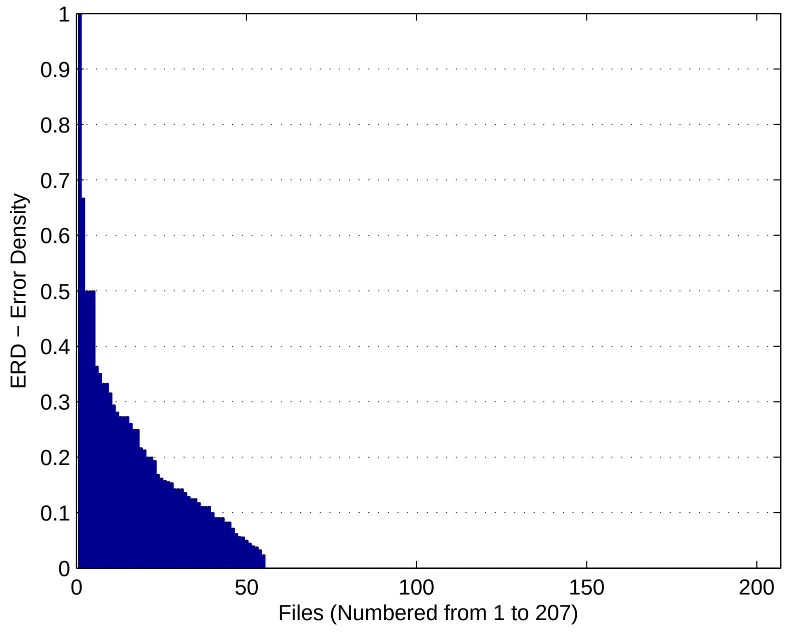


Figure 4.2: The bug density of the org.eclipse.compare files

$$TEVD = \frac{\sum_{i=1}^{n} EVD_i}{n} \tag{4.5}$$

$$TERD = \frac{\sum_{i=1}^{n} ERD_i}{n} \text{ or } TERD = 1 - TEVD \tag{4.6}$$

For the results from table 4.11 the value for TERD is 0.054. This value expresses that 5.4% of all activity in the project is due to bug fixing activity, and 94.6% for functional extension. [Boehm, 1981] made extensive research on the percental spreading of software engineering task. As result of this research the tasks are weighted as follows.

- **16%** Specification and architecture design
- **8%** Detailed design and coding
- **16%** Testing
- **12%** Adaption
- **36%** Extension, improvement
- **12%** Bugfixing

This results were collected by investigating developers as well as the analysis of cost reports. The results from this thesis seem to contradict the conclusions made by Boehm, as they differ

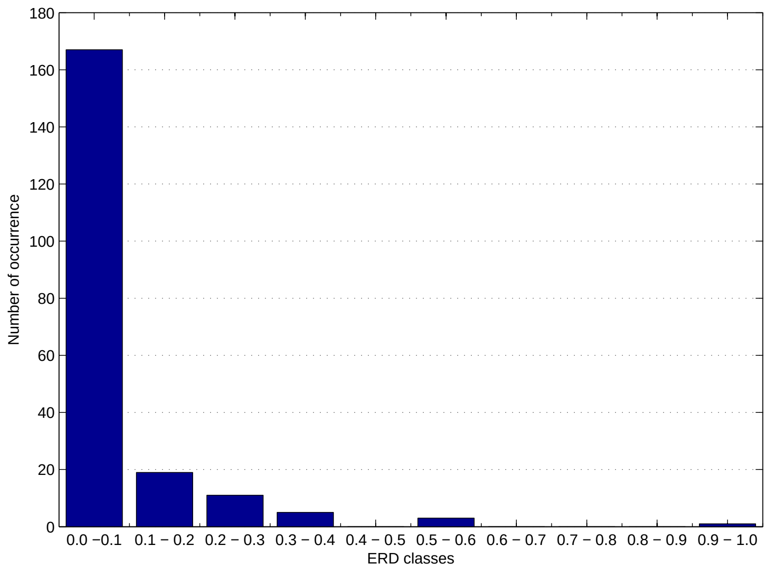


Figure 4.3: The spreading of the ERD metric

by 50% (5.4% vs. 12%). In the subsequent section some facts will be mentioned to illustrate this divergence of the two values.

Life cycle [Boehm, 1981] examines a software project over its whole life cycle. The 12% bug fixing are meant from the beginning of a project until the very end (death) of the software. The org.eclipse.compare plug-in as well as the whole eclipse project is in the mid of its life cycle. Most likely, the share of bug fixing will grow, because after implementing the required features, the focus will be set on finding bugs and improving existing features.

Effort of time The underlying data for the TERD metric is the check-in activity on the version control system. There is no information about how much time a developer spent before he could commit the change. It is well possible that there is more time necessary to fix a bug than to create a functional extension (in average).

Bug reporting discipline The queried data can only consider the reported bugs. These bugs may be only a part of all the defects in a software system. Bugs detected by the developer itself may be fixed without reporting it in a bug tracking system.

Bug linking discipline A bugfix has to be marked by the developer during the check-in action. If this linkage is omitted, the results will be distorted with a leverage effect. The created revision is not recognized as bugfix and therefore, it is considered as functional extension. So not only the ERD is smaller than it really is but also the EVD is falsely raised.

Project characteristics A last reason, why a software project may have a smaller share of bug fixing activity could be a special nature of the project. Maybe some special development techniques prevent the raising of bugs. An example could be 'pair programming' where two developers create the code together controlling each other. This technique can lower the number of bugs in code, but will extend the development time.

Commit message summarizing Software development has often a chaotic touch. A developer with the intention to fix a bug may, while browsing through the code, find a way to extend the functionality of a certain piece of code. He will do the extension and also fix the bug. When committing the made changes the message is often summarized and all changes are checked-in at once. A message could look like this: *"fixed bug 123456 and improved gui responsivity"* Both, the file with the bug fix and the file(s) with the gui improvement are now linked to the bug with number 123456. This fact has the opposite effect than the above mentioned bug linking discipline.

Most of the above reasons could explain the gap betwenn Boehms 12% and the queried 5.4%.

4.3 Patterns

Software engineering has to cope with returning sets of problems to solve. A high-level problem can be most effectively solved by using an existing solution for that specific problem. Such a problem could be the implementation of a communication protocol or a set of user interface controls. On a lower level, there are also recurring problems but they cannot be solved by common-of-the-shelf code because the domain of application is too specific. Solutions for these problems can only be a best-practice advice but never a complete working piece of code. These advices are known as patterns and were summarized by [Gamma et al., 1995]. On the other hand, there are also patterns describing bad design. These *anti-patterns* may increase the error-proneness of the code or generally decrease the manageability of the code. The last form of patterns are the *code smells*[Fowler, 1999]. These are mostly on a very low level and describe in general some bad style of software engineering.

4.3.1 Software Patterns

Software patterns can be divided into two categories: Structural and behavioral patterns. Structural patterns describe the way objects are arranged and linked to each other where behavioral patterns express how objects interact with each other. Often a developer tags a class if it is part of a design pattern by giving it some conventional name or by adding a comment with a reference to the used pattern. But patterns are also used intuitionally. Being able to detect patterns could help to document the code and to make it better understandable.

Behavioral Patterns

The given source code ontology model reaches its limits when trying to detect behavioral patterns. There is a lot of external context and knowledge necessary to identify such a pattern. The observer pattern is a representative of the behavioral patterns. It mainly consists of two classes, the observer and the observable. The observable maintains a list of the registered observers and informs them if a certain event has happened. Therefore the observable needs to have some operations to add, remove and inform observers. To figure out if a class belongs to the `Observer` pattern, we would need to analyze this class if it really maintains a list of `Observer`s and informs them when an event happens. To do this analysis, we would need to follow certain execution paths which is impossible with the current ontology models.

Structural Patterns

In contrast to the behavioral patterns, the structural patterns do not describe an action, but a structural arrangement of elements and their interconnection. This focus on the structure instead of the execution suits better the approach made in the ontology models. Therefore, the instance files can be queried to detect structural patterns.

Proxy Pattern

The proxy pattern is a representative of a structural pattern. Proxies are used to delegate calls to another object. A caller has no reference to the object providing the functionality. It communicates with the proxy object that will do the method call as proxy on the target object. Figure 4.4 shows a UML[2] class diagram of the proxy pattern [Gamma et al., 1995].

