2010

Improving software quality using an ontology-based approach

Yixin Luo
Louisiana State University and Agricultural and Mechanical College

IMPROVING SOFTWARE QUALITY USING AN ONTOLOGY-BASED APPROACH

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

In

The Department of Computer Science

By

Yixin Luo
B.S., Wuhan University of Technology, 1991
M.S., Southern University, 2001
May 2010

谨将此论文
献给敬爱的父亲，罗孝杨
对您的怀念将伴我一生
和我的家人
母亲，蒋亚平
爷爷， 罗重秀 奶奶， 蒋金莲
外公， 蒋荣陪 外婆， 黄淑华
公公， 周垠庚 阿婆， 蒋宏英
岳母， 曾桂芳
你们的关爱是我完成此文的动力之源

Acknowledgements

I express my deepest gratitude to Dr. Doris Carver, my supervising professor, for her guidance, patience, and encouragement throughout my graduate studies. It is an honor and a blessing for being her student. She has a great knowledge and her wisdom is second to no one. Also, her honesty and integrity is my moral model.

I wish to express sincere thanks to Dr. Allyson Hoss for her advice. She was persistent in her encourage and support. She had spent a lot of time on my research.

As a full time employee while trying to pursue the degree, I would express my heartfelt appreciation to my employers, Dr. Gregory Stone, the director of Coastal Studies Institute, Dr. James Lupo, my supervisor at Computer Center for Technologies (CCT), and Dr. Honggao Liu, the director of High Performance Computing (HPC) for their active support and fully cooperation.

Also, my gratitude extends to my committee members Dr. Jianhua Chen, Dr. Don Kraft, Dr, Yesho Chen, and Dr. Roberto Barbosa for their advice and cooperation.

I give a special acknowledgment to all the software engineers who participated in my statistical survey and gave their advices on my research.

I am grateful for my parents, Mr. and Mrs. Xiaoyang Luo and Yaping Jiang as well as my sister, Yunfei who are always my source of motivation and encourage me to study by their examples. I would like to sincerely thank my mother-in-law, Guifang Zeng, who handled the tasks that were due to me.

I would like to thank my wife, Yao and my daughters, Muzi and Mulan for everything.

Table of Contents

List of Tables

List of Figures

Abstract

Ensuring quality in software development is a challenging process. The concepts of anti-patterns and bad code smells utilize the knowledge of reoccurring problems to improve the quality of current and future software development. Anti-patterns describe recurring bad design solutions while bad code smells describe source code that is error-free but difficult to understand and maintain. Code refactoring aims to remove bad code smells without changing a program's functionality while improving program quality. There are metrics-based tools to detect a few bad code smells from source code; however, the knowledge and understanding of these indicators of low quality software are still insufficient to resolve many of the problems they represent. Minimal research addresses the relationships between or among bad code smells, anti-patterns and refactoring. In this research, we present a new ontology, Ontology for Anti-patterns, Bad Code Smells and Refactoring (OABR), to define the concepts and their relation properties. Such an ontological infrastructure encourages a common understanding of these concepts among the software community and provides more concise definitions that help to avoid overlapping and inconsistent description. It utilizes reasoning capabilities associated with ontology to analyze the software development domain and offer new insights into the domain. Software quality issues such as understandability and maintainability can be improved by identifying and resolving anti-patterns associated with code smells as well as preventing bad code smells before coding begins.

Chapter 1
Introduction

The production and maintenance of quality software continue to provide challenges to software developers. Many problems that cause failure of software products are chronic and reoccurring. Attributes describing software quality include reliability, efficiency/effectiveness, human engineering, understandability, modifiability/reusability, testability/functionality, and portability/extendibility [Glass][Bansiya]. Researchers have suggested pattern-based concepts and solutions such as design patterns, anti-patterns, bad code smells, and refactoring to help improve software quality by giving general descriptions of time-tested solutions to reoccurring problems. Using knowledge from these pattern-based concepts in software development, software developers can improve software quality.

One clear and concise definition of the term pattern is "a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves" [Gabriel]. The major benefits of pattern concepts are that they provide proven solutions to solve software common issues, and they can improve understanding among software agents, making communication between agents more efficient.

1.1 Design Patterns, Anti-patterns and Bad Code Smells

Design patterns are descriptions of communicating objects and modules that are customized to solve a general design problem in a particular context [Gamma et al.]. A design pattern refers to both the description of a solution and an instance of the solution

for solving a particular problem.

Anti-patterns, like their design pattern counterparts, are literary forms that describe a commonly occurring solution to a problem that generates decidedly negative consequences [Brown]. Bad code smells refer to source code structures problems, and refactoring addresses the resolution for anti-patterns and code smells.

This research addresses chronic problems that arise in software development by identifying relations between anti-patterns and bad code smells and using the relations to help solve or prevent problems. Figure 1.1 shows a high-level view of the applications of anti-patterns, bad code smells, and refactoring solutions in software development. Design patterns and anti-patterns "guide" the design of software. Software design "creates" source code. Source code can "contain" bad code smells related problems. Refactoring helps "solve" the source code problems described by bad code smells and the problems created by anti-patterns. This research highlights the differences between anti-patterns and bad code smells. Anti-patterns usually occur at design, and code smells occur at the coding stage. Design patterns and anti-patterns can prevent bad code smells related problems.

1.2 Costs of Solutions

It is widely accepted in the software industry that the cost of fixing a problem rises dramatically when its discovery occurs in later phases of the software life cycle, because there are more deliverables affected by each correction. No data accurately reflects the exact cost differences, but rough estimations exist. Table 1.1 shows the significant cost differences for fixing software problems at different stages of the software life cycle [Mogyorodi]. This data was gathered from software products

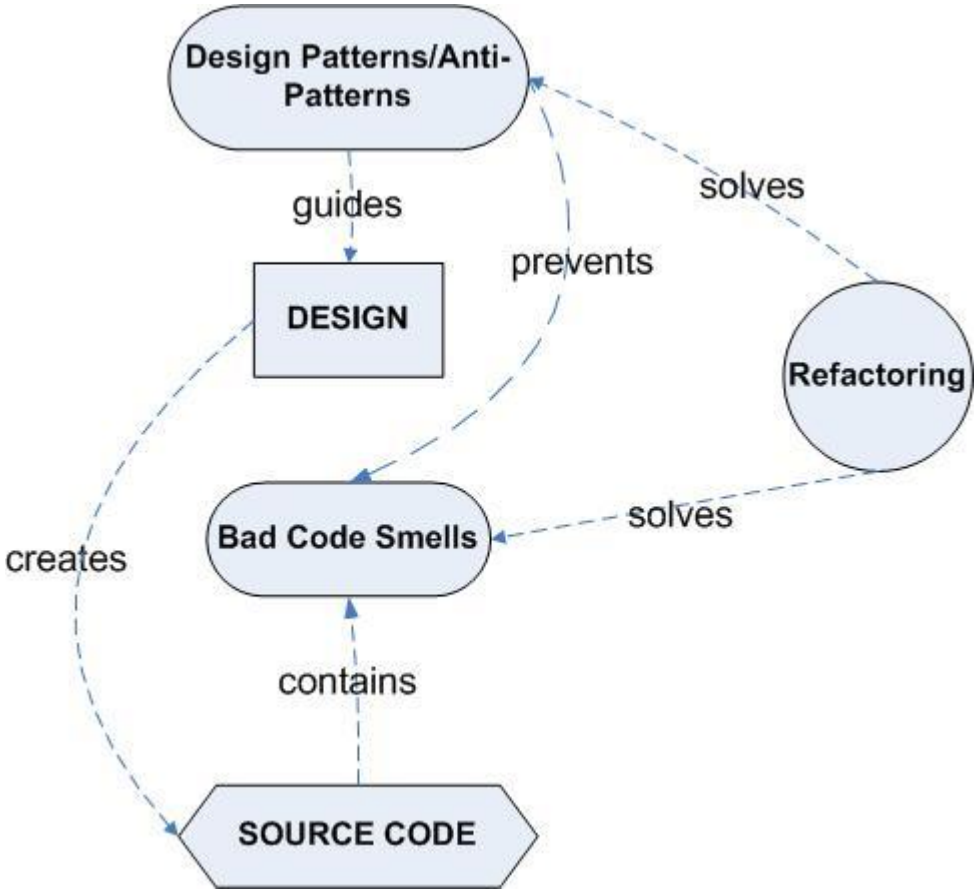


Figure 1.1: Relations among Design Patterns, Anti Patterns, Bad Code Smells, and Refactoring

developed by software industries such as IBM and GTE. For example, if the cost of fixing a problem (bug) at the requirements level were one dollar, the cost of fixing it at the testing level would be 15 to 40 dollars.

Table 1.1: Relative Cost to Fix a Problem [Mogyorodi]

Phase	Cost Ratio
Requirements	1
Design	3-6
Coding	10
Testing	15-40
System/Acceptance Testing	30-70
Production	40-1000
(IBM, GTE, et al.)	

Bad code smells related problems are found only in the source code, making them costly to fix. Anti-patterns could help software developers to identify and prevent the problems at the early stage of the software life cycle, thus reducing the cost.

1.3 Research Objective

This research is motivated by the need to eliminate chronic software problems and the lack of research consistently defining anti-patterns, bad code smells, and refactoring as well as detailing the relations between anti-patterns, bad code smells and refactoring. Existing taxonomies and techniques identifying and defining bad code smells and anti-patterns are based on informal human intuition that is both manual and heuristic. Previous work lacks sufficient formal descriptions and classifications of bad code smell

anti-pattern, and refactoring. Also, existing identification and solutions of anti-patterns and code smells are neither automatic nor systematic and are often overlapping, inconsistent, and inaccurate.

The research objective is to improve software quality by detecting and removing software problems that are defined by anti-patterns and bad code smells. We also aim to improve sharing and understanding of anti-patterns, bad code smells, and refactoring as well as their relations in the software community.

The general methodology is to develop and utilize a new ontology, Ontology for Anti-patterns, Bad Codes Smells, and Refactoring techniques (OABR). We develop the conceptual domain model for OABR to identify the concepts and the relations between the concepts. Next, we define the properties for the OABR foundational concepts like anti-patterns, bad code smells, and refactoring. We also define priority indexes for each bad code smell that will help identify which bad code smells should be removed or prevented and which should be tolerated. We develop templates for code smells and refactoring to provide a consistent outline for documentation. The formally defined concepts and relations will improve the understanding and help to find a new taxonomy. The relations between code smells and anti-patterns will help prevent problems related to bad code smells by detecting and resolving software problems early in the software development life cycle, thereby saving the cost of identification and removal at coding level. Finally, we apply ontological tools to implement and validate OABR.

1.4 Dissertation Overview

Chapter 2 describes anti-patterns, bad code smells, and refactoring. It also reviews

other works that are related to this research. Chapter 3 presents the background and benefits of ontology as well as a conceptual model of anti-patterns, bad code smells, software problems, and refactoring in OABR. Chapter 4 presents the development and process of the OABR infrastructure. Chapter 5 describes the application and validation of OABR. Finally, Chapter 6 summarizes this research and discusses the future work.

1.5 Summary

Design patterns, anti-patterns and bad code smells with refactoring aim to describe and help solve chronic software problems that cause failure of software projects. An anti-pattern is a bad solution that causes problems, a bad code smell refers to source code structure problems, and refactoring addresses their resolution. Many anti-patterns happen at early stages of the software life cycle, and all the bad code smells occur in source code. This research develops an ontology-based approach to provide a detailed description of anti-patterns, bad code smells, refactoring, detection, and their relations with the goal of improving understanding about these concepts among software developers. Detecting and removing software problems at early stages of the life cycle help developers to reduce the costs of development and maintenance of software projects.

Chapter 2
Background and Related Research

2.1 Introduction

Reusable solutions such as design patterns, anti-patterns, bad code smells and related refactoring have been shown to be efficient for improving understandability of software reoccurring problems among software developers [Fowler][Brown et al.]. Use of these concepts provides guidance to improve the quality and standards of the software industry, map a general situation to a specific class of solutions, and improve understanding among the software communities by providing a common vocabulary for identifying problems and discussing solutions. Section 2.2 provides an overview on the foundational concepts of this research. Section 2.3 describes software quality attributes and the impact of design patterns, anti-patterns, bad codes smells and refactoring on the software quality attributes. Section 2.4 reviews existing related research.

2.2 Related Concepts

This section elaborates on anti-patterns, bad code smells, and refactoring as well as their application in software developments. It includes an introduction to design patterns as it relates to the other foundational concepts.

2.2.1 Design Patterns

In 1995, Gamma, Helm, Johnson, and Vlissides (frequently referred to as the Gang of Four (GoF)) introduced and described design patterns as “Recurring solutions to software design problems that are repeatedly found in real-world application development” [Gamma et al.], to help address software development problems. Currently, there are 23 Gamma Patterns defined by the GOF, 17 Buschmann Patterns,

72 Analysis Patterns, 38 CORBA Design Patterns, and 95 Anti-patterns. Gamma patterns describe how to design [Gamma]. Buschmann Patterns cover core elements of building concurrent and network systems such as service access and configuration, event handling, synchronization, and concurrency [Buschmann]. Analysis Patterns focus on object-oriented analysis and design [Fowler]. CORBA patterns give solutions for designing and building distributed object-oriented systems [Mowbray & Malveau]. Anti-patterns are the extension and the counterpart of design patterns [Brown].
Each pattern definition typically includes some of the following fundamental elements [Gamma][Brown]:

- Name describes the problem and solutions by vocabulary.
- Intent describes the goal behind the pattern and the reason for using it.
- Problem gives the context and description of problems that would be solved by design patterns.
- Forces introduce a scenario in which the pattern can be used.
- Solutions give an abstract description of the solution and its constraints.
- Consequences provide the results and tradeoffs.

An example of a design pattern is the Factory Method Pattern. This pattern addresses the problem when the application class does not know when to instantiate a new object of a class or what kind of subclass to create. The Factory Method Pattern offers the solution of creating a pattern that helps to model an interface for creating an object, which can let its subclass decide which class to instantiate at creation time.

2.2.2 Anti-Patterns

The original work of GoF does not mention the concept of anti-patterns. In 1998, Brown, Malveau, McCormick, Mowbray, the Anti Gang of Four (AGoF), suggested the

concept of an anti-pattern as "a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences" to help address recurring bad design solutions [Brown et al.]. Like design patterns, the main goal of anti-patterns is to prevent chronic problems from reoccurring. AGof claimed that anti-patterns are a natural extension to, but the opposite of, design patterns. When a design pattern creates more problems than it solves, it becomes an anti-pattern. Anti-patterns are studied as a category so that they can be avoided.

Software design involves making choices that are often complex with many issues to consider such as reliability, cost, schedule, and adaptability. Usually, anti-patterns originate from lack of experience of software developers or from the use of good design patterns in the wrong context [Smith & Lioyd].

