# Quality Validation through Pattern Detection – A Semantic Web Perspective

David Walsh

A Thesis

In

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Sciences (Software Engineering) at

Concordia University

Montreal, Quebec, Canada

May 2012

**CONCORDIA UNIVERSITY**

**School of Graduate Studies**

This is to certify that the thesis prepared

By: David Walsh

Entitled: Quality Validation through Pattern Detection – A Semantic Web Perspective

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

**Master of Applied Sciences (Software Engineering)**

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

Signed by the final Examining Committee:

| | |
|---|---|
| Dr Joey Paquet | Chair |
| Dr Gregory Butler | Examiner |
| Dr Olga Ormandijeva | Examiner |
| Dr Juergen Rilling | Supervisor |

Approved by ________________________________________________

Chair of Department of Graduate Program Director

______________2012 ____________________________

Dean of Faculty

# Abstract

Quality Validation through Pattern Detection – A Semantic Web Perspective

David Walsh

Given the ongoing trend towards a globalization of software systems, open networks and distributed platforms, validating non-functional requirements and qualities becomes an essential requirement. Our research addresses this challenge from two different perspectives: (1) the integration of knowledge and tool resources through semantic web technologies as part of our SE-PAD environment in order to reduce or eliminate existing traditional information and analysis silos. (2) The ability to reason upon linked resources to infer both explicit and implicit patterns to support the validation of quality aspects. We illustrate the flexibility and applicability of our approach through several use cases, including the detection of security and design patterns, violations of secure programming guideline violations.

## Table of contents

List of Figures ........ v
List of Tables ........ v
1. Introduction ........ 1
 1.1 Contributions ........ 3
 1.2 Thesis Outline ........ 4
2. Quality Validation through Pattern Detection – a Semantic Web Perspective ........ 5
 2.1 Motivation ........ 5
 2.2 Goals and Requirements ........ 6
 2.3 Research Hypothesis ........ 8
3. Background ........ 10
 3.1 Ontologies and Semantic Web Technologies ........ 10
  3.1.1 RDF ........ 11
  3.1.2 RDFS ........ 12
  3.1.3 OWL-DL ........ 12
  3.1.4 Reasoning ........ 14
  3.1.5 SPARQL queries ........ 17
  3.1.6 Putting it all together ........ 18
 3.2 Software Patterns and Quality Inspection ........ 19
  3.2.1 Software Quality Inspection ........ 20
  3.2.2 Functional Inspections ........ 21
  3.2.3 Design Inspection ........ 22
  3.2.4 Inspection of Source Code ........ 24
 3.3 Ontologies in Software Engineering ........ 27
  3.3.1 Knowledge modeling: information silos ........ 28
  3.3.2 Conceptualization ........ 29
4. SE-PAD as a Semantic Web based quality validation platform ........ 32
 4.1 SE-PAD Architecture ........ 32
 4.2 SE-PAD and the use of Ontologies ........ 33
  4.2.1 Design (*t-box*) ........ 34
  4.2.2 Population and Realization (*a-box*) ........ 38
 4.3 A Semantic Web Approach to Software Quality ........ 42
 4.4 Programming Guidelines ........ 44
  4.4.1 General Programming Guidelines ........ 44
  4.4.2 Secure Coding Guidelines ........ 53
 4.5 Modeling Design Patterns ........ 59
  4.5.1 Adapter Pattern ........ 61
  4.5.2 Proxy Pattern ........ 64
  4.5.3 Strategy Pattern ........ 67
  4.5.4 Template Pattern ........ 71
 4.6 Tool integration ........ 74
 4.7 Cross artefact analysis ........ 77
5. Case Studies ........ 80
 5.1 Security violations ........ 80
 5.2 Design Pattern Automated Recovery ........ 82
 5.3 Integration of external static analysis tool results (PMD and FindBugs) ........ 83

5.4 Integration of SCM Ontology ........ 85
6. Discussion ........ 88
6.1 Revisiting the hypothesis ........ 88
6.2 The Open World Assumption problem ........ 89
6.3 Threats to Validity ........ 91
6.3.1 Semantic web technologies fail in bridging the information silos (R1). ........ 91
6.3.2 Implicit versus explicit knowledge (R2) ........ 92
6.3.3 Establishing Trustworthiness (R3) ........ 92
6.3.4 Pattern and knowledge base have to be extensible (R4) ........ 94
7. Conclusions and Future Work ........ 95
8. References ........ 97

## List of Figures

Figure 1-1: High level view of our approach ........ 4
Figure 3-1: A simple semantic network ........ 11
Figure 3-2: Semantic Web technologies working together ........ 18
Figure 3-3: Software abstraction scale ........ 21
Figure 3-4: Information Silos for a generic domain of knowledge ........ 29
Figure 4-1: SE-PAD's architecture ........ 32
Figure 4-2: Sample of SE-PAD's *t-box* ........ 38
Figure 4-3: SE-PAD's partial resulting ontology example ........ 40
Figure 4-4: SE-PAD's support for external tool rules ........ 41
Figure 4-5: The semantic levels supported by SE-PAD ........ 42
Figure 4-6: Mutable class realization ........ 56
Figure 4-7: SCM rule violation search algorithm ........ 78

## List of Tables

Table 3-1: Research goal, sub-goals, and related requirements ........ 6
Table 2-1: Examples of software patterns ........ 19
Table 2-2: Software engineering responsibilities ........ 28
Table 2-3: SOM Ontology relations, domain and range concepts ........ 31
Table 4-1: SE-PAD's set of extended data properties ........ 35
Table 4-2: SE-PAD's set of extended object properties ........ 36
Table 4-3: Extension of SOM's Attribute concept ........ 36
Table 4-4: Extension of SOM's Method concept ........ 37
Table 4-5: Extension of SOM's Class concept ........ 37
Table 4-6: Pattern presentation template ........ 43
Table 5-1: JabRef population summary ........ 80
Table 5-2: SE-PAD vs. PMD ........ 81
Table 5-3: SE-PAD vs. DPR ........ 82
Table 5-4: ArgoUML population summary ........ 83
Table 5-5: Ratio of unsafe classes in ArgoUML ........ 84
Table 5-6: Integration of SCM Ontology results ........ 86
Table 6-1: Semantic classification and identification of pattern support ........ 89

# 1. Introduction

Syntax errors are part of any programmer's daily life: missing parenthesis, misspelled variable names are easily caught by compilers and immediately fixed by the programmer. In contrast to these syntax errors, other forms of programmer mistakes, such as violating coding guidelines, programming patterns, introducing vulnerabilities and lack of good programming practices often lead to mistakes that are not discovered by traditional IDEs.

These problems are often part of what constitutes code quality, a Non-Functional Requirements (NFR). Such qualities can be divided into two main categories: (1) *Execution* qualities, such as performance and usability, which are observable at run time; and (2) *evolution* qualities, such as testability, maintainability, extensibility, and scalability, which are embodied in the static structure of the software system.

Given the large, complex and global systems being developed, it becomes essential for organizations to validate and assess software qualities. However, a lack of requirements and artifact traceability often results in situations where validating qualities in post-mortem systems becomes an inherently difficult task.

Patterns and programming guidelines have been promoted for some time to help to detect problem areas and improve various quality aspects of the final software product. The challenge is that these already implemented problems lead to situations where people (1) reuse code with flaws without being aware of it or even worst (2) the same mistakes or bad programming practices are repeated and become recurring patterns. These issues become even more aggravating, in our global software economy, with its collaborative workspaces and diversified knowledge distribution among project stakeholders.

One approach to detect such common coding problems or for enforcing best practice coding patterns are manual code reviews [18]. However, the quality of these reviews will largely depend on the expertise (pattern and guidelines to be validated) of the reviewer and the thoroughness of the review process itself. Source code analysis can be applied to automatically detect many of these common coding problems. Furthermore, these analysis tools can capture relevant domain expertise in their analysis without requiring the tool operator to have the same expertise level as required during manual code reviews. Many analysis techniques have been developed over the years to detect different form of patterns in software, with many of these techniques relying on formal methods and sophisticated program analysis. Most existing source code analysis approaches have failed to address these challenges associated with the new global software economy. As a result, most of the existing analysis tools have remained in technology silos, focusing mainly on improving precision and performance rather than outreaching and integrating with other tools or deal with data at a global scale.

More precisely, we will describe an approach relying on a Semantic Web based automated source code quality analysis tool that can perform the following tasks:

- detect the violations of coding good practices, including security related guidelines
- integrate knowledge reported by external static analysis tools to enrich its own knowledge base
- recover well design patterns such as those described in well known GoF book [17]

- share concepts with knowledge bases of other artefacts such as Source Code Management tools to perform historical quality mining

Research results presented in this thesis have been published in proceedings of two recognized international conferences including:

- Proceedings of the 35th IEEE Conference on Computers, Software, and Applications (COMPSAC 2011) where our complete SE-PAD approach was presented.

- 2nd IEEE International Workshop on Software Engineering for Context Aware Systems and Applications (SECASA 2009) where we introduced the theoretical aspects of a contextual approach to security pattern detections based on Semantic Web technologies.

## 1.1 Contributions

The objective of our research is to provide a novel approach that takes advantage of Semantic Web technologies to represent software artifacts and related knowledge resources. The research builds the basis (1) to integrate and eliminate some existing technology and knowledge silos found in the software engineering community. (2) Taking advantage of Semantic Web technologies to support both, implicit and explicit pattern detection to support the use of quality guidelines. (3) Develop a prototype (SE-PAD) as a proof of concept that takes advantage of the Semantic Web to support both knowledge integration and pattern detection. (4) Demonstrate through several motivating examples and case studies, the flexibility and applicability of our approach. Figure 1-1

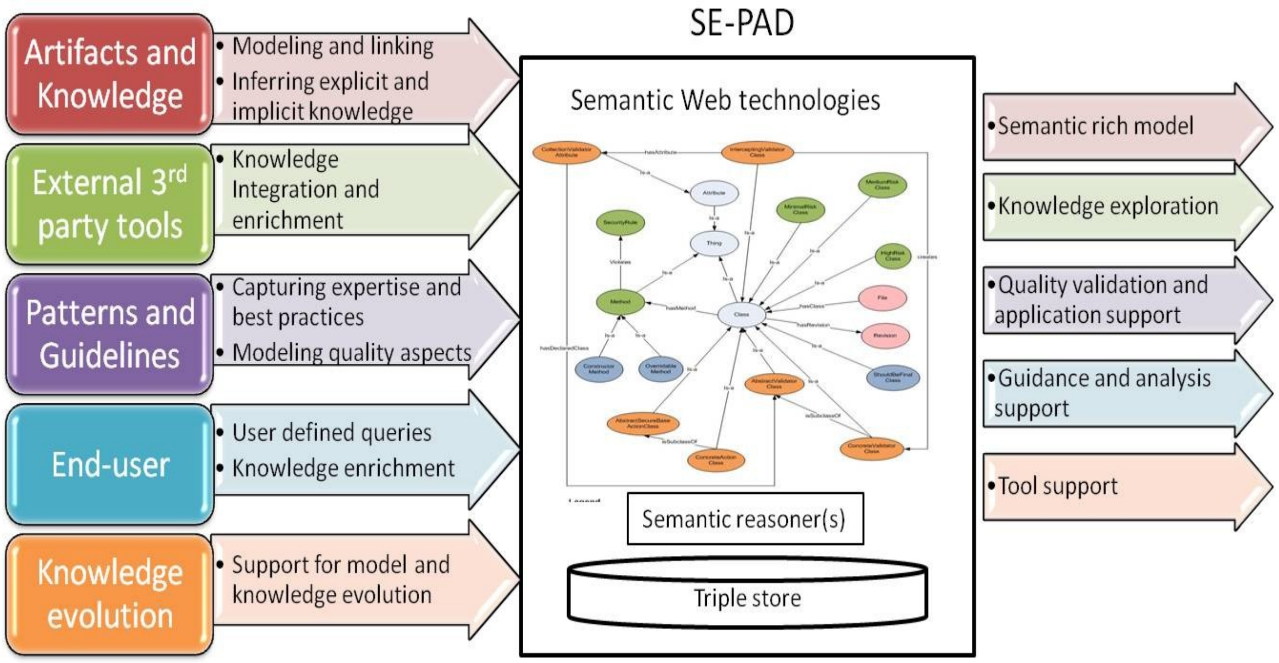


**Figure 1-1: High level overview of SE-PAD**

## 1.2 Thesis Outline

Section 2 presents the research hypothesis and goals are exposed. The relevant theoretical background on which this research is based is presented in section 3, followed by the description of the approach in section 4. Case studies are detailed in section 5 followed by a discussion which includes the Open World Assumption and threats to validity in section 6. Finally, section 7 concludes the thesis and discusses future work.

# 2. Quality Validation through Pattern Detection – a Semantic Web Perspective

## 2.1 Motivation

Nowadays, driven by a globalization of economical activities [35], software is being more and more developed in a global and distributed fashion. Teams are geographically scattered across the globe: they live in different time zones, use different languages, have different cultural background, etc. Not only does this pose organizational, cultural and technical challenges but software development has to adapt to the generally increased complexity of such an environment to maintain an acceptable level of quality.

At the same time, relevant knowledge is being distributed across multiple resources, making the assessment and maintenance of the quality of a system inherently more difficult. In this thesis, we address some of these challenges for the next generation of software engineering quality validation tools, the need to unify quality patterns with system engineering and models. We in particular focus on the second challenge, the integration of resources and knowledge related to patterns within a common Ontological representation. We introduce our SE-PAD tool implementation, to illustrate how Semantic Web technologies can support the integration of knowledge resources at various abstraction and semantic levels. We also show how the Semantic Web can provide the foundation for a knowledge base that supports the extension of new resources and patterns.

A key motivation for our approach is to guide maintainers and developers to validate and ensure that source code meets certain qualities. Supporting global software development processes with artefacts and knowledge resources distributed across organization boundaries and systems will require the modeling and integration of these knowledge resources.

Quality software increases its stakeholders trust and is perceived by many as an indicator of improved evolvability and lower maintenance cost. Semantic information represents a basis for the validation of quality aspects in source code and other resources. We argue that the Semantic Web can provide the required technologies to create semantic rich models unifying knowledge resources and enabling semantic rich forms of source code analysis [39, 40, 48].

## 2.2 Goals and Requirements

The main goal of our research is to provide a novel approach that takes advantage of Semantic Web technologies to represent software artefacts and related knowledge resources in order to support the assessment of quality aspects of post-mortem systems in a distributed and global setting. What follows details how each requirement is linked to its respective sub-goal and consequently to the main goal, as summarized in table 3-1.

| **Main Goal** | **Sub-Goal** | **Requirement** |
|---|---|---|
| Support the assessment of quality aspects of post-mortem systems in a global setting | Knowledge dissemination | (R1) Bridge information silos |
| | Knowledge modeling | (R2) Retrieve implicit and explicit knowledge |
| | Quality validation | (R3) Establish trustworthiness |
| | Knowledge enrichment | (R4) Model extendibility |

**Table 2-1: Research goal, sub-goals, and related requirements**

**Requirement #1**: **Bridging information silos (R1)**

Maintenance activities are often performed across organizational boundaries with knowledge and expertise being distributed across these resources. In order to avoid the creation of information silos in heterogeneous development environments, knowledge dissemination and integration among stakeholders or resources has to become an essential part of global software system.

**Requirement #2**: **Retrieve implicit and explicit knowledge (R2)**

From a programmer perspective, locating and extracting relevant knowledge and resources becomes a major challenge. In order to support programmers and maintainers in their current work context, new modeling techniques and representations have to be applied to support both explicit and implicit knowledge retrieval. In a global and distributed development environment, developers are often geographically separated and located in different time zones, making communication difficult. The ability to automatically retrieve implicit and explicit knowledge becomes crucial to achieve many software engineering tasks, such as program comprehension.

**Requirement #3: Establishing Trustworthiness** (R3)

Establishing trustworthiness in the quality of a system requires maintainers to apply organization and application domain specific processes and activities to document that an application meets or exceeds the expected quality. Validating system qualities creates trustworthiness, by ensuring that that various patterns (e.g. design and security patterns) have been respected and good programming practices and guidelines have not been violated.

**Requirement #4**: **Model extendibility (R4)**

Pattern and knowledge base have to be extendible in order to detect new patterns and to ensure the future quality of systems. There is a need to support knowledge enrichment in the form of modeling and integrating new patterns and guidelines as part of an existing and global knowledge base.

## 2.3 Research Hypothesis

We argue that it is possible to build an Ontology based prototype that can support the four requirements described in the previous section, and therefore address the main research goal of supporting the assessment of quality aspects of post-mortem systems by its associated research sub-goals. Consequently, the hypothesis will hold if and only if our SE-PAD prototype can provide support for all four requirements.

As discussed earlier in section 2, SE-PAD extends the SOM Ontology [4, 5] with new concepts and relationships to form a semantically richer Ontology. Once populated with source code and external tools quality related information, the knowledge base can be queried and enriched. Users can introduce additional concepts and relationships or share existing concepts with other Ontologies through Ontology alignment [56].

Capturing programming expertise and best practices is a well recognized research and application domain. Many forms of patterns (e.g. security, design, guidelines) and programming guidelines have been established describing benefits, limitations and their potential application contexts. The objective of our research is not to compete with existing specialized tools. Rather, we see SE-PAD as a complementary, knowledge integration approach. Furthermore, given the ongoing globalization of software development processes, we do believe that a standardized representation using Semantic Web technologies will become an important enabling factor. It allows for information and knowledge while providing supporting technology infrastructures.

A key objective of SE-PAD is to provide a flexible approach that allows queries to be applied either in a standalone fashion or as  embedded within an IDE (e.g. Eclipse [73]), to retrieve different type of source code related knowledge: good practices violations, design pattern implementations, code metrics, module usages, etc.

# 3. Background

This chapter reviews background literature relevant to the research presented in this thesis. The topics include Semantic Web Technologies and Ontologies, software design and inspection, as well as code quality.

## 3.1 Ontologies and Semantic Web Technologies

This section examines features of Semantic Web technologies, going from the most trivial (RDF) to the more sophisticated (OWL-DL). Both their querying and reasoning features will also be discussed.

The Semantic Web was originally introduced to organize knowledge available on web sites to make it interpretable by machines [69]. As with any information intensive systems and similarly to those backed by relational databases, Semantic Web technologies involve modeling formalization. Whereas entity-relationship models are often used to represent database information, the Semantic Web paradigm relies on Ontologies.

More formally, Ontologies consist of graphs whose nodes are linked by named and directed vertices representing relationships. Nodes may symbolize concrete Individuals or Classes. To compare with the Set theory, Ontology Classes equate to Sets and Ontology Individuals equate to items belonging to zero or more of these sets.