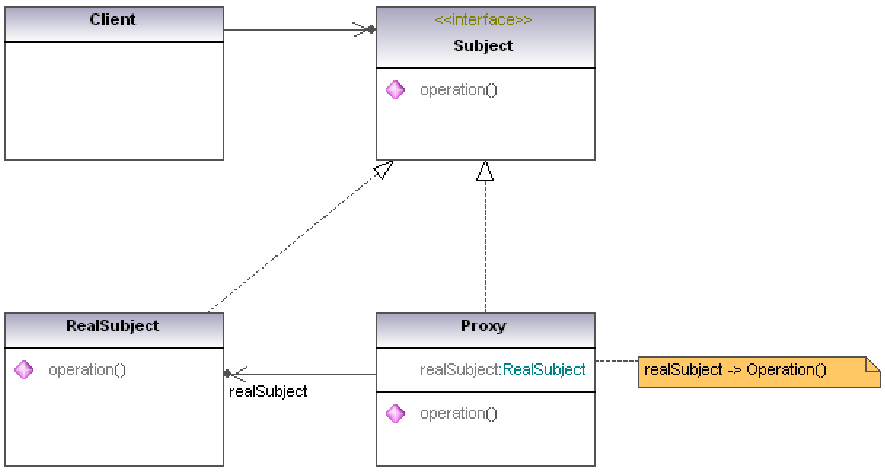


Figure 4.4: The structure of the proxy pattern

When trying to detect a proxy pattern using a SPARQL query, different steps are required. Two classes implementing the same interface are strong candidates for the proxy pattern. If one of the two implementations (proxy) of this interface has a reference to the other one (subject) we consider these classes as a proxy pattern. Query 4.13 shows the SPARQL query to detect the proxy pattern.

```
PREFIX som: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/som#>
SELECT ?client ?interface ?subject ?proxy
WHERE {
        ?proxy som:isSubtypeOf ?interface .
        ?proxy som:hasAttribute ?proxyreference .
        ?subject som:isSubtypeOf ?interface .
        ?interface som:isDeclaredClassOf ?clientreference .
        ?client som:hasAttribute ?clientreference .
        ?proxyreference som:hasDeclaredClass ?subject .
}
```

Listing 4.13: Query to detect the proxy pattern

Query 4.13 can be proofed when using a engineered example (sample implementation of the proxy pattern). When querying the data of the org.eclipse.compare plug-in, no results are returned. This is because the org.eclipse.compare does not use the proxy pattern.

[2]Unified Modeling Language

4.3.2 Anti-Patterns

Anti-Patterns [Fowler, 1999], in contrast to design patterns, generally describe bad design approaches. The danger of these patterns is that they look attractive for a developer to solve a problem. They might seem to be the best way to reach a goal, but, in a longer term they cause many problems and endanger software stability and manageability. To test our ontology models, we tried to detect two *Anti-Patterns*.

Alien Spider Anti-Pattern

The first is the *AlienSpider* pattern, describing a group of classes where every class has a reference to every other class of this group. This will lead to $n*(n-1)$ references between these classes. The major problem underlying this pattern is the tight coupling of the pattern members. When two classes use the functionality of each other, every change to class A will most likely lead to a change of class B. Query 4.14 is used to retrieve an alien spider pattern with two members (classes).

```
PREFIX som: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/som#>
SELECT ?class1 ?class2
WHERE {
        ?class1 som:hasAttribute ?var1 .
        ?class2 som:hasAttribute ?var2 .
        ?var2 som:hasDeclaredClass ?class1 .
        ?var1 som:hasDeclaredClass ?class2 .
        FILTER(?class1 != ?class2)
}
```

Listing 4.14: Query to detect the *Alien Spider* anti-pattern

Query 4.14 for classes belonging to the alien spider pattern. If a variable of type class1 shows up in class2 and a variable of type2 shows up in class1, we can infer that class1 and class2 are part of the alien spider pattern. Table 4.12 shows the results of Query 4.14. The result shows, that the org.eclipse.compare plug-in has one pair of classes holding references of each other. However, there are two occurrences in the query. This is due to the fact, that the query will match every pair twice, once with the first class as class1 and once with the second class as class1. A custom property function could be written for SPARQL to check these double results and eliminate them.

class1	class2
PatchWizard	InputPatchPage
InputPatchPage	PatchWizard

Table 4.12: Result set of Query 4.14

Shotgun Surgery

Our next *Anti-Pattern* is the *Shotgun Surgery*. This pattern describes a method in a software system that is intimately connected with many other classes and methods. If a change is made to this method, all the connected classes and methods are affected of this modification. The danger of this pattern is the human memory. When a change is made, a developer most likely does not know every point in the software any more, that uses the functionality of this method. So this change will lead to defects in the software. *Shotgun Surgery* is a crossover of a pattern and a metric calculation. There are two metrics includes in the detection of the pattern: CC (changing classes) and CM (changing methods, see 4.2.3). CC is a modification of CM that retrieves the number of classes having methods accessing a class. We can extract those two metrics to decide if a class is a shotgun surgery pattern. Query 4.15 extracts the two metrics CC and CM.

```
PREFIX som: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/som#>
PREFIX agg: <java:ddis.evoont.evaluation.extensions.>
SELECT ?class ?method ?CC ?CM
WHERE {
        ?class som:hasMethod ?method .
        ?class som:uniqueName ?className .
        FILTER regex(?className, "compare", "i") .
        ?CC agg:changingClasses ?method .
        ?CM agg:changingMethods ?method .
}
ORDER BY DESC(?CM)
```

Listing 4.15: Query to detect a *Shotgun Surgery*

Query 4.15 uses two custom implemented property functions because determining two different metrics in one query lead our generic counting function to its limits. The code of these property functions can be found in Appendix A. Table 4.13 shows the strongest candidates for a Shotgun Surgery pattern in the org.eclipse.compare plug-in.

Class	Method	CC	CM
CompareUIPlugin	getDefault()	10	30
Utilities	getString(Ljava.util.ResourceBundle,Ljava.lang.String)	14	26
Utilities	getString(Ljava.lang.String)	12	24
ICompareInput	getLeft()	9	16
ICompareInput	getRight()	8	15
ITypedElement	getName()	10	13
CompareUIPlugin	getShell()	6	11

Table 4.13: Parts from the result set of Query 4.15

In Table 4.13 we can see that the method `getDefault()` in class `CompareUIPlugin` is used by 30 methods spread over 10 different classes. In other words, a change to the `getDefault()` method could need a adaption of 30 methods in 10 different locations (classes). The danger to forget one of these 10 locations is high, and, therefore, this class could be considered harmful for the software.

4.3.3 Code Smells

Code Smells [Fowler, 1999] are, in contrast to *Anti-Patterns*, on a lower level. They could be generally described as a bad style of coding. Where a pattern affects the architecture of a software system, a smell is limited to the implementation of a software system. Such a smell is not necessarily an error, but the readability and usability of the source code suffers significantly.

Weak Encapsulation

We tested the ability of our ontology models to detect *Code Smells* by trying to find weak encapsulated attributes of a class. Such attributes are accessible (and manipulable) from outside their defining classes. In Java, this would refer to public attributes. Query 4.16 lists such attributes, accessed from outside their classes. This can be problematic, as an attribute is part of the inner state of a class and should not be exposed to the outside, except by using an accessor (getter, setter) method.

```
PREFIX som: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/som#>
SELECT DISTINCT ?WeakEncapsulationClass ?Field ?AccessorClass
WHERE {
        ?WeakEncapsulationClass oomodel:hasAttribute ?Field .
        ?Field oomodel:accessControlQualifier ?accessControlQualifier .
        ?Field oomodel:isFinal ?final .
        FILTER(?final = false) .
        FILTER(?accessControlQualifier = "public") .
        ?BadMethod oomodel:accesses ?Field .
        ?AccessorClass oomodel:hasMethod ?BadMethod .
        OPTIONAL {
                ?subclass oomodel:isSubclassOf ?WeakEncapsulationClass .
                FILTER(?AccessorClass != ?subclass) .
        }
        FILTER(?AccessorClass != ?WeakEncapsulationClass) .
}
```

Listing 4.16: Query to find attribute accesses from outside the declaring class

In Query 4.16 the `OPTIONAL` block is worth mentioning. It is used to exclude accesses to attributes from super classes. An access from a subclass to a superclass' attribute cannot be seen as an external access because the attributes of the superclass are made internal to the subclass through inheritance. Table 4.14 shows the result set of Query 4.16.

Class	Field	Accessor Class
PatchMessages	InputPatchPageNothingSelectedmessage	PatchTargetPage
PatchMessages	PreviewPatchPageFuzzFactortooltip	PreviewPatchPage
PatchMessages	InputPatchPageSelectInput	PatchTargetPage
PatchMessages	PreviewPatchPageIgnoreWhitespacetext	PreviewPatchPage
...	...	...

Table 4.14: Parts from the result set of Query 4.16

After further investigation of the results from Query 4.16, it arose that all of the matched attributes in Table 4.14 are `static`. Static attributes do not express a inner state of an object, therefore, this cannot be considered harmful for a software system. Ultimately, we can say that the org.eclipse.compare plug-in is free of the *weak encapsulation* code smell.