Anti-patterns are an effective way to capture knowledge, transfer ideas, and foster communication. They provide the following benefits to software development and maintenance [Brown et al.]:

- "a method of efficiently mapping a general situation to a specific class of solutions"
- "real world experience in recognizing recurring problems in software industry"
- "a common vocabulary for identifying problems and discussing solutions"
- "a holistic resolution of conflicts."

The following fundamental elements describe anti-patterns [Vesa]:

- Name describes the problem and solutions by vocabulary;
- Anti-Pattern Solution gives the symptoms, consequences and abstract description of the problematic solution;

- Refactored Solution provides the description of refactoring methods and positive consequences of the refactoring.

No standard classification or taxonomy exists for describing anti-patterns. AGoF categorizes 42 types of anti-patterns based on different stages of the software life cycle. The AGoF defined the design anti-patterns, architectural anti-patterns, and management anti-patterns categories shown in Table 2.1. Other classifications include project management anti-patterns, general design anti-patterns, programming anti-patterns, methodologies anti-patterns, and configuration anti-patterns [Brown & Thomas].

Utilizing anti-patterns to prevent chronic problems from reoccurring includes the following three steps:

1. Anti-pattern Identification - describes how to recognize the general form;
2. Anti-pattern Removal - describes the refactored solutions to change the anti-patterns into a sound design pattern;
3. Verification – describes the validation methods to prove that the anti-pattern has been removed.

There are two widely accepted basic rules to recognize and process anti-patterns [Laplante]:

- Rule 1: (“Rule of three”) “Someone must have experiences and report each anti-pattern (and a successful refactoring) in three separate instances” [Laplante];
- Rule 2: “It is a high risk to process several anti-patterns simultaneously.” [Brown].

Table 2.1: Categories of Anti-patterns [Brown et al.].

Categories	Examples
Management Anti-Pattern	Metric Abuse: the malicious or incompetent use of metrics and measurement
Project Management Anti-pattern	Smoke and Mirrors: demonstrating how unimplemented functions will appear
General Design Patterns	Ambiguous viewpoint: Presenting a model without specifying its viewpoint
OO Design Patterns	God object: Concentrating too many functions in single part of the design
Programming Patterns	Lava flow: Retaining undesirable (redundant or low-quality) code because removing it is too expensive or has unpredictable consequences
Methodological Management Anti-patterns	Copy and paste programming: Copying (and modifying) existing code rather than creating generic solutions
Configuration Management Anti-patterns	DLL hell: Problems with versions, availability and multiplication of Dynamic-Link Library|DLLs, specifically on Microsoft Windows

Rule 1 shows that anti-pattern definition and identification is heuristic. The logical basis for the Rule of Three is that the first occurrence shows that the design does not work; the second occurrence shows that the design problem is interesting; and the third

occurrence suggests that it appears to have a wider applicability. The informal concept behind the Rule of three is: “the first occurrence is an event, the second occurrence is a coincidence, and the third occurrence may be a pattern” [Sabt et al.].

Rule 2 refers to resolving anti-patterns. It suggests that the processing of anti-patterns is not easy and that simultaneously processing of several anti-patterns is hard to control and can cause new problems.

Existing research organizes anti-patterns using templates. Examples of anti-pattern are the Blob and the Spaghetti Code described in template form developed by [Brown et al.] and shown in Figures 2.1 and 2.2, respectively. These templates describe anti-patterns by providing knowledge such as an informal cause analysis and refactoring solutions for solving the anti-pattern.

THE BLOB
Anti-pattern Name: The Blob
Also Known As: Winnebago and The God Class
Most Frequent Scale: Application
Refactored Solution Name: Refactoring of Responsibilities
Refactored Solution Type: Software
Root Causes: Sloth, Haste
Unbalanced Forces: Management of Functionality, Performance, Complexity
Anecdotal Evidence: “This is the class that is really the heart of the architecture”.

Figure 2.1: An Example of Anti-pattern Blob [Brown et al.]

SPAGHETTI CODE
Anti-pattern Name: Spaghetti code
Also Known As: N/A
Most Frequent Scale: Application
Refactored Solution Name: Software Refactoring, Code Cleanup
Refactored Solution Type: Software
Root Causes: Sloth, Ignorance
Unbalanced Forces: Management of Change, Complexity
Anecdotal Evidence: “It is for future modification and extension”.

Figure 2.2: An Example of Anti-pattern Spaghetti Code [Brown et al.]

2.2.3 Bad Code Smells

Fowler suggested the concept of bad code smells [Fowler] with the following introductory definition: “A bad code smell is a structure that needs to be removed from the source code by refactoring to improve the maintainability of the software.” Bad code smells are defined and organized in an informal manner.

A bad code smell itself is not a problem but a sign of a problem. It shows poor structure and poor qualities of software products. Bad code smells are not the same as syntax errors or compiler warnings. Bad code smells are indications of bad program design or bad programming practices. Bad code smells are not errors, but they could make software projects difficult to develop and maintain when the program needs modification. Examples of bad code smells include “Large Class” which means a class is doing too many things and that results in too many instance variables, “Duplicated Code” which means the same code structure exists in more than one place, and “Long Methods” which are methods that are too long to understand and reuse. Bad code smells can be removed by applying refactoring methods.

2.2.4 Differences between Anti-patterns and Bad Code Smells

Although both bad code smells and anti-patterns describe re-occurring software problems, a major difference is that the development of anti-patterns is generally at a more abstract and higher level, like design level [Mika]. However, as current definitions of anti-patterns and bad code smells are heuristic, thus, incomplete and inconsistent, the definitions of some anti-patterns, such as the software development anti-patterns, are similar to bad code smells and even have some overlap.

Table 2.2 shows comparisons between anti-patterns and bad code smells.

Table 2.2: Anti-Pattern vs. Bad Code Smell

	Bad Code Smell	Anti-pattern
Number of Distinct Types	23	95 (growing)
Software Development Stage(s)	Source code level	Entire Software Life-cycle
Contents	Symptoms	Causes + Solutions
People	Programmers	Managers, Architects, Designers, and Developers
Goals	Tells developers when to refactoring	How to prevent chronic design problems
Identification	Heuristics + Metrics	Heuristics
Proof of existence	None	Rule of three
Solutions	Refactoring	Refactoring
Format	English expression	More formal templates
Known Causes	No	Yes
Removal cost	Expensive	N/A

Bad code smells are the symptoms of problems existing in the source code and indicate when refactoring is needed. Anti-patterns give developers a way to recognize software problems in advance to help avoid most common pitfalls. Bad code smells give warnings to programmers that something may be wrong with the source code, while anti-patterns provide software managers, architects, designers, and developers a common vocabulary for recognizing possible sources of problems in advance.

Metrics-based tools can detect some bad code smells automatically while the identification of most anti-patterns is based on heuristic analysis. The proof of existence of anti-patterns is based on the obscure "Rule of three". No rules exist for proving the existence of bad code smells. Although the solution to both anti-patterns and bad code smells is refactoring, the refactoring methods for bad code smells are more technical and programming-based while refactoring for anti-patterns are "approaches for evolving the solution into a better one" [Brown et al.]. Anti-patterns are organized in a semi-formal template while code smells are described in plain English. Each anti-pattern is given cause analysis, while bad code smells are descriptions of symptoms. The identification of bad code smells is at the developing, testing, and maintenance levels of life cycle and, therefore, the cost of removal is high.

2.2.5 Refactoring

When a design pattern becomes an anti-pattern, it is useful to have an approach for evolving the anti-pattern back into a good design pattern. Also, removal of a bad code smell will improve the structure and quality of source code. Solutions for both bad code smells and anti-patterns are based on refactoring. "This process of change, migration, or evolution is called refactoring" [Ciupke]. Refactoring refers to the algorithms or

methodologies that are used to remove bad code smells/anti-patterns. A series of small refactoring could require a significant restructuring of code. Table 2.3 lists examples of some of the 80 types of refactoring given in [Fowler] along with bad code smells. For example, "Extract Class" could be used to fix the code smells of Large Class, Duplicated Code, Data Clumps, and Divergent Change. Code smells such as "Shot Gun Surgery" would need two or more refactoring methods to solve, such as "Move Method" and "Inline Class".

Table 2.3: Refactoring and Bad Code Smells

Refactoring Technique	Bad Code Smell
Extract Class	Large Class
	Duplicated Code
	Data Clumps
	Divergent Change
Move Method	Alternative Classes with Different Interfaces
	Data Class
	Feature Envy
	Shotgun Surgery
Inline Class	Lazy Class
	Short Gun Surgery
	Speculative Generality

We define refactoring for bad code smells as:

(Functionality) R(C) ~ C

where

R is a refactoring operation

C is a code segment

"~" equivalent

Figure 2.3 shows an example of a refactoring named "Extract Class".

The class "Lab" on the left describes the name and obtains the address. The extracting of this class into the Lab and Address classes will make the classes more understandable and easier to reuse.

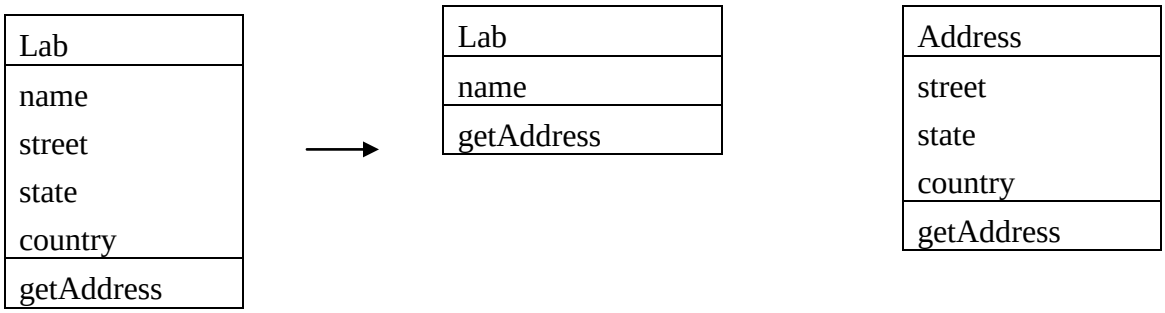


Figure 2.3: An Example of Refactoring (Extract Class)

2.3 Software Quality Attributes

ISO 9126 is the software product evaluation standard from the International Organization for Standardization. ISO 9126 part one, also referred as ISO 9126-1, defines the following six software quality attributes [ISO9126]:

- Functionality – "A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs."
- Reliability – "A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.

- Usability – “A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.”
- Efficiency – “A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions. “
- Maintainability – “A set of attributes that bear on the effort needed to make specified modifications.”
- Portability – “A set of attributes that bear on the ability of software to be transferred from one environment to another.”

The application of design patterns, anti-patterns, bad code smells, and refactoring have benefits for improving the software quality attributes. Table 2.4 summarizes their impacts on related attributes.

Table 2.4: Impact of Design Patterns, Anti-patterns, Code Smells and Refactoring on Software Quality [Gammar][Brown][Fowler][Mens]

	functionality	reliability	usability	efficiency	maintainability	portability
Design Pattern	N/A	+	+	N/A	+	N/A
Anti-Pattern	N/A	-	-	N/A	-	N/A
Code Smell	N/A	N/A	-	N/A	-	N/A
Refactoring	N/A	+	+	N/A	+	N/A

‘ + ’ positively impact ‘ – ‘ negative impact ‘N/A’ not available

The applications of design patterns will improve quality attributes such as reliability, usability, and maintainability of software products while anti-patterns, the opposite of design patterns, negatively impact software quality attributes. Bad code smells mostly affect the usability and maintainability software quality attributes [Fowler].
Refactoring by definition aims to improve the reliability, maintainability and usability of software products [Mens]. However, the impacts of anti-patterns and bad code smells on other quality attributes are unclear.

2.4 Related Work

Numerous researchers have shown interest in anti-patterns, bad code smells and refactoring in recent years. Most research is about representation, description, and classification of anti-patterns and bad code smells. Also, the software community has developed metric-based tools for the identification and removal of anti-patterns and code smells. Table 2.5 gives an overview of the selected works focus on the identification and classification of anti-patterns, bad code smells as well as refactoring methods. The detection column indicates how detection is performed (by metrics-based tools or heuristically) and what is detected (code smells or anti-patterns). The classification column describes how to classify anti-patterns and bad code smells, and what techniques are used, such as taxonomy or ontology.
Akroyd was the first one that documented problematic software constructs [Akroyd], and Konig introduced the term of anti-pattern [Konig]. Brown et al. prompted the term by suggesting 42 types of anti-patterns and describing them by using a uniform template. Their work analyzed the causes, root causes, symptoms, and solutions of their anti-patterns [Brown et al.].

Laplante developed a new catalog of anti-patterns that extends and complements Brown's work. The catalog covers management anti-pattern, environmental or cultural anti-patterns, and personality anti-patterns that help to correct problem identification and provide solutions [Laplante et al.]. They listed and summarized 21 management anti-patterns and 27 environmental anti-patterns. The structure described for each anti-pattern involved 'Name', 'Central Concept', 'Dysfunction', 'Vignette', 'Explanation', 'Band Aid', 'Self-Repair', 'Refactoring', 'Observations', and 'Identification'. The structures are not a formal structure.

Fowler et al. showed the related refactoring methods for bad code smells. They also suggested that no substitution can replace human intuition when it comes to deciding whether a certain code smell should be refactored and that no precise criteria for evaluating code smells can be given [Fowler & Becker].

Mika challenged Fowler's viewpoints by suggesting the following [Mika et al.]:

1. Automatic bad code smell measurement was possible if their measurability could be addressed; Mika applied source code metrics for certain code smells;
2. Fowler's description of bad code smells was not organized and was not clear to understand. Mika presented a taxonomy based on shorter concepts from a larger context;
3. Programmers should have a common view in order to utilize bad code smells as indicators of software defects.

Mika also presented an initial empirical study on the subjective evaluation of bad code smells and provided a new view by using perceived evaluations. They selected Large Class, Long Parameter List, and Duplicate code as a basis for subjective code

Table 2.5: Comparison of Related Work on Anti-patterns and Bad Code Smells

	Anti-patterns		Bad Code Smells	
	Detection	Classifications	Detection	Classification
Akroyd, Konig, Brown, Laplante	Heuristics	Software life cycle	None	None
Fowler & Beck	None	None	Heuristics	None
Mika et al.	None	None	Metrics-based analysis	Taxonomy based on symptom analysis
Radu et al.	Metrics-based detection on three anti-patterns	None	None	None
Emden, Moonen and Slinger	None	None	Metrics-based tools	None
Cheng et al.	None	None	None	Ontology classification of bad smells
Moha et al.	Metric-based heuristics and structure and semantic information	Classification based on key concepts	Metric-based heuristics and structure and semantic information	Classification based on key concepts

evaluation**.** They compared the results from subjective evaluation to those obtained from metrics-based tools and found that the results between subject evaluation and source code metrics do not correlate, suggesting that subjective evaluations are "greatly affected by conflicting perceptions of different developers" [Mika&Lassenius]. On the

other hand, Mika acknowledged that metrics-based tools are not always reliable.