Nodes in Ontologies are linked by means of relationships. They may link individuals amongst themselves or to Classes to explicit an Individual's types. In the latter case, a pre-determined relationship exists and is systematically involved to make an Individual's types explicit.

### 3.1.1 RDF

The Resource Description Framework (RDF) consists of a set of W3C specifications which were initially introduced by Ramanathan V. Guha [51] to implement a metadata data model for the Web. Technically, RDF models typically contain a number of directed sub-graphs. Figure 2-1 shows a simple graph linking node A to node B through the named edge C.

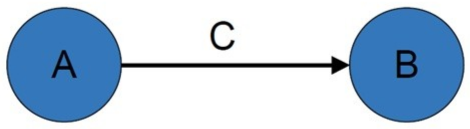


**Figure 3-1: A simple semantic network**

Such graphs are typically represented in the form of triples, which may formally be denoted as:

$$\tau =< A, B, C >$$

In RDF, the graph semantics has been enhanced by defining the participants in a triple. For instance, using the previous example, node A is called the Subject, node B the Object and the edge C is a Relationship. RDF models also have the particularity of identifying its elements with Uniform Resources Identifiers (URI), which are easily processed by a computer program. Subjects and Relationships must be URIs whereas Objects can be either a URI or a literal such as a String, an Integer, etc. For instance, the previous triple can be expressed as follows:

```
τ =<http://www.example.com/test#A,
http://www.example.com/test#C,
http://www.example.com/test#B>
```

RDF injects larger semantics to semantic networks by predefining richer axioms to construct Ontologies. One of the most notables is the `rdf:Property` axiom which allows Ontology designers to create relationships bearing specific meanings. In the previous example, the Relationship C could be replaced by "hasChild", the subject A by "Julius Caesar" and the object B by "Brutus" to model the parent relationship between Julius Caesar and his son Brutus.

### 3.1.2 RDFS

RDFS extends RDF by adding more predefined constructs. Classes are introduced and with them, further semantics including sub/super-classing, inheritance, etc. Relationships are enhanced through additional restrictions, like *domain/range* restrictions and new properties, such as the Sub/Super properties. RDFS also introduces a clearer separation between the two main constituents of an Ontology: the *t-box* is composed of domain Concepts and Relationships whereas Instances, or *a-box*, refers to concrete individuals whose type (*class*) and possible relationships are defined in the *t-box*. To recall the set theory again, the set of Sets equates to the *t-box*, the *a-box* to the available items and their union, the Ontology, to the set theory's Universe.

### 3.1.3 OWL-DL

OWL-DL[13] increases Ontologies expressivity by adding a rich set of semantics to RDF(S). However, contrarily to the pre-existing Semantic Web architecture, which is based on the Semantic Web Stack [70, 71], it does not so by building a new layer through the extension of RDFS. Instead, OWL combines RDFS with Description Logics (DL) [53] by borrowing its XML syntax and style to define rich, formal DL-based class restrictions, hence giving birth to the decidable language OWL-DL. Here are a few examples of the semantic constructs introduced in OWL-DL [13, 69]:

1. **Local scope of properties**

   The `rdfs:range` statement defines the range of a relationship for all the classes of a given Ontology. For example, defining the range of the relationship **eats** as **plants** in RDFS means that everything eaten is a plant whereas OWL-DL allows restrictions to be declared for some classes only. It would then be possible to state that cows eat plants while other animals eat meat.

2. **Disjointness of classes**

   RDFS only allows for subclass relationships amongst classes. Classes in OWL-DL can also be defined as disjointed from one another, as having no intersection so that no individuals can be part of both. For instance, the class **male** would be modeled as disjoint from the class **female**.

3. **Boolean combination of classes**

   OWL-DL supports a combination of first order logic and set operations by allowing classes to be combined with other classes using union, intersection, and complement operators. For example, **drivers** could be modeled as the union of **car_driver** and **bus_driver**.

4. **Cardinality restrictions**

   Restrictions in OWL-DL can include the number of range items a property should or must bear. For instance, a class **car** may restrict the **has_wheels** property so that the cardinality is equal or greater than 4.

5. **Special characteristic of properties**

   In OWL-DL, properties may have special features. Taking for example the various familial relationships, transitivity is illustrated by **is_ancestor_of** and inversion by **has_parent/ is_parent_of**.

### 3.1.4 Reasoning

Reasoners process OWL-DL Ontologies to explicit facts implicitly presented in the *a-box* [14] based on DL formal class restrictions or in other words, to perform automated inferences. More specifically, this research makes use of the following features offered by typical reasoners [15, 52]:

1. **Computation of reverse properties**

   Assume X and Y are sets and L is a relation from X to Y. Then, the reverse relation of L, $L^{-1}$, is formally defined in first order logic as:

   $$L^{-1} = \{(y, x) \in Y \times X \mid (x, y) \in L\}$$

   In Ontologies relationships are represented by triples and they can be declared as the reverse of another one. As an example, suppose the following triple exists in an Ontology's knowledge base:

$$\tau = <A,C,B>$$

Suppose also that Z is a property that was defined as the reverse property of C. In that case, the reasoner can infer the following triple:

$$\tau' = <B,Z,A>$$

2. **Computation of transitive properties**

Assume X is a set and R a relation from X to X. The transitivity of R is formally defined in first order logic as:

$$\forall\, x, y, z \in X : (xRy \wedge yRz \Rightarrow xRz)$$

Suppose an OWL knowledge base contains the following two triples:

$$\tau = <A,C,B>\,,\ \tau' = <B,C,D>$$

If C was labelled as a transitive property, the reasoner can infer the following triple:

$$\tau'' = <A,C,D>$$

3. **Computation of sub-properties**

Assume X is a set and suppose R and S are relationships over X. In first order logic, sub-properties can be expressed as follows:

$$R \sqsubseteq S \Leftrightarrow (\forall(x,y) | (x \in X) \wedge (y \in X) : R(x,y) \Longrightarrow S(x,y))$$

Suppose the following triple exists in an Ontology's knowledge base:

$$\tau = <A,C,B>$$

If C was labelled as a sub-property of D, the reasoner can infer the following triple:

$$\tau' = <A,D,B>$$

**4. Classification**

An OWL-DL reasoner calculates the subclass mapping amongst all identified class in an Ontology's *t-box* to infer the entire class and subclasses hierarchy of the knowledge base. During the classification process, the reasoner can detect inconsistencies in the hierarchy. A class is inconsistent if it is restricted in such a way that it cannot have any instances. For example, a class restricted for humans that are male and not male at the same time will always remain empty, and therefore is considered inconsistent. In first order logic, an inconsistent set R restricted along the relationship S can be represented by the following expression:

$$x \in R \Leftrightarrow \{\exists y \mid S(x,y) \land \neg S(x,y)\}$$

**5. Realization**

Once a hierarchy is computed, it becomes possible to infer all the types to which an Ontology's individuals (*a-box*) belong. The realization process determines the most precise classes to which the individuals base belong by relying namely on the computation of sub-properties. It then becomes possible to determine all the classes an individual is part of through the hierarchy computed in the classification process. As an example, a reasoner can compute the following implication, based on the formal definition of sub-properties:

$$R(x,y) \wedge (R \sqsubseteq S) \Rightarrow S(x,y)$$

### 3.1.5 SPARQL queries

SPARQL[16] is a query language for RDF, i.e. a language to query triple stores. A query consists of triple patterns, conjunctions, disjunctions, and optional patterns. Here is a SPARQL query example. It queries a fictitious Ontology for all the cities and states in the whole Asia, except China.

```
SELECT
     ?city ?state
WHERE {
     ?x   name ?city;
          isCityOf ?y.
     ?y   stateName ?state;
          isInRegion "Asia".
}
Filter ?state != "China".
```

Each SPARQL `SELECT` query includes an ensemble of ordered sections. A query begins with prefix definitions; next a `SELECT` section describes which variables will be listed in the results. The following `WHERE` clauses describe the graph patterns the results are expected to match, including the variables defined by the `SELECT` clause (e.g. the statements contained within the `WHERE` clause brackets in the example above).

The next section of a SPARQL query is where solution modifiers like `FILTER` are applied. The `FILTER` keyword limits a query's results by forcing constraints on the values assigned to the variables defined in the `SELECT` section. Constraints are implemented by logical expressions that result in Boolean. For example, a query returning a set of string values could be modified with a filter to return only the ones matching a specific regular expression. In the last example, the `FILTER` statement is used to remove all cities in China from the results.

### 3.1.6 Putting it all together

The Semantic Web encompasses many technologies. The following directed and named graph shows how we take advantage of the different Semantic Web technologies in our research.

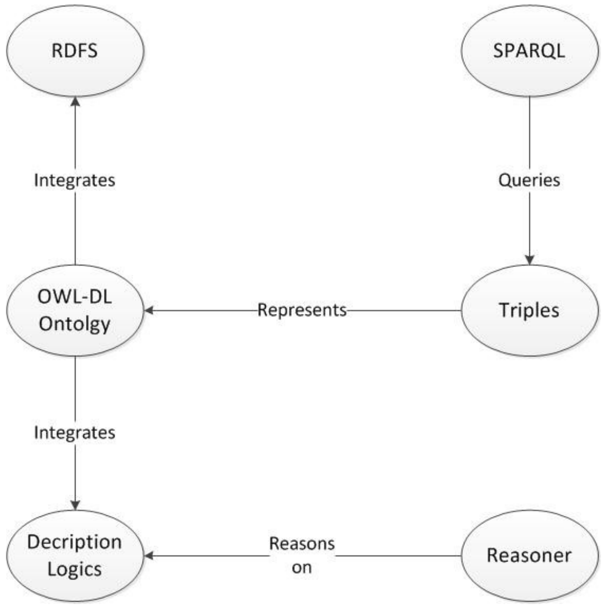


**Figure 3-2: Semantic Web technologies working together**

The most important component is the OWL-DL Ontology. It combines elements of RDFS (e.g. XML syntax, notions of Classes and Relationships) with Description Logics to create the powerful and expressive language OWL-DL. During the realization process, a reasoner is applied on instances of OWL-DL Ontologies to infer new facts based on the Description Logics they contain.

Everything composing a knowledge base, including the new facts inferred by the reasoner, is materialized by triples as discussed in section 2.1.1. SPARQL queries are ultimately applied to the resulting set of triples of an Ontology, e.g. the *a-box*, in order to retrieve the inferred knowledge.

## 3.2 Software Patterns and Quality Inspection

Software patterns are reusable solutions applicable to repeatable problems in software engineering. Since such solutions are reputed as valuable and working once adapted to a specific context and properly implemented, their presence or absence directly affects the overall software quality. Patterns have been defined for different software abstraction levels such as architecture, design and implementation. Table 2-1 presents a few well known patterns for each abstraction level [17, 24, 31].

| **Abstraction Level** | **Pattern example** |
|---|---|
| Architecture patterns | Pipe and Filters<br>Peer to Peer<br>Client-Server<br>… |
| Design patterns | Strategy<br>Composite<br>Adaptor<br>Decorator<br>… |
| Implementation patterns | Naming conventions<br>Exception processing<br>Secure coding<br>… |

**Table 3-1: Examples of software patterns**

Architectural patterns will not be discussed any further as they are outside the scope of this research. They are presented as examples only. Design patterns are best known from the GoF book [17] and capture reoccurring design level solutions. Implementation patterns are directly concerned with code constructs, most of the time language specific.

### 3.2.1 Software Quality Inspection

Quality inspection refers to "examining a product by following a prescribed, systematic process that is intended to determine whether or not the product is fit for its intended use" [18]. Inspections often occur on items of a product at their exit the production lines. Then, a statistically representative lot is collected and analyzed.

In the Software Engineering domain, the inspection principle relates to the detailed and organized examination of a program's source code. Similarly to classic inspections, the goal of software inspection is to assess overall quality. They are defined in [54] as an "approach that involves a well-defined and disciplined process in which qualified personnel analyse a software product using a reading technique for the purpose of detecting defects".

They are typically performed before new code goes in production in order, for example, to prevent side-effects in the case of a modification performed in the maintenance phase. Inspections can also be applied in the development phase to prevent defects which would be costly to fix post-mortem and to maintain a high level of quality during the whole life-cycle.

Inspections can be divided in three types, according to the granularity factor:

- Functional inspection (coarse grained)
- Design inspection (medium grained)
- Implementation inspection (fine grained)

### 3.2.2 Functional Inspections

Functional modification, the modification of an application feature or the addition of a new feature, is the main type of activity occurring during the maintenance phase of the software lifecycle [22]. Such changes are significant because application features involve many software modules, as illustrated in the software abstraction scale (figure 2-3).

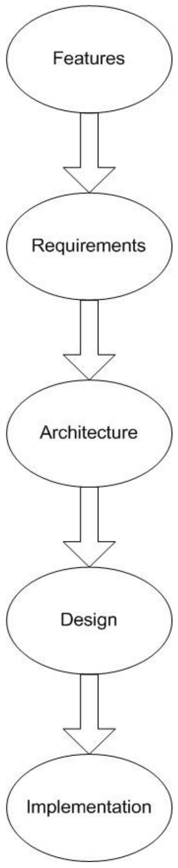


**Figure 3-3: Software abstraction scale**

Features are abstract requirements, which constitute the first step down the scale. Then, requirements abstract architectural components, which abstract design components which in turn abstract code or implementation components. Consequently, any functional change may potentially involve an exponential number of modifications to many abstraction layers of a software application.

Functional inspections are performed prior to feature modifications, by applying impact analysis to determine which software components might be affected by the modification. The complexity of the task is such that it is hard to automate [47]. Human intervention is often necessary to namely create the necessary traceability links [46] amongst the components across the abstraction scale.

### 3.2.3 Design Inspection

At the design level, patterns have been promoted as a good way to ensure software quality. Bushman et al [24] have the following view of design patterns:

> "*A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.*"

In other words, design patterns are object-oriented elements (classes, methods and attributes) organized to implement a solution to a recurring software problem. The key idea is reusability: patterns are to be applied in a specific context and can be reused whenever an appropriate context presents itself.

Design Patterns are classified in three categories: structural, behavioural and creational. Structural patterns are solutions applicable to the composition of classes or objects. Behavioural patterns are concerned with the way objects or classes interact, exchange messages and assume various responsibilities. Creational patterns describe solutions involving the instantiation of objects.

Inspections of design patterns can take various shapes in terms of software quality:

- applicability assessment: inspect whether a pattern is appropriately applied and fits the context
- implementation assessment: inspect whether a pattern is properly implemented, according to the theoretical definition
- impact assessment: in the case of a change performed in the maintenance phase, the inspection should ensure that the modification does not break a previously implemented pattern

Applicability [11] can be inspected by assessing the context of a specific component. If it contains a design pattern, its context must be similar to the one described in the reference of the pattern. For example, suppose a component X handles files and folders organized in a tree-like structure. This context is very similar to the one used to exemplify the *Composite* pattern [17]. Consequently, the inspection should indicate that the context of X is suitable for the Composite design pattern.

Implementation assessment requires that a sound documentation of the component under inspection is available to the code reviewer. When it is established that the component is supposed to implement a design pattern, the inspection must ensure that:

1. the implemented object oriented structure -e.g. classes, methods, parameters and attributes- match the theoretical pattern definition

2. the implemented pattern constituents bear the responsibilities required by the theoretical definition

3. the pattern constituents have the relationships required by the theoretical definition

The impact assessment involves the same process with the difference that the inspection targets components to be modified, for example, in the case of a maintenance intervention. If one of these components is supposed to implement a design pattern, the inspection must verify that the modification will not break it by performing the three assessments previously described.

### 3.2.4 Inspection of Source Code

#### *3.2.4.1 Static Source Code Analysis*

At the implementation level, software inspection focuses almost exclusively on non-functional requirements pertaining to one or many qualities. Generally speaking, these qualities evolve around the maintainability concept, that is "…*the ease with which a software system can be modified*" [28]. More precisely and according to [27], maintainability is defined as:

"*…an integrated measure of many characteristics of software like readability of source code, documentation quality and understandability of the software*".

In other words, the maintainability quality comprises various sub-qualities such as code readability and software understandability.

Since source code is essentially composed of text which will potentially be read by humans, it benefits from being organized and presented in a way that eases its grasping. Readability refers to the physical aspect of source code: its style, its syntax, etc. It is the consequence of the way language elements are organized and presented in text files, making code apprehension variably easy.

Some languages have conventions with respect to readability. For instance, the Java language [23, 32] has a set of guidelines in that matter and tools to support the detection of their violation [29]. Guidelines include recommendations for lines length and spacing, conventions for naming object oriented elements (packages, classes, methods, attributes, etc.) and more.

However, readability goes beyond the aesthetics of text appearance. The concept can be extended to include some elements of understandability, meaning that not only is the code readable but it is understandable. Software understandability is put to the test during the comprehension activity [12], which is when a developer reads code in order to understand a software module. To achieve comprehension, it is necessary to decode the semantics of said module. However nowadays, systems have become large and complex, comprising millions of lines of codes. In these conditions, the comprehension task through manual source code analysis may become overwhelming for a human being. The activity would consequently benefit from a form of automated assistance.

Automated source code analysis is a topic of interest for researchers as well as practitioners. More specifically, the latter often rely on CASE tools to perform static source code analysis [21]. For the Java language, FindBugs [49] and PMD [50] are amongst the most commonly used tools. They parse Java source or compiled code and report units that violate pre-defined rules.

These reports constitute a form of quality evaluation: the more violations a tool detects, the lower the quality of a software module is. Moreover, in [30], the authors show that there is a direct correlation between the bugs reported by static analysis tools and the readability and understandability of a source code unit.

#### *3.2.4.2 Secure Coding Guidelines*

Nowadays, the quality of a system often implies how secure the software is [31, 36, 45, 59, 67]. Indeed, software is present in critical systems in domains such as finance, military, health, transportation, etc., making the assessment of security features more relevant than ever. Security covers a broad range of topics, including hardware, network and software related issues, depending on the threat at stake. For instance, Denial of Service (DoS) [33] vulnerabilities might be handled by a proper network configuration whereas malware or virus vulnerabilities require software solutions.

Security is also important to consider in the daily life of a developer. Some programming languages are more vulnerable to be exploited by an attacker, such as C and C++ that are subject to buffer overflow attacks [34] by taking advantage of the memory manipulation features of the languages. In Java, memory is automatically managed, making buffer overflow attacks harder. However, it does not mean the language is immune to malicious intentions: developers are strongly encouraged to follow the Java Secure Coding Guidelines [31], to reduce the risk of introducing vulnerabilities in their Java programs.

Similarly to the previous description of static source code analysis, one of the goals of this research is to perform an automated detection of security flaws caused by violations of Java Secure Coding Guidelines. In particular we are interested in the following specific guideline [72]:

- **Design APIs to avoid security concerns**: it is preferable to pre-emptively design for security than to reactively apply corrections once vulnerabilities are discovered.