4.4 Similarities

To calculate similarities within the models, different frameworks are used in combination. Figure 4.5 gives a brief overview of how these components are linked together. The remaining sections describe the components in detail. The Jena components are not explained in detail any more.

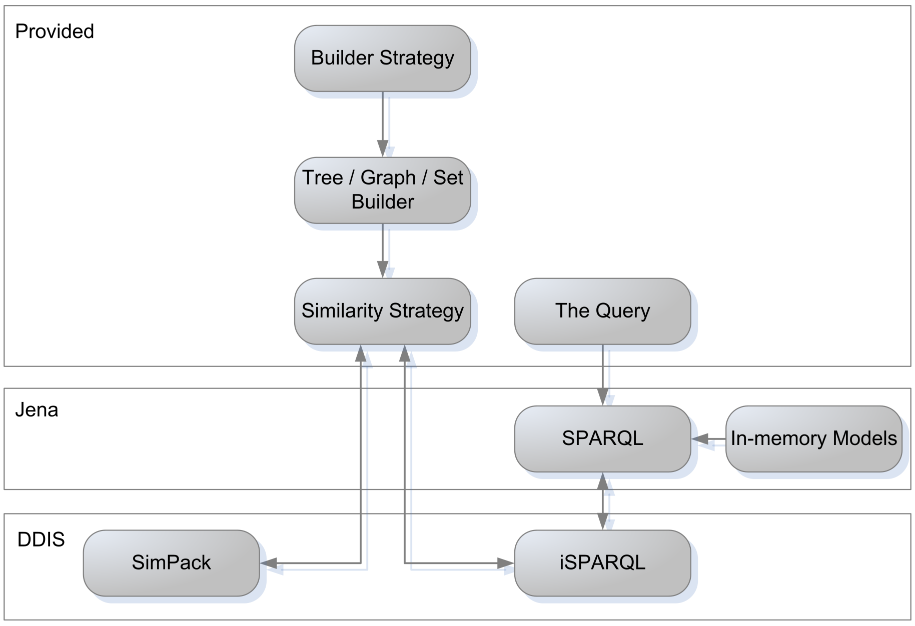


Figure 4.5: Component overview for similarity measures

4.4.1 iSPARQL

This section succinctly introduces the relevant features of the iSPARQL[Stocker, 2006] framework that serves as the technical foundation to all following experiments. iSPARQL is an extension of SPARQL[Prudhommeaux and Seaborne, 2006]. iSPARQL extends the traditional SPARQL grammar but does not make use of additional keywords. Instead, iSPARQL introduces the idea of virtual triples. Virtual triples are not matched against the underlying ontology graph, but used to configure similarity joins: they specify which pair of variables (that are bound by SPARQL to resources) should be joined and compared using what type of similarity measure. Thus, they establish a virtual relationship between the resources bound to the variables describing their similarity. A similarity ontology defines the admissible virtual triples and links the different measures to their actual implementation in the library of similarity measures called *SimPack*. The similarity ontology also allows the specification of more complicated combinations of similarity measures, which we call similarity strategies (or simply strategies) in the remainder of this thesis. An iSPARQL query always needs at least three steps: First, the desired similarity strategy has to be selected. Sencond, the input arguments have to be passed to iSPARQL and third, a variable has to be passed to iSPARQL where the results of the similarity calculations are stored to. Listing 4.17 shows an example configuration of an iSPARQL query. The name of a similarity strategy needs to be a registered iSPARQL similarity strategy.

```
PREFIX isparql: <java:ch.unizh.ifi.isparql.query.property.>
...
?strategy isparql:name "SimilarityStrategy" .
?strategy isparql:arguments (?input1 ?input2)
?strategy isparql:similarity ?sim.
```

Listing 4.17: iSPARQL configuration example

4.4.2 SimPack

SimPack[3] is a generic set of similarity algorithms. First, the data has to be transformed to a structure supported by *SimPack*. Two of these structures are trees and graphs. Then, the algorithms can be applied to these data structures to calculate similarities in various ways.

4.4.3 Evaluation of Compare Algorithms

4.4.4 Similarity Strategy

Whenever using a query with the iSPARQL extension, a similarity strategy is needed. Every triple passed to iSPARQL will be forwarded to the similarity strategy. While there are some generic strategies in the iSPARQL package, these cannot consider the specific features of our software ontology. Thus, some particular adapted strategies were created to respect the special nature of the ontology models. For every pair of arguments the strategy is called once, where an argument is a graph node from the underlying RDF graph maintained by Jena. This RDF graph is accessible from within the strategy and is needed to regain the OntModel from Jena for further data retrieval. In other words, the strategy's input are two resources and the expected output is a double value describing the similarity between these two resources. In the strategy itself the decision has to be made which compare algorithm is used, and how many connected nodes to the passed resources are considered for the similarity calculation. To apply a compare algorithm, the data needs to be brought into the expected format for the algorithm. In a first place we considered two classes of algorithms, namely tree and graph algorithms. Due to the hierarchical structure of source code entities (i.e. classes, methods etc.), our first chosen data structure and compare algorithms were trees. The technique to build up this tree is described in the following section.

Building a Tree from the Ontology

Like described above, the only information to build up a tree is the root node. This root node is passed as parameter from the similarity strategy. Before we can use a compare algorithm, we need to build up the tree for the later comparison of these two trees. To receive expressive results, it is absolutely necessary that both trees are built up by the same methods and rules. To achieve this, we created a generic tree builder class taking the the RDF node as starting point. The idea was for the builder to decide whether to go deeper into some entities or not. As an example, a passed parameter could be the RDF node of a class. This would be a tree with only one (the root) node. When comparing the two tree nodes, the algorithm would always result in a structural similarity of 1.0 because no further elements of the class are considered. The builder should also attach the methods, attributes etc. to the tree. To keep this builder as flexible as possible, we

[3] `http://www.ifi.unizh.ch/ddis/research/semweb/simpack/`

extracted a builder strategy to be able to configure the builder to create trees with custom depths. The configurable parameters are:

- **GoIntoAnonymousClasses** This property is considered when creating a subtree of a method. Whenever a method declares an anonymous class and this property is true, the tree builder will be called recursively and the anonymous class is attached under the declaring method node.
- **GoIntoAttributeAccesses** A method can access attributes of a class. Whenever such an access is encountered, and this property is true, the attribute node will be added to the tree.
- **GoIntoAttributes** This property is slightly different to the above described GoIntoAttributeAccesses. When the above property will attach the attribute subtree to a method node whenever an access happens, this property will add the attribute node under a class node when this class declares this attribute. This node will also be added if an attribute is never accessed.
- **GoIntoClasses** The builder will attach a class node if this property is set to true. This is needed at the very beginning of a build action (when setting the root node) and when a declared class or an anonymous class is found. In the latter case, the anonymous class node is attached to the method and the builder will go on to add methods and attributes of the anonymous class (if demanded).
- **GoIntoDeclaredClasses** This property controls the depth of a class serving as a type of a variable. When a class node is attached by the GoIntoDeclaredTypes property, this property controls wheter to go into the whole class definition of the variable type or not.
- **GoIntoDeclaredTypes** A declared class is the type of an attribute or variable. When enabling this property, the builder will attach only the class node under the attribute or variable node (no recursion).
- **GoIntoLocalVariables** Local variables are defined inside a method. If this property is true, the local variable will be attached to its parent method.
- **GoIntoMethodInvocations** A method can invoke other methods. The invoked method will be added under the invoking methods node if this property is true.
- **GoIntoMethodReturnTypes** A method can have a return value. This value is typed with a class. Whenever a return type is found and this property is true, the class node of the return type is added as a leaf of the returning method's node.
- **GoIntoMethods** If this property is true, a method node is added under its declaring class node.
- **GoIntoParameters** Parameters are the variables passed to a method. Whenever a method takes parameters and this property is true, the parameter node is added under its method node.
- **AnonymousClassDepht** A method can declare anonymous classes and inside these classes methods can again declared other anonymous classes. To control the depth of these declarations, this property takes an integer value describing the number of levels the builder goes into these class declarations before stopping.

- **TypeDeclarationDepht** Similar to the anonymous class depth, this property tells the builder how many levels it should go into type declarations. This will also prevent the builder from running into an endless loop. When a method's return type is from the same class than it is declared by (for instance the singleton pattern), the builder would go into this declared class, and find the initial method, go into the declared class and so on.