Mika defined a simple taxonomy of bad code smells based on heuristic analysis from his programming experiences [Mika]. Although the taxonomy can improve the understanding of bad code smells and suggest common properties of several bad code smells, it is not as powerful as an ontological representation because it cannot explain what causes a bad code smell; how to prevent it from occurring; or the internal relationships between code smells and refactoring, anti-patterns, and detections.

Radu et al. developed a set of detection strategies based on software metrics to detect several anti-patterns [Redu et al.][Chidamber & Kemerer][Trifu & Marinewscu]. They later refined the methodologies by using "Historical information of the suspected flawed structures" [Radu]. They showed how to detect anti-patterns such as God Classes and Data Classes, and they indicated that their approach refines the properties of anti-patterns, which leads to a two-fold benefit:

1. identify "harmless" anti-patterns with the help of history information by a single-version detection strategy, and
2. using additional information over their analyzed history could identify "most dangerous" anti-patterns [Radu].

However, they found that the selection of metrics is heuristic, and their metrics based tools cannot find most other bad code smells and anti-patterns.

Emden and Mooned developed a bad code smell detecting tool named jCOSMO, a Java-based code smell detector that can also be used with other tools. They showed how to automatically detect code smells by breaking them up [Emden & Moonen], and they also described how the bad code smell concept might be expanded to include

coding standard conformance. They suggested that their bad code smell detector is fully automated and can show bad code smells graphically. Slinger developed a prototype of code smell detection plug-in for the Eclipse IDE framework [Slinger].

Cheng and Liao created a taxonomy for bad code smells that is closely related to anti-patterns. Their taxonomy contained three-levels, including description, detection symptoms, and properties. They referred to their taxonomy as "an ontology based taxonomy" because it included semantic relations between objects in the domain. The semantic relations include that "companion smells" describes bad code smells that often accompany conditional statements smells, and "causal links" describes the possible relations between bad code smells [Cheng & Liao].

Moha introduced a methodology based on a meta-model to detect and correct high-level design defects utilizing refactorings. She defined design defects as bad code smells and anti-patterns that include the Blob, the Functional Decomposition, the Spaghetti Code and the Swiss Army Knife. Moha's meta-model approach and our ontology-based approach share a common goal of improving software quality by identifying and removing design and programming defects [Moha][Moha2][Moha et al.]. Other similarities include analysis of textual descriptions of design defects to identify key words used to define a common vocabulary, development of taxonomy to describe design defects, and validation methods that include a survey within the software community.

Meta-model and ontology are different in that an ontology is descriptive and belongs to the domain of the problem, but a meta-model is prescriptive and belongs to the domain of the solution. Ontology is specially suited for knowledge models. Some researchers

suggest that using the same meta-model without an ontology can cause different knowledge representations of the same domain to be incompatible [Lee].

This research distinguishes itself from other works via its ontological approach that covers several related concepts and capitalizes on the advantages ontologies offer such as ontological tools and associated platforms to facilitate the establishment of class hierarchy, development of rules and axioms, and reasoning about relationships.

2.5 Summary

Anti-patterns are the extension and opposite of the design patterns, and they are classes of bad solutions to the problems. Bad code smells are the symptoms of problems in the software source code. The existence of many bad code smells is a strong indication of poor source code structure. Refactoring provides step-by-step solutions to improve the quality of software by removing bad code smells and anti-patterns. However, as the refactoring of bad code smells occur usually after the coding level of the software life cycle, it can be very costly.

There is a lack of efficient methods to define, identify and analysis these concepts and their relations. Chapter 3 introduces the ontology concept and its application in this research.

Chapter 3
Ontological Representations

This chapter provides a general introduction to ontologies; a high-level view of the OABR representation of anti-patterns, bad code smells, refactoring, and software problems; and the relevance of ontologies to this research.

3.1 Ontology Background

In computer science, the term "ontology" refers to a "data model that presents a specific part of the real-world and is used to reason about the relationships of objects or concepts in the world" [Lee & Meier]. Another definition of ontology focuses on the form of an ontology – "An ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary" [Calero & Piattini]. Ontology is interpreted as the formal representation of a conceptualization and as an efficient knowledge engineering technique useful in representing concepts in a formal way. A general ontology contains the following parts [Noy & McGuinness]:

- "Instances are the basic components of ontology such as people, animals, tables, automobiles, molecules, and planets.
- **"**Class (Concept) represents a concept in a domain or a collection of elements of an instance of the concept with similar properties such as a person (the class of all people), a molecule (the class of all molecules), and car (the class of all cars)."
- **"**Property describes the structure of a concept."

- **“**Relations show the properties and values.”
- **“**Attributes are data types that consist of name and values.”

Generally, an ontology is not a taxonomy or classification, though they often look alike. An ontology provides more richness of information than a classification. In addition to the concepts that both ontology and classification provide, ontology also includes the relations among the concepts [Reinout].

A taxonomy is “a system for naming and organizing things into groups, which share similar qualities” [Cambridge Dictionary]. A taxonomy can make objects easier to understand, help recognize the relationship between the objects, and help understand the larger context for each object.

The major difference between an ontology and a taxonomy is that rules and constraints are defined in ontology:

vocabulary + structure = taxonomy

taxonomy + relationship + constraint and rules = ontology [TopQuadrant]

Figure 3.1 shows a portion of a simple ontology for pets. In the pet ontology, the related terms could be cat, dogs, pet, color, location, weight, hair, and age. Pet is the top class while Cat and Dog are subclasses of Pet. The Pet class has the attributes or properties such as color, location, weight, age, and hair. All the sub-classes of Pet also inherit these properties. A cat name “Romeo” is an instance of the Cat class. The non-taxonomic relations in this ontology are like “Dogs->chase->cats”

Figure 3.2 provides a more complex example of an ontology that contains concepts and relations similar to the ontology used in this research. It describes interrelationships between abnormal life, symptoms of diseases, detection, and treatments.

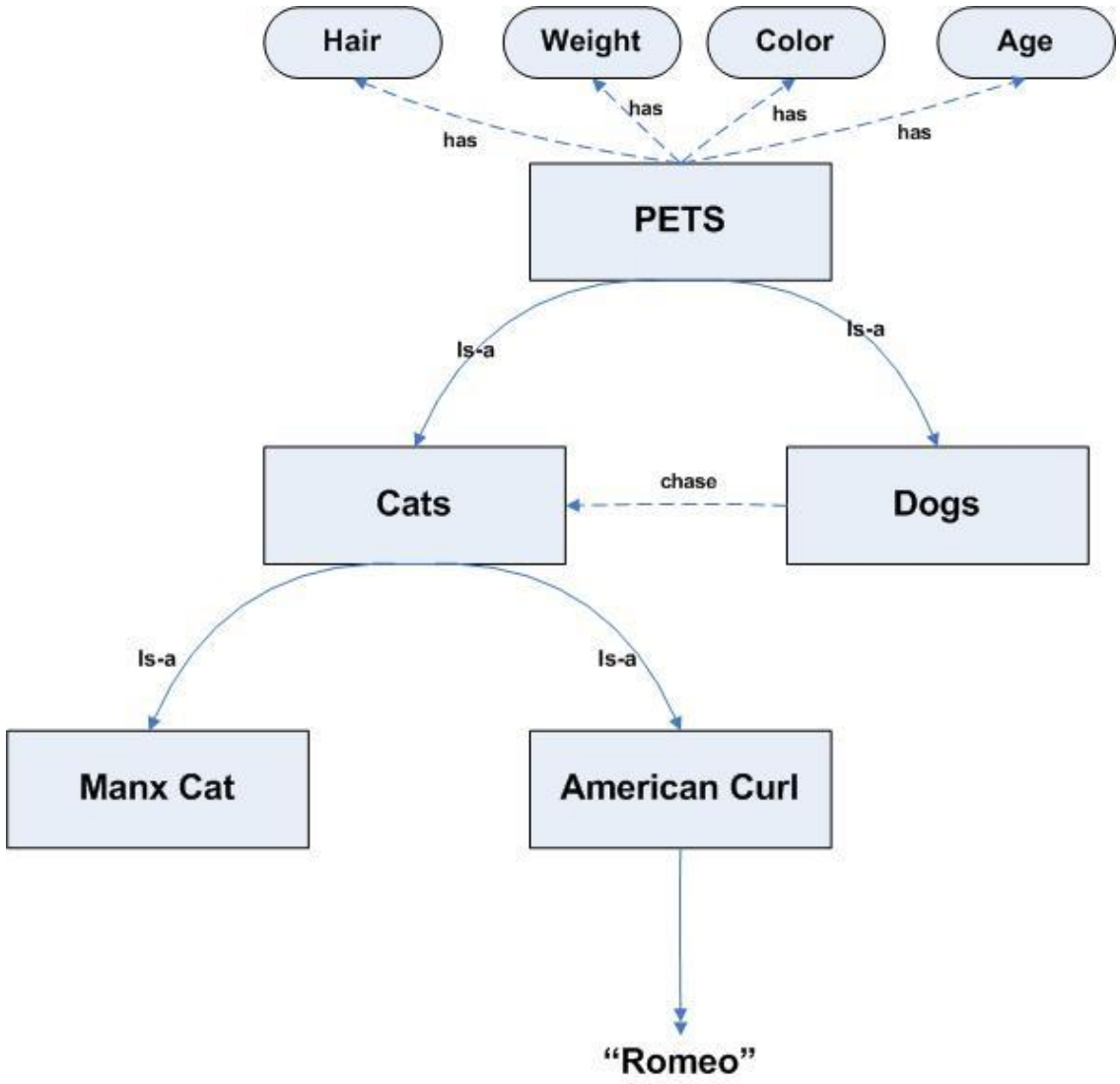


Figure 3.1: A Simple Example of Ontology

It shows a partial ontology representing human illnesses and its interrelationships with partial ontologies for anti-patterns in the form of abnormal life, bad code smells in the form of disease and symptoms, refactoring in the form of treatment, and metrics-based tools in the form of medical detection tools. It also shows Symptoms, Illness (disease), Medical Detection Tools, Treatment, and Abnormal life. Symptoms describe diseases that are the signs of problems in human bodies. There are thousands of symptoms

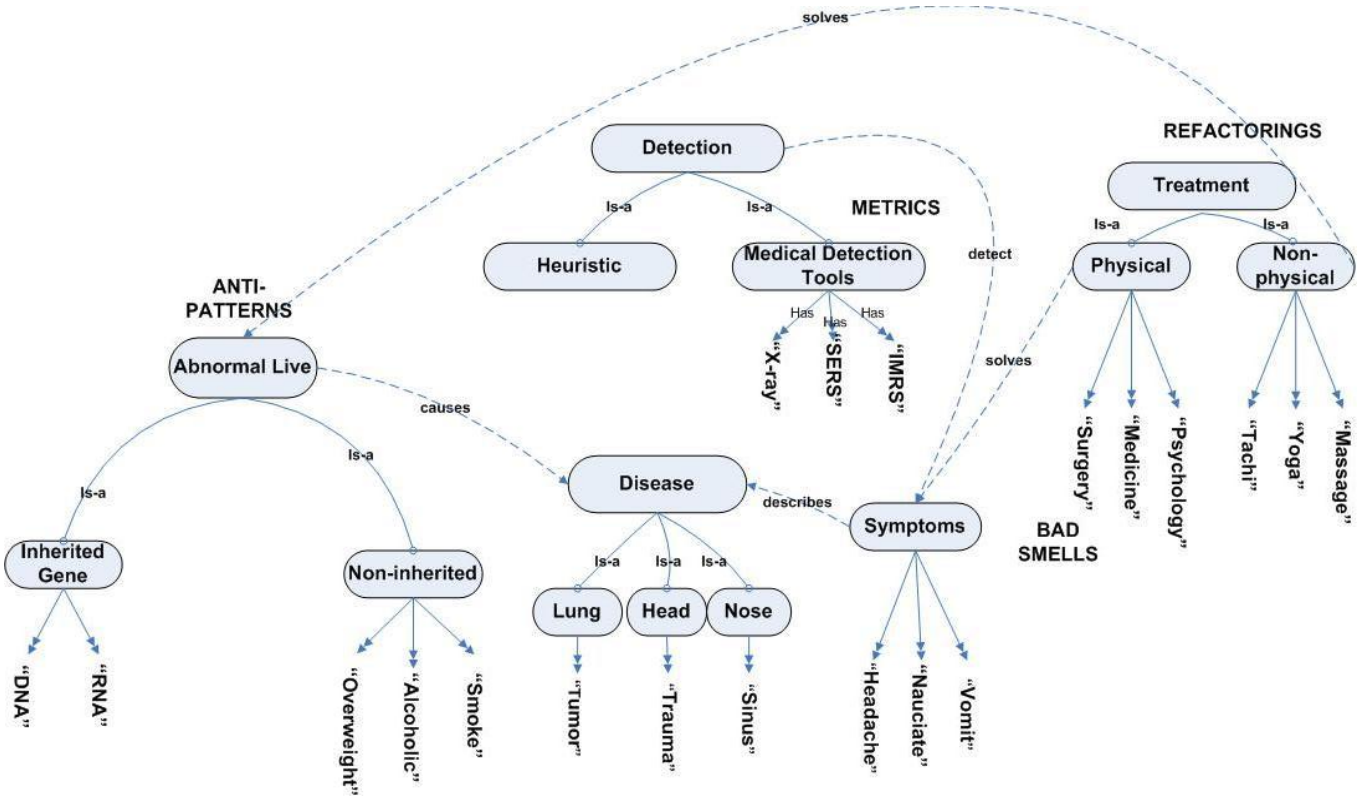


Figure 3.2: A Partial Ontology Representing Human Illnesses

that might be signs of disease/illness. In the diagram, we show three common symptoms - "Headache", "Nausea", and "Vomiting". Diseases are classified according to the different parts of human body such as head, heart, and lung. For example, various diseases might cause a headache. A disease could be very serious such as a tumor, or not so serious such as sinus. Medical tools detect symptoms based disease. For example, an X-ray instrument could detect tumors but might not be reliable. Tools could detect diseases as for treatments. Some diseases must be treated while others might be tolerated considering the medical cost. Finally, abnormal Life causes diseases. Examples of abnormal life include "overweight", "too little rest", or gene problems. If people could solve the problems of abnormal life, the diseases might be prevented. Thus, medical treatment cost could be reduced.

In general, the ontology includes the following non-taxonomic relations:

Abnormal life->causes->disease

Treatment->solves->symptoms

Detection->detects->symptoms

Symptoms->describe->diseases.

3.2 Ontological Representations of Anti-patterns, Code Smells, and Refactoring

Figure 3.3 shows a high-level view of the OABR conceptual domain model. For example, anti-patterns cause software problems that can be source code problems or other problems. The source code problems can be either errors or poor codes. Bad code smells describe the symptoms of poor codes. Metrics can detect bad code smells, and refactoring methods can solve bad code smells. Section 4.7 contains a detailed view of OABR that facilitates the depiction of the interrelationships between and

among these software concepts.