In other words, the guideline states that it is desirable to prevent security issues than to correct them post-mortem. If flaws or bad practices are detected at design time, the overall security of a system should improve.

## 3.3 Ontologies in Software Engineering

Ontologies and Semantic Web technologies have generated a significant amount of interest as a mean to model various aspects of software engineering [1, 2, 3], a knowledge intensive domain [7]. Recent research has explored its application on activities of the development process: from requirements engineering to system verification and validation [8]. Our research focuses on the implementation activity or more precisely, the source code artefact.

### 3.3.1 Knowledge modeling: information silos

Nowadays, organizations like businesses and governments have adopted a decentralized structure. One aspect of this decentralization is the assignment of responsibilities to sub-groups, teams and departments contributing to the organization reaching its goals. This principle can be applied to software engineering. Table 2-2 shows an example of non-overlapping responsibilities related to a typical software development project:

| **SE Activity** | **Responsibilities** |
|---|---|
| Requirements Analysis | Stakeholders identification<br>Users interview<br>Requirements elicitation<br>Use cases<br>… |
| Design | High level architecture<br>Modules modeling<br>Responsibility assignation<br>… |
| Implementation | Interpretation of design documents<br>Coding<br>Unit testing<br>… |
| Testing | Test cases creation<br>Quality assurance assessment<br>… |

**Table 3-2: Software engineering responsibilities**

This decentralization ultimately leads to instances of the "silo syndrome" [6, 9]. Concretely, silos can be thought of as large cylindrical containers. The fact that their content is isolated from the outside world leads to the "information silo" metaphor. It describes the knowledge circulation in a decentralized organization. More formally, information silos are a structural scheme in which knowledge and activities related to an organization's 2..n sub-domains occur mostly exclusively amongst their direct stakeholders, although a sub-domain would benefit from the knowledge generated by one or many of the other 1..n-1 sub-domains to improve the general state of the organization they compose. Figure 2-5 illustrates this definition of information silos.

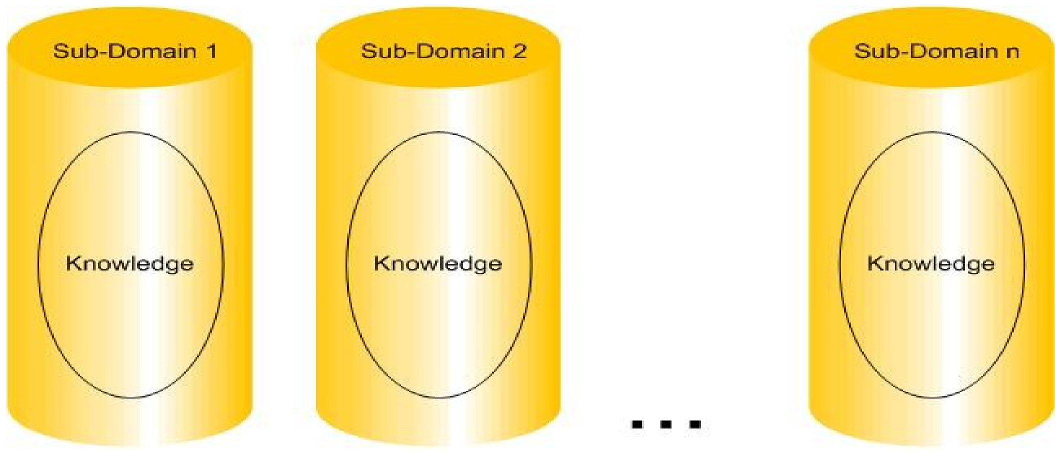


**Figure 3-4: Information Silos for a generic domain of knowledge**

Knowledge related to software engineering shares the characteristics of the knowledge system resultant of a decentralized organization [10] where communication across sub-domains is often not actively promoted. Indeed, software development involves a significant amount of supporting tools and resources covering all steps of any given process. Knowledge relevant to a particular activity is typically dispersed over a range of artefacts in different representational formats and at different abstraction levels. In other words, the software engineering domain spans a wide set of sub-domains.

### 3.3.2 Conceptualization

Ontologies as a modeling technique have been promoted to conceptualize software engineering artefacts, processes, metrics, terminology, development environment, and more. For example, in [55], the authors designed an Ontology for the software maintenance process, including concepts such as developers' skills, development process, their steps and tasks. They performed a post-mortem analysis to gather facts from the stakeholders and improve knowledge on the system to be maintained. This is a knowledge base that maintainers can use, for example, to increase their comprehension of the code.

Developing a structured Ontology for the object-oriented source code artefact implies the analysis of concepts and relationships in this specific area of discourse. In other words, from a software practitioner's point of view, the Ontological model must include concepts and relationships that match those found in the object oriented world.

Our research is based on previous work that has addressed different aspects of such modeling: the SOM Ontology from the Dynamic and Distributed Information Systems Group at the University of Zurich [4, 5]. This Ontology includes the necessary object-oriented concepts such as `Class`, `Method`, `Attribute`, etc. These concepts are linked amongst themselves by relationships like `hasAttribute`, `hasMethod` and `hasFormalParameter`. Table 2-3 shows a few relevant roles and for each, the concepts part of their ranges and the concepts part of their domains.

| **Range Concept** | **Relation** | **Domain Concept** |
|---|---|---|
| Class | hasMethod | Method |
| Class | hasAttibute | Attribute |
| Method | hasFormalParameter | FormalParameter |
| FormalParameter | hasDeclaredClass | Class |
| Attribute | hasDeclaredClass | Class |
| Method | hasDeclaredReturnClass | Class |
| Method | hasLocalVariable | LocalVariable |
| LocalVariable | hasDeclaredClass | Class |

**Table 3-3: SOM Ontology relations, domain and range concepts**

It has to be noted that these general semantics roughly represent a Class defined in UML 2.0 [60] notation although some details are missing, such as access control and visibility information. In other words, it is possible to "translate" the definition of an object-oriented Class in a graph interpretable by Semantic Web technologies enabled tools.

This research incorporates the SOM Ontology and extends it with richer object-oriented concepts and relationships in order to (1) obtain a sounder representation of code constructs and by using more expressive description logics, to (2) optimize the usage of the Ontology to support the use of DL-Reasoners such as Racer [43] or Pellet [52]. These reasoners support the classification of constructs and compute transitive, reflexive and reverse relationships and to (3) fill the semantic gap of the SOM Ontology.

# 4. SE-PAD as a Semantic Web based quality validation platform

In this chapter, we present SE-PAD, a semantic web based tool implantation which supports a pattern based approach to the analysis and assessment of different code qualities. In section 4.1, we introduce the overall architecture of SE-PAD and its main components. Section 4.2 presents how Ontologies are an integral part of SE-PAD. The remaining sections focus on the different quality aspects and how they are supported within our SE-PAD approach.

## 4.1 SE-PAD Architecture

This section provides a general overview of the SE-PAD architecture and its major components (Figure 4-1).

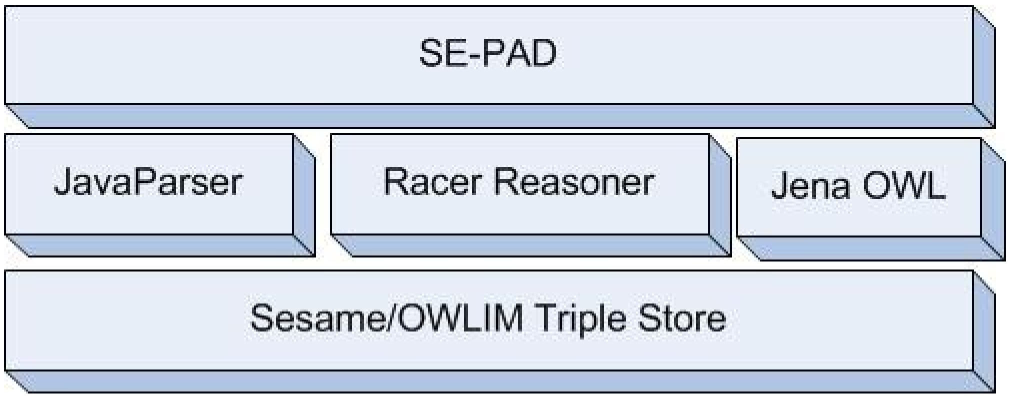


**Figure 4-1: SE-PAD's architecture**

Java Parser [41] is an open source library that parses Java source code to extract the corresponding Abstract Syntax Tree (AST). It supports the Visitor pattern to browse the results, is fairly easy to use and has excellent performances –thousands of classes are parsed in seconds.

SE-PAD uses the results of Java Parser's API to extract facts about Java programs. It converts AST information into Ontology triples through the Jena OWL API [42]. Jena is an open-source framework providing libraries for the creation of Semantic Web applications. Its main role is to help SE-PAD manipulate Ontology triples programmatically.

Racer [43] is an OWL-DL reasoner which performs the realization operation described in section 3.1.4. It has been selected because it is reputed to provide timely support for cardinality restrictions higher than one. It is applied on SE-PAD's Ontology to realize OWL individuals who belong to concepts restricted in this fashion.

Sesame [57] is a RDF triple store. It provides features, among others, for semantic knowledge storage and querying. The OWLIM reasoner plug-in [44] was integrated to the triple store to perform the remainder of the reasoning. It complements RACER because of it is limited to cardinality restrictions of 0 (zero) or 1 (one). It also allows results inferred statements to be included in the queries results.

## 4.2 SE-PAD and the use of Ontologies

As discussed earlier, the Semantic Web and more specifically Ontologies have been widely accepted as a knowledge modeling platform [39, 40, 48]. Ontologies provide direct support for addressing the research requirements stated previously. This section describes the design (*t-box*) and the population mechanism (*a-box*) of SE-PAD's Ontology.

### 4.2.1 Design (*t-box*)

The SE-PAD Ontology is an extension of the SOM Ontology [4, 5]. It takes advantage of the extensibility feature [58] of Semantic Web technologies by adding new OWL data properties, object properties, and concepts in order to achieve a richer set of formal semantics representing the various object-oriented code constructs specific to the Java language. This section describes SE-PAD's *t-box*, the predefined concepts and data/object relationships.

SE-PAD's extended data properties are shown in table 4-1, which also includes their *domain*, *range*[1] and a brief description of their representation in SE-PAD. Here are a few notes on these properties:

- Most properties are based on Java keywords, access or visibility modifiers.
- All properties are functional, e.g. each element of the range must be assigned one and only one value from the domain to create an RDF triple.
- Qualified names are built according to the Java namespace principle, extended to Class attributes and methods, as well as method parameters and local variables.

| **Data Property** | **Domain** | **Range** | **Representation in SE-PAD** |
|---|---|---|---|
| `hasRank` | Parameters | Integer | The rank of a parameter in a method's signature |
| `isAbstract` | Classes<br>Methods | Boolean | Whether the element bears the `abstract` Java keyword or not |
| `isConstructor` | Methods | Boolean | Whether the method is a constructor or not |
| `isDefault` | Classes<br>Methods<br>Attributes | Boolean | Whether the element has the `default` access modifier or not |

[1] The domain and ranges mentioned here are not formally parts of SE-PAD's ontology as it is considered a bad design practice. They are listed here to illustrate with which concepts a relationship's Subject and Object are populated.

| | | | |
|---|---|---|---|
| `isInterface` | Classes | Boolean | Whether the class is an Interface or not |
| `isPrivate` | Classes<br>Methods<br>Attributes | Boolean | Whether the element has the `private` access modifier or not |
| `isFinal` | Classes<br>Methods<br>Attributes | Boolean | Whether the element bears the `abstract` Java keyword or not |
| `isProtected` | Classes<br>Methods<br>Attributes | Boolean | Whether the element has the `protected` access modifier or not |
| `isPublic` | Classes<br>Methods<br>Attributes | Boolean | Whether the element has the `public` access modifier or not |
| `isStatic` | Methods<br>Attributes | Boolean | Whether the element bears the `static` Java keyword or not |
| `Name` | All | String | The element's short name |
| `hasVersion` | Classes | Integer | The source code management system generated revision number |
| `qualifiedName` | All | String | The element's qualified name |

**Table 4-1: SE-PAD's set of extended data properties**

SE-PAD's extended object properties are shown in table 4-2, which also includes their *domain*, *range* and a brief usage description. These properties all have their respective reverse properties which were omitted in this table for the sake of simplicity.

| **Object Property** | **Domain** | **Range** | **Representation in SE-PAD** |
|---|---|---|---|
| `invokes` | Method | Method | A method invokes another method of any scope: the same class, an attribute, a parameter or a local variable |
| `overrides` | Method | Method | A method overrides another, e.g. in a subclass or in an interface implementation. Transitive. |
| `isSubClassOf` | Class | Class | A class subclasses another. Transitive. |
| `implements` | Class | Interface | A class implements an interface. Transitive. |
| `isSubInterfaceOf` | Interface | Interface | An interface subclasses another interface. Transitive. |
| `violates` | Method | Rule violation | A method is reported by a static source code analysis tool to violate a coding rule |
| `assignsValueTo` | Method | Attribute | A method assigns a value to a |

| | | | |
|---|---|---|---|
| | | | class attribute |
| `isAppliedTo` | Method | Local variable, FormalParameter, Attribute | The scope of a method invocation in a method implementation |

**Table 4-2: SE-PAD's set of extended object properties**

SE-PAD's extended concepts are based on the following SOM's Ontology [4, 5] concepts: `Class`, `Method` and `Attribute`. They form the foundation of a BNF-Grammar style concept definition which will be used to form DL classes representing richer object oriented semantics. Tables 4-3, 4-4 and 4-5 respectively detail the how SE-PAD extends SOM `Attribute`, `Method` and `Class` concepts. The related DL is expressed in Manchester-OWL [38] syntax and forms restrictions from which the set of individuals belonging to the concept will be inferred by the reasoner during the realization process.

| **Attribute Concept** | **Description Logics Restriction** | **Representation within SE-PAD** |
|---|---|---|
| `InstanceAttribute` | `Attribute and isStatic value false` | Attributes part of a class' state, as opposed to static attributes |
| `PackageAttribute` | `Attribute and isDefault value true` | Attribute bearing the default access modifier |
| `PrivateAttribute` | `Attribute and isPrivate value true` | Attribute bearing the `private` access modifier |
| `ProtectedAttribute` | `Attribute and isProtected value true` | Attribute bearing the `protected` access modifier |
| `PublicAttribute` | `Attribute and isPublic value true` | Attribute bearing the `public` access modifier |
| `StaticAttribute` | `Attribute and isStatic value true` | Attributes bearing the `static` Java keyword |

**Table 4-3: Extension of SOM's Attribute concept**

| **Method Concept** | **Description Logics Restriction** | **Representation within SE-PAD** |
|---|---|---|
| `AbstractMethod` | `Method and isAbstract value true` | Methods bearing the `abstract` Java keyword |
| `ConcreteMethod` | `Method and isAbstract value false` | Methods not bearing the `abstract` Java keyword |
| `ConstructorMethod` | `Method and isConstructor value` | Class constructors |

| | | |
|---|---|---|
| | true | |
| PackageMethod | Method and isDefault value true | Methods bearing the default access modifier |
| FinalMethod | Method and isFinal value true | Methods bearing the `final` Java keyword |
| InstanceMethod | Method and isStatic value false | Methods not bearing the `static` Java keyword |
| NonFinalMethod | Method and isFinal value false | Methods not bearing the `final` Java keyword |
| NonPrivateMethod | Method and isPrivate value false | Methods not bearing the `private` access modifier |
| PrivateMethod | Method and isPrivate value true | Methods bearing the `private` access modifier |
| ProtectedMethod | Method and isProtected value false | Methods bearing the `protected` access modifier |
| PublicMethod | Method and isPublic value true | Methods bearing the `public` access modifier |
| StaticMethod | Method and isStatic value true | Methods not bearing the `static` Java keyword |

**Table 4-4: Extension of SOM's Method concept**

| **Class Concept** | **Description Logics Restriction** | **Representation within SE-PAD** |
|---|---|---|
| ConcreteClass | Class and isAbstract value false | Classes not bearing the `abstract` Java keyword |
| ExtensibleClass | Class and isFinal value false | Classes not bearing the `final` Java keyword |
| AbstractClass | ExtensibleClass and isAbstract value true | Classes bearing the `abstract` Java keyword |
| FinalClass | Class and isFinal value true | Classes bearing the `final` Java keyword |
| PackageClass | Class and isDefault value true | Classes bearing the default access modifier |
| ProtectedClass | Class and isProtected value true | Classes bearing the `protected` access modifier |
| PublicClass | Class and isFinal value true | Classes bearing the `public` access modifier |

**Table 4-5: Extension of SOM's Class concept**

Figure 4-2 shows a sample of SE-PAD's resulting *t-box*:

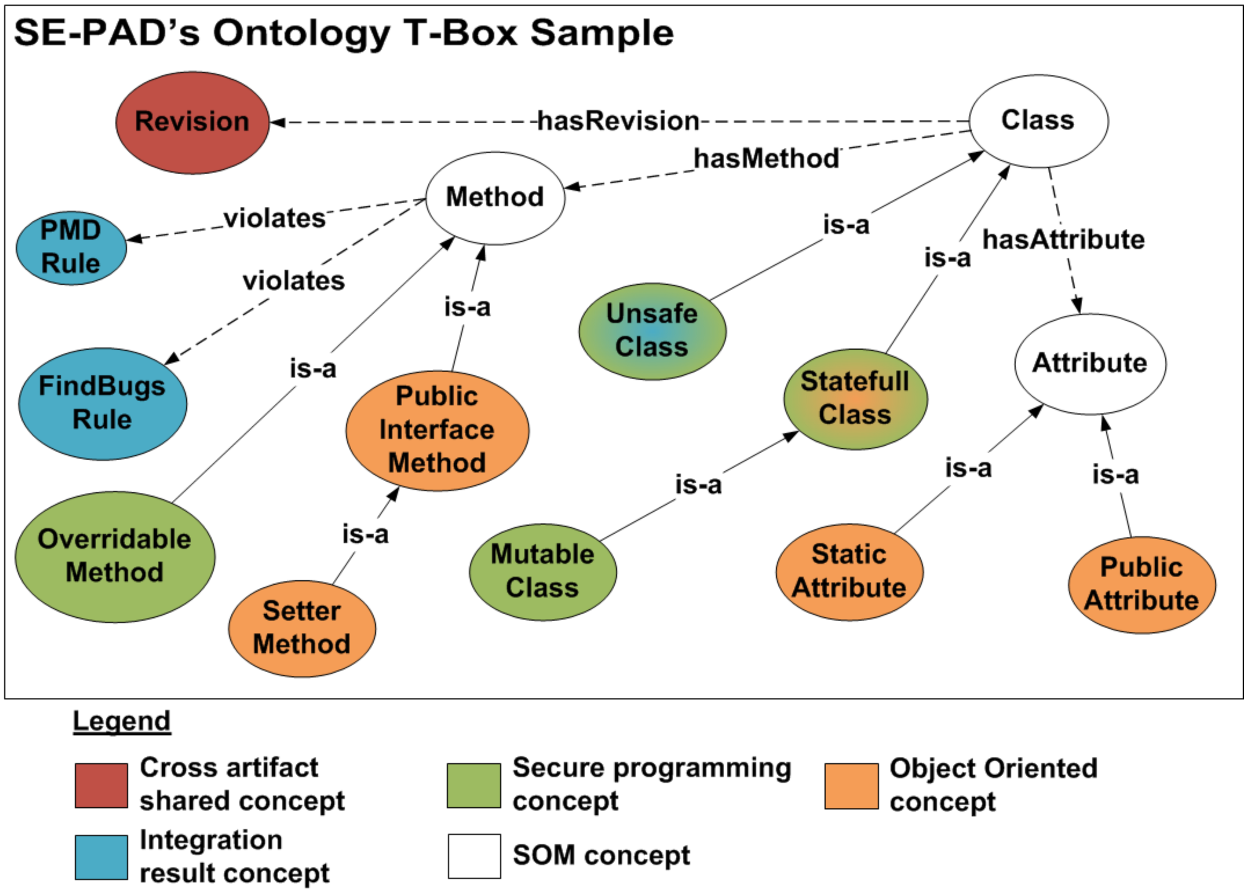


**Figure 4-2: Sample of SE-PAD's *t-box***

### 4.2.2 Population and Realization (*a-box*)

An Ontology's *a-box* refers to its OWL individuals. Recall that in the mathematical set theory, individuals amount to items part of one or more sets. Populating an Ontology means explicitly assigning OWL individuals to Classes through the `rdf:type` relationship. SE-PAD completes its Ontology population in multiple steps:

1- AST Population phase 1

2- AST Population phase 2

3- External Static Analysis tool population

4- Realization

In the first population phase, SE-PAD populates the concepts and relationships along with the data properties introduced earlier. After completion, the Ontology's content is equivalent to a Java application's set of classes typically described in a UML class diagram. Figure 4-2 shows an example of SE-PAD's resulting Ontology after the class `FooBar` of package `foo` has been parsed.