Following, the visualized output of the tree builder algorithm is presented. The input was the RDF node representing the org.eclipse.compare.CompareViewerPane class of release 3.2.1 of the org.eclipse.compare plug-in. The builder ran with the configuration: GoIntoAnonymousClasses=false, GoIntoAttributeAccesses=false, GoIntoAttributes=true,GoIntoClasses=true, GoIntoDeclaredClasses=false, GoIntoDeclaredTypes=false, GoIntoLocalVariables=false, GoIntoMethodInvocations=true, GoIntoMethodReturnTypes=false, setGoIntoMethods=true, setGoIntoParameters=false, AnonymousClassDepht=0 and TypeDeclarationDepht=0. For better readability, the output has compressed package names and parameter lists.

```
Class: CompareViewerPane
  |- Method: CompareViewerPane.getToolBarManager(...)
  |  +- Method invocation: CompareViewerPane.getToolBarManager()
  |- Method: clearToolBar(...)
  |  |- Method invocation: ToolBarManager.update(...)
  |  |- Method invocation: CompareViewerPane.getToolBarManager(...)
  |  +- Method invocation: ContributionManager.removeAll()
  |- Method: CompareViewerPane.setImage(...)
  |  |- Method invocation: CLabel.setImage(...)
  |  +- Method invocation: ViewForm.getTopLeft()
  |- Method: CompareViewerPane.setText(...)
  |  |- Method invocation: CLabel.setText(...)
  |  +- Method invocation: ViewForm.getTopLeft()
  |- Method: CompareViewerPane.<init>(...)
  |  |- Method invocation: ViewForm.setTopLeft(...)
  |  |- Method invocation: Widget.addDisposeListener(...)
  |  |- Method invocation: CompareViewerPane$3.<init>()
  |  |- Method invocation: CompareViewerPane$2.<init>()
  |  |- Method invocation: CompareViewerPane$1.<init>(...)
  |  |- Method invocation: ViewForm.<init>(...)
  |  +- Method invocation:Control.addMouseListener(...)
  |- Method: CompareViewerPane.getToolBarManager()
  |  |- Method invocation: ViewForm.setTopCenter(...)
  |  |- Method invocation: ToolBarManager.<init>(...)
  |  +- Method invocation: ToolBar.<init>(...)
  +- Attribute: CompareViewerPane.fToolBarManager
```

Building a Graph from the Ontology

The other class of compare algorithms, graph algorithms, need a graph as input. As a class or software in general has a hierarchical structure, the graph of a class looks almost identical to the above presented tree. Therefore we adapted to tree builder and all of its functionality to now build a graph instead of a tree. This new graph builder uses the same configuration properties as the tree builder but is feasible as input for SimPack's graph algorithms.

4.4.5 Selection of Algorithms

To receive a similarity between two trees or graphs, we need to select proper compare algorithms [Valiente, 2002]. For the tree comparison we chose TreeEditDistance[Valiente, 2002]. This algorithm compares two trees by determining the number of steps needed to merge one tree into the other one. These steps can be insertions, deletions or replacements. If a large number of steps is necessary to transform one tree into the other, this will result in a lower similarity. The algorithm used for graph comparison was SubgraphIsomorphism[Baggenstos, 2006]. This algorithm tries to find the maximum common subgraph of two graphs. Depending on the size of the common subgraph and the input graphs the similarity is calculated. Both of the above mentioned algorithms can compare the structure and the nodes of two graphs or trees. If only the structure would be cons idered, the algorithms would return a similarity of 1.0 whenever the number of nodes and their hierarchical structure is the same. A class with ten methods and five attributes will be considered identical to a class with fifteen methods (the number of nodes is identical and they are all on the same level of hierarchy). To have a finer granulation of the similarity measures, another algorithm can be provided and will be used to calculate the node similarity. In SimPack, a node can contain a user object, a Java object that is wrapped inside the node object. The node compare algorithm can access this object to base the calculations onto the similarity between two such user objects. For our experiments, we used a simple String object as user object containing the uniqueName property of the entities. The used node comparison algorithm was a Levenshtein[Levenshtein, 1966] string similarity. This algorithm uses a similar technique to the TreeEditDistance. Levenshtein counts the number of steps needed to transform one String into the other one. By setting the number of transformations in relation to a worst case distance, the similarity can be calculated.

4.4.6 Criticism on the used Algorithms

The used TreeEditDistance algorithm considers the order of the siblings in a tree. Whenever two elements in the tree are swapped, the algorithm will detect this as a change and therefore the similarity will be decreased. Although, in our case this has no effect on the software itself, if a method is declared before an attribute or afterward, this behavior of the algorithm will lead to blurred results. Figure 4.6 shows another problem when using TreeEditDistance. The figure shows two identical trees except Tree 2 is missing node C. When applying the TreeEditDistance algorithm to the trees 1 & 2, the algorithm will compare the nodes one-by-one. So it will compare A with A, B with B, C with D and so on. By comparing node C from Tree 1 with node D from Tree 2, the calculated similarity of the two trees is smaller than it actually is. A more realistic result would be achieved if a node would be compared to every of its siblings and the one value would be taken with the highest similarity.

The Subgraph Isomorphism[Baggenstos, 2006] algorithm has a very high complexity. To keep the number of calculations small, the algorithm provides a grouping functionality to summarize nodes and treat them as one. However this functionality seems to fail for unknown reasons. For the above mentioned flaws of the used algorithms we decided to implement a specialized, yet simple algorithm to calculate the similarities by ourselves.

4.4.7 Engineered Compare Algorithm

We created our own compare algorithm to respect the nature of source code in the calculation of the similarity. The basic idea of the above presented algorithms were adapted. In the remaining part of this thesis this algorithm will be called *custom algorithm*. At a first stage we needed to create

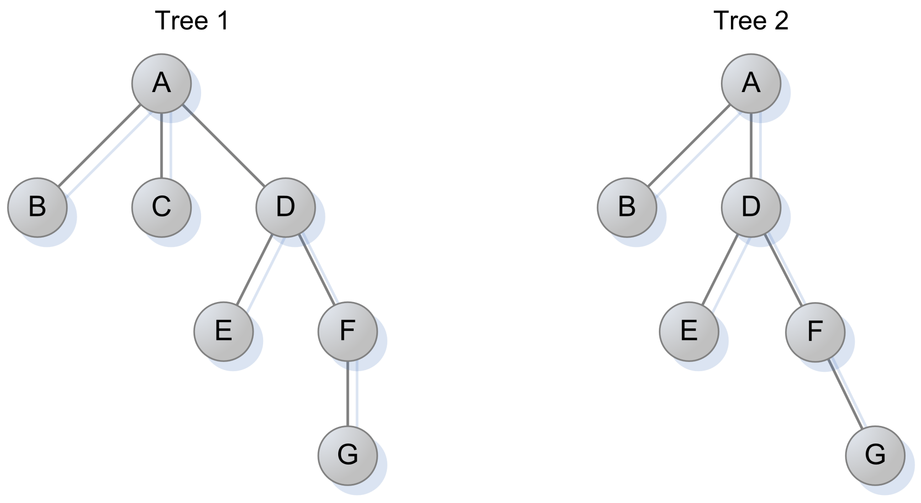


Figure 4.6: Example trees

a data structure to store the to compare entities. We decided to create a object called FamixOntologyObject containing the entity's uniqueName property and its type. The type is as well a simple string containing the kind of entity such as *Class*, *Method*, or *Attribute*. This information is needed later on to calculate the structural similarity. All the FamixOntologyObjects are stored inside a simple Java set (unordered). The first step our algorithm takes is to group the nodes by their types. This will result in a set of entities for each type (i.e. *Classes*, *Methods* etc.). Then, the structural similarity is calculated for every type in the set. As single type similarity is defined as in Equation 4.7.

$$Sim_{type} = \frac{min(TC_{type_{tree1}}, TC_{type_{tree2}})}{max(TC_{type_{tree1}}, TC_{type_{tree2}})} \tag{4.7}$$

In Equation 4.7, TC is the number of a type occurring in the trees. Next, the type similarity is cumulated for all types resulting in the structural similarity.

$$StructSim = \frac{\sum_{type=0}^{numTypes} Sim_{type}}{EC} \tag{4.8}$$

In Equation 4.8, EC is the total number of elements in the trees. To illuminate the calculation of the structural similarity, we can apply it to an example class. Lets assume *Class1* has 10 methods and 5 attributes. *Class2* has 8 methods and 7 attributes. Applying Equation 4.7 to those two classes would lead to a structural similarity shown in Equation 4.9

$$StructSim = 0.7571 = \frac{\frac{8}{10} + \frac{5}{7}}{2} \tag{4.9}$$

After the structural similarity is calculated we extract the node similarity. Because we know about the data's context, we can omit the comparison of nodes, that are not of the same type. If we would compare a method and an attribute, both with the same name, this would lead to a node similarity of 1.0, but from the context we know that between these two entities can never be a similarity because the type does not fit. With this fact in mind we calculate the node similarity type by type. Again, a Levenshtein string compare algorithm was used. The node comparison has an optimistic assumption to find the most similar entities. An entity is compared to every sibling entity of the same type. The highest similarity value is the one to be selected. In a last step the similarity value of all the nodes is summarized and divided by the number of nodes. This will result in the overall node similarity. To retrieve the overall similarity, the structural and node similarity have to be added. The addition can be weighted if one similarity should be considered more or less in the result. In our experiments we weighted them equally.

$$sim = 0.5 * structSim + 0.5 * nodeSim \tag{4.10}$$

4.4.8 Querying the Models for Similarity

In this section, different queries are presented that we used to do our similarity calculations. The queries use the SPARQL syntax with the iSPARQL extension to use SimPack's functionality.