The ontological representations provide a systematic approach toward defining the properties of anti-patterns, code smells and refactoring as well as analyzing the interrelationships between them. Figure 3.3 has similarities in common with the concepts, relationships, and constraints shown in Figure 3.2. Bad code smells are like symptoms of disease. Both concepts describe something wrong, and they can be removed by addressing related problems. Both symptoms and bad code smells can be detected heuristically or by tools. Anti-patterns and abnormal life are the causes of problems. If they could be prevented, related chronic problems could be prevented from reoccurring. Thus, the cost and risk would be reduced. Finally, refactoring is similar to treatments that aim to solve problems.

3.3 Why Use Ontological Representation

We utilize a new ontology, OABR, to provide a formalized description of anti-patterns, bad code smells, refactoring, and their relations. The use of OABR offers the following advantages:

- sharing and improving common understanding of bad code smells, anti-patterns, refactoring, and detection techniques among the software community because they share the same underlying ontology of the terms;
- providing the reasoning capabilities associated with ontologies to analyze the domain and offer new insights. The ontology can assist to find new anti-patterns, new bad code smells, new identification methods, and refactoring methods;
- enabling reuse of domain knowledge by other researchers to develop additional

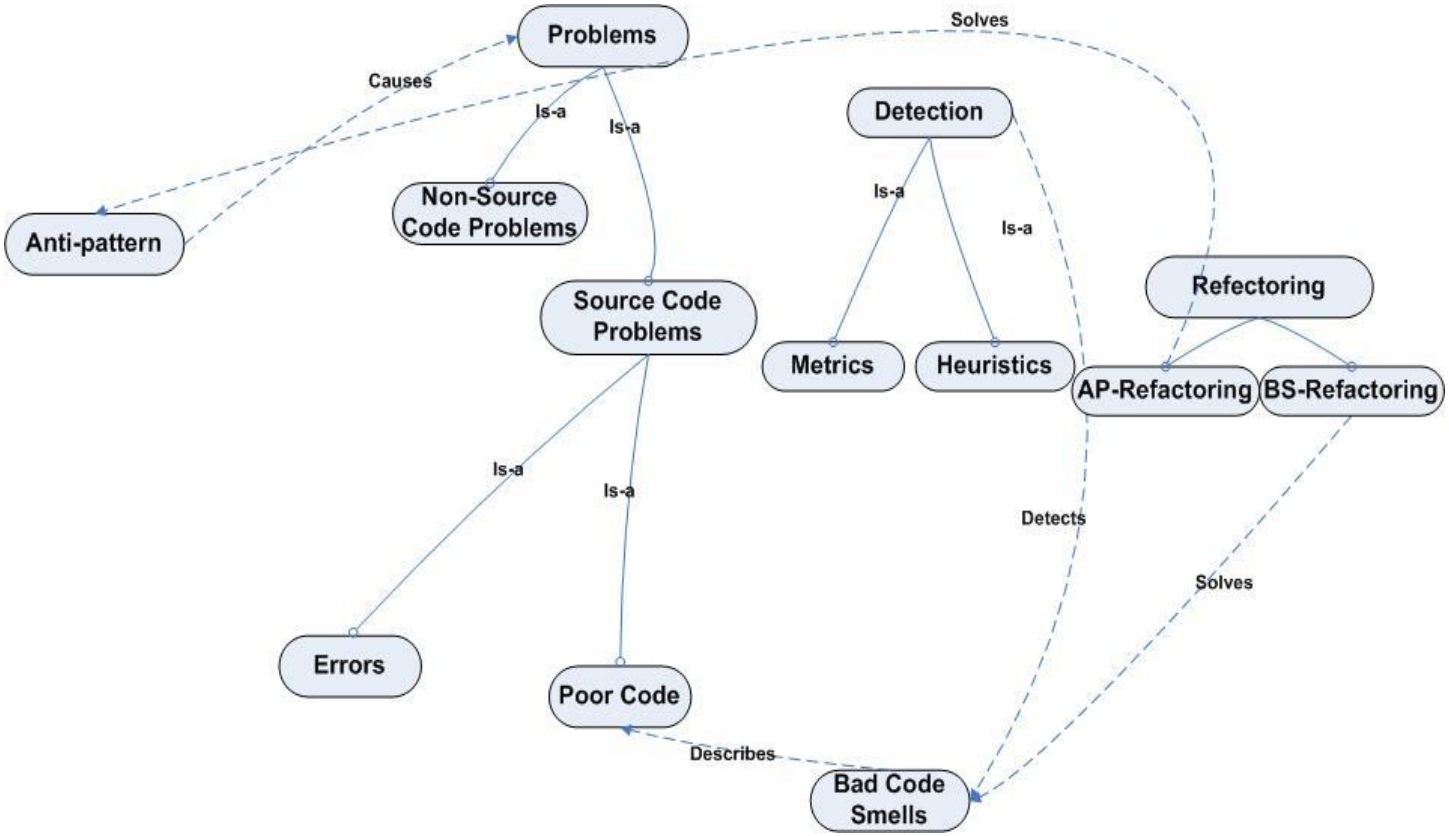


Figure 3.3: Simplified Conceptual Model of OABR

ontologies for software domains such as software qualities attributes, software design patterns, software cost estimation, software reuse, and maintenance.

An ontology mainly consists of lightweight ontologies and heavyweight ontologies. Lightweight ontologies, the most commonly occurring type [David], are a subclass of heavyweight ontologies. A lightweight ontology includes concepts, concept taxonomies, relationships between concept, and properties that describe these concepts. A heavyweight ontology has axioms and constraints plus the features of a lightweight ontology [Calero]. We classify OABR in this research between lightweight and heavyweight because OABR has constraints on bad code smells.

3.4 Summary

This chapter described the ontology concept, its application in this research and the benefits of the ontology for improving the understanding of anti-patterns, bad code smells, and refactoring. Chapter 4 will present the methodology used to develop the OABR infrastructure.

Chapter 4
Research Method

4.1 Introduction

This chapter presents an ontology-based approach to define and apply the properties of anti-patterns, bad code smells, refactoring, and the relations between them. We collected, organized, and classified the properties of the related concepts for further analysis (refer to Section 4.2). We expanded the properties for bad code smell by creating a quality index used to prioritize bad code smells with the goal of providing support for identifying which bad code smells should be removed, or tolerated (refer to Section 4.3). We then created templates based on properties for additional bad code smells and refactoring analysis (refer to Section 4.4). We also developed taxonomies for anti-patterns, refactoring, and bad code smells to provide hierarchical classifications (refer to Section 4.5). Section 4.6 shows the terminologies and relations represented by the basic Descriptive Logics (DL) used to define ontology language. We developed an OABR infrastructure including anti-patterns, bad code smells, related software problems, detections, and refactoring based on the properties, taxonomy, and non-taxonomy relations (refer to Section 4.7). Finally, we describe creating, accessing, storing, querying, and mapping of OABR with the ontological tools, platforms, and ontology registries/repositories in Section 4.8 and Section 4.9. Figure 4.1 provides the inputs to OABR.

4.2 Definitions of Class Properties

Class properties provide organized information and internal structure for each class. For

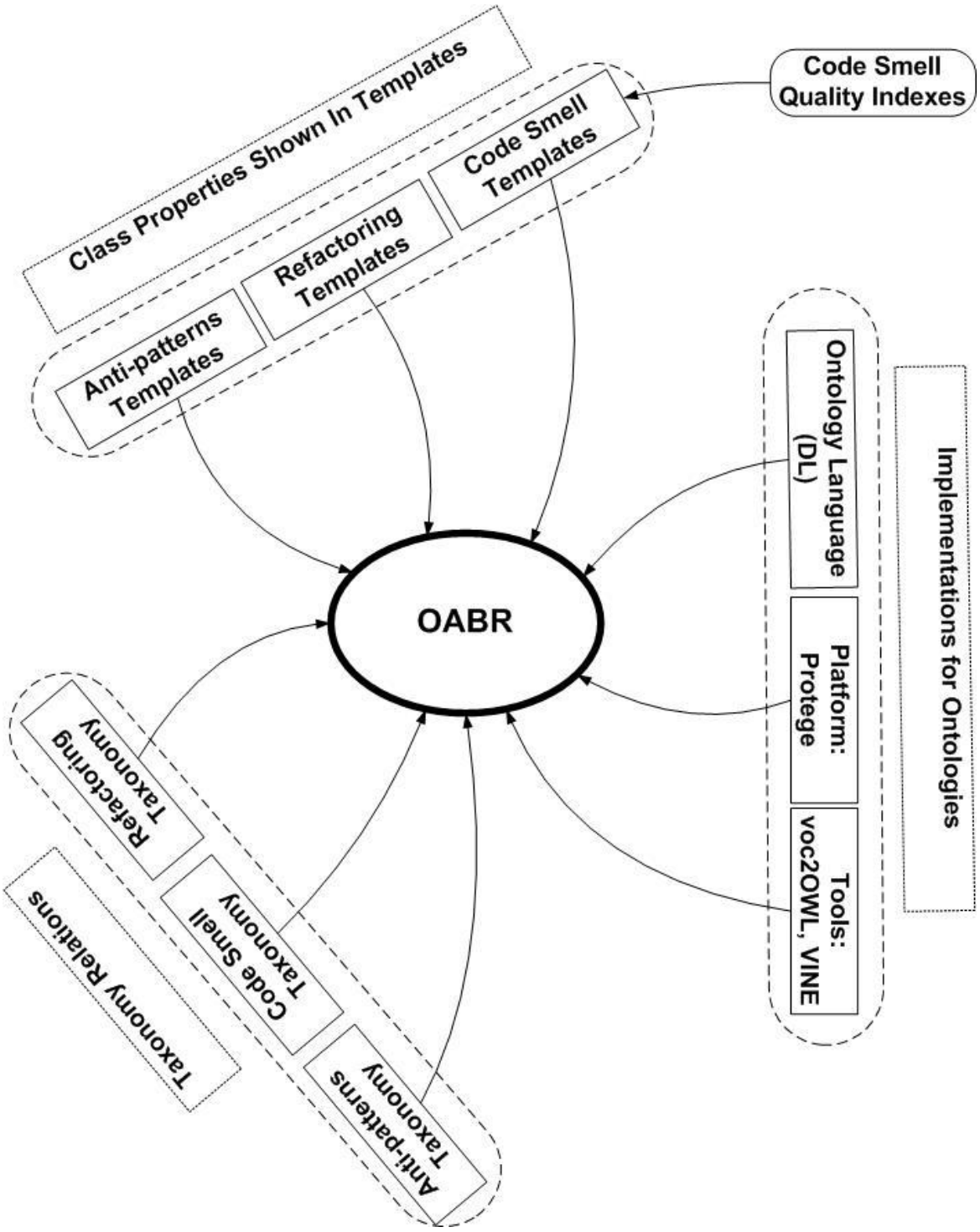


Figure 4.1: Outline Showing the Inputs to OABR

each class, one part describes the class name, and the remaining part is properties of the class describing various features and attributes of the concept such as causes, solutions, and symptoms. Each property belongs to a certain class. A property is also called role or slot.

The types of properties could be intrinsic properties such as the causes and symptoms of an anti-pattern or the symptom of bad code smells; extrinsic properties such as name; or the relations to other individuals. All the sub-classes of a class inherit the properties of that class. In this section, we will enumerate and briefly describe the properties for anti-patterns, refactoring, and bad code smells. As there are few formal and consistent descriptions about bad code smells, we describe and define code smells properties in more details.

4.2.1. Anti-pattern Properties

Many properties of existing anti-patterns were defined by [Brown] and [Laplante] separately. In this research, we selected properties such as name, causes, consequences, symptoms, and refactoring. Other properties such as root causes, variations, background, and general forms are not included as they are dependent on the software developers' personal experiences. Also, too many properties for a specific concept will increase the complexity in accordance with the basic principle that "the more expressive the language, the harder the reasoning" [Horridgy et al.].

An example of the Blob anti-pattern with its properties is shown in Table 4.1.

4.2.2 Refactoring Properties

We defined the refactoring properties as name, scenario, and mechanics. The name of a

Table 4.1: Examples of Blob Anti-pattern Properties

Anti-pattern
Name: Blob Causes: Lack of (an object-oriented architecture, architecture enforcement), too limited intervention Consequences: Too complex for reuse and testing, expensive to load into memory Symptoms: Large number of attributes Refactoring: Change responsibilities

refactoring usually consists of an operation and an object. For example, for the "Remove Middle Man" refactoring, "Remove" is an operation while "Middle Man" is an object. The scenario property provides description to each refactoring about when it will be applied. The mechanics property describes how to apply methods step by step for each refactoring to solve the related problem.

Refactoring could be classified as design refactoring and code refactoring. We focus on existing code refactoring. Design refactoring is beyond to this research.

Figure 4.2 lists 35 code refactoring techniques, each with a numbering label that has been applied to solve bad code smells related problems by software developers [Fowler]. We use the number label rather than the full name of the refactoring methods in subsequent tables and figures. Refactorings not related to bad code smells are not included in Figure 4.2.

R1. Change Bidirectional Association to Unidirectional
R2. Collapse Hierarchy
R3. Decompose Conditional
R4. Encapsulate Collection
R5. Encapsulate Field
R6. Extract Class
R7. Extract Interface
R8. Extract Method
R9. Extract Subclass
R10.Form Template Method
R11.Hide Delegate
R12.Inline Class
R13.Inline Method
R14.Introduce Assertion
R15.Introduce Foreign Method
R16.Introduce Local Extension
R17.Introduce Null Object
R18.Introduce Parameter Object
R19.Move Field
R20.Move Method
R21.Preserve Whole Object
R22.Pull up Methods
R23.Replace Array with Object
R24.Replace Conditional with Polymorphism
R25.Replace Data Value with Object
R26.Replace Delegation with Inheritance
R27.Replace Inheritance with Delegation
R28.Replace Method with Method Object
R29.Replace Parameter with Explicit Method
R30.Replace Temp With Query
R31.Replace Type Code with State/Strategy
R32.Replace Type Code with Subclass
R33.Remove Middleman
R34.Remove Parameter
R35.Rename Method

Figure 4.2: Subset of Refactoring Methods

4.2.3. Bad Code Smell Properties

An objective of this research is to provide a more formal and consistent documentation of properties to make each bad code smell easier to identify and compare. Current definitions of bad code smells are described and organized in a rather informal and inconsistent manner.

We analyzed initial properties of bad code smells as name, symptoms, metrics, and refactoring. Symptoms property describes how to find a bad code smell. Table 4.2 contains subsets of the 23 bad code smells named by [Fowler] with the symptoms property. We defined the symptoms property based on descriptions from both [Fowler] and [Mika]. We expressed each bad code smell using key words instead of sentences to describe symptoms in order to simplify comparison and inclusion as ontological

properties. For example, the existence of “Divergent Change” means a frequently changed class. The symptom of “Duplicate Code” is defined in the key words as “Redundant Code”.