The `rdf:type` relationship as well as part of an individual's URI were omitted for the sake of readability. For instance, according to figure 4-2, the class attribute `attr1` would be the subject of the following triple, added to SE-PAD's Ontology:

```
<a_foo.FooBar.attri1, rdf:type, Attribute>
```

Individuals URI naming convention is as follows:

- the prefix is built by concatenating:
  - the Ontology prefix ending with a pound sign (e.g. `"http://aseg.sepad.org/argoUML#"`)
  - the acronym of the individual's RDF type (e.g. "a" for `Attributes`, "fp" for `FormalParameters`, "c" for `Classes`, etc.)
  - the underscore character ("_")
- the suffix is built with the element's Java qualified and unique name

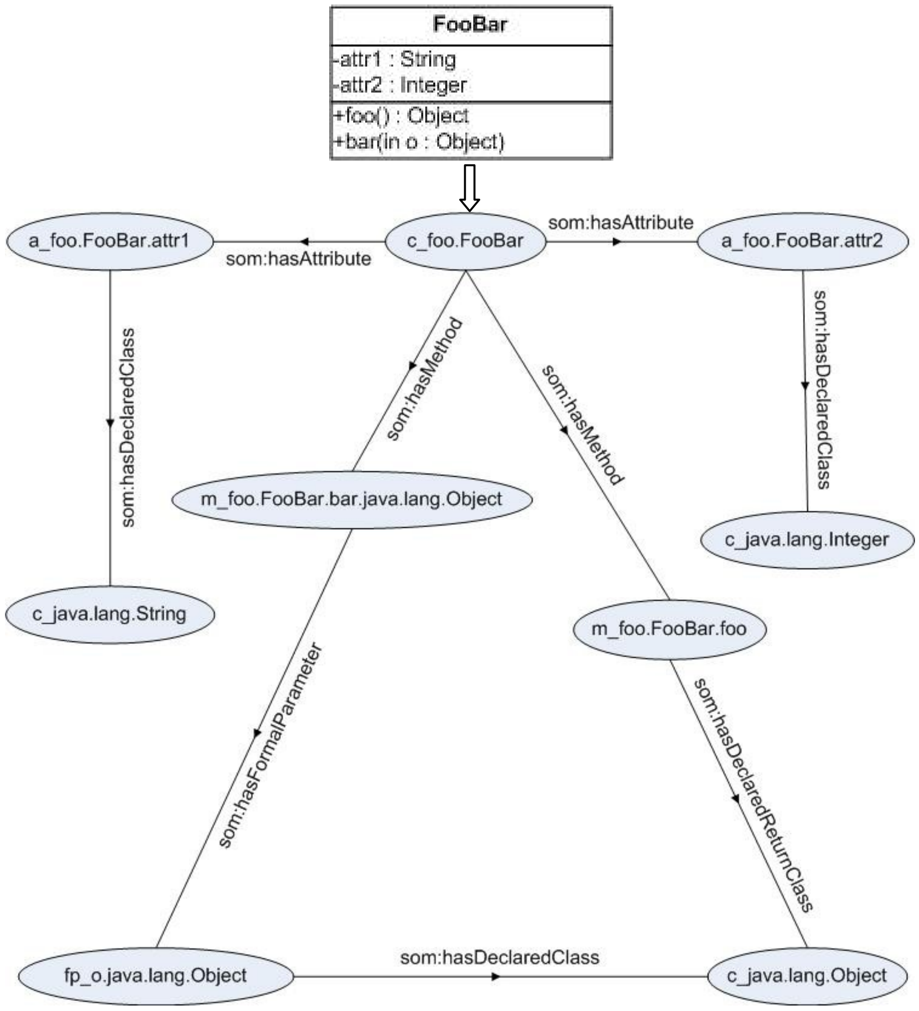


**Figure 4-3: SE-PAD's partial resulting Ontology example**

During the second population phase, SE- PAD re-parses the AST but this time it focuses on method implementations, in order to extract the relationships shown in table 4-2. For instance, to populate the `INVOKES` relationship which models method calls within a method implementation, SE-PAD queries its populated Ontology from the first population phase and determines the individuals representing the caller and the called method. Once identified, SE-PAD completes the triple with the `INVOKES` relationship.

SE-PAD currently supports the integration of results from two major static Java source code analysis tools: PMD [49] and FindBugs [50]. In order to limit the scope of the thesis, we focused on the detection of security concerns that can be detected by these tools. Security concern results obtained from both of these tools were then analyzed, categorized and added explicitly to SE-PAD's *a-box* as individuals belonging to the class hierarchies shown in figure 4-3. For a detailed description of the security concerns and the rules used to detect them, we refer the reader to the respective tool documentation [49, 50].

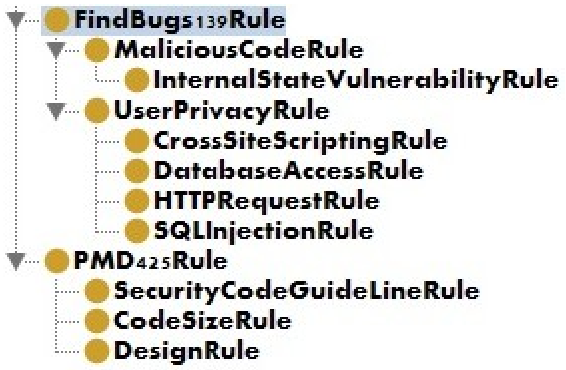


**Figure 4-4: SE-PAD's support for external tool rules**

SE-PAD identifies the OWL individual representing the Java Method M violating a rule and creates a new triple with M as a subject, `violates` as a property and the pre-populated violated rule individual as an object.

As part of the population process, new *a-box* elements' `rdf:type` are inferred based on the *t-box*'s description logics and the use of OWL reasoners (in our case Racer and OWLIM [44]). More specifically, after the population phase, the Ontology is uploaded to an OWLIM enabled Sesame triple store –OWLIM is a semantic reasoner implementation that is bundled as a Sesame plug-in. OWLIM realizes SE-PAD's Ontology upon reception and by doing so, populates the richer concepts presented in tables 4-3, 4-4 and 4-5.

## 4.3 A Semantic Web Approach to Software Quality

Figure 4-4 provides a high-level view of the semantic support provided by SE-PAD for our pattern-based quality validation approach. As shown, our approach focuses on the integration of knowledge and resources, which are essential aspects to eliminate current

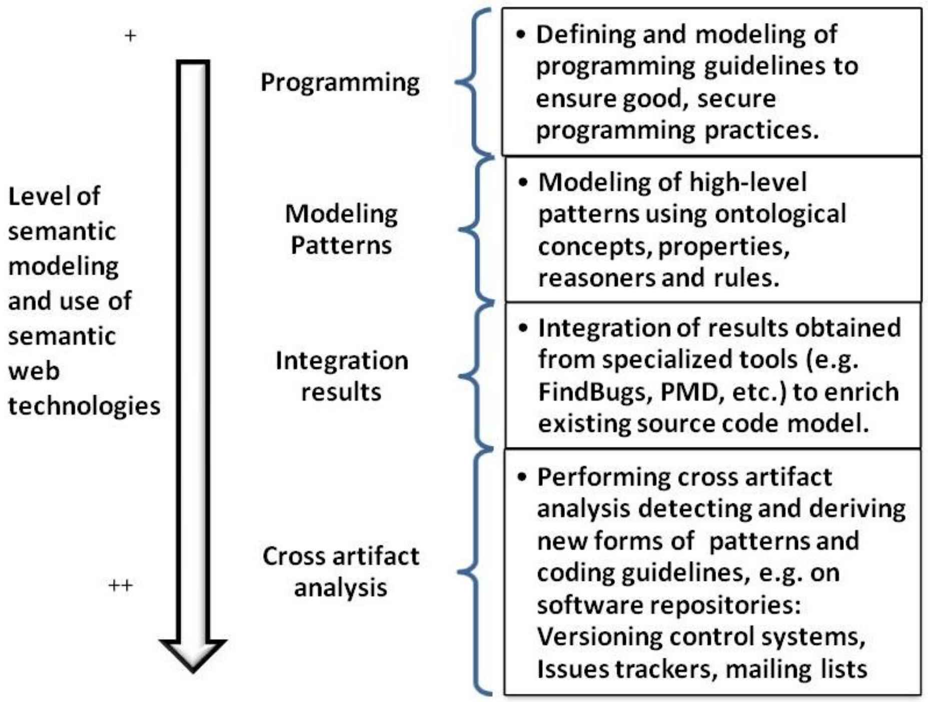

+
++
Level of semantic modeling and use of semantic web technologies
Programming
Modeling Patterns
Integration results
Cross artifact analysis
• Defining and modeling of programming guidelines to ensure good, secure programming practices.
• Modeling of high-level patterns using ontological concepts, properties, reasoners and rules.
• Integration of results obtained from specialized tools (e.g. FindBugs, PMD, etc.) to enrich existing source code model.
• Performing cross artifact analysis detecting and deriving new forms of patterns and coding guidelines, e.g. on software repositories: Versioning control systems, Issues trackers, mailing lists

**Figure 4-5: The semantic levels supported by SE-PAD**

information and analysis tool silos and to support a more global quality validation perspective.

The next sections detail our SE-PAD approach combining semantic analysis and Semantic Web technologies. For the programming and patterns levels, table 4-6 introduces our pattern presentation template. It will be reused for the programming and pattern semantic modeling levels.

| | |
|---|---|
| **Ontology Extension** | A visual representation of the Ontology concepts involved. Shows concepts and concept inheritance through the `Is-A` relationship as well as the `rdf:type` relationship. The following visual notation is used:<br>⬭ \| SOM concepts<br>▨ \| SE-PAD concepts<br>⬬ \| Concepts extending SE-PAD's Ontology to support the current pattern<br>The section will be omitted if the pattern does not require an extension to SE-PAD's Ontology. |
| **OWL-DL involved** | Description logics (DL) used as class restrictions in Manchester-OWL syntax [38]. The concepts are often defined in a recursive BNF grammar style. |
| **SPARQL_Query** | The SPARQL query retrieving relevant knowledge from SE-PAD's extended knowledge base. For improved readability, the required URI prefixes (SPARQL `PREFIX` keyword) will be omitted in the queries, except for the `rdf:type` prefix given its important semantic value. |
| **Role of Semantic Reasoner** | The concepts realized and the relationships computed by the semantic reasoner in the pattern detection process. |
| **Requirements covered** | Lists and explains which requirements are covered by the current pattern detection and by extension, which research sub-goals are consequently reached. |

**Table 4-6: Pattern presentation template**

## 4.4 Programming Guidelines

There exist a significant number of code review and analysis tools that support the detection of good coding practices and guidelines at the source code level [45, 49, 50, 62]. These approaches range from string matching (codifiers) to semantic analysis tools that allow the modeling of these guidelines as queries and rules to be executed over a source code model [49, 63]. Most of these semantic query or rule based approaches use an AST for extracting facts and then store these facts in their tool proprietary models. These models are typically static (XML or relational), restricting the model typically to the information available at design time of the tool.

SE-PAD takes advantage of its semantic rich representation of source code to support the detection of basic violations of coding standards and guidelines, similar to ones supported by existing tools [e.g., 49, 50]. We exemplify this guidelines support by automating the detection of four violations, already supported by either Findbugs or PMD [49, 50].

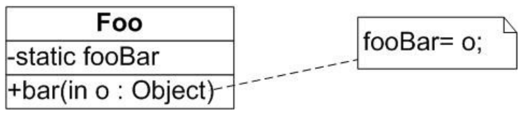
### 4.4.1 General Programming Guidelines

This section describes how SE-PAD is able to support the detection of Java coding guidelines violations. The violations are also detected by Findbugs and were selected to show that SE-PAD offers comparable static analysis features. It should also be pointed out that SE-PAD has the advantage of running on text files whereas Findbugs needs compiled Java bytecode, which means an increased flexibility for SE-PAD. Indeed, Findbugs requires either the context of an IDE or significant human intervention to run, whereas SE-PAD is applicable to any Java project as long as its source code is accessible in a file system.

Users can run the SPARQL query on a project's SE-PAD generated Ontology to assess the state of the application with respect to the bad practice or guideline currently discussed, helping developers perform the required modifications in order to redesign the code.

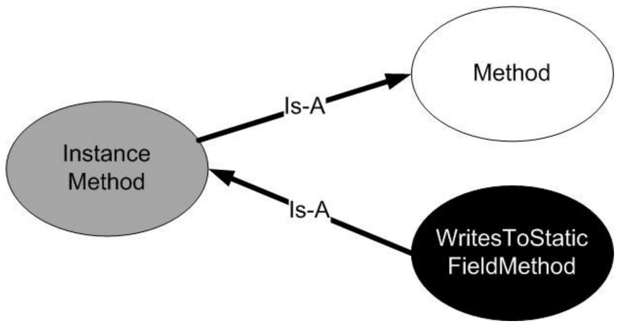
### *4.4.1.1 Guideline: Do not write to static fields from instance methods*

**Pattern Description:**



Since `fooBar` is static, only one instance exists in the Java Virtual Machine (JVM) whereas `Foo` might be instantiated many times. Consequently, `bar()` or any other of Foo's methods has the opportunity to write different values to `fooBar` which can become hazardous to manage in terms of concurrency, amongst other things. FindBug's id for this pattern is: ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD.

**Ontology Extension:**



**OWL-DL involved:**

```
StaticAttribute ≡ Attribute and isStatic value true

InstanceMethod ≡ Method and isStatic value false

WritesToStaticFieldMethod ≡ InstanceMethod and
assignsValueTo some StaticAttribute
```

The first 2 concepts are basic SE-PAD concepts whereas the third one, added by SE-PAD's client to its *t-box*, categorizes the faulty methods with the following restriction: they must not be static and they must assign a value to a static attribute.

**SPARQL Query:**

```
SELECT ?WritesToStaticFieldMethod ?Class WHERE
{
     ?WritesToStaticFieldMethod
     rdf:type WritesToStaticFieldMethod;
     som:isMethodOf ?Class.
}
```

This query returns the list of instance methods assigning values to static attributes and the classes to which they belong.

**Role of Semantic Reasoner**

Computes the reverse relationship `isMethodOf` part of the SPARL query and realizes the individuals belonging to concepts `StaticAttribute`, `InstanceMethod` and `WritesToStaticFieldMethod`.

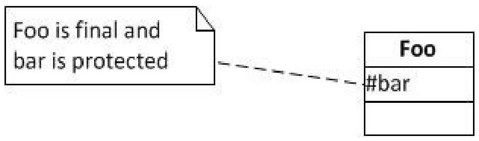
**Requirements covered**

| | |
|---|---|
| **Establish trustworthiness** | This pattern is a bad practice. The fewer instances are reported, higher is the quality of the software analyzed. |
| **Retrieve implicit and explicit knowledge** | The implicit knowledge retrieved by the SPARQL query determines the existence of the pattern in the code and the explicit knowledge is the identification of the software modules implementing it. |
| **Model extendibility** | The concept `WritesToStaticFieldMethod` was created and added to SE-Pad's Ontology to support the detection of this pattern. |

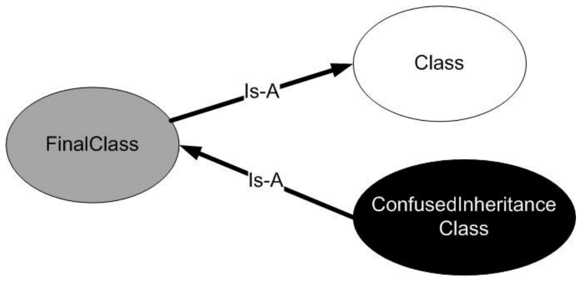
### *4.4.1.2 Guideline: Final classes should not have protected attributes*

**Pattern Description**

A Java class C is marked as `final` (e.g. it cannot be sub-classed) and has a `protected` attribute A. This is inconsistent because the `protected` access modifier makes A accessible throughout C's subclass hierarchy. Since A belongs to C which cannot have a hierarchy, marking the attribute as protected is pointless. FindBug's id for this pattern is: CI_CONFUSED_INHERITANCE.



**Ontology Extension**



**OWL-DL involved**

**`ProtectedAttribute`** `≡ Attribute and isProtected value true`

**`FinalClass`** `≡ Class and isFinal value true`

**`ConfusedInheritanceClass`** `≡ FinalClass and hasAttribute some ProtectedAttribute`

The first two concepts are basic SE-PAD concepts whereas the third one categorizes the faulty classes based on following restriction: the resulting classes must be `final` and they must have at least one `protected` attribute.