Release Similarity

First, we calculated the difference between two releases. We wanted to have the similarity between all classes of one release and all classes of another release. At first the query joins all the classes in the two releases to pairs. So, the 127 classes of the org.eclipse.compare plug-in will result in 127*127=155'829 pairs between each other the similarity is calculated.

```
PREFIX isparql: <java:ch.unizh.ifi.isparql.query.property.>
PREFIX som: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/som#>
PREFIX vom: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/vom#>
SELECT ?uniqueName1 ?uniqueName2 ?sim
WHERE {
        ?release1 a vom:Release .
        ?release2 a vom:Release .
        ?release1 vom:name "R3_2" .
        ?release2 vom:name "R3_1" .
        ?revision1 vom:hasRelease ?release1 .
        ?revision2 vom:hasRelease ?release2 .
        ?file1 som:hasRelease ?release1 .
        ?file2 som:hasRelease ?release2 .
        ?file1 som:hasClass ?class1 .
        ?file2 som:hasClass ?class2 .
        ?class1 som:uniqueName ?uniqueName1 .
        ?class2 som:uniqueName ?uniqueName2 .
        ?strategy isparql:name "TreeEditDistance" .
        ?strategy isparql:arguments (?class1 ?class2) .
        ?strategy isparql:similarity ?sim .
}
```

Listing 4.18: Query for similarity measures between two releases

To visualize the results, we exported the returned data into a CSV[4] format to be able to process it using Matlab. The CSV files for the queries can be found on the accompanying DVD (see appendix C). We used Matlab to generate heatmaps of the queried data to visualize the similarity. On each axis of the graph (x and y) one release and its source files are listed. On every

[4]comma separated values

intersection of two classes their calculated similarity value is presented using a color code. The color changes depending on the similarity from black (no similarity), red (few similarity), yellow (strong similarity) to white (equality). Figure 4.7 shows the heatmap of the comparison between release 3.1 and 3.2 of the org.eclipse.compare plug-in using the TreeEditDistance algorithm. At a first glance, the diagonal light line stands out of the figure. This line represents the comparison of the same class, once from release 3.1 and once from 3.2. This line marks the effective change made to the software project between the two releases. The other parts of the figure show the similarities between the other classes. The underlying similarity strategy of figure 4.7 does not regard the similarity between the nodes. It only compares the structure between two trees. With a next experiment we wanted to see the effect on the similarity when the nodes (the uniqueNames) are considered in the similarity calculations. Figure 4.8 shows the same measure as in figure 4.7 except the node similarity is considered by using a Levensthein string compare algorithm. The result show a generally decreased similarity except the diagonal line seems unchanged. In another cycle we conducted this experiment using the SubgraphIsomorphism algorithm. Again, a Levenstein string comparator was used to calculate the similarities of the nodes. Figure 4.9 shows the resulting heatmap. In a last experiment we applied our engineered custom algorithm (Figure 4.10).

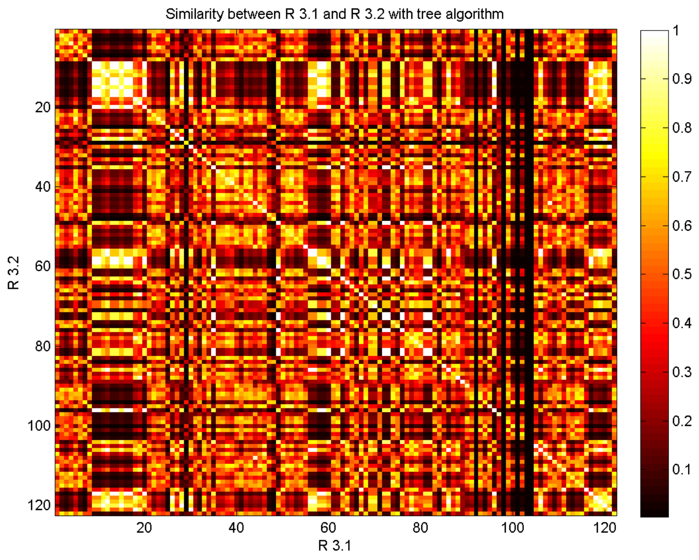


Figure 4.7: Similarity between release 3.1 and 3.2 using a TreeEditDistance algorithm

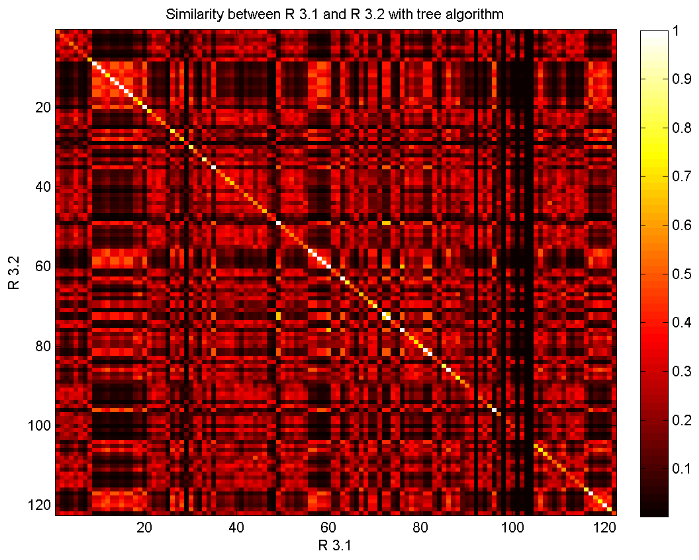


Figure 4.8: Similarity between release 3.1 and 3.2 using a TreeEditDistance algorithm and a Levenshtein node comparator

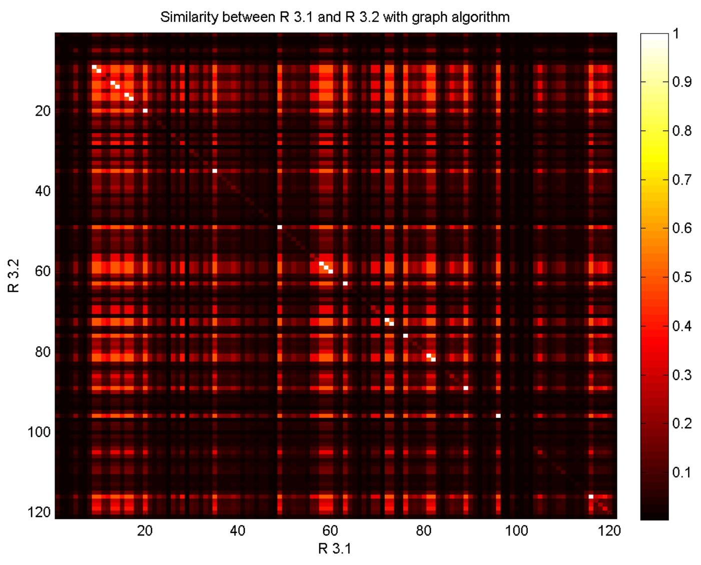


Figure 4.9: Similarity between release 3.1 and 3.2 using a SubgraphIsomorphism algorithm and a Levenshtein node comparator

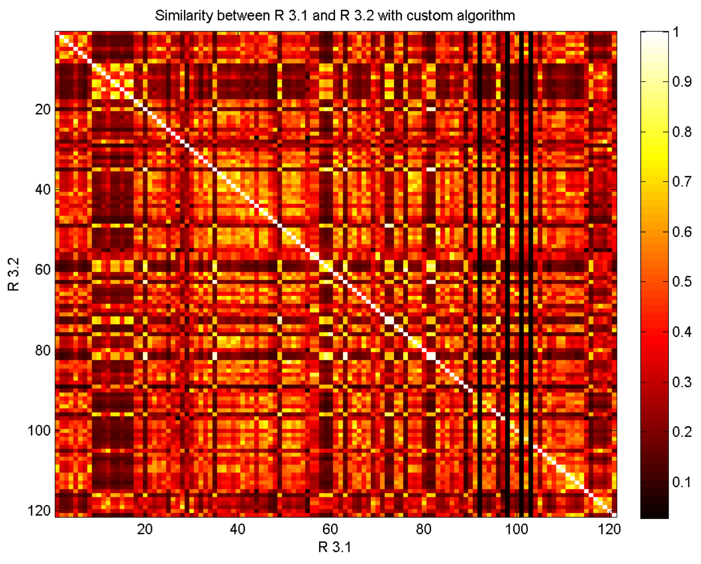


Figure 4.10: Similarity between release 3.1 and 3.2 using a custom algorithm and a Levenshtein node comparator