Table 4.2: Subset of Bad Code Smells of Which Symptoms Are Described by Keywords [Fowler][Mika]

Smell Name	Symptoms
Alternative Classes with Different Interfaces	A class operating with two classes with different interfaces
Comments	Poor structure code
Data Class	Data with no logic
Data Clumps	Data dependent each other
Divergent Change	Frequently changed class
Duplicate Code	Redundant code
Feature Envy	Use other classes than itself
Inappropriate Intimacy	Too tightly coupled
Incomplete Library Class	Using incomplete library
Large Class	Too many functions
Lazy Class	Doing little things
Long Method	Too long method
Long Parameter List	Too long parameter list
Message Chains	Coupling problems
Middle Man	Delegating jobs to subsequent classes
Parallel Inheritance Hierarchies	Parallel class hierarchies exist
Primitive Obsession	Using primitives instead of class
Refused Bequest	Child class not support its inherited methods
Shotgun Surgery	Change one leading to changing others
Speculative Generality/Dead code	Code for future
Switch Statements	No polymorphism and pass on methods
Temporary Field	Occasionally used variables

Bad code smells can be detected heuristically depending on a programmers' experiences or through the application of traditional software metrics such as these metrics shown in Table 4.3 [Ronningen] [Rosenberg][Chidamber & Kemerer]. The metrics property is how traditional software metrics is used to identify a bad code smell.

Table 4.3: Goals and Application of Traditional Software Metrics

Metric	Application	Goals
LOC (Lines of Code)	measure the size of a class	understandability, reusability, and maintainability
CP (Comment Percentage)	understandability and maintainability	understandability, reusability, and maintainability
WMC (Weighted Methods per Class)	sum of the complexities of the methods-weighed methods per class	understandability, reusability, and maintainability
RFC (Response For a Class)	number of methods can be invoked	understandability, maintainability, and testability
LCOM (Lack of Cohesion of Methods)	measure the dissimilarity of methods in a class	efficiency and reusability
CBO (Coupling between Objects)	count the number of coupled classes	efficiency and reusability
Halstead	measure a program module's complexity	complexity

The metrics listed in Table 4.3 are defined as the follows.

- Weighted methods per class (WMC): WMC is the number of methods included in a class weighted by the complexity of each method. High WMC indicates a high complexity.

- Response for a class (RFC): RFC is a count of methods implemented within a class plus the number of methods accessible to an object of this class type due to inheritance.
- Cohesion (LCOM): A measure that indicates how well the parts of a component belong together. Cohesion should be maximized to promote encapsulation.
- Coupling (CBO): A measure of the extent to which interdependencies exist between software modules. Loose coupling decreases the complexity.

Software product metrics measure software products at different development stages, ranging from measuring the complexity of software design to the size of the final source code. The measurability of a bad code smell depends on the size, the complexity, and the structure of the bad code smell. Some bad code smells such as "Long Method" can be easily detected by traditional software metrics such as Cyclomatic complexity and Halstead measures. However, the selected metrics types used for each bad code smells are based on heuristic analysis. Some code smells such as "Dead Code" and "Middle Man" are difficult to detect by software metrics. Many bad code smells appear to be undetectable by software metrics. Mika developed an index based on his heuristic analysis to show the measurability of each code smell, where 0 means impossible to measure by metrics while 5 means easiest to use metrics to measure [Mika].

The refactoring property describes the solutions to each code smell. Some refactoring could solve several code smells, and some code smells may need several refactoring methods to remove.

We developed Table 4.4 to describe bad code smells in an organized manner.

Table 4.4: Properties of Bad Code Smells

Code Smell Name	Symptoms	Metrics		Refactoring Solutions
		Types	M	
Alternative Classes with Different Interfaces	A class operating with two classes with different interfaces	None	0	R20,R35
Comments	Poor structure code	None	1	R8,R14
Data Class	Data with no logic	CC/Number of fields	4	R4,R5,R20
Data Clumps	Data dependent each other	None	0	R6,R18,R21
Dead Code		Static/dynamic detection	3	
Divergent change	Frequently changed class	None	0	R6
Duplicate code	Redundant code	LOC	4	R6,R8,R10,R22
Feature Envy	Use other classes than itself	Coupling	4	R8,R19,R20
Inappropriate Intimacy	Too tightly coupled	Coupling	4	R1,R11,R19, R20,R27
Incomplete Library class	Using incomplete library	None	0	R15,R16
Large class	Too many Functions	NLOC Cohesion	4	R6,R7,R9,R25
Lazy Class	Doing little things	NLOC/CC	4	R2,R12
Long Method	Too long method	NOLOC/CC/Halstead	5	R3,R8,R28,R30
Long Parameter List	Too long parameter list	Number of parameters	5	R18,R28,R21
Message Chains	Coupling problems	Coupling	3	R11
Middle man	Delegating jobs to subsequent classes	Coupling/CC	2	R13,R26,R33
Parallel Inheritance Hierarchies	Parallel class hierarchies exist	None	0	R19,R20
Primitive Obsession	Using primitives instead of class	None	0	R6,R18,R23, R25,R31,R32
Refused Bequest	Child class not support its inherited methods	None	0	R27
Shotgun surgery	Change one leading to changing others	None	0	R12,R19,R29
Speculative Generality	Code for future	Static/dynamic detection	3	R2,R12,R34, R35
Switch statements	No polymorphism and pass on methods	LOC/CC/running time detection	3	R17,R24,R29, R31,R32,
Temporary field	Occasionally used variables	Methods counting	3	R6,R17

Properties of each bad code smell include name, symptoms, metrics-based detection analysis, and refactoring solutions. Each number label in the "Refactoring Solution" column refers to one of refactoring methods shown in Figure 4.2.

From Table 4.4, we can define each bad code smell based on their properties. However, it is difficult to compare them quantitatively because there is no computed value for each attribute.

4.3 Quality Indexes of Bad Code Smells

To improve the software product quality, the best scenario is to identify and remove all the bad code smells from source code. However, in reality, some code smells are difficult or impossible to identify or too costly to remove. For example, some bad code smells, such as "Large Class", are difficult to remove while others, such as "Lazy Class", are easier to remove. Some bad code smells can be easily identified, but the removal process is costly, such as "Divergent Change". On the other hand, not all bad code smells have the same level of importance to the source code. There is a tradeoff between cost of identifying/removing bad code smells and improving the software quality.

In this section, we define a quality index to prioritize bad code smells based on their properties with the goal of comparing bad code smells and identifying which bad code smells should be removed, prevented, or tolerated (Refer to Figure 4.3).

Identification is a mixture of heuristic analysis and metrics based tool detection that indicates whether a bad code smell is easy or difficult to find. *Remove* reflects the heuristic analysis of refactoring methods for removing a code smell and indicates whether the code smell is easy to remove or not. *Impact* refers to the consequence and

potential danger of the problem that a bad code smell refers to and whether its impact is in a small region or a large part of source code.

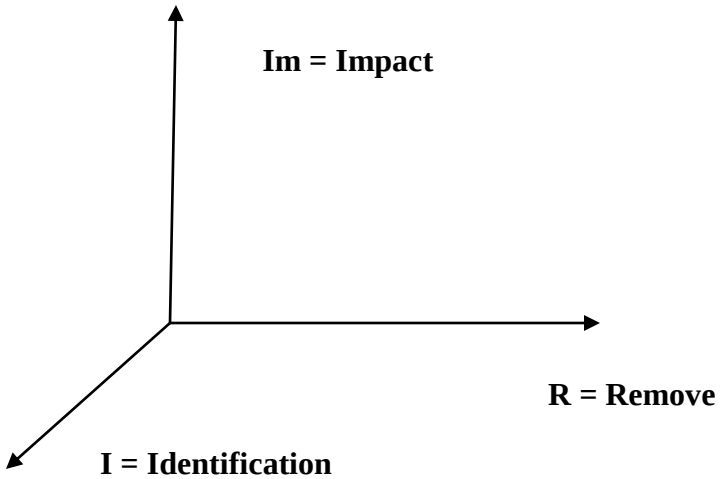


Figure 4.3: Quality Index of Bad Code Smell

For each property, we set a scale of 0 – 5 where "0" means impossible to identify, difficult to remove, and has little impact while "5" means easy to identify, easy to remove, and has strong impact on source code.

We rate a quality index as strong, medium, and weak are shown in Table 4.5.

Table 4.5: Definition of Quality Index

Quality Index	Identification(I)	Remove(R)	Impact (Im)
Strong	3 < I < 5	3 < R < 5	3<Im<5
Medium	2 < I < 3	2 < R < 3	2 < Im < 3
Weak	I<2	R < 2	Im < 2

Strong code smell related problems have a strong impact on the source code, and, thus, should be removed. Weak code smell means is not easy to identify or to remove, as well as its related problems do not have a strong impact on source code. Thus, it can be tolerated. An example of a strong code smell is "Large Class", and an example of a

Table 4.6: Bad Code Smells Indexes

Code Smell Name	Identification	Remove	Impact	Quality
Alternative Classes with Different Interfaces	0	5	3	Medium
Comments	4	4	1	Weak
Data Class	4	3	3	Medium
Data Clumps	4	5	1	Weak
Dead Code	4	5	2	Medium
Divergent Change	0	4	-	Medium
Duplicate Code	4	4	4	Strong
Feature Envy	4	-	-	-
Inappropriate Intimacy	4	4	5	Strong
Incomplete Library Class	0	-	-	-
Large Class	4	3	5	Strong
Lazy Class	2	2	1	-
Long Method	5	3	4	Medium
Long Parameter List	5	-	2	Weak
Message Chains	3	-	-	-
Middle Man	2	-	-	-
Parallel Inheritance Hierarchies	0	-	3	Medium
Primitive Obsession	0	4	-	-
Refused Bequest	3	-	-	-
Shotgun Surgery	0	-	-	-
Speculative Generality	3	-	-	-
Switch Statements	3	3	3	Medium
Temporary Field	3	-	-	-

‘ – ‘ = not able to assign

weak code smell is “Lazy Class”. These grouping are subjective and can be altered based on the domain of use.

Table 4.6 shows the quality index applied to represent bad code smells based on our heuristic analysis. For example, we evaluate Dead Code as easily detectable by dynamic checking (rating as 4 for identification). It would be easy to remove (rating as 5 for remove index). Dead Code will affect the readability and understandability of

source code, but its impact is not significant. We rate the impact index for Dead Code as 2. Another example is Large Class. The Large Class, which is easy to identify with the cohesion metrics such as Lack Cohesion Metrics, thus, its index is rated as 4. The removal of Large Class will use refactoring methods such as Extract Class, Extract Interface, and Replace Data Value with Object. We define the rate of its remove index as 3. The Large Class will not only affect the understandability and reusability, it will also make source code hard to maintain. We rate the impact index for Large Class as 5. The rating of the quality indexes like identification, remove, and impact of code smells is subjective and intuitive. The '-' symbols in the Table 4.6 mean that we do not assigned values to these code smell quality indexes.

To provide empirical support, we conducted a survey on the quality indexes of bad code smells among senior software engineers. The questionnaire consists of two parts. The first part collected background information on the responders such as IT related degrees, years of programming experiences, preferred programming language, and current projects. The second part provides a description of each code smell in plain text and requests the responses to rate each code smell based on the properties we defined. The questionnaire was distributed through emails and accessible via the following websites (Part A – http://www.questionpro.com/akira/TakeSurvey?id=771431 and Part B http://www.questionpro.com/akira/TakeSurvey?id=771311).

Fourteen responders have at least a Masters or above degree in IT related major, and their average working experiences are 6 years. The software engineers came both from industry (75%) and academia (25%). Results (refer to Table 4.7) affirm that bad code smells differ with regard to their identification, remove, and impact.

Table 4.7: Code Smell Quality Indexes from Survey

Code Smell Name	Identification		Removal		Impact	
	Mean	Std.	Mean	Std.	Mean	Std.
Alternative Classes with Different Interfaces	2.82	1.40	2.91	1.38	3.18	1.54
Comments	3.45	2.02	3.91	1.70	2.36	2.11
Data Class	3.09	1.30	2.82	1.40	2.82	1.33
Data Clumps	3.82	1.33	3.5	1.17	2.00	1.15
Divergent Change	3.56	1.23	2.20	1.40	2.50	1.43
Duplicate Code	4.27	1.27	3.5	2.01	2.45	1.57
Feature Envy	2.00	1.33	2.50	0.97	3.22	1.20
Inappropriate Intimacy	2.27	1.62	1.82	1.25	3.55	1.13
Incomplete Library Class	3.00	1.61	2.1	1.60	3.00	1.86
Large Class	2.55	2.07	1.82	1.60	3.60	0.84
Lazy Class	3.00	1.79	3.64	1.29	1.36	1.43
Long Method	2.82	2.04	2.09	1.38	3.2	1.32
Long Parameter List	3.45	1.64	3.00	1.27	2.45	1.81
Message Chains	3.10	1.10	2.55	1.44	3.55	1.04
Middle Man	2.36	1.50	1.91	1.58	3.36	1.43
Parallel Inheritance Hierarchies	2.33	1.32	2.33	1.22	3.11	1.17
Primitive Obsession	3.09	1.22	3.36	1.29	3.00	1.33
Refused Bequest	3.00	1.49	3.30	1.57	2.56	1.59
Shotgun Surgery	2.40	1.58	1.70	1.49	3.80	1.47
Speculative Generality	2.9	1.37	3.18	1.25	2.40	1.07
Switch Statements	3.00	1.61	2.91	1.38	1.90	1.45
Temporary Field	2.55	1.51	3.36	1.36	1.55	1.29

The survey shows that the quality indexes of bad code smells are perceived differently. While some code smell are easy to remove, to identify, and do not have a significant impact on the source code, others are costly to identify and remove and have a considerable impact on software products. We applied the survey results to assign values to each code smell data property, calculating the average values of a property for each code smell, and then filled in the values for the properties of each code smell to assist software developers in determining whether a bad code smell related problem should be removed or tolerated.

4.4 Templates for Bad Code Smell and Refactoring Analysis

Following the analysis of the properties of bad code smells and refactorings in Section 4.2 and Section 4.3, we created templates to express the properties of code smells and refactoring. The example for the template of anti-pattern is shown in Table 4.1. Templates provide a consistent outline for documentation, and they are used as a reference for the input or output of OABR.

Figures 4.4 through 4.7 show the formats of templates for expressing the properties of bad code smells and refactoring, along with examples.

Figure 4.5 shows an example of the “Middle Man” bad code smell represented using the template defined in Figure 4.4. Other examples of bad code smells are shown in Appendix A.

We organized a refactoring a template for properties as shown in Figure 4.6.

Figure 4.7 shows an example of the refactoring method named “Collapse Hierarchy” represented using the refactoring template defined in Figure 4.6.

BAD CODE SMELL
Name: Code smell name Symptoms: the Definitions of the code smell Solutions: The refactoring method(s) Identification: How is it easy/difficult to detect Remove: How is it easy/difficult to remove Impact: How much does the code smell impact the source code quality

Figure 4.4: Bad Code Smell Template

MIDDLE MAN
Name: Middle Man Symptoms: A class delegating most of its tasks to subsequent classes Solutions: Inline Methods, Replace Delegation with Inheritance Detection: Many methods coupled to one class with a low cyclomatic complexity Identifications: Medium Remove: Difficult Impact: Strong

Figure 4.5: An Example of Bad Code Smell – "Middle Man"

REFACTORING
Name: What is the refactoring called? Scenario: When is the refactoring needed? Mechanics: How does the refactoring work?