**SPARQL Query**

```
SELECT ?ConfusedInheritanceClass WHERE {
     ?ConfusedInheritanceClass
          rdf:type ConfusedInheritanceClass;
}
```

The query returns the list of classes matching the captured sub-classing hierarchy violation.

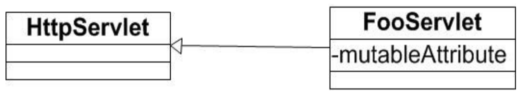
**Role of Semantic Reasoner:** For this guideline, the reasoner realizes the individuals belonging to concepts `ProtectedAttribute`, `FinalClass` and `ConfusedInheritanceClass`.

**Requirements covered:**

| | |
|---|---|
| **Establish trustworthiness** | This pattern is a bad practice. The fewer instances are reported, higher is the quality of the software analyzed. |
| **Retrieve implicit and explicit knowledge** | The implicit knowledge retrieved by the SPARQL query determines the existence of the pattern in the code and the explicit knowledge is the identification of the software modules implementing it. |
| **Model extendibility** | The concept `ConfusedInheritanceClass` was created and added to SE-Pad's Ontology to support the detection of this pattern. |

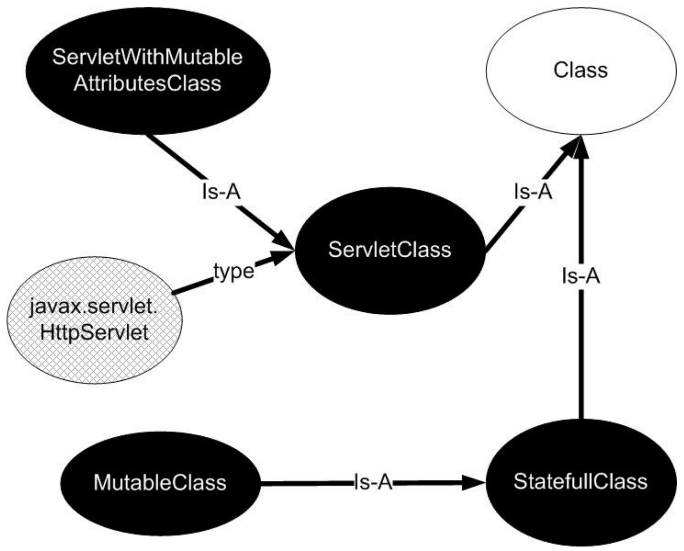
### *4.4.1.3 Guideline: Servlet classes should not have mutable attributes*

**Pattern Description**



A class is a subclass of Java's Servlet (`javax.servlet.http.HttpServlet`) and also has mutable instance variables. This creates race conditions because web containers treat Servlets as singletons -e.g. 1 instance of each servlet class at most at runtime [64]. FindBug's id for this pattern is: MSF_MUTABLE_SERVLET_FIELD.

**Ontology Extension**



`javax.servlet.http.HttpServlet` is an OWL individual assigned at design time to the `ServletClass` concept.

**OWL-DL usage**

The description logics related to the `MutableClass` concept will be described in detail later in this thesis. The other concepts involved in this detection are as follows:

```
ServletClass ≡ Class and isSubclassOf value
javax.servlet.http.HttpServlet
```

**`ServletWithMutableAttributesClass`** `≡ ServletClass and hasAttribute (hasDeclaredClass some MutableClass)`

**SPARQL Query:**

```
SELECT ?ServletWithMutableAttributesClass WHERE {
?ServletWithMutableAttributesClass
     rdf:type ServletWithMutableAttributesClass;
}
```

The query returns the list of Servlet classes having mutable attributes, and therefore violating the "*Servlet classes should not have mutable attributes" guideline.*

**Role of Semantic Reasoner**

- realizes individuals belonging to concepts `MutableClass` and `FinalClass`
- computes the `isSubClassOf` transitive relationship in order to support the detection of classes located at any level of `javax.servlet.http.HttpServlet`'s subclass hierarchy

**Requirements covered**

| | |
|---|---|
| **Establish trustworthiness** | This pattern is a bad practice. The fewer instances are reported, higher is the quality of the software analyzed. |
| **Retrieve implicit and explicit knowledge** | The implicit knowledge retrieved by the SPARQL query determines the existence of the pattern in the code and the explicit knowledge is the identification of the software modules implementing it. |
| **Model extendibility** | The concepts `ServletClass`, `ServletWithMutableAttributesClass`, `StatefullClass` and `MutableClass` were created and added to SE-PAD's Ontology to support the detection of this pattern. |

*4.4.1.4 Guideline: Respect Naming Conventions*

**Pattern Description**

Java follows a mixed case naming convention for methods and attributes whereas CamelCase[65] is recommended for classes. FindBugs rules detect method and attribute names with uppercase first letters and class names with lowercase first letters. SE-PAD provides the same type of detection through SPARQL queries. The specific rules covered are, by their Findbugs denomination: NM_CLASS_NAMING_CONVENTION, NM_FIELD_NAMING_CONVENTION and NM_METHOD_NAMING_CONVENTION.

**SPARQL Queries**

All queries use SPARQL's regex function to filter improperly named Java elements.

This query returns the list of classes whose names have a lowercase first letter:

```
SELECT DISTINCT ?ClassName WHERE
{
   ?Class
     name ?ClassName;
     rdf:type ?Class.
   Filter (regex(?ClassName,"^[a-z]"))
}
```

This query returns the list of methods whose names have an uppercase first letter and the class to which they belong.

```
SELECT DISTINCT ?MethodName ?Class WHERE
{
   ?Method
     name ?MethodName;
     isMethodOf ?Class;
     rdf:type Method.
   Filter (regex(?MethodName,"^[A-Z]"))
}
```

This query returns the list of attributes whose names have an uppercase first letter and the class to which they belong.

```
SELECT DISTINCT ?AttributeName ?Class WHERE
{
  ?Attribute
    name ?AttributeName;
    isAttributeOf ?Class;
    rdf:type Attribute.
  Filter (regex(?AttributeName,"^[A-Z]"))
}
```

**Role of Semantic Reasoner**

Computes the `isAttributeOf` and `isMethodOf` reverse properties used in the queries retrieving attributes and methods.

**Requirements covered**

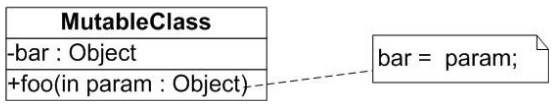
| | |
|---|---|
| **Establish trustworthiness** | These patterns are bad practices. The fewer instances are reported, higher is the quality of the software analyzed. |
| **Retrieve implicit and explicit knowledge** | The implicit knowledge retrieved by the SPARQL query determines the existence of the pattern in the code and the explicit knowledge is the identification of the software modules implementing it. |

## 4.4.2 Secure Coding Guidelines

Nowadays, security as a non-functional requirement is an increasingly important concern for various stakeholders within a software development process, especially given (1) the level of code reuse through third party libraries, frameworks, etc. and (2) that systems increasingly communicate across distributed networks or have communication ports opened on the Internet. This section illustrates how SE-PAD supports security related tasks by identifying violations based on Java's secure coding guidelines [31]. In what follows, we demonstrate the automated detection of two of such coding violating guidelines.

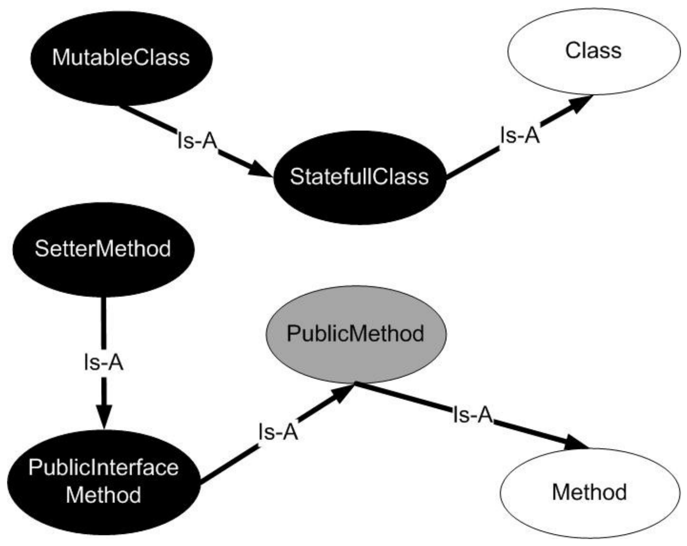
### *4.4.2.1 Coding Violation: Prefer immutable classes*

**Pattern Description**



An immutable class is a class whose state does not change after it is instantiated, as opposed to mutable classes. The volatile nature of mutable classes makes them a hazardous design choice. They force developers to take extra precautions, especially when objects of mutable classes are part of other classes' state, making immutable classes preferable. Consequently, mutable classes can be seen as implementing a coding violation whose detection is desirable.

**Ontology Extension**



**OWL-DL involved**

**`PublicMethod`** `≡ Method and isPublic value true`

**`PublicInstanceMethod`** `≡ PublicMethod and isStatic value false`
**`SetterMethod`** `≡ PublicInstanceMethod and assignsValueTo some Attribute`

**`StateFullClass`** `≡ Class and hasAttribute some InstanceAttribute`

**`MutableClass`** `≡ Class and hasMethod some SetterMethod`

The first concept defines public methods with the `isPublic` data property. The second concept subsumes `PublicMethod` and adds the restriction of not being static to categorize `PublicInstanceMethods`, which require an object instance as an invocation scope contrarily to static class methods. `SetterMethod` specializes methods further by classifying those who assign a value to a class attribute, thus modifies its state. The value assigned can be of any origin: parameter, computation, local variable, result of a method call, etc. Next, `StatefullClass` defines classes who have instance attributes, or a state. Finally, `MutableClass`es are `StatefullClass`es who also have `SetterMethod`s, meaning that their state may change after their creation as the pattern prescribes.

**SPARQL Query**

```
SELECT ?MutableClass WHERE{
?MutableClass rdf:type MutableClass.}
```

The query returns the list of mutable classes ordered by their fully qualified names.

**Role of Semantic Reasoner**

- realizes individuals belonging to concepts `PublicMethod`, `PublicInstanceMethod`, `SetterMethod`, `StateFullClass` and `MutableClass`

Figure 4-5 is a visual representation of the reasoner's work in the case of `MutableClass` realization. The full arrows represent the explicit type assignment performed by SE-PAD whereas the dotted arrows represent the type assignment performed by the reasoner. In that case, `m_foo` is a method part of class `c_Bar` and assigns a value to one of the class variables.

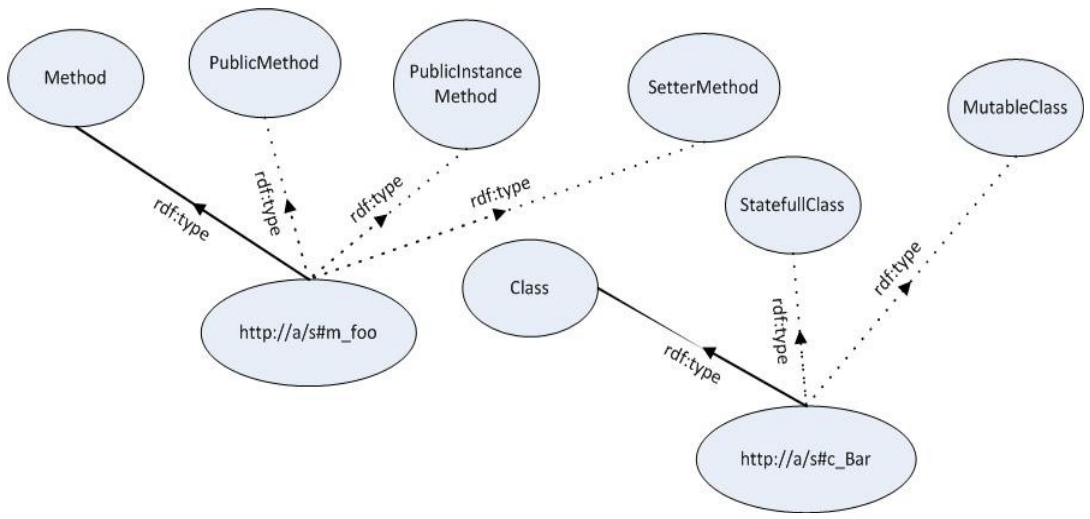


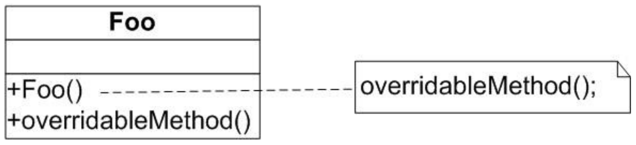
**Figure 4-6: Mutable class realization**

**Requirements covered**

| | |
|---|---|
| **Establish trustworthiness** | Identifying mutable classes can be part of a major software security improvement to enforce secure coding guidelines. Once mutable classes are identified, the inspection of their usage will reveal the need to perform one or many of the following tasks to improve the security of these mutable classes: (1) Make sure mutable classes have a clone (deep copy) operation. (2) If a class has an internal mutable object used as a method output, make sure the method clones (deep copies) the object before returning it. (3) In cases then a class has an internal mutable object, make sure a method assigning one of its parameters to it first performs a clone (deep copy) operation. |
| **Retrieve implicit and explicit knowledge** | The implicit knowledge retrieved by the SPARQL query is used to detect the pattern in the code and the explicit knowledge is the identification of the software modules implementing it. |
| **Model extendibility** | The concepts `SetterMethod`, `PublicInterfaceMethod`, `StatefullClass` and `MutableClass` were created and added to SE-Pad's Ontology to support the detection of this pattern. |

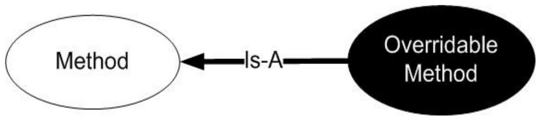
### *4.4.2.2 Prevent constructors from calling overridable methods*

**Pattern Description**



A class has a constructor method that delegates control to an overridable method. It gives eventual subclasses access to a reference to the object being constructed by overriding the overridable method. Attackers could exploit this vulnerability to alter the behaviour of the object being constructed and therefore occurrences of this pattern should be detected.

**Ontology Extension**



**OWL-DL involved**

**OverridableMethod** ≡ `isFinal value false and isPrivate value false and isConstructor value false and isStatic value false`

The central concept is `OverridableMethod`. In Java, a method is overridable if it bears none of the following modifiers: `Private`, `Final`, and `Static`.

**SPARQL Query**

```
SELECT DISTINCT ?MethodName WHERE {
     ?Method rdf:type ConstructorMethod;
          qualifiedName ?MethodName;
          invokes ?OverridableMethod;
          isMethodOf ?Class.
     ?OverridableMethod rdf:type OverridableMethod;
          isMethodOf ?Class.
     ?Class isFinal ?isFinal.
     FILTER (?isFinal=false)
}
```

The query returns the list of classes with constructors calling overridable methods. An additional constraint is included that selected methods cannot be part of a `Final` Java class.

**Role of Semantic Reasoner**

Realizes individuals belonging to concept `OverridableMethod` and computes the transitive relationship `Invokes`, part of the SPARQL query, which puts a method in relationship with a method call within its implementation. Consequently, if a constructor delegates to a non-overridable method delegating to an overridable one, SE-PAD will detect the violation, no matter how far the overridable method call is located in the graph. In other words, if a `private` method delegates to another `private` method, which also delegates to a `private` method and so on, SE-PAD will detect any call to an overridable method within such a chain.

**Requirements covered**

| | |
|---|---|
| **Establish trustworthiness** | These patterns are bad practices. The fewer instances are reported, higher is the quality of the software analyzed. |
| **Retrieve implicit and explicit knowledge** | The implicit knowledge retrieved by the SPARQL query is used to detect the pattern in the code and the explicit knowledge is the identification of the software modules implementing it. |
| **Model extendibility** | The concepts `CallsOverridableMethodConstructor` , `OverridableMethod`, `NonFinalMethod`, `NonPrivateMethod` and `ConstructorMethod` were created and added to SE-Pad's Ontology to support the detection of this pattern. |

## 4.5 Modeling Design Patterns

Patterns [17, 20] represent existing solutions in a structured way and allow knowledge from these existing solutions to be reused. Patterns can speed up the development process by providing tested, proven development paradigms that might often not become visible until later in the implementation or the evolution of a system. Furthermore, they also prevent subtle issues that may cause major problems later on, and they can improve code readability and comprehensibility for programmers familiar with the patterns. Design patterns, the most widely known type of patterns, provide a formal way to document a solution to a design problem. Within our SE-PAD environment, we have formalized a small subset of these GoF [17] patterns - four of the most common patterns. Detected design pattern can convey important design and implementation decision associated with the use of a pattern. This information can provide maintainers with additional insights in software comprehension.

SE-PAD's support for the automated recovery of design patterns meets the following research requirements:

*Establish trustworthiness*: A critical aspect of software maintenance is the risk for the code modification to cause undesirable side effects by introducing new defects. Suppose a maintenance developer has to rewrite code that is part of a design pattern to perform a code fix and the related documentation is either outdated and does not mention the pattern implemented. If the maintainer is otherwise unaware that a pattern is present in the area of the code to be modified, there is a significant risk that the modification breaks the pattern implementation. In other words, there is a need for impact analysis [47] to prevent this from happening. Concretely, an IDE plug-in could detect the file a maintainer has started to edit and query SE-PAD's Ontology to see if a pattern is involved in the area. If so, the user receives a warning detailing the pattern implementation. Then, when the code is committed, the plug-in would rerun the query to validate that no patterns were broken by the developer's intervention.

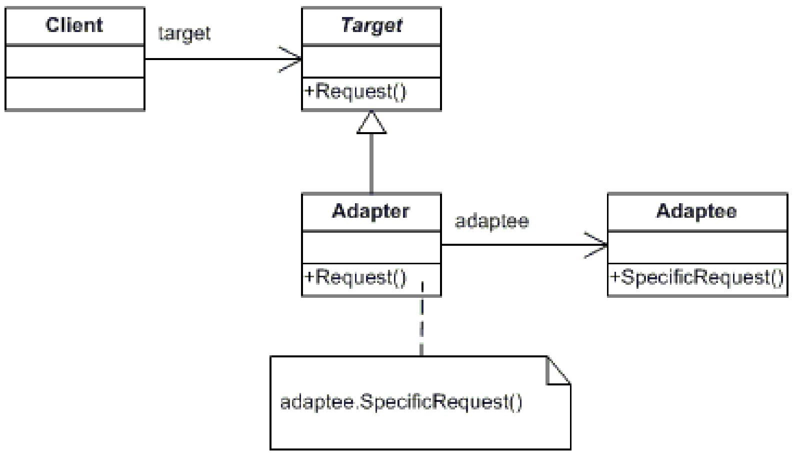
Retrieve implicit and explicit knowledge: Sometimes, developers might implement a GoF pattern without being aware of it, as shown in our preliminary case studies analysis: the template pattern was for instance found without being documented as such. By recovering instances of these patterns in code, SE-PAD can play a significant role in an effort to update typical software engineering documentation such as code comments, design diagrams, wiki pages, etc.