Class History

Another interesting information is the history of a class. Usually, a class experiences different changes over its life cycle. If a class underlies many changes, this can be an indicator of importance but, as well, be a sign of error-prone code. There are different approaches to measure the change made to a class. A very simple way to measure the change could be by comparing the filesize of two classes. A more sophisticated approach is made by textual `diff` tools, counting the added, removed and changed lines. However, this is made on a textual base. The `diff` tool cannot separate code from for example comments. With our approach, we can directly calculate the change made to the code and omit changes made to comments, the moving of methods or attributes etc. Where in the above section all the classes from two releases were examined we here focus on single classes but try to consider their whole lifetime. Therefore, we need to compare a single class in every release of the software. Because a software project can have hundrets of releases, this would be cumbersome to measure in one query. Therefore, our query (4.19) compares in a first step the same classes of only two releases. The compare algorithm is our foregoing mentioned custom compare algorithm.

```
PREFIX isparql: <java:ch.unizh.ifi.isparql.query.property.>
PREFIX som: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/som#>
PREFIX vom: <http://www.ifi.unizh.ch/ddis/evoont/2007/02/vom#>
SELECT ?uniqueName ?sim
WHERE {
        ?release1 a vom:Release .
        ?release2 a vom:Release .
        ?release1 vom:name "<release2>" .
        ?release2 vom:name "<release1>" .
        ?file1 som:hasRelease ?release1 .
        ?file2 som:hasRelease ?release2 .
        ?file1 som:hasClass ?class1 .
        ?file2 som:hasClass ?class2 .
        ?class1 som:uniqueName ?uniqueName .
        ?class2 som:uniqueName ?uniqueName2 .
        FILTER (?uniqueName = ?uniqueName2) .
        ?strategy isparql:name "SimpleListCompare" .
        ?strategy isparql:arguments (?class1 ?class2)
        ?strategy isparql:similarity ?sim.
}
```

Listing 4.19: Query for similarity measures between the same classes of two releases

In a next step we took our list of all releases in chronological order and walked through that list executing Query 4.19 for every pair of releases. Again the results were visualized using *Matlab*. The output of this measure are 127 figures. Three of them are presented here. The three figures show the class history over the whole lifetime of their classes. The height of the bars describe the amount of change.

Classes show different lifecycle types. Where figure 4.11 shows a class with increasing changes over time, the class shown in figure 4.12 seems to underly some equally spread, constant changes. 4.13 shows a class that seemed to have an activity peak in the middle of its history and then, the change amount stabilized at a more or less constant level.

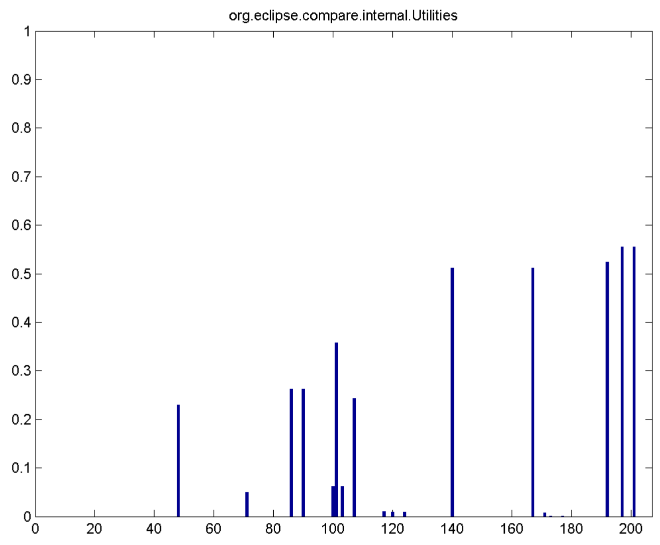


Figure 4.11: Change history of class org.eclipse.compare.internal.Utilities

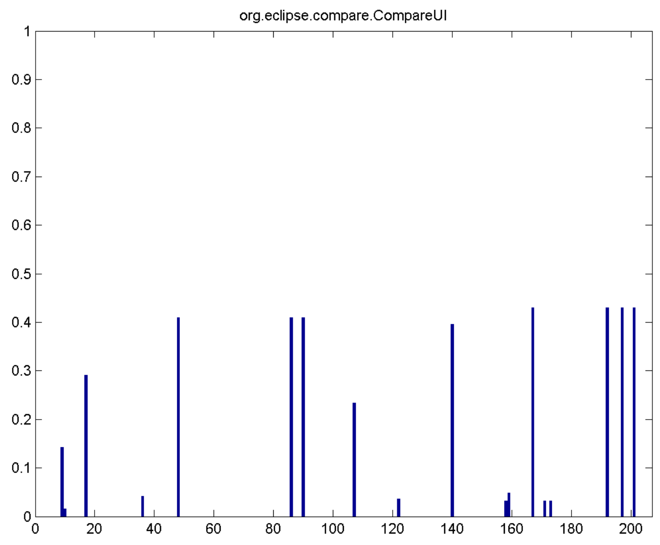


Figure 4.12: Change history of class org.eclipse.compare.CompareUI

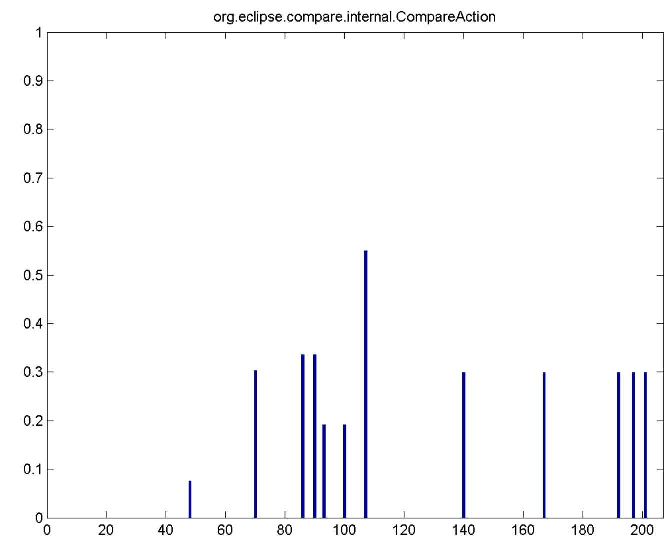


Figure 4.13: Change history of class org.eclipse.compare.internal.CompareAction

5 Conclusions

In this thesis we successively showed the application of Semantic Web techniques and formats to the area of software engineering. First, we introduced the meta models for the three software repositories, source code, versioning system and bug tracking system. These meta models serve as a base for the implemented tools to retrieve and store the data from the three repositories. We presented an eclipse plug-in able to extract the repositories' data in an automated manner. In the last chapter we conducted excessive experiments to proof our approach. The experiments have shown, that OWL/RDF can be used as data exchange format for software meta data, and that *SPARQL*, *iSPARQL* and *SimPack* are feasible for executing queries and retrieving meaningful data from the models. The major advantage of the presented approach is the open, web-capable format which allows future extension with further repositories such as email systems, data from social networks etc. At last, the possibility to link web-wide spread semantic annotation to a large documentation and annotation network offers a interesting approach for future software engineering.

5.1 Limitations

The limitations of the proposed approach can be divided into two categories – 'Technical Limitations' and 'Conceptual Limitations'. Where the technical limitations are externally given and may vanish over time, the conceptual limitations need a reconsideration of the used technologies and proposed meta models.

5.1.1 Technical Limitations

The Semantic Web technology is grown out of different research projects. It has not the pragmatism certain industry-invented technologies have, but, therefore, is elaborate and proof of concept. The technical limitations of such technologies are mostly at a second place because computers will get faster and Internet connections will gain bandwidth. In this thesis, the limits were often given by CPU power or memory size. For example, the measurement of certain queries was in the range of several hours to days. To create the in-memory models and store them to OWL/RDF the Jena framework is used. When parsing the data of a larger project, the model created by Jena is too complex for the underlying Java virtual machine resulting in a java.lang.StackOverflowError. Owl as part of the semantic web uses a lot of information to annotate the data. For example a number is annotated with its proper XSD type to enable a reading computer system to identify the kind of data and how to handle it. Next, the underlying XML is more expressive than it technically would need to be. For instance a closing tag always repeats the name of the opening tag like <example>...</example> where for a computer system <example>...</> would be enough. All these redundancies are brought in to increase the readability for a human. The price for this readability is a larger file size and therefore a slower processing. The models generated for this thesis of the org.eclipse.compare plug-in which are the versioning model, the bug model and 286 code models have a file size of together 1.67 GB. This large filesizes make the models cumbersome to handle and the queries long to execute.