Figure 4.6: Refactoring Template

COLLAPSE HIERARCHY
Name: Collapse Hierarchy Scenario: Subclass and parent class is similar. Mechanics: 1. Select the class to be removed 2. Merge the class 3. Adjust references and remove the empty class

Figure 4.7: An Example of Refactoring Template – "Collapse Hierarchy"

The templates describe the anti-patterns, bad code smells and refactoring properties in a uniform way that are later folded into OABR. More examples of refactoring are shown in Appendix B.

4.5 Anti-pattern, Refactoring, and Code Smells Taxonomy

We arranged and organized the classes of anti-pattern, refactoring, and bad code smells in hierarchical taxonomies. A taxonomy represents an "is-a" relation (a class A is a subclass of B if every instance of B is also an instance of A [Noy]) and a taxonomic relation is as a "kind-of" relation. For example, Dead Code is a kind of bad code smell. The taxonomy not only makes the related concepts more understandable but also identifies relations at a higher classification as well as improves the clarity and reuse of an ontology.

Figure 4.8A shows the anti-pattern taxonomy. The anti-pattern taxonomy is based on its application for software developments, software management or software maintenance. There are currently six categories of anti-patterns, including software design, project management, software analysis, programming, and methodology. We used software design, methodology, and programming to this research. The anti-patterns categories about organizational anti-patterns, project management, or analysis are beyond this research.

Figure 4.8B shows a code refactoring taxonomy based on operations. A refactoring method consists of an operation part and object part. For instance, we define the Extract Method, Extract Interface, Extract Sub classes, and Extract Super classes as a category of "Extract" as they apply the same extract operation on different objects such as class, super class, sub class, method, or interface. Other categories include the operations of replace, remove, and introduce. The label for each refactoring term refers to the labels shown in Figure 4.2.

Figure 4.8C shows the bad code smell taxonomy that is based on the comparison of

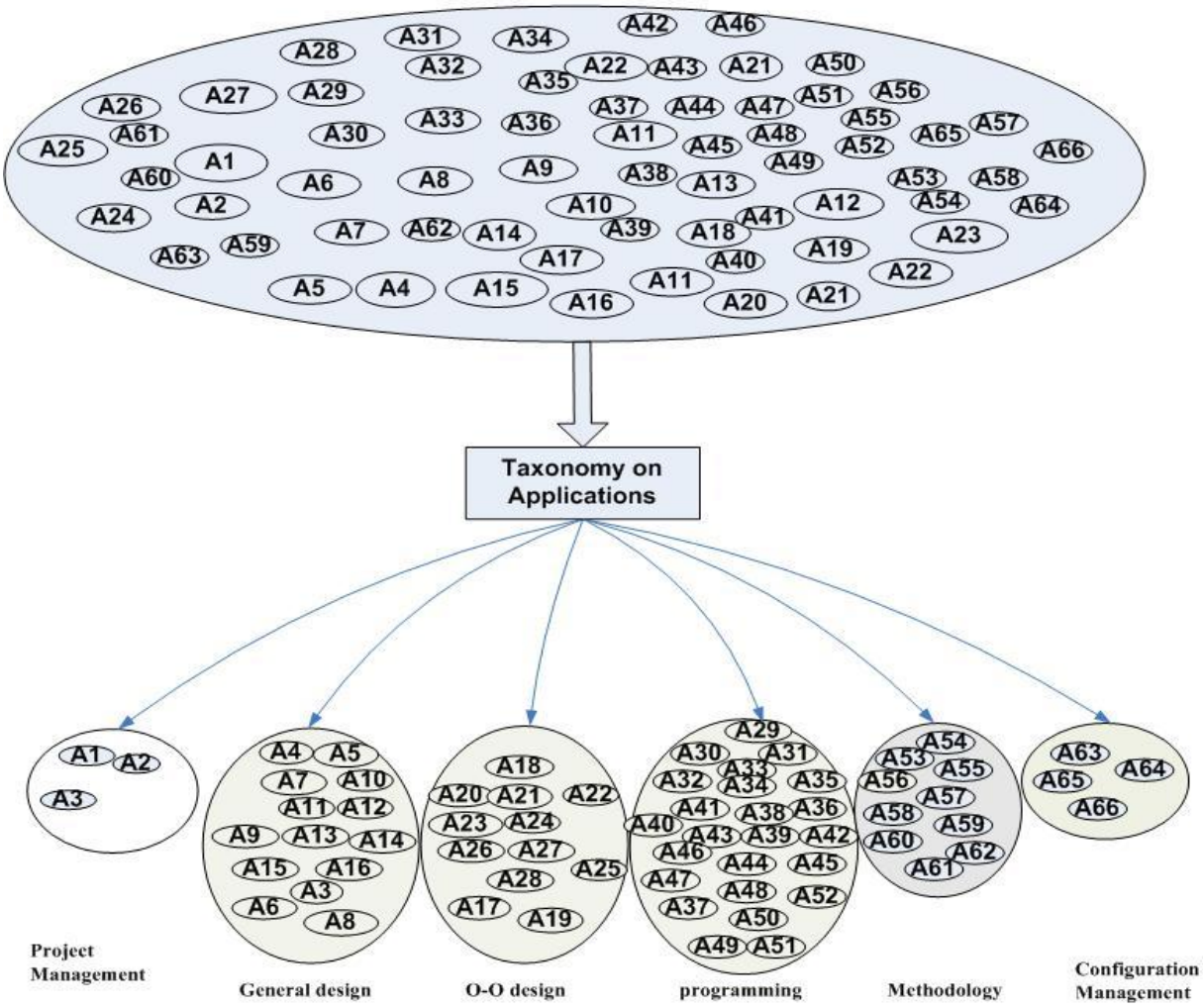
refactoring solutions to each code smell. Each code smell has one or more related refactoring methods (solutions). The name of each category in the taxonomy reflects the solution for the bad code smell. Our taxonomy is solution-based in contrast to Mika's taxonomy that is based on symptoms [Mika].

The bad code smell taxonomy includes the following six categories:

- Extracting - describes bad code smells that could be removed by using the refactoring method named "expanding classes, methods, subclass, interfaces";
- Object Introducing - describes a group of bad code smells that could be solved by applying the refactoring methods of adding new objects to the source code;
- Inline - describes a category of bad code smells that could be solved by applying the refactoring method of inline class or inline method with other refactoring methods;
- Moving - describes a category of bad code smells that could be solved by applying the refactoring method of transferring method or field or along with other refactoring methods;
- Delegation - describes a category of code smells that could be solved by applying the refactoring methods related to OOP's Delegations with other refactoring methods; In OOP, delegation "is a technique of delegating or deferring the implementation of an interface to the result of a function. The purpose of delegation is for dynamic inheritance." [Christopher];
- Others - include all the bad code smells that apply other refactoring methods.

With the solution-based taxonomy of bad code smells, the types of refactoring are usually specified; thus, the analysis is based on a fixed set of concepts (keywords)

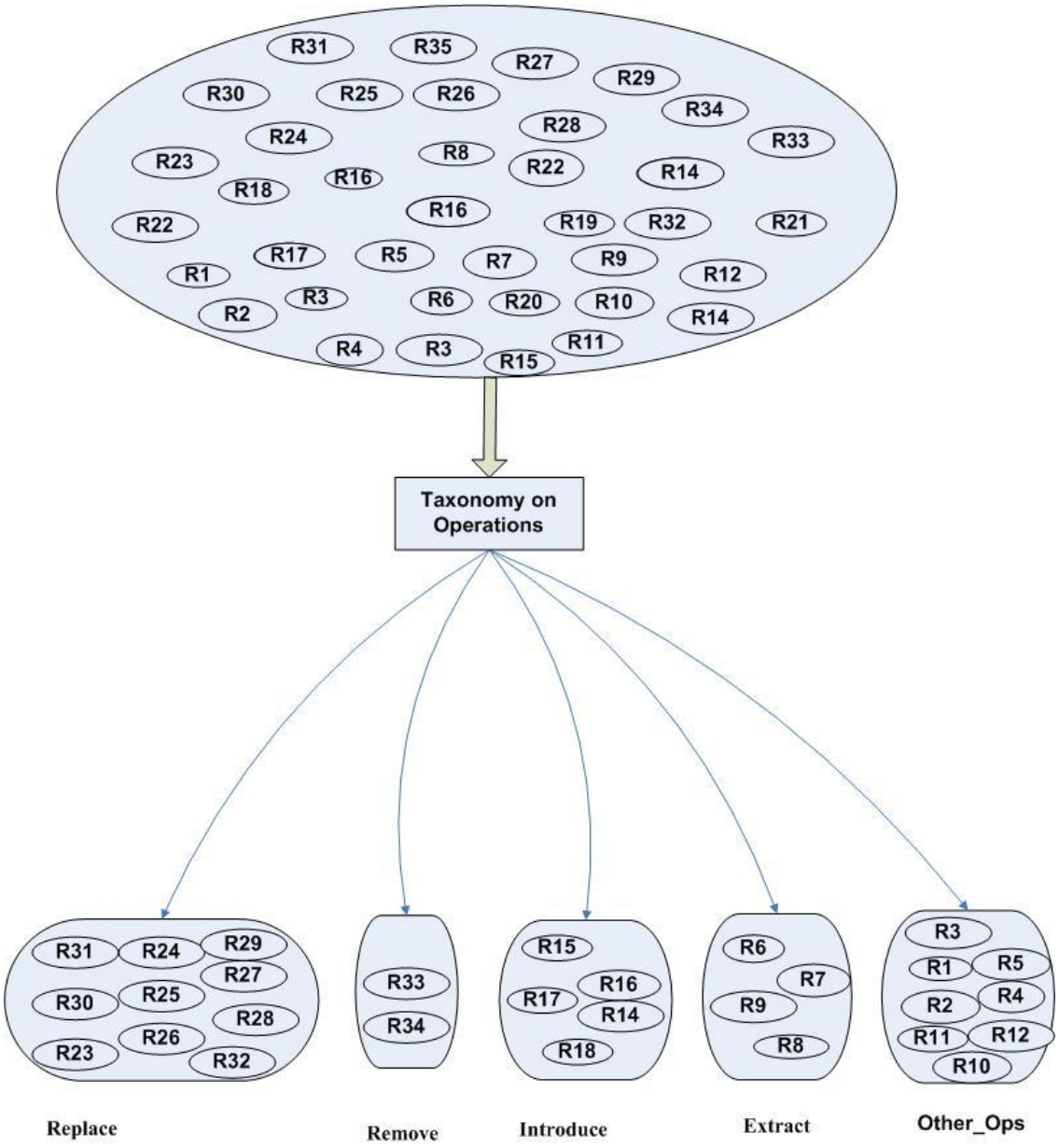
A1-Smoke and mirrors A2-Software bloat A3-Bad Management A4-Abstraction inversion A5-Ambiguous viewpoint A6-Blob (Anti-pattern) A7-BOMQ A8-Database as an IPC A9-Gas factory A10-Input kludge A11-Interface bloat A12-Magic pushbutton A13-Race hazard A14-Re-Coupling A15-Stovepipe system A16-Anemic Domain Model A17-BaseBean A18-Call super A19-Circle-ellipse problem A20- Empty subclass failure A21-God object A22-Object cesspool A23-Poltergeist A24-Sequential Coupling A25-Singletonitis A26- "'YAFL"' (Yet Another Layer) A27-Yo-yo problem A28-Accidental complexity A29-Action at a distance (computer science) A30-Accumulate and fire A31-Blind faith (computer science A32-Boat anchor A33-Busy spin A34-Caching failure A35-Cargo cult programming A36-Checking type instead of membership A37-Code momentum A38-Coding by exception A39-Error hiding A40-Exception handling A41-Full monty A42-Hard code A43-Lava flow A44-Loop-switch sequence A45-Magic number (programming) A46-Magic string (programming) A47-Ravioli code A48-Spaghetti code A49-Superboolean logic A50- Useless exception handling A51-Copy and paste programming A52-De-factoring A53-Golden hammer A54-Improbability factor A55-Optimization (computer science) A56-Programming by permutation A57-Reinventing the wheel A58-Reinventing the square wheel A59-Silver bullet A60-Tester Driven Development A61-Dependency hell A62-DLL hell A63-JAR hell A64-Extension conflict



A. Anti-pattern Taxonomy

Figure 4.8: Anti-pattern, Code Refactoring, and Bad Code Smell Taxonomies

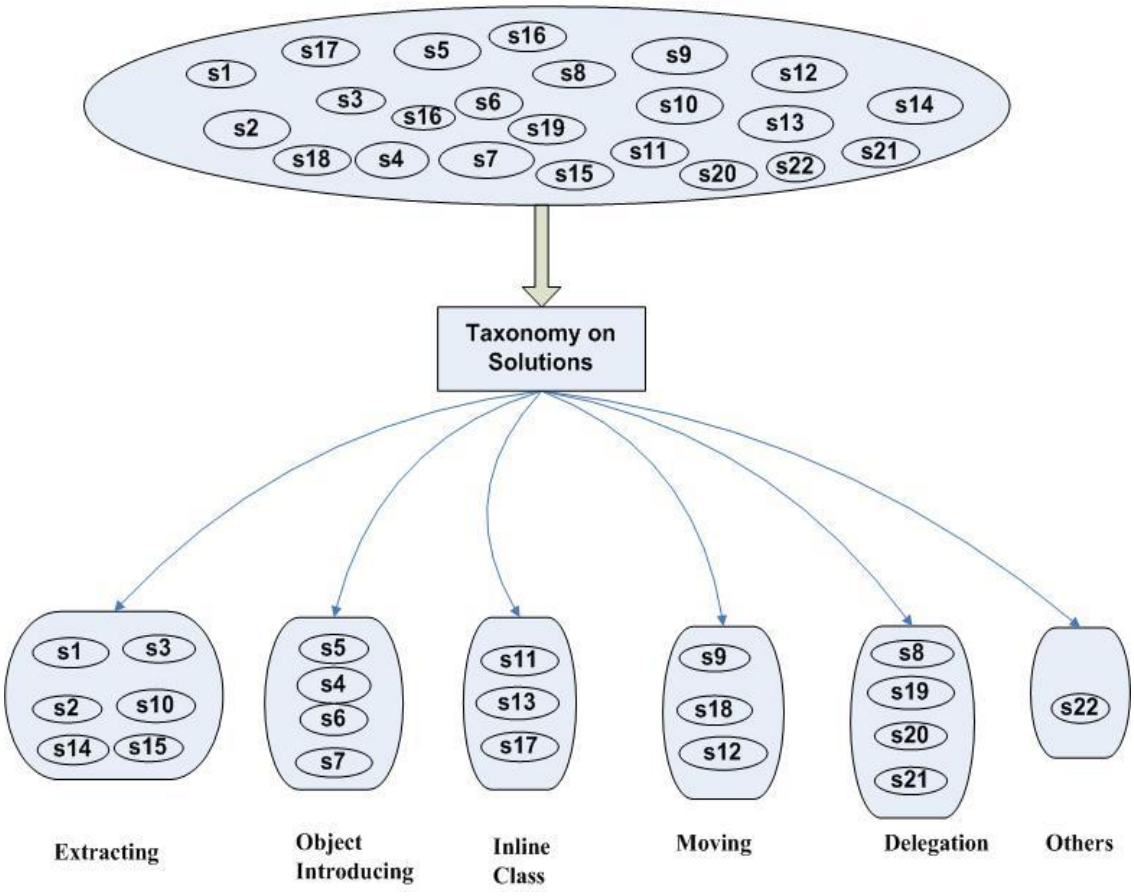
(Figure 4.8 continued)



B. Code Refactoring Taxonomy

(Figure 4.8 continued)

s1-Long Method s2-Large Class s3-Primitive Obsession s4-Long Parameter List s5-Data Clumps s6-Switch Statements s7-Temporary Field s8-Refused Bequest s9-Alternative Classes with Different Interfaces s10-Divergent Change s11-Shotgun Surgery s12-Parallel Inheritance Hierarchies s13-Lazy class s14-Data class s15-Duplicate Code s16-Dead Code s17-Speculative Generality s18-Feature Envy s19-Inappropriate Intimacy s20-Message Chains s21-Middle Man s22-comments



C: Bad Code Smell Taxonomy

because the names and types of refactoring methods usually do not change.