Model extendibility: The design patterns automated recovery is rather SPARQL querying intensive, making heavy use of SE-PAD's basic Ontology. However, the Strategy design pattern recovery required the addition of new concepts.

Bridge Information Silos: The source code and design model artifacts can be considered as information silos. The best proof of this is how often they become out of sync: developers change code but do not update the design documents, making the latter more and more obsolete and useless. Given its reverse engineering nature, automated design pattern recovery helps bridging the source code silo and the software design silo by keeping the knowledge contained both artefacts synchronized. Also reverse engineering and abstracting design patterns from low-level source code representations allows for the recovery of some design related domain knowledge.

### 4.5.1 Adapter Pattern

#### Pattern Description



The *Adapter* pattern's main purpose is to encapsulate the behaviour of a component (the Adaptee) for usage in new contexts. Note that the Client class was purposely left out as it is the pattern's consumer rather than part of the pattern itself.

#### SPARQL Query

The query is based on the UML class diagram of the pattern and details each component as well as their relationships with the other elements:

- the Target class has the Target Request Method in its interface and the Adapter as a subclass
- the Adapter class has the Adapter Request as a method and has an Adaptee in its attributes
- the Adaptee class has a Specific Request as a method and is the declared class of one of the Adapter's attributes

- the Adapter Request delegates to the Adaptee Specific Request and overrides the Target Request Method
- the Adaptee Specific Request is applied to the attribute of the adapter that has the Adaptee as a declared class

```
SELECT DISTINCT

?TargetClassName
?AdapterClassName
?AdapteeClassName
?AdapterSpecificRequestName
?AdapteeSpecificRequestName


WHERE {
?Target
     isInheritedBy ?Adapter;
     hasMethod ?TargetRequestMethod;
     shortName ?TargetClassName.

?Adapter
     hasMethod ?AdapterRequest;
     hasAttribute ?AdapteeAttribute;
     shortName ?AdapterClassName.

?Adaptee
     isDeclaredClassOf ?AdapteeAttribute;
     hasMethod ?AdapteeSpecificRequest;
     shortName ?AdapteeClassName.

?AdapterRequest
     invokes ?AdapteeSpecificRequest;
     overrides ?TargetRequestMethod;
     name ?AdapterSpecificRequestName.

?AdapteeSpecificRequest
     isAppliedTo ?AdapteeAttribute;
     name ?AdapteeSpecificRequestName.

Filter (?Adapter != ?Adaptee).
}
```

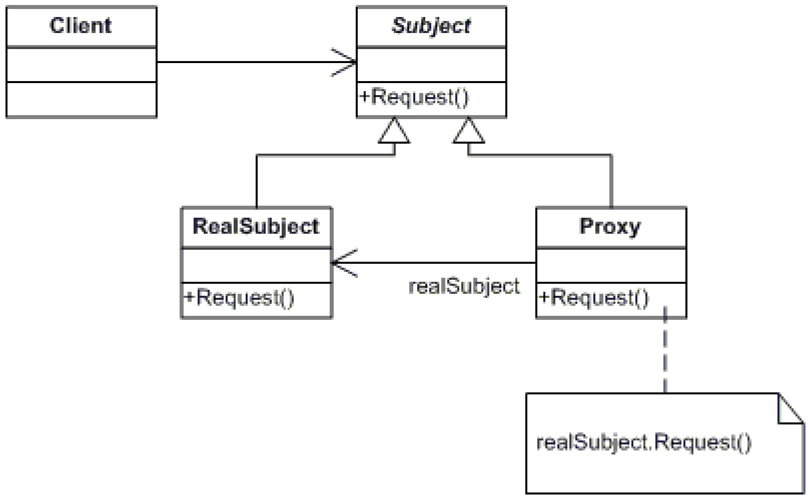
The query results are represented by the 5-tuple (`TargetClass`, `AdapterClass`, `AdapteeClass`, `AdapterRequest`, `AdapteeSpecificRequest`), each item representing a formal participant. The ending Filter SPARQL clause prevents the query from returning results in which a class is retrieved as both an Adapter and an Adaptee – ensuring a significant improvement in the false positives detection rate.

**Role of Semantic Reasoner**

- computes the transitive properties `isInheritedBy` and `overrides`, which means that the pattern will be detected even if the Adapter class is not the immediate child of the target class
- computes the reverse object property `isDeclaredClassOf`

### 4.5.2 Proxy Pattern

**Pattern Description:**



The Proxy pattern's main purpose is to provide light objects (*Proxies*) that have the same interface as their heavy counterparts. Proxies are responsible for instantiating heavy objects only when strategically needed. Otherwise, the light weight proxy is used. For instance, an image viewer could load picture objects only partially to show information like the picture name, date taken, etc. in a *Proxy* object to display in a list and only load the image itself in a real subject when it is required to be rendered.

**SPARQL Query**

The query is based on the UML class diagram representation of the pattern and details the relationships within its elements:

- the *Subject* class has the *Request* method in its interface and the *Proxy* and *RealSubject* as subclasses

- the Proxy and the Real Subject respectively have the *Proxy Request* and the *Real Subject Request* in their interfaces
- the *Proxy Request* overrides the parent's *Subject Request* and invokes the *Real Subject Request*, which also overrides the parent's *Subject Request*

The query results are represented by the 5-tuple (`Subject`, `RealSubject`, `Proxy`, `ProxyRequest`, `RealSubjectRequest`) each item of the tuple representing a formal participant and the whole set representing a pattern implementation instance. The last SPARQL Filter clause prevents the query from returning results in which a class is retrieved as both a *RealSubject* and a *Proxy* since their semantics are very similar, ensuring a significant improvement in the false positives detection rate.

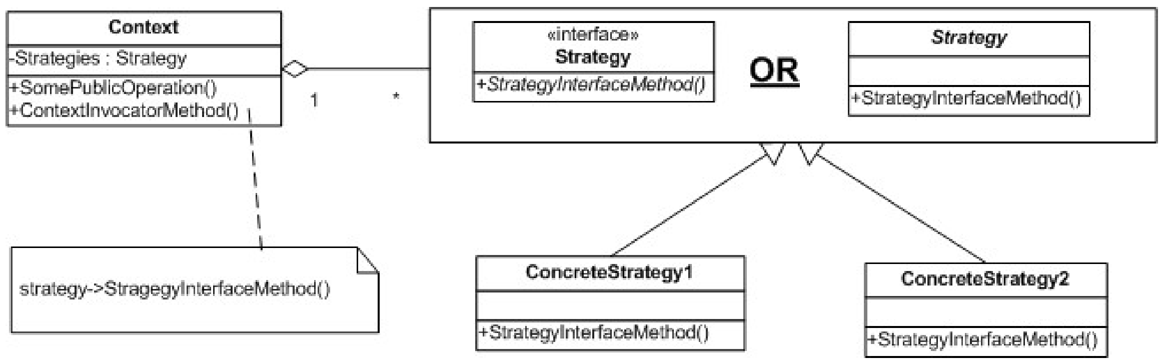
```
SELECT DISTINCT

?SubjectName
?RealSubjectName
?ProxyName
?ProxyRequestName
?RealSubjectRequestName

WHERE {
?Subject
     isInheritedBy ?RealSubject;
     isInheritedBy ?Proxy;
     hasMethod ?SubjectRequest;
     shortName ?SubjectName.
?Proxy
     hasMethod ?ProxyRequest;
     shortName ?ProxyName.

?ProxyRequest
     overrides ?SubjectRequest;
     name ?ProxyRequestName;
     invokes ?RealSubjectRequest.

?RealSubject
     hasMethod ?RealSubjectRequest;
     shortName ?RealSubjectName.

?RealSubjectRequest
     overrides ?SubjectRequest;
     name ?RealSubjectRequestName.
Filter (?RealSubject != ?Proxy).
}
```

**Role of Semantic Reasoner**

- computes the transitive properties `isInheritedBy` and `overrides`, which allows for the detection of the pattern even in cases then the Proxy and RealSubject classes are not immediate children of the Subject class

## 4.5.3 Strategy Pattern

### Pattern Description:


Context
-Strategies : Strategy
+SomePublicOperation()
+ContextInvocatorMethod()
1
*
«interface»
Strategy
+StrategyInterfaceMethod()
OR
Strategy
+StrategyInterfaceMethod()
strategy->StragegyInterfaceMethod()
ConcreteStrategy1
+StrategyInterfaceMethod()
ConcreteStrategy2
+StrategyInterfaceMethod()

The objective of the Strategy pattern is to provide a family of algorithms through a common interface. The algorithm being used depends on the context and is selectable at runtime. The pattern therefore decouples a class' behaviour from the class itself. Note that the original GoF pattern has been extended to include the implementation of abstract strategies in the form of both an abstract class and an interface.

### SPARQL Query

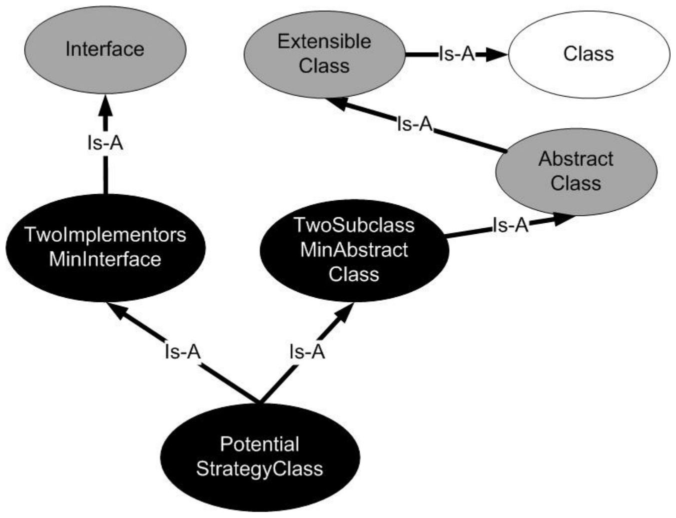
The query is based on the UML class diagram representation of the pattern and details the relationships within its elements.

- the *Context* holds a reference to an abstract Strategy and has one or more methods that invoke one or more methods from the abstract Strategy's public interface –the RDF type *PotentialStrategyClass*
- the abstract *Strategy* can either be implemented through an abstract class or an interface that is subclassed by at least two concrete classes

```
SELECT DISTINCT

?ContextName
?StrategyInterfaceName
?StrategyAttributeName
?ConcreteStrategyClassName

WHERE {

?Context
     hasAttribute ?StrategyAttribute;
     hasMethod ?ContextInvocatorMethod;
     name ?ContextName.

?ContextInvocatorMethod
     invokes ?StrategyInterfaceMethod.

?StrategyInterfaceMethod
     isAppliedTo ?StrategyAttribute.

?StrategyAttribute
     hasDeclaredClass ?StrategyInterface;
     name ?StrategyAttributeName.

?StrategyInterface
     rdf:type ?PotentialStrategyClass;
     hasMethod ?StrategyInterfaceMethod;
     isInheritedBy ?ConcreteStrategyClass;
     name ?StrategyInterfaceName.

?ConcreteStrategyClass
     name ?ConcreteStrategyClassName;
     rdf:type ?ConcreteClass.
}
```

The query results are represented by the 4-tuple (`Context`, `StrategyInterface`, `StrategyAttribute`, `ConcreteStrategy`) each capturing a formal participant to the pattern implementation. Records with the same `Context`, `StrategyInterface` and `StrategyAttribute` represent the same pattern and a record will be returned for each `ConcreteStrategy`.

**Ontology Extension**



**OWL-DL involved:**

```
ExtensibleClass ≡ Class and isFinal value false

AbstractClass ≡ ExtensibleClass and isAbstract value true

TwoSubclassMinAbstractClass ≡ AbstractClass and hasSubclass
min 2 ConcreteClass

Interface ≡ Class and isInterface value true

TwoImplementorsMinInterface ≡ Interface and isImplementedBy
min 2 ConcreteClass
```

The first two concepts define `ExtensibleClass`es and `AbstractClass`es using the data properties previously presented. The `TwoSubclassMinAbstractClass` concept and the `TwoImplementorsMinInterface` concept respectively define all abstract classes which have at least 2 concrete subclasses and all interfaces implemented by at least two concrete classes. This is meant to model the fact that to implement an instance of the Strategy pattern, there must at least be a choice of two strategies available at runtime. Finally, the `PotentialStrategyClass` concept is a value partition of `TwoImplementorsMinInterface` and `TwoSubclassMinAbstractClass`, meaning that any individual belonging to the partitioned values automatically belongs to the parent concept `PotentialStrategyClass`.

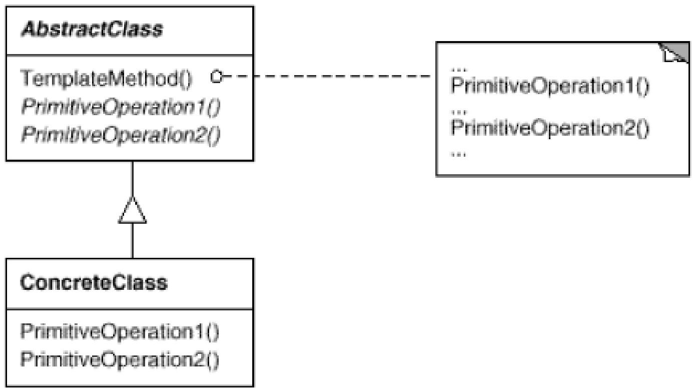
**Role of Semantic Reasoner**

- realizes individuals that belong to concepts `ExtensibleClass`, `AbstractClass`, `TwoSubclassMinAbstractClass`, `Interface`, `TwoImplementorsMinInterface` and `PotentialStrategyClass`
- computes the transitive object relationships `isSubclassOf` and `isImplementedBy`, part of the restrictions on concepts `TwoSubclassMinAbstractClass` and `TwoImplementorsMinInterface`

Consequently, SE-PAD will detect occurrences of the pattern even if the participating concrete strategy classes do not directly inherit from or implement the *Abstract Strategy,* but are at a lower level of the *Strategy* subclass hierarchy. This pattern involves 2 DL concepts (`TwoSubclassMinAbstractClass` and `TwoImplementorsMinInterface`) defined with minimum cardinality restrictions of 2. Since the default reasoner OWLIM [44] does not support cardinality restrictions higher than 1, Racer [43] which supports this type of cardinality restrictions was utilized.

## 4.5.4 Template Pattern

### Pattern Description



The *Template Method* pattern's objective is to define the outline of an algorithm in a template operation where some tasks are delegated to subclasses. Template Method allows classes at a lower hierarchical level to implement an algorithm's operations without altering the main algorithm's organization. It implements the famous Hollywood Design Principle: “don't call us we'll call you”, referring to the fact that subclasses do not need to know the main algorithm, only which of its operations they must implement to be invoked by the template method.

### SPARQL Query

The query is based on the UML class diagram of the pattern and details the relationships within its elements.

- the *AbstractClass* must bear the abstract keyword modifier and have at least one abstract operation invoked by the *TemplateMethod*

- the *ConcreteClass*es must extend the *AbstractClass* and override each of its primitive operations with concrete methods
- in order to comply with the theoretical model, results are restricted to *TemplateMethod*s implemented as final and protected

```
SELECT DISTINCT

?AbstractClass
?TemplateMethod
?TemplatePrimitiveOperation
?ConcreteClass
?ConcretePrimitiveOperation

WHERE {

?AbstractClass
hasMethod ?TemplateMethod;
rdf:type AbstractClass.

?TemplateMethod
invokes ?TemplatePrimitiveOperation;
rdf:type FinalMethod;
rdf:type ProtectedMethod.

?TemplatePrimitiveOperation
isMethodOf ?AbstractClass;
rdf:type AbstractMethod.

?ConcreteClass
isSubclassOf ?AbstractClass;
hasMethod ?ConcretePrimitiveOperation.

?ConcretePrimitiveOperation
rdf:type ConcreteMethod;
overrides ?TemplatePrimitiveOperation.
}

Order by ?AbstractClass
```

The query result is the 5-tuple (`AbstractClass`, `TemplateMethod`, `TemplatePrimitiveOperation`, `ConcreteClass`, ConcretePrimitiveOperation) each item representing a formal participant. Records with the same AbstractClass are considered part of the same pattern implementation, so each implementation might have many records depending on the number of concrete classes, primitive operations, etc.

**OWL-DL involved**

**AbstractClass** ≡ ExtensibleClass and isAbstract value true

**ExtensibleClass** ≡ Class and isFinal value false

**FinalMethod** ≡ Method and isFinal value true

**AbstractMethod** ≡ Method and isAbstract value true

AbstractClass subsumes ExtensibleClass because abstract classes cannot be marked as final in Java, hence the added restriction along the isFinal data property. The other concepts define Java final, abstract and concrete methods using the data properties previously described.

**Role of Semantic Reasoner**

- realizes individuals belonging to concepts AbstractClass, ExtensibleClass, FinalMethod, AbstractMethod and FinalMethod

- computes the transitive object relationship isSubclassOf

Consequently, SE-PAD will detect occurrences of the pattern even if the participating concrete classes do not directly inherit from or the Abstract class but are at a lower level of the abstract subclass hierarchy.

## 4.6 Tool integration

As stated in the introduction and motivation section of the thesis, the objective of this research is neither to re-invent nor compete with existing pattern detection tools. Instead, we intend to highlight the flexibility of Semantic Web technologies to support different forms of pattern detection and knowledge integration. As shown in the previous sections, our SE-PAD approach can support many of the pattern detection approaches found in tools like PMD [49] or FindBugs [50].

FindBugs is a static source code analysis tool which identifies a set of pre-defined bugs in Java code. A subset of bugs identified by the tool relates to security issues such as malicious code vulnerabilities. More specifically, some of reported bugs correspond to code blocks as part of a class that either expose the internal state of the class or stores references to external mutable objects in the attributes of that class. Both situations make instances of such classes subject to security threats if accessed by untrusted code. As part of our approach we do support the import security reports from external tools (e.g. FindBugs) and make them an integrated part of our source code model. SE-PAD populates the `violates` relationship as previously described and when the reasoner is applied, the security relevant concepts are realized. Ultimately, all the Java classes deemed unsafe according to the results of at least one of the static analysis tools are identified.

**SPARQL Query**

```
SELECT DISTINCT ?ClassName

WHERE {?Class rdf:type UnsafeClass.}
```

The query returns all the classes reporting safety issues in static analysis tools.