5.1.2 Conceptual Limitations

The concept used has some flaws, especially when trying to extract metrics. There are many metric calculations that need finer grained model components like *if* clauses, loops etc. These elements are not included in the ontology models and parsers, therefore, not all metric calculations and pattern detections can be made. Another limitation of the meta model is the fixation on Bugzilla. It would be desirable to have a generic meta model of a bug tracking system. This may be part of a future work.

5.2 Future Work

The presented ontology models and parser show a first approach. The ontology models as well as the parsers should be extended to regard more information stored in the repositories. The concrete extensions of the meta models would be:

- Extend the version ontology model to include more entities of the CVS system. Also, other versioning systems should be regarded such as Subversion.
- Adapt the bug ontology model to be compatible with other bug tracking systems such as Mantis[1] or Scarab[2].
- Extend the software ontology model to include more facets of the Java language (like some features of version 1.5 i.e *Generics*) and different programming languages such as C++ or SmallTalk.

Every extension made to the meta models results in a necessary extension of the parsers. Whenever adding support for a new programming language, bug tracking system or versioning system, a completely new parser needs to be written to be able to retrieve data from that system. When extending the meta model, the existing parsers need to be adapted. Another extension that may be part of future work is the extension with completely new types of repositories. This could include email conversation, organization hierarchy data or data from message forums. These new repositories can be easily attached to the existing models and meta models. Interesting experiments using data mining techniques could be adapted. The here shown experiments only use simple queries and some post-processing steps. Probably a lot more data and information can be retrieved by applying data mining to the generated models.

[1]http://www.mantisbt.org
[2]http://scarab.tigris.org

A

Code Listings

Some selected source code listings are presented in this section. The selection of the code snippets is based on the usefulness for understanding its functionality. Far not all the source code generated in this thesis is listed here. A complete collection of all programs can be found on the accompanying DVD. See appendix C for details.

A.1 A Generic Function for Counting with ARQ

This function is inspired by the `group` class created by Andy Seabourne at the HP labs. The original file is not part of the standard Jena or ARQ distribution and can be found on Jena's CVS server in the package `com.hp.hpl.query.extension.library`. Below, only the central method, `exec`, is presented.

```java
public QueryIterator exec(QueryIterator input, List args, String uri,
    ExecutionContext execCxt) {

  //argument 1 (the variable holding the input
  Expr expr = (Expr)args.get(1);
  NodeVar saveTo = (NodeVar)args.get(0);
  HashMap<NodeValue,List<Binding>> count = new HashMap<NodeValue,List<
      Binding>>();
  while(input.hasNext()){

    Binding binding = input.nextBinding();
    // we evaluate the expression for the actual binding
    NodeValue nodeValue = expr.eval(binding, execCxt);

    //check if the hashmap already contains this nodevalue
    if(count.containsKey(nodeValue)){
      //add the binding (grouping)
      count.get(nodeValue).add(binding);
    }else{
      //not yet in list. create the new list and add this binding
      List<Binding> list = new ArrayList<Binding>();
      list.add(binding);
      count.put(nodeValue, list);
    }

  }

  Collection<List<Binding>> list = count.values();
  List<Binding> returnList = new ArrayList<Binding>();
  for(List<Binding>bindingList:list){
      if(bindingList.get(0) != null){

      Binding save = bindingList.get(0);
      //the number members in this group (count) is bindingList.size()
      save.add(saveTo.getVarName(), NodeValue.makeInteger(bindingList.
         size()).asNode());
      returnList.add(bindingList.get(0));
    }

  }

  return new QueryIterPlainWrapper(returnList.iterator(),execCxt);

}
```

Listing A.1: countChilds.java

The next two classes, `changingClasses` and `changingMethods` also count entities within a query but are much more sophisticated. These two classes only work with our ontology models and their respective instances files. The first class, `changingClasses`, counts the number of classes, a method has invokers in. The second, `changingMethods`, counts the number of invokers. These two classes were created for the *Shotgun Surgery* pattern as the above presented, generic counting function has problems when being called mulitple times in the same query.

```
public QueryIterator exec(QueryIterator input, List args, String uri,
   ExecutionContext execCxt) {

  //argument 1 (the variable holding the input
  Expr expr = (Expr)args.get(1);
  NodeVar saveTo = (NodeVar)args.get(0);
  HashMap<NodeValue,List<Binding>> count = new HashMap<NodeValue,List<
     Binding>>();
  while(input.hasNext()){

    Binding binding = input.nextBinding();
    // we evaluate the expression for the actual binding
    NodeValue nodeValue = expr.eval(binding, execCxt);

    //check if the hashmap already contains this nodevalue
    if(count.containsKey(nodeValue)){
      //add the binding (grouping)
      count.get(nodeValue).add(binding);
    }else{
      //not yet in list. creat the new list and add this binding
      List<Binding> list = new ArrayList<Binding>();
      list.add(binding);
      count.put(nodeValue, list);
    }

  }

  Collection<List<Binding>> list = count.values();
  List<Binding> returnList = new ArrayList<Binding>();
  for(List<Binding>bindingList:list){
      if(bindingList.get(0) != null){

      Binding save = bindingList.get(0);
      //the number members in this group (count) is bindingList.size()
      save.add(saveTo.getVarName(), NodeValue.makeInteger(bindingList.
         size()).asNode());
      returnList.add(bindingList.get(0));
    }

  }

  return new QueryIterPlainWrapper(returnList.iterator(),execCxt);

}
```

Listing A.2: changingClasses.java

```
  public QueryIterator exec(QueryIterator input, List args, String uri,
       ExecutionContext execCxt) {

    OntModel model = ModelFactory.createOntologyModel(OntModelSpec.
       OWL_DL_MEM ,ModelFactory.createModelForGraph(execCxt.
       getActiveGraph()));

    Expr expr = (Expr)args.get(1);
    NodeVar saveTo = (NodeVar)args.get(0);
    List<Binding> bindings = new ArrayList<Binding>();
    while(input.hasNext()){

      Binding binding = input.nextBinding();
      NodeValue nodeValue = expr.eval(binding, execCxt);
      NodeIterator ni = model.listObjectsOfProperty(model.getResource(
         nodeValue.asNode().getURI()), model.getProperty(Namespace.SOM
         +"isInvokedBy"));

      int i = 0;
      while(ni.hasNext()){
        RDFNode node = ni.nextNode();
        System.out.println("Processing␣method␣invocation␣node:␣"+node.
           toString());
        NodeIterator ni2 = model.listObjectsOfProperty(model.
           getResource(node.asNode().getURI()), model.getProperty(
           Namespace.SOM+"isMethodOf"));
        i++;

      }

      NodeValue nv = NodeValue.makeInteger(i);
      bindings.add(new Binding1(binding, saveTo.getVarName(), nv.asNode
         ()));

    }

    return new QueryIterPlainWrapper(bindings.iterator(),execCxt);

  }
```

Listing A.3: changingMethods.java

B

UI Elements of the Plug-in

In the following section, the different steps during the user interaction of the plug-in is presented.

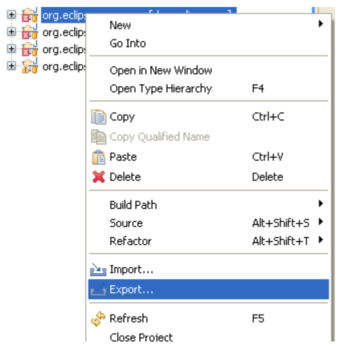


Figure B.1: Entry point where exporters can be activated

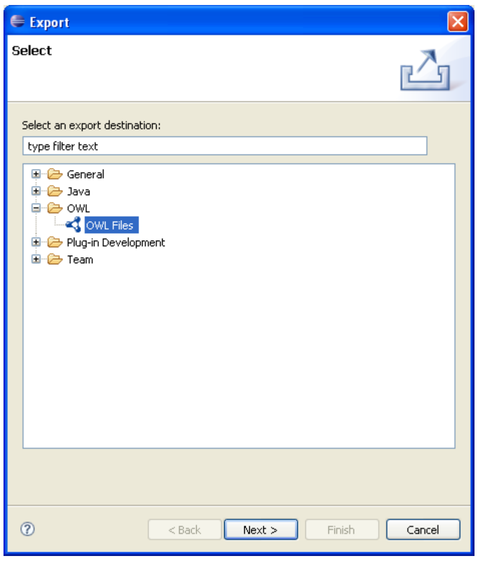
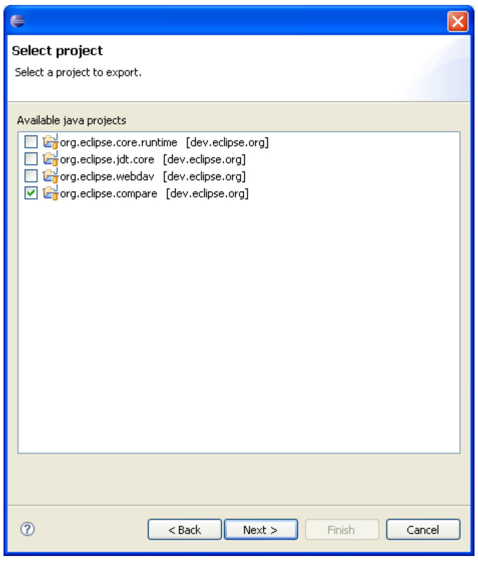
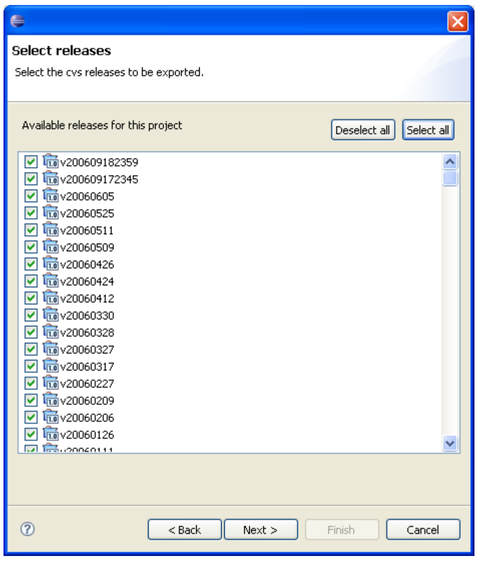