A solution-based taxonomy of code smells has some similarities to Mika's symptom-based taxonomy. For example, we put Long Method, Large Class, Primitive Obsession,and Data Clumps into the "Extracting" category as Mika did by putting them into the "the Bloaters" category. The result makes sense as both the symptoms and solutions have relationships with each other by reflecting same problems.

4.6 Description Logics to Express Terminologies and Relations of OABR

Description Logics (DL) is expressive, objective, and an ideal starting point for describing concepts, properties, relations, and individuals in a domain [baader]. Also, DL provides a useful tool for defining, integrating, and maintaining an ontology.

We apply DL to express terminologies and relations among concepts for OABR. Figure 4.9 shows examples of the basic DL's expression for OABR.

Atomic symbols consist of atomic concepts and atomic roles from which we build complex descriptions.

A TBox describes concept hierarchies like relations between concepts by sentences. For example, CodeSmell can be defined as poor code showing symptoms by writing this declaration: "CodeSmell ≡ ∃showSymptoms.PoorCode".

An Abox consists of concept assertions and role or property assertions that describe the relations between instances and classes. An Abox is also called individuals or membership assertions [Baader]. For instance, Large Class and Extract Class are instances, CodeSmell(Large Class) means that Large Class is an instance of code smell class, Refactoring(Extract Class) means that Extract Class is an instance of Refactoring method class, and hasRefactoring(Large Class, Extract Class) means that

Symbol
≡ (concept equivalence/definition) ∃ (existential restriction) ¬ (negation) ∀ (universal restriction) ∩ (intersection or conjunction of concepts) ∪ (union or disjunction of concepts) ( ) (Concept/role assertion)
Atomic Symbols
SoftwareProblems, SoftwareChronicalProblems, SourceCodeProblems, Solution, Good solution, PoorCode, Refactoring
TBox
DesignProblems≡∀hasDesignProblem.softwareChronicalProblems Sourcecodeproblems≡∀hasSourcecodeProblem.softwarechronicalProblems Badsolution ≡ ¬ goodsolution DesignPattern≡∀hasDesignProblem.softwareChronicalProblems ∪ Solution AntiPattern≡DesignProbelms ∪ BadSolution CodeSmell ≡ ∃showSymptoms.SourceCodeProblems MediumCodeSmell≡((∃hasImpact.CodeSmell) ∩(≥2 hasImpact ∩ ≤3 hasImpact)) ∩((∃hasIdentification.CodeSmell) ∩ (≥2 hasIdentification ∩ ≤3 hasIdentification)) ∩ ((∃hasRemoval.CodeSmell) ∩(≥2 hasRemoval ∩ ≤3 hasRemoval)) StrongCodeSmell≡≡((∃hasImpact.CodeSmell) ∩ (3 ≤ hasImpact)) ∪ ((∃hasIdentification.CodeSmell) ∩ (≥3 hasIdentification ∩ ≤5 hasIdentification) ∪ ((∃hasRemoval.CodeSmell) ∩∩(≥3 hasRemoval ∩ ≤5 hasRemoval)) WeakCodeSmell≡((∃hasImpact.CodeSmell) ∩ (2≥ hasImpact)) ∪ ((∃hasIdentification.CodeSmell) ∩ (2≥ hasIdentification)) ∪ ((∃hasRemoval.CodeSmell) ∩(2≥ hasRemoval))
ABox
Concept Assertions: CodeSmell(LongMethod), CodeSmell(LargeClass), CodeSmell(PrimitiveObsession), CodeSmell(LongParameterList), CodeSmell(DataClumps) AntiPattern(Blob), AntiPattern(StovepipeSystem), AntiPattern(GasFactory) Refactoring(ExtractClass), Refactoring(InlineClass), Refactoring(ForeignMethod), Refactoring(RenameMethod), Refactoring(ExtractMethod)
Role Assertions: hasSymptoms, showSymptoms, hasDesignProblems, hasSourcecodeProblems, causeProblems, hasRefactoring, hasImpact, hasRemoval, hasIdentification, hasProblem, hasContext, hasConsequences, hasRootCause, hasSolution

Figure 4.9: Examples of DL Description of OABR

Refactoring(Extract Class) could be used to solve the CodeSmell(Large Class) related problems. More examples are shown in Appendix D.DL provides the ability to capture different kinds of relationships. However, the DL's exponential computational complexities usually make the automatic computation impractical. Also, some relation properties like "causes" or "addresses" in OABR are impossible or very difficult to be expressed in DL. There are non-standard inferences that support building and maintaining DL knowledge bases [Baader].

4.7 Ontological Infrastructure

Based on the properties for each foundational concept and the taxonomic relations defined in the previous sections, we developed the OABR infrastructure. Figure 4.10 shows a detailed view of the OABR representation. The OABR graphically shows the interrelationships between and among the related software concepts.

The root class of OABR, the Problems class, refers to chronic software problems. Source code problems and non source code problems are subclasses of software problems. Non source code problems include all chronic problems at the different software development cycles except coding level. The "Poor Code" subclass is an example subclass of "Source Code Problems", and it shows some instances such as SourceCodeProlem(Large_Class_Low_Cohesion), SourceCodeProblem(Not_doing_enough_Class), and SourceCodeProblem (Many_Object_High_Coupling).

The anti-patterns class causes software development problems. There are five classes of antipatterns [Laplante] (refer to Figure 4.8A). OABR currently includes only the anti-

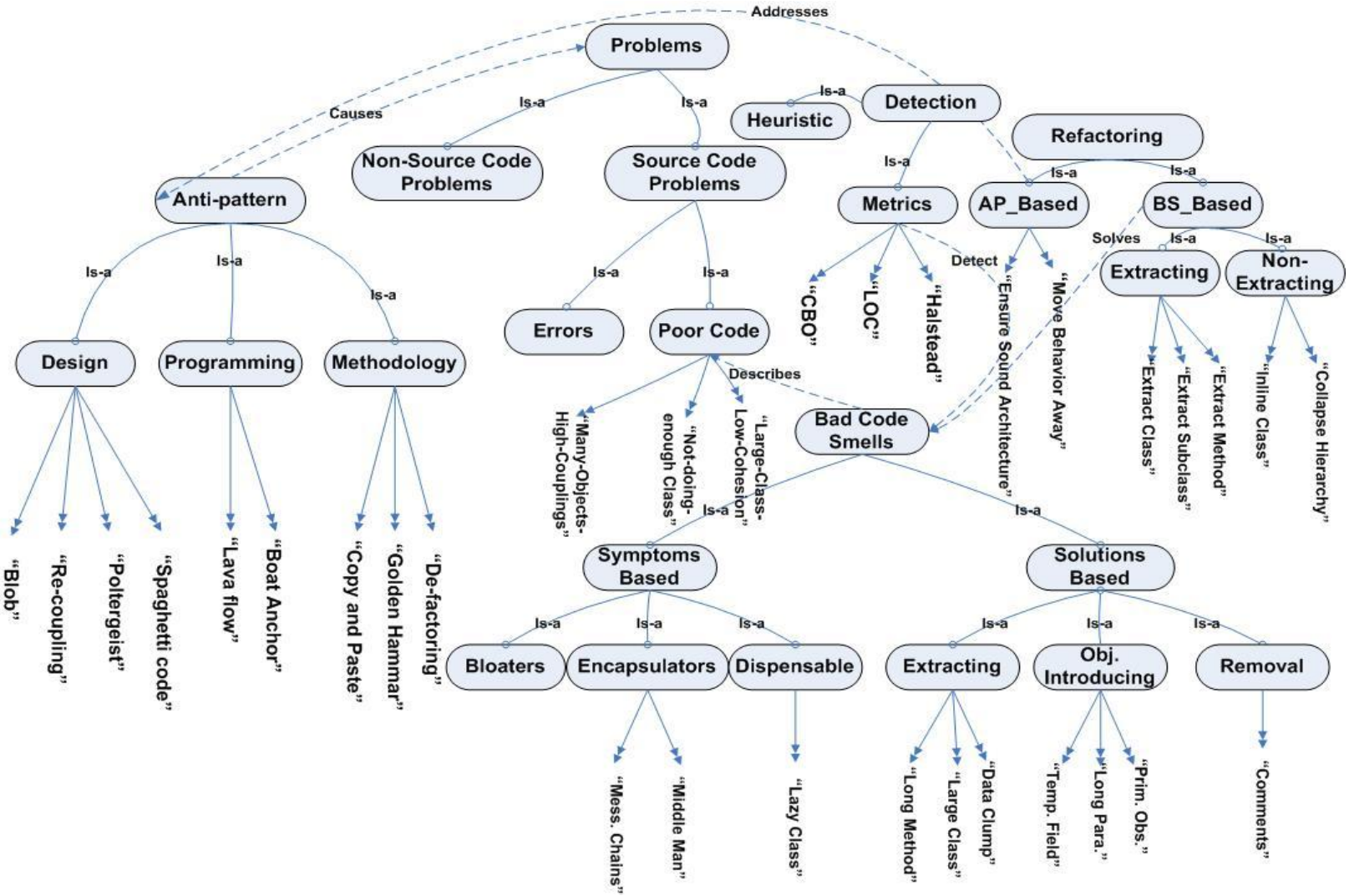


Figure 4.10: Conceptual Models of OABR

patterns categories that are related to designing , programming, and methodology. We merge the anti-patterns in general design with those in OO design to design class.

Bad code smells are symptoms of problems in the source code and are classified as either Symptoms-based on the symptoms properties or Solutions-based on the refactoring properties of each code smell. Sub-classes of the Symptoms-based are from [Mika et al.] and include: Bloaters, O-O Abusers, Change Preventers, Dispensable, and Encapsulations [Mika]. Sub-classes of Solutions-based are from our taxonomy that analyzes the refactoring methodologies on each existing bad code smell and includes: Extracting, Object-Introducing, Inline, Moving, Removal, and Delegation (refer to Figure 4.8C).

The detection class and refactoring class include the identification methods and refactoring methods to indentify and remove bad code smells and anti-patterns. The detection class contains methods for detecting anti-patterns and bad code smells. Detection could be performed via heuristics analysis or software metrics.

Figure 4.10 gives three instances of Metrics: “Coupling between Object Classes (CBO)”, “Lines of Code (LOC)”, and “Halstead”. Refactoring consists of two sub-classes according to the refactoring objects. BS-BASED refactoring is applied to fix a bad code smell related problems while AN-BASED refactoring are applied to address Anti-patterns. BS-refactoring have two sub-classes according to refactoring behaviors, “Extracting” and “Non_Extracting” We merged other subclasses like “Remove”, “Introduce”, and “Replace” into Non-Extracting class. The instances of each refactoring sub-class including Refactoring(Extract Class), Refactoring(Extract Subclass), and Refactoring(Extract Method) for “Extracting” category; and

Refactoring(Inline Class) and Refactoring(Collapse Hierarchy) for "Non_Extracting" category.

OABR facilitates the depiction of the interrelationships between and among these foundational concepts. For example, anti-patterns cause software problems that can be source code problems or other problems. Source code problems can be either errors or poor code. Bad code smells describe poor code. Metrics can detect code smells, and refactoring methods can solve some code smells.

Table 4.8: The Domain and Range of OABR Non-taxonomy Relation Properties

Property name	Property type	Domain	Range
Describe	Object Property	Bad-Smells	Poor-Code
Cause	Object Property	Anti-Patterns	Problems
Address	Object Property	AP_Based Refactoring	Anti-patterns
Detect	Object Property	Detection	Bad-Smells
Solve	Object Property	BS-Based Refactoring	Bad-Smells
Identification	Data Property	Bad-Smells	Float
Remove	Data Property	Bad-Smells	Float
Impact	Data Property	Bad-Smells	Float

Relations can also be called properties. Properties may specify a domain and a range and link individuals from the domain to the individuals from the range. Table 4.8 shows relation properties of OABR.

To demonstrate how the OABR represents the inter-relationships among these software concepts, we focus on the Poor Code instance "Not-doing-enough-class". This instance refers to a software problem in which the functionality of a class does not justify the costs to maintain and understand that class. This instance is associated with the Anti-pattern instances AntiPattern(Poltergeist); the bad code smell instance CodeSmell(Lazy Class); and the Refactoring instances Refactoring(Collapse Hierarchy) and Refactoring(Inline Class) to provide solutions to CodeSmell(Lazy Class) related problems. The other related refactoring instances Refactoring(Sound-Architecture-Precedes-Production-Code-Development), Refactoring(Establish-System-Level-Software-Interfaces), and Refactoring(Object-Oriented-Architecture) provide solutions to AntiPattern(Poltergeist). The PoorCode(Large-Class-Low-Cohesion) is associated with the AntiPattern(Blob), and it has the symptoms shown by the bad code smell instance CodeSmell(Large Class). The refactoring instances Refactoring(Extract Class) and Refactoring(Extract subclass) correspond to provide the solutions to the code smell of CodeSmell(Large Class) and the Refactoring instance Refactoring(MoveBehaviorAway) to provide the solutions to the AntiPattern(Blob).

The OABR infrastructure provides a more systematic approach toward analyzing the interrelationships between anti-patterns, code smells and refactoring.

4.8 Tools and Platforms

We implemented the conceptual model for OABR shown in Figure 4.10 with Protégé, a powerful ontological editor with a library of plug-ins that adds more functionality to the environment of the ontology. Protégé with the Protégé-OWL plug-in [http://protege.stanford.edu/] were developed by Stanford University [Horridge et al.].