**OWL-DL involved**

In terms of OWL individuals and relationships, an unsafe class is a class that violates specific security related PMD or FindBugs rules through one of its methods. The main DL definitions involved are as follows, in Manchester-OWL syntax:

```
UnsafeClass ≡  ConcreteClass and
InternalStateVulnerableClass or PrivacyVulnerableClass


InternalStateVulnerableClass ≡  Class and hasMethod some (
(violates some MaliciousCodeRule) or
(violates some SecurityCodeGuideLineRule)or
(violates value ConstructorCallsOverridableMethod))


PrivacyVulnerableClass ≡ Class and hasMethod some
(violates some UserPrivacyRule)
```

**Role of Semantic Reasoner**

For this pattern, the reasoner plays a crucial role by automatically classifying the individuals belonging to the `UnsafeClass`es concept, whose complex definition include the integration of PMD (e.g. `ConstructorCallsOverridableMethod`, `SecurityCodeGuideLine`) and FindBugs results (e.g. `PrivacyVulnerability`, `MaliciousCodeVulnerability`).

**Requirements covered**

| | |
|---|---|
| **Establish trustworthiness** | Unsafe classes reported by static source code analysis tools represent significant risks exploitable by malicious developers. The more are reported, the less safe is the software analyzed. |
| **Retrieve implicit and explicit knowledge** | The implicit knowledge retrieved by the SPARQL query is used to detect the pattern in the code and the explicit knowledge is the identification of the software modules implementing it. |
| **Model extendibility** | All the concepts involved in this detection are added to SE-PAD's Ontology by software clients. |
| **Bridge information silos** | By integrating other source code analysis tools results to SE-PAD's Ontology and inferring knowledge on the source code analyzed by these tools based on them, SE-PAD helps bridging the source code artefact with rule violation results. |

## 4.7 Cross artefact analysis

SE-PAD's Ontology can be combined with knowledge resources from other software related artefacts. In our cross artefact analysis, we combine the results of static source code analysis tools with knowledge from different software artefact Ontologies, namely a source code management system (SCM). The goal is to identify the developer who first committed code in a SCM that generates a violation report. Ultimately, the analysis could serve as a developer profiler tool, based on the quality of the code they commit. However, such a tool is outside the scope of this research.

SE-PAD's supports this detection by identifying the first revision (or version) in which a given method M is reported as being a violation (based on the results obtained from a static source code analysis tool). SE-PAD then retrieves code from the SCM and populates its Version concept inherited from SOM's Ontology, which assigns a revision id to each Java class parsed. The client is then able to query SE-PAD to determine if a violation is found. In the case of a positive outcome, since the Version concept is also present in the Version Ontology Model (VOM), SE-PAD's clients can query the VOM and share version related knowledge. The following algorithm uses a binary search inspired approach to browse the SCM since they typically contain a large number of revisions.

```
revMax <- violationRevNumber
revMin <- 0

while (revMax-revMin > 10)
     revCurrent <- (revMax+revMin)/2
     populateSEPAD(revCurrent)
     if (violationInOntology())
          revMin <- revCurrent
          revCurrent <- (revMax-revMin)/2 + revMin
     else
          revMax <- revCurrent
     end if
end while

revId <- searchViolationSequential(revMax, revMin)
```

**Figure 4-7: SCM rule violation search algorithm**

The following reuses sections of the pattern presentation template to describe in detailed SE-PAD's support for cross artefact analysis.

**SPARQL Query**

There are two queries involved in this detection. Both queries are part of the client software module performing the SCM rule violation search. The first one is:

```
SELECT ?RevisionId WHERE
{
     {0} violates {1}.
     ?Class hasMethod {2};
            hasRevision ?Revision.
     ?Revision revisionId ?RevisionId.
}
```

It is applied on the SE-PAD's Ontology and retrieves the revision id of a class in which a method violates a rule. The token {0} and {2} represent the method M originally reporting a violation of the rule represented by {1}. If the query returns a result, it means that M reports a violation for the current revision.

The second query retrieves the author of the revision where the rule violation was first reported from the VOM:

```
SELECT ?Author WHERE
{
     ?Revision rdf:type Revision;
          hasCommitAuthor ?Author;
          id ?id.
}
FILTER (str(?id) = {0}}
```

**Ontology Extension**

This Ontology extension includes concept sharing with the Version Ontology Model (VOM) [5]. The shared concept we use for aligning our SE-Pad Ontology with the VOM Ontology is `Version` and the sharing is materialized in the SPARQL queries shown above through the Revision id.

**Requirements covered**

| | |
|---|---|
| **Retrieve implicit and explicit knowledge** | The identification of developers committing rule violations is a form of historical data mining. SE-PAD can then constitute, for instance, the basis of a developer profiling tool which can eventually help in assigning developers to relevant formations. |
| **Model extendibility** | As previously described, SE-PAD's Ontology is extended by sharing a concept with the VOM. |
| **Bridge information silos** | By combining results of static source code analysis results with the VOM, SE-PAD helps bridge the silos related to 3 artefacts: source code, versioning system and static analysis tools results. |

# 5. Case Studies

This chapter describes the evaluations we performed using SE-PAD in different application contexts. Unless specified otherwise, all case studies were performed on JabRef[2] (Version 2.6) a small-sized open source project offering bibliography reference management features. Ohloh[3] reports for this version of JabRef 136 992 lines of code, 29 091 lines of comments and 42 894 blank lines. After being parsed by SE-PAD, the resulting Ontology contains 285 385 triples which are stored in the Sesame triple store. Table 5-1 shows a summary of the population:

| | |
|---|---|
| Number of Java classes | 859 |
| Number of method invocations | 27668 |
| Number of class method overrides | 504 |
| Number of interface method overrides | 303 |
| Number of class attribute assignations | 1292 |

**Table 5-1: JabRef population summary**

For the case studies, we used PMD Version 4.2.5 and FindBug**s** Version 2.0. All experiments were conducted on a PC, with an Intel I7 processor, 6 GB of RAM and running Windows 7 64 bits.

## 5.1 Security violations

The first case study evaluates the results generated by PMD and SE-PAD during the detection of violations of a secure coding guideline.

**Experimental setting:** For the evaluation JabRef was analyzed to detect the following secure coding violation: "a class constructor should not invoke a method that can be

[2] http://jabref.sourceforge.net/
[3] http://www.ohloh.net/p/jabref/analyses/latest

overridden” [31, 72]. For the experiment SE-PAD used the following query introduced in section 4.4.2.2:

```
SELECT DISTINCT ?MethodName WHERE {
     ?Method rdf:type ConstructorMethod;
          qualifiedName ?MethodName;
          invokes ?OverridableMethod;
          isMethodOf ?Class.
     ?OverridableMethod rdf:type OverridableMethod;
          isMethodOf ?Class.
     ?Class isFinal ?isFinal.
     FILTER (?isFinal=false)
}
```

**Evaluation results**: SE-PAD reports 60 violations whereas PMD reports 66. A manual evaluation of the results obtained from both tools was performed to determine their precision and recall. Table 5-2 summarizes the results from our analysis. A security violation baseline in JabRef was established by summing up the true positives reported by both tools (not counting duplicate instances). As a result, a baseline of 80 true positives for the “a class constructor should not invoke a method that can be overridden” violation by JabRef could be established. This baseline value was used for the later recall calculation of both tools.

| | **Detected** | **True positives** | **Recall** | **False positives** | **Precision** | **F1 score** |
|---|---|---|---|---|---|---|
| **PMD** | 66 | 65 | 0.83 | 1 | 0.99 | 0.91 |
| **SE-PAD** | 60 | 59 | 0.74 | 1 | 0.98 | 0.81 |

**Table 5-2: SE-PAD vs. PMD**

**Evaluation discussion:** Table 5-2 shows that the recall and precision of SE-PAD was lower (recall by 9%) and precision by 1%. As previously discussed, we do not claim to perform as well as specialized tools do but that our approach is flexible enough to obtain significantly close results. Given the circumstances, we feel the results are comparable, with the F1 score falling in the 10% difference range. Further analysis of the results showed that the lower recall is due to a problem while populating methods involving the `Default` and `Protected` modifiers. Both types of modifiers were in some cases not correctly categorized as modifies are part of our Ontology population.

## 5.2 Design Pattern Automated Recovery

We also used JabRef to evaluate SE-PAD's applicability in detecting design patterns in the source code.

**Experimental setting:** The challenge was to find a tool to support the same subset of patterns. DPR[4] a reverse engineering tool able to perform the recovery of the Adapter design patterns fit our needs.

**Evaluation results:** Both tools reported together 122 different instances of the Adapter pattern in JabRef. This total was used as a baseline to compute recall. DPR detected 91 occurrences versus 70 for SE-PAD. Here are the statistics:

| | **Detected** | **True positives** | **Recall** | **False positives** | **Precision** | **F1 score** |
|---|---|---|---|---|---|---|
| **SE-PAD** | 70 | 70 | 0.57 | 0 | 1 | 0.73 |
| **DPR** | 245 | 91 | 0.74 | 154 | 0.37 | 0.49 |

**Table 5-3: SE-PAD vs. DPR**

**Evaluation discussion:** Further analysis of the results showed that the differences in the recall results are mainly due to our more conservative interpretation of what constitutes an actual adapter pattern implementation. SE-PAD's query did not report any false positives which compares favourably to the 154 false positives report by the DPR tool.

[4] http://www.sesa.dmi.unisa.it/dpr/

Ultimately, according to the F1 score, SE-PAD outperformed DPR due to its lower number of false positive. The quality of our results highlights one important aspect of our approach, especially with respect to the false positive rate. When patterns are semantically modeled in a way mimicking the theoretical implementation in UML, finding corresponding code constructs not implementing the expected pattern is very hard.

## 5.3 Integration of external static analysis tool results (PMD and FindBugs)

In this case study, we apply the tool integration's query (see section 4.6) to initiate a statistical analysis based on the detection of `UnsafeClass`es as determined by the results of PMD and FindBugs.

**Experimental setting:** We used revision 19000 of ArgoUML[5], a mid-size open source project providing a UML integrated development environment. For this revision, Oholoh[6] reported 910 411 lines of code, 233 787 line of comments and 123 089 blank lines. After the code was parsed and SE-PAD's Ontology was populated, the resulting Ontology was uploaded to our Sesame repository which reported 1 996 526 triples. Here is a summary of the population:

| | |
|---|---|
| Number of Java classes | 2625 |
| Number of method invocations | 67098 |
| Number of class method overrides | 6862 |
| Number of interface method overrides | 725 |
| Number of class attribute assignations | 1748 |

**Table 5-4: ArgoUML population summary**

[5] http://argouml.tigris.org/
[6] http://www.ohloh.net/p/argouml

The objective is to determine the percentage of unsafe classes in each Java package of ArgoUML. The implementation of the solution includes three SPARQL queries surrounded by code which performs the calculations and generates a comma separated value file as an output. The algorithm uses the following query to retrieve all the packages:

```
SELECT DISTINCT ?Packages WHERE
{?Packages rdf:type Package.}
```

Then, for each package, the query of section 4.6 was applied to retrieve and count unsafe classes. The following query was applied to retrieve and count the total number of classes, in which the wildcard * is replaced in code by the current package's URI:

```
SELECT DISTINCT ?Classes
WHERE {<*> hasClass ?Classes.}
```

**Evaluation results:** For revision 19000 of ArgoUML, the packages showing the highest rate of unsafe classes are:

| **Package** | **Number of unsafe classes** | **Number of classes** | **Ratio of unsafe classes** |
|---|---|---|---|
| `jdepend.textui` | 1 | 1 | 100% |
| `org.argouml.language.ui` | 1 | 1 | 100% |
| `org.argouml.notation.ui` | 1 | 3 | 33% |
| `org.argouml.uml.diagram.deployment.ui` | 6 | 19 | 31.58% |
| `org.argouml.uml.diagram.use_case.ui` | 5 | 17 | 29.41% |
| `jdepend.framework` | 4 | 16 | 25% |
| `org.argouml.uml.ui.model_management` | 3 | 14 | 21.43% |
| `org.argouml.uml.diagram.ui` | 28 | 131 | 21.37% |
| `org.argouml.activity2.diagram` | 4 | 20 | 20% |
| `org.argouml.uml.util.namespace` | 1 | 5 | 20% |

**Table 5-5: Ratio of unsafe classes in ArgoUML**

**Evaluation discussion:** Based on this sample, user interface (UI) related packages seem to be especially unsafe. The UI layer of a project is often developed by programmers specialized in visual designs and focus on usability and aesthetic of GUIs rather than on secure implementations. From a managerial standpoint, the results we obtained for ArgoUML can trigger the need to have these programmers implementing the GUI follow stricter quality assurance procedures or be provided with additional training related to secure programming guidelines. As expected, we were able to combine external tools results to SE-PAD's original Ontology. We created new concepts based on the supported violations to significantly enrich our knowledge base by providing valuable insight into the quality of a software project.

As part of the SE-PAD implementation a Quartz job [61] has been created to allow SE-PAD to be automatically invoked in regular time intervals. Using this script, SE-PAD can be used to monitor the progress of software securing efforts over time or mine past revisions to determine trends in safety development.

## 5.4 Integration of SCM Ontology

In this case study, we combine SE-PAD's Ontology with SOM's Version Model Ontology [4, 5]. As described in section 4.7, the goal is to identify developers who commit code that contains code violation detected by static source code analysis tools.

**Experimental setting:** For revision 19000 of ArgoUML, PMD reports that five methods violate the rule stating that a constructor should not call a method that can be overridden. They are all class constructors and are listed below with their fully qualified Java names:

- `org.argouml.util.ItemUID.ItemUID`

- `org.argouml.kernel.ProfileConfiguration.ProfileConfiguration.org.argouml.kernel.Project`
- `org.argouml.kernel.ProfileConfiguration.ProfileConfiguration.org.argouml.kernel.Project.java.util.Collection`
- `org.argouml.uml.diagram.DiagramSettings.DiagramSettings.org.argouml.uml.diagram.DiagramSettings`
- `org.argouml.uml.diagram.ui.FigStereotypesGroup.FigStereotypesGroup.java.lang.Object.java.awt.Rectangle.org.argouml.uml.diagram.DiagramSettings`

The algorithm presented in section 4.7 detects the revision of the code in which a committer potentially introduced a bug.

**Evaluation results:** The program searched ArgoUML's source code management (SCM) repository for the revision id where these violations were introduced. The results of this search are shown below:

| Method | Revision Id | Author |
|---|---|---|
| `argouml.util.ItemUID.ItemUID` | 14536 | tfmorris |
| `org.argouml.kernel.ProfileConfiguration.ProfileConfiguration.org.argouml.kernel.Project` | 13846 | euluis |
| `org.argouml.kernel.ProfileConfiguration.ProfileConfiguration.org.argouml.kernel.Project.java.util.Collection` | 13963 | tfmorris |
| `org.argouml.uml.diagram.DiagramSettings.DiagramSettings.org.argouml.uml.diagram.DiagramSettings` | 16431 | tfmorris |
| `org.argouml.uml.diagram.ui.FigStereotypesGroup.FigStereotypesGroup.java.lang.Object java.awt.Rectangle.org.argouml.uml.diagram.DiagramSettings` | 16252 | tfmorris |

**Table 5-6: Integration of SCM Ontology results**

**Evaluation discussion:** In this case study, we illustrate the advantage of the Ontological representation by supporting the integration of results obtained from external source code analysis tools with knowledge already represented in our knowledge base (Version Control Ontology). The resulting knowledge base can provide different stakeholders such as managers with additional insights regarding organizational development processes and practices. For example, the query results can be used to identify developers that show a reoccurring pattern towards producing unsafe code. Management might also use this information for various purposes (e.g. training, additional secure programming guidelines) and specifically targeted quality improvements.

# 6. Discussion

## 6.1 Revisiting the hypothesis

The objective of the research presented in this thesis was to provide a novel approach that takes advantage of Semantic Web technologies to represent software artefacts and related knowledge resources in order to support the assessment of quality aspects of post-mortem systems in a distributed and global setting. In section 4 and 5 we have introduced a variety of quality patterns to address the different requirements associated with our research hypothesis and its sub-goals. The following is a review of these initial requirements:

R1. Bridge information silos

R2. Retrieve implicit and explicit knowledge

R3. Establish trustworthiness

R4. Model extendibility

Table 6-1 summarizes these requirements and how they are addressed in the thesis. As shown, each requirement was addressed through a concrete example of how SE-PAD can support it and therefore also the research hypothesis that Semantic Web technologies can not only represent software artefacts and related knowledge resources but also support the assessment of quality aspects of post-mortem systems.

| **Supported Pattern Detections** | | | **R1** | **R2** | **R3** | **R4** |
|---|---|---|---|---|---|---|
| Programming Guidelines [section 4.4] | General Programming Guidelines | *Do not write to static fields from instance methods* | | √ | √ | √ |
| | | *Final classes should not have protected attributes* | | √ | √ | √ |
| | | *Servlet classes should not have mutable attributes* | | √ | √ | √ |
| | | *Respect Naming Conventions* | | √ | √ | |
| | Secure Coding Guidelines | *Prefer immutable classes* | | √ | √ | √ |
| | | *Prevent constructors from calling overridable methods* | | √ | √ | √ |
| Modeling Design Patterns [section 4.5] | *Adapter* | | √ | √ | √ | √ |
| | *Proxy* | | √ | √ | √ | √ |
| | *Strategy* | | √ | √ | √ | √ |
| | *Template* | | √ | √ | √ | √ |
| Tool Integration [section 4.6] | | | √ | √ | √ | √ |
| Cross-Artefact Analysis [section 4.7] | | | √ | √ | | √ |

**Table 6-1: Semantic classification and identification of pattern support**

## 6.2 The Open World Assumption problem

As part of our future work, we plan to enrich our existing Ontological model with additional artefacts and security concerns and further evaluate the applicability of our SE-PAD tool in detecting additional types of patterns. The goal is to deal with one important feature of OWL-DL Ontologies: the Open World Assumption (OWA). The main effect of the OWA occurs in the reasoning phase. When a reasoner is not explicitly told a fact X is true, it will not consider it as false like it is the case for relational database. Based on the OWA the truth-value of X will be considered unknown (neither true nor false). Given is the following example (fact):

```
Paul lives in Montreal
```

If this fact is modeled in a relational database (closed world) and the query "Does Paul live in Toronto?" executed, the result would be false. This is due to the closed world, where it is assumed that no other knowledge exists than the one at hand. If the same example (fact) was modeled in an Ontology, a reasoner would entail "Unknown" as a truth-value to an equivalent query. Since an Ontology's world is open, a reasoner could not assume that a fact X is either false or true unless explicitly stated as such because other knowledge might exist in the open world which will influence its truth-value. The OWA has many benefits during the pattern detection, like the ability to deal with incomplete or incremental populated knowledge, while still supporting pattern detection.