Figure B.2: Location of the OWL-Exporter in the list of exporters



- [ ] org.eclipse.core.runtime [dev.eclipse.org]
- [ ] org.eclipse.jdt.core [dev.eclipse.org]
- [ ] org.eclipse.webdav [dev.eclipse.org]
- [✓] org.eclipse.compare [dev.eclipse.org]

Figure B.3: Selection of the project to export



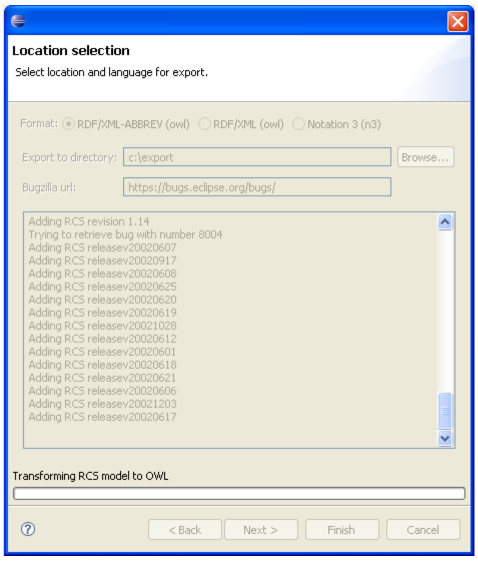
Select releases

Select the cvs releases to be exported.

Available releases for this project

- ✓ v200609182359
- ✓ v200609172345
- ✓ v20060605
- ✓ v20060525
- ✓ v20060511
- ✓ v20060509
- ✓ v20060426
- ✓ v20060424
- ✓ v20060412
- ✓ v20060330
- ✓ v20060328
- ✓ v20060327
- ✓ v20060317
- ✓ v20060227
- ✓ v20060209
- ✓ v20060206
- ✓ v20060126
- ✓ v20060111

Figure B.4: Selection of the software releases to export



Location selection

Select location and language for export.

Format: [✓ RDF/XML-ABBREV (owl)] RDF/XML (owl) Notation 3 (n3)

Export to directory: c:\export

Bugzilla url: https://bugs.eclipse.org/bugs/

```
Adding RCS revision 1.14
Trying to retrieve bug with number 8004
Adding RCS releasev20020607
Adding RCS releasev20020917
Adding RCS releasev20020608
Adding RCS releasev20020625
Adding RCS releasev20020620
Adding RCS releasev20020619
Adding RCS releasev20021028
Adding RCS releasev20020612
Adding RCS releasev20020601
Adding RCS releasev20020618
Adding RCS releasev20020621
Adding RCS releasev20020606
Adding RCS releasev20021203
Adding RCS releasev20020617
```

Transforming RCS model to OWL

Figure B.5: Providing an export folder and Bugzilla URL. Start of the parsing action

C

DVD

All the created and generated files within this thesis are stored on the accompanying DVD. Table C.1 gives a brief overview of the files and folders on the disc and their contents. The need of using a DVD instead of a CD is the large file size of the generated org.eclipse.compare models using about 1.67 GB of space.

Name	Description
Code	All source code used and created in this thesis. The sub folders of this folder are Eclipse projects
Text	This document as PDF as well as the LaTeXsource code
Ontologies	All the OWL files used in this thesis
Output	The output of the different queries and measures presented above. Also the images to visualize the data created with Matlab
Parsed	The generated source code, versioning and bug-models from the org.eclipse.compare plugin.

Table C.1: An overview over the files and folder on the DVD and their contents

Bibliography

[Baader et al., 2003] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F. (2003). *The Description Logic Handbook: Theory, Implementation, Applications.* Cambridge University Press, Cambridge, UK.

[Baggenstos, 2006] Baggenstos, D. (2006). *Implementation and Evaluation of Graph Isomorphism Algorithms for RDF-Graphs*. PhD thesis, University of Zurich.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web (berners-lee et. al 2001).

[Boehm, 1981] Boehm, B. W. (1981). *Software Engineering Economics*. Prentice-Hall.

[D'Ambros and Lanza, 2006] D'Ambros, M. and Lanza, M. (2006). Software Bugs and Evolution: A Visual Approach to Uncover Their Relationships. In *Proc. of the 10th European Conf. on Software Maintenance and Reengineering (CSMR '06)*, pages 227–236. IEEE CS Press.

[Demeyer et al., 1999] Demeyer, S., Tichelaar, S., and Steyaert, P. (1999). Famix 2.0. Technical report, University of Berne.

[Dietrich and Elgar, 2005] Dietrich, J. and Elgar, C. (2005). A Formal Description of Design Patterns Using OWL. In *Proc. of the 2005 Australian Software Engineering Conf. (ASWEC '05)*, Brisbane, Australia.

[Fischer et al., 2003] Fischer, M., Pinzger, M., and Gall, H. (2003). Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32, Amsterdam, Netherlands. IEEE Computer Society Press.

[Fowler, 1999] Fowler (1999). *Refactoring. Improving the Design of Existing Code.* Addison-Wesley.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns.* Addison-Wesley, Boston, MA.

[Happel et al., 2006] Happel, H.-J., Korthaus, A., Seedorf, S., and Tomczyk, P. (2006). KOntoR: An Ontology-enabled Approach to Software Reuse. In *Proc. of the 18th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE '06)*, San Francisco, CA.

[Hyland-Wood et al., 2006] Hyland-Wood, D., Carrington, D., and Kapplan, S. (2006). Toward a Software Maintenance Methodology using Semantic Web Techniques. In *Proc. of the 2nd Int. IEEE Was. on Software Evolvability at IEEE Int. Conf. on Software Maintenance (ICSM '06)*, pages 23–30, Philadelphia, PA.

[Lanza and Marinescu, 2006] Lanza, M. and Marinescu, R. (2006). *OO-Metrics in Practice*. Springer, Berlin.

[Levenshtein, 1966] Levenshtein, V. I. (1966). *Binary Codes Capable of Correcting Deletions, Insertionsand Reversals*. Soviet Physics Doklady.

[Mäntylä et al., 2003] Mäntylä, M., Vanhanen, J., and Lassenius, C. (2003). A Taxonomy and an Initial Empirical Study of Bad Smells in Code. In *Proc. of the Int. Conf. on Software Maintenance (ICSM '03)*, Washington, DC. IEEE Computer Society.

[Prudh́ommeaux and Seaborne, 2006] Prudh́ommeaux, E. and Seaborne, A. (2006). Sparql query language for rdf. Technical report, W3C.

[Sager et al., 2006] Sager, T., Bernstein, A., Pinzger, M., and Kiefer, C. (2006). Detecting Similar Java Classes Using Tree Algorithms. In *Proc. of the 2006 Int. Was. on Mining Software Repositories (MRS '06)*, New York, NY. ACM Press.

[Shatnawi and Li, 2006] Shatnawi, R. and Li, W. (2006). A Investigation of Bad Smells in Object-Oriented Design Code. In *Proc. of the 3rd Int. Conf. on Information Technology : New Generations (ITNG'06)*, Washington, DC. IEEE Computer Society.

[Stocker, 2006] Stocker, M. (2006). The fundamentals of isparql. Master's thesis, University of Zurich.

[Valiente, 2002] Valiente, G. (2002). *Algorithms on Trees and Graphs*. Springer.

[W3C, 2004a] W3C (2004a). Owl web ontology language overview.

[W3C, 2004b] W3C (2004b). Xml schema part 2: Datatypes second edition.