However, an open world assumption does not always reflect software source code's reality. In many cases, source code can be treated as a closed finite set of known or knowable elements. For example in Object-Oriented programming one typically deals with a finite set of knowable classes, methods, interfaces, method calls, variables, external libraries, frameworks, containers, patterns, etc.

The OWA has other important implications with respect to source code and its closed nature. More specifically, source code relevant reasoning is impaired. For example, reasoners cannot assert truth-values of class restrictions involving *smallerThan* cardinality comparisons (<), *allValuesFrom* axioms (∀), disjunction axioms (OR) or complement axioms (NOT).

Not being able to reason on these axioms means that some interesting patterns cannot be detected by SE-PAD. For instance, the unsupported `ImmutableClass` pattern implies that no method other than the constructor of a class is allowed to change its state by assigning values to its attribute. So, to be able to infer a class X as part of the `ImmutableClass` concept, a reasoner would require X to be closed in terms of the number of methods it contains otherwise, it will entail "Unknown" when asked to decide on X's belonging to `ImmutableClass` since, as explained earlier, it assumes other knowledge pertaining to X might exist. Consequently, for the detection of some patterns such as micro-patterns [25] which are otherwise non-detectable using the OWA, it will be desirable to close SE-PAD's world.

## 6.3 Threats to Validity

In what follows, we identify, analyze and discuss different threats that could affect the validity of our approach and the requirements we introduced in Section 2. Each subsection details a threat and shows the threatened requirement.

### 6.3.1 Semantic web technologies fail in bridging the information silos (R1).

Providing tool support for a domain like software engineering is inherently different due to the variations in users' contexts, the abstraction levels and the semantics of knowledge that needs to be modeled [66]. Consequently, knowledge representation becomes an essential part of the modeling challenge [67]. Formal semantics provide a means of representing and ensuring some consistency in modeling knowledge. However, they still do not guarantee that either sufficient or the right information is captured. We do not claim that our approach is able to capture all domain specific knowledge. Instead, we argue that by using an Ontological model for the knowledge representation in our SE-PAD environment, we can support the modeling of incomplete and often inconsistent knowledge found in software artifacts – see "Classification" in section 2.1.4. Previous work [48, 68] has also shown that Ontologies can be applied to create a uniform representation for different types of artifacts and link them successfully.

In spite of these modeling techniques, there will always remain a gap between the actual and modeled knowledge. However, we believe that our approach provides an important step towards modeling and retrieval of knowledge relevant to guide further analysis and comprehension of software systems.

### 6.3.2 Implicit versus explicit knowledge (R2)

Information or artifacts might often not be available or consistent. An Ontological representation can provide us with the flexibility to support an open world assumption. Furthermore, the use of semantic reasoners enables the exploration of both explicit and implicit knowledge. We have shown as part of our SE-PAD environment how reasoners can be applied during pattern detection to resolve transitive closure in inheritance hierarchies or can be applied to classify program parts based on their violations of security guidelines. Remaining threats to validity are that the design of the Ontological model has to be such that it supports the capabilities of the semantic reasoner being used. However, this challenge is not unique to Semantic Web technologies and has also to be taken into consideration by other knowledge modeling approaches.

### 6.3.3 Establishing Trustworthiness (R3)

In order to establish trustworthiness, one has to analyze two different issues:

1. Is the implemented approach able to capture the required patterns and guidelines required to validate trustworthiness of the system being analyzed?
2. Are the results obtained by the tool itself trustworthy?

For the first challenge, the issue of being able to capture the right patterns has been addressed by our semantic modeling approach by allowing:

1- the creation and addition of new knowledge to our model, by formalizing new patterns or guidelines as queries and populate the model with the result of these queries

2- the provision of a set of predefined queries to detect design and security patterns, as well as supporting the validation of secure programming guidelines. The provided queries are similar to the ones supported by other specialized tools [49, 63]

3- the integration of knowledge from often specialized third party tools and integrating this knowledge directly as part of our knowledge base

For the second challenge, the analysis of the trustworthiness of the SE-PAD results, we conducted some experiments comparing our SE-PAD pattern detection results to results obtained from other tools. The case study performed in section 5.2 is an example of such evaluation and validation, however, it has to be pointed out that the general quality and trustworthiness of detection approaches depends on the ability to formalize and express these patterns and based on the objective of the algorithm, to maximize either recall or precision. Furthermore, depending on the type of pattern, pattern detection not only in SE-PAD but in general becomes an inherently complex problem with threats regarding the trustworthiness of the results remaining.

### 6.3.4 Patterns and knowledge base have to be extensible (R4)

Given the existence of many, often highly specialized analysis (both static and dynamic) tools to detect patterns and validation of guidelines, we see our approach as complementary to them. Our goal was not to replace these tools. Instead, we focused on the integration of knowledge resources from existing tools (e.g. PMD) to enrich our Ontological KB. Given our common unified and semantically representation for various artifacts, we can support tracing of concerns and pattern across various abstraction levels which was also a concern. Through our Ontological representation, users can enrich the existing knowledge base with new concepts as they become available (e.g. new security patterns, rules). Furthermore, knowledge derived from other tools or artifacts can be integrated in the form of new concepts or automatically linked through the use of upper Ontologies, shared concepts or semantic links across Ontologies. In our research we were able to demonstrate that our source code Ontology can be extended with new concepts (see figures in section 4) that were derived from FindBugs and PMD. Semantic technologies allow our SE-PAD to be extended by integrating new knowledge and enrich existing knowledge. It has to be noted that both Ontology modeling and consistency management of the model are not trivial tasks. Lack of design expertise can limit the knowledge exploration and the use of semantic reasoners for inferring implicit knowledge.

# 7. Conclusions and Future Work

With relevant knowledge being distributed across multiple resources, the assessment and maintenance of the quality of these systems becomes inherently difficult. In this research, we address some of the challenges for the next generation of software engineering quality validation tools, the need to provide an extensible and unified knowledge representation. Our motivation was to integrate resources and knowledge related to quality patterns within a common Ontological representation to support post-delivery quality analysis of these systems. As part of the thesis research SE-PAD was developed, which supports the fact extraction from different artifacts, as well as the integration of Semantic Web technologies. The use of Semantic Web not only provides the enabling technology for the integration of knowledge resources at various abstraction and semantic levels, it also provides the foundation for an evolving knowledge base that supports the extension of new resources and patterns. We also showed through several case studies how SE-PAD can support the detection of various quality patterns and the knowledge integration from external tools.

More precisely, in this research, we introduced SE-PAD, a Semantic Web based automated source code quality analysis tool that supports several analysis tasks:

- the ability to detection violations of good coding practices, including security related guidelines
- the ability to integrate knowledge created by external (third party) static analysis tools to eliminate information silos by enriching our knowledge base

- the support for recovery of design patterns such as some of the GoF patterns [17]
- the ability to share knowledge across repositories boundaries to support different types of data and knowledge mining

As a result we are able to

- eliminate information silos by supporting result sharing among tools and integrating knowledge across different knowledge bases
- retrieve both,  implicit and explicit knowledge
- enhance the trustworthiness of software systems, by detecting coding and best practice violations
- support additional knowledge exploration through user defined queries and by enriching our existing knowledge base
- illustrate how our approach supports the detection of semantic rich patterns, while achieving at the same time reasonable precision

As part of our future work, we plan to enrich our existing Ontological model with additional artifacts and security concerns and further evaluate the applicability of our SE-PAD tool in detecting additional types of patterns.

## 8. References

1. K. Bontcheva and M. Sabou. "Learning Ontologies from Software Artifacts: Exploring and Combining Multiple Sources". In Proc. of the 2nd International Workshop on Semantic Web Enabled Software Engineering, Athens, GA, U.S.A., 2006.

2. H.-J. Happel and S. Seedorf. "Applications of Ontologies in Software Engineering". In Proc. of International Workshop on Semantic Web Enabled Software Engineering, 2006.

3. Y. Zhao, J. Dong, and T. Peng. "Ontology classification for Semantic Web based software engineering". IEEE Transactions on Services Computing, 2(4), pp. 303-317, 2009.

4. J. Tappolet, C. Kiefer and A. Bernstein. "Semantic Web enabled software analysis". Journal of Web Semantics: Science, Services and Agents on the World Wide Web, pp. 225-240, 8 July 2010.

5. Dynamic and Distributed Information Systems Group website (http://www.ifi.uzh.ch/ddis/)

6. J. A.Vayghan, S. M. Garfinkle, et al. "The internal information transformation of IBM". IBM Systems Journal, 46(4), pp. 669-683, 2007.

7. K.C. Desouza. "Barriers to effective use of knowledge management systems in software engineering". Communications of the ACM, 46 (1), pp. 99-101, 2002.

8. D. Gaševic, N. Kaviani and M. Milanovic. "Ontologies and Software Engineering". Handbook on Ontologies, pp. 593-615, 2009.

9. M. Côté. "A matter of trust and respect". CA Magazine [serial online]. March 2002. Available at: http://www.camagazine.com/archives/print-edition/2002/march/columns/camagazine23400.aspx. Accessed April 13, 2012.

10. H.R. Nemati, D.M. Steiger, L.S. Iyer and R.T. Herschel. "Knowledge warehouse: an architectural integration of knowledge management, decision support, artificial intelligence and data warehousing". Decision Support Systems, 33, pp. 143-161, 2002.

11. C. Larman. "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design". Prentice Hall, pp. 435-472, 2005.

12. M.-A. D. Storey. "Theories, methods and tools in program comprehension: Past, present and future". In Proc. of the 13th International Workshop on Program Comprehension (IWPC 2005), St. Louis, MO, USA, pp. 181-191, 2005.

13. G. Antoniou and F. van Harmelen. “Web Ontology Language: OWL”. Handbook on Ontologies. Springer, pp. 67-92, 2004.

14. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider. “The Description Logic Handbook: Theory, Implementation and Applications”. Cambridge University Press, chapter 3, 2003.

15. E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur and Y. Katz. “Pellet: A practical OWL-DL reasoner”. Journal of Web Semamtics, 5(2), pp. 51-53, 2007.

16. E. Prud’hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3CCandidate Rec. 6 April 2006. (http://www.w3.org/TR/rdf-sparql-query/)

17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison Wesley, Reading, MA, USA, 1995.

18. D. Parnas and M. Lawford. “The role of inspection in software quality assurance”. IEEE Transaction on Software Engineering, 29(8), pp. 674-676, August 2003.

19. E. van Emden and L. Moonen, “Java quality assurance by detecting code smells”. In Proc. of IEEE Computer Society Working Conference on Reverse Engineering, pp. 97-108, 2002.

20. W. Crawford and J. Kaplan. “J2EE Design Patterns”. O’Reilly and Associates, chapter 12, 2003.

21. P. Anderson, T. W. Reps, T. Teitelbaum and M. Zarins. “Tool support for fine-grained software inspection”. In Proc. of IEEE Software, 20(4), pp. 42-50, 2003.

22. A.M. Vans, A. von Mayrhauser and G. Somlo. “Program Understanding Behavior during Corrective Maintenance of Large-Scale Software”. Int’l Journal of Human-Computer Studies, 51(1), pp. 31-70, July 1999.

23. D. Flanagan. “Java in a Nutshell”. O’Reilly & Associates, Inc., 1st edition, Feb. 1996.

24. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. “Pattern-oriented Software Architecture - A System of Patterns”. J. Wiley and Sons Ltd., 1996.

25. J. Y. Gil and I. Maman. “Micro patterns in Java code”. In Proc. of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications, New York, NY, USA, pp. 97-116, 2005.

26. K. Beck and M. Fowler. “Bad smells in code”. In M. Fowler, editor, Refactoring: Improving the Design of Existing Code, Addison Wesley, pp. 75-88, 1999.

27. K. K. Aggarwal, Y. Singh, and J. K. Chhabra, “An Integrated Measure of Software Maintainability,” In Proc. of IEEE Annual Reliability and Maintainability Symposium, Seatle Westin,U.S.A, pp.235-241, 2002.

28. IEEE Std. 610.12-1990. Standard Glossary of Software Engineering Terminology, IEEE Computer Society Press, Los Alamitos, CA, 1993.

29. Code Conventions for the Java Programming Language (http://www.oracle.com/technetwork/java/codeconvtoc-136057.html)

30. R. P. L. Buse and Westley Weimer. “A metric for software readability”. In Proc. of International Symposium on Software Testing and Analysis, pp. 121-130, 2008.

31. Secure Coding Guidelines for the Java Programming Language, Version 3.0 (http://www.oracle.com/technetwork/java/seccodeguide-139067.html)

32. Checkstyle Homepage (http://checkstyle.sourceforge.net)

33. G. Carl, G. Kesidis, R. Brooks, and S. Rai. “Denial-of-Service attack detection techniques”. In Proc. of IEEE Internet Computing, 10(1), pp. 82-89, 2006.

34. C. Cowan, P. Wagle, C. Pu, S. Beattie and J.Walpole. “Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade”. In Proc. of DARPA Information Survivability Conference and Expo, 1(2), pp. 119-129, 2000.

35. D. Damian and D. Moitra. “Guest Editors' Introduction: Global Software Development: How far Have We Come?”. In Proc. of IEEE Software, 23(5), pp.17-19, 2006.

36. B. Blakley, C. Heath and Members of the Open Group Security Forum. “Security design patterns”. 2004.

37. F. Lee Brown, J. Di Vietri, G. Diaz de Villegas and E. Fernandez. “The authenticator pattern”. In Proc. of the Sixth Conference on Pattern Languages of Programming, 1999.

38. M. Horridge, N. Drummond, J. Goodwin, A. Rector, R. Stevens and H.H. Wang. “The Manchester OWL Syntax”. In Proc. of the OWL Experiences and Directions Workshop at the ISWC’06, 2006.

39. M. Uschold and M. Grüninger. “Ontologies: Principles, Methods and Applications”. Knowledge Engineering Review, 11(2), pp. 93-135, 1996.

40. Y. Zhang, R. Witte, J. Rilling, and V. Haarslev. “An Ontological Approach for the Semantic Recovery of Traceability Links between Software Artifacts”. IET Software, Special issue on Language Engineering, 2(3), pp. 185-203, 2008.

41. Java Parser project (http://code.google.com/p/javaparser/)

42. Jena Semantic Web Framework (http://jena.sourceforge.net/index.html)

43. Racer Systems (http://www.racer-systems.com/index.phtml)

44. OWLIM Semantic Repository (http://www.ontotext.com/owlim/)

45. V. B. Livshits and M. S. Lam. “Finding security vulnerabilities in Java applications with static analysis”. In Proc. of USENIX Security Symposium, pp.271-286, 2005.

46. M. Bashir and M. Qadir. “Traceability Techniques: A Critical Study”. In Proc. of IEEE Multitopic Conference, pp. 265-268, 2006.

47. S. Bohner and R.S. Arnold. “Software Change Impact Analysis”. IEEE Computer Society Press, 1996.

48. R. Witte, Y. Zhang and J. Rilling. “Empowering Software Maintainers with Semantic Web Technologies”. In Proc. of the 4th European Semantic Web Conference, pp. 37-52, 2007.

49. PMD (http://pmd.sourceforge.net/)

50. FindBugs (http://findbugs.sourceforge.net/)

51. D, Brickley and R.V. Guha. “Resource Description Framework (RDF) Schema Specification”, World Wide Web Consortium, 1999.

52. Pellet (http://clarkparsia.com/pellet/features/)

53. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. “The Description Logic Handbook”. Cambridge University Press, In print. 2002.

54. O. Laitenberger. “A Survey of Software Inspection Technologies”. In Handbook on Software Engineering and Knowledge Engineering, World Scientific Publishing, volume 2, pp. 517-555, 2002.

55. C. Calero, F. Ruiz and M. Piattini. “Ontologies for Software Engineering and Software Technology”. Springer, Berlin, Heidelberg, 2006.

56. J. Euzenat, A. Isaac, C. Meilicke, P. Shvaiko, H. Stuckenschmidt, O. Svab, V. Svatek, W. van Hage, and M. Yatskevich. “Results of the Ontology alignment evaluation initiative 2007”. In Proc. of the workshop on Ontology Matching at ISWC/ASWC, pp. 96-132, 2007.

57. Sesame (http://www.openrdf.org/)

58. W. Liu, A. Weichselbraun, A. Scharl and E. Chang. “Semi-Automatic Ontology Extension Using Spreading Activation”. Journal of Universal Knowledge Management, no. 1, pp. 50-58, 2005.

59. G. McGraw. “Software Security”. In Proc of IEEE Security & Privacy, vol. 2, pp. 80-83, March-April 2004.

60. G. Booch, J. Rumbaugh and I. Jacobson, “The Unified Modeling Language User Guide”. Addison Wesley, Reading MA, 1999.

61. Quartz (http://quartz-scheduler.org/)

62. N. Nagappan and T. Ball. “Static Analysis Tools as Early Indicators of Pre-Release Defect Density”. In Proc. Int’l Conf. Software Engineering, pp. 580-586, 2005.

63. W3C XQuery recommendation of December 14th 2010 (http://www.w3.org/TR/xquery/)

64. The Servlet Life Cycle (http://docstore.mik.ua/orelly/java-ent/servlet/ch03_01.htm)

65. D. Binkley, M. Davis, D. Lawrie, and C. Morrell. "To CamelCase or Under_score". In Proc. of 17th IEEE International Conference on Program Comprehension (ICPC), Vancouver, Canada, pp. 158-167, 2009.

66. M.-A. D. Storey, K. Wong and H. A. Müller. “How do program understanding tools affect how programmers understand programs?”. Science of Computer Programming, 36(2-3), pp. 183-207, March 2000.

67. C. Lai. “Java Insecurity: Accounting for Subtleties That Can Compromise Code”. In Proc. of IEEE Software, 25(1), pp. 13-19, 2008.

68. Y. Zhang, R. Witte, J.  Rilling, and V. Haarslev. “An Ontological Approach for the Semantic Recovery of Traceability Links between Software Artifacts”. IET Software, Special issue on Language Engineering, 2(3), pp. 185-203, June 2008.

69. D.L. McGuinness and F. van Harmelen. “OWL Web Ontology Language Overview,” World Wide Web Consortium (W3C) recommendation, 2004.

70. A. Gerber, A. van der Merwe and A. Barnard. “Towards a Semantic Web Layered Architecture”. In Proc. of IASTED International Conference on Software Engineering, Innsbruck, Austria, pp. 353-362, 2007.

71. Semantic Web Stack (http://en.wikipedia.org/wiki/Semantic_Web_Stack)

72. G. McGraw. “Software Security: Building Security In”. Addison Wesley Professional, pp 163-165, 2006.

73. J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy. “Java Developer's Guide to Eclipse”. Addison-Wesley Professional, 2004.