The Protégé and related tools are open source software and can be installed locally. They provide required functionality for this research, such as definition of classes, hierarchies, and properties as well as relations analysis. Also, Protégé has a rich set of operators.

Figures 4.11 and 4.12 provide snapshots produced by the information browser for Protégé, Jambalaya. They graphically portray views of the OABR representation.

Figure 4.11 shows a general picture of the concepts and the relations between the classes in OABR. The structure is similar to the concept domain model shown in Figure 4.10. In Figure 4.11, the solid lines show the taxonomy relations “is-a” between the subclass and super class. For example, classes (triangles) in the diagram are the subclasses of Protégé root class [Brown et al.] shown by a rectangle. The dashed lines show non-taxonomy relation properties between classes like “Solve”, “isSolvedBy”, “Causes”, “isCausedBy”, “Detect”, “isDetectedBy”, “Describes”, and “isDescribedBy”. The brown dashed line from “Bad-Smells” to “Poor-Code” is the property of “Describes” and the blue dashed line from “Poor-Code” to “Bad-Smells” is the property “isDescribedBy” which is the inverse function of property “Describes”. The domain and range of some non-taxonomy relations are shown in Table 4.8.

We showed a general template describing the properties about bad code smells in Figure 4.4. Figures 4.12A and 4.12B show examples of template representations for the CodeSmell(Large Class) and CodeSmell(Long Method) using Protégé with Jambalaya respectively. These figures show the code smell related source code problems, detections, symptoms, refactoring solutions, and quantitative values for the

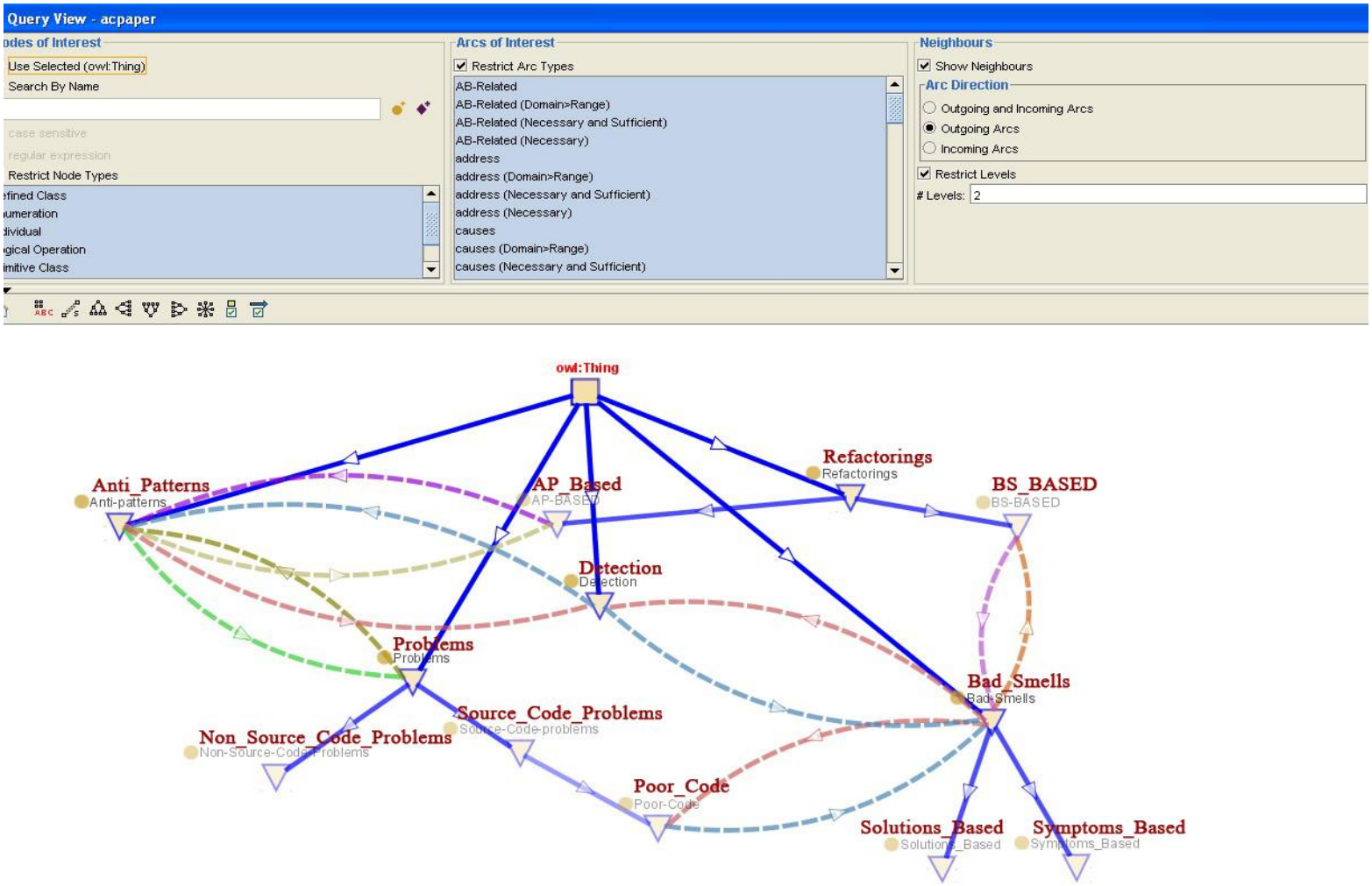


Figure 4.11: Hierarchy Relations among Classes

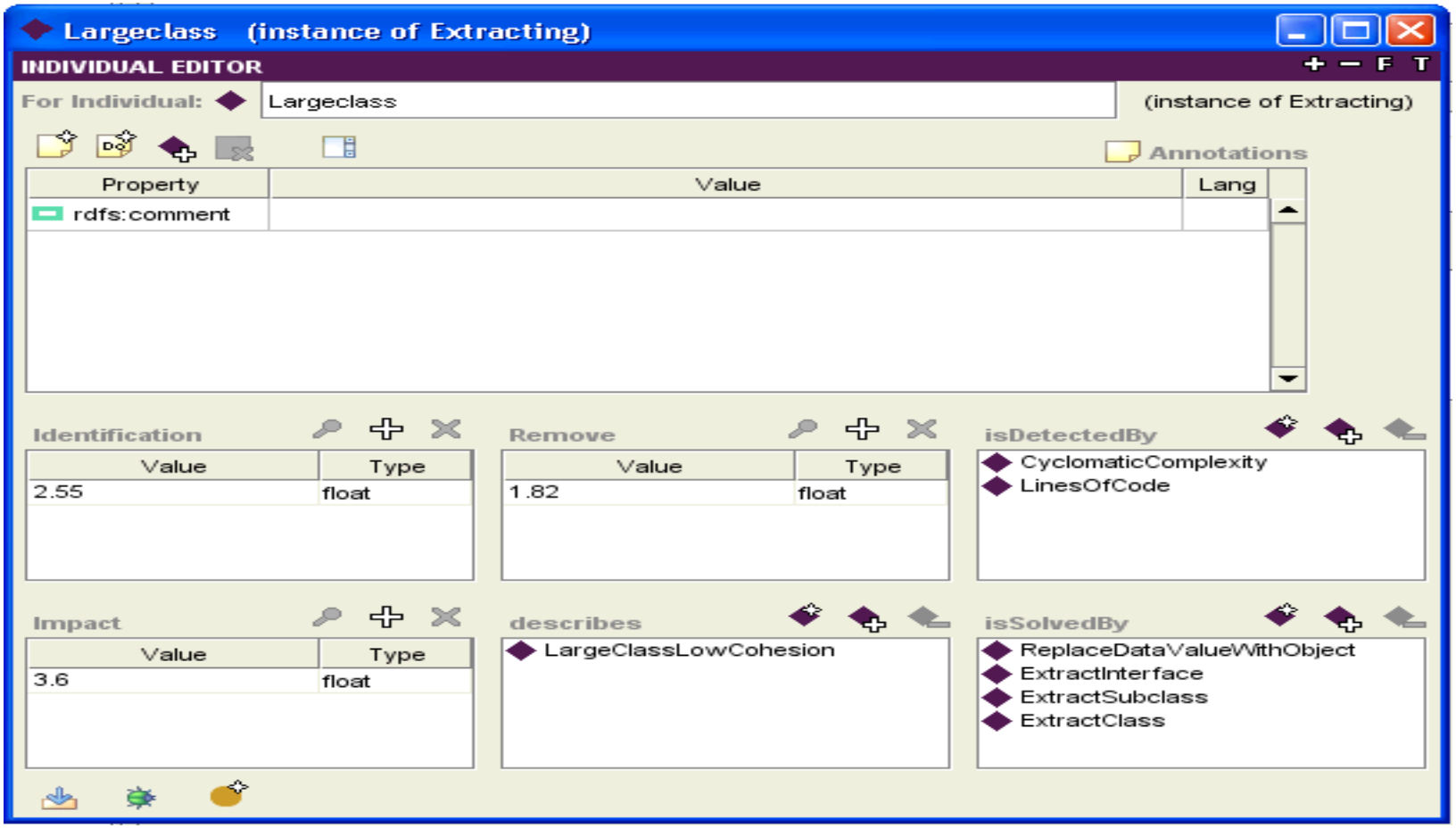
quality indexes of identification, impact, and remove for code smells.

The data properties such as quality indexes are the results from the survey. The information will not only improve the understanding of software developers about each code smell but also provide guidance for which code smell should be removed and which one could be tolerated. For example, the CodeSmell(Large Class) (Refer to Figure 4.12A) is difficult to remove and has a significant impact on the source code quality though it is not difficult to identify. The OABR shows how to remove CodeSmell(Large Class) through the instance of Refactoring(Extract Class) or Refactoring(Extract Subclass) of BS-BASED refactoring, the sub class of refactoring class. The CodeSmell(Long Method) (Refer to Figure 4.12B) is not difficult to remove and identify, but it has a significant impact on source code quality according to its quality indexes. For the removal of CodeSmell(Long Method), the OABR shows that it can be solved by the instances such as Refactoring(Extract-Method), Refactoring(Replace-Temp-with-Query) and Refactoring(Preserve-Whole-Object) of BS-BASED refactoring of class of refactoring.

4.9 Technologies in Support of OABR for Communities' Uses

Ontology is an open system promoting wide use and sharing. Its expansion and validation depend on the input from the users of related community. The normal way is to register the ontology with an ontology search engine, or with a repository to make the ontology visible to the community. The responses from the community will make the ontology more consistent and reliable.

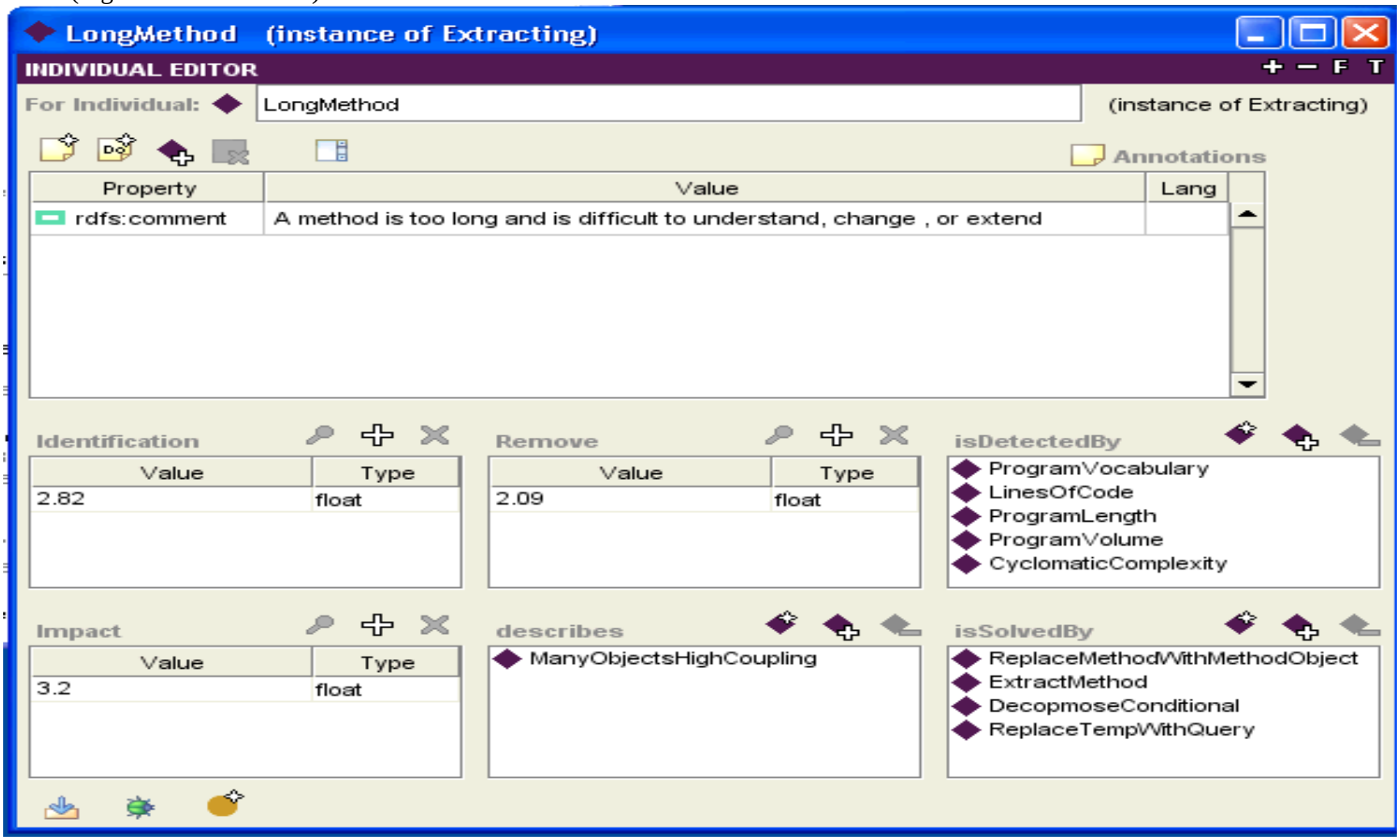
In this research, tools from Marine Metadata Interoperability (MMI) were modified for



A: Example of CodeSmell(Large Class) with Properties

Figure 4.12: Code Smell Templates Represented by Protége

(Figure 4.12 Continued)



B: Example of CodeSmell(Long Method) with Properties

use to develop the OABR infrastructure. The MMI project is a successful example of ontological applications though it is still under development. In the past five years, this project has already developed usefully technologies and tools that are implemented for the ocean observation ontology in North Gulf of Mexico. The tools and technologies include Voc2OWL for creating Web Ontology Language (OWL), Ontology Registry and Repository for users' registering, and The Vocabulary Integration Environment (VINE) for mapping concepts [Bermudez][Graybeal]. The related open source software tools can be downloaded from http://marinemetadata.org/tools/.

Voc2OWL can convert an ASCII Tab-delimited set of terms and definitions, i.e. the templates of anti-patterns, code smells, and refactoring to the related OWL. The OWL Web Ontology Language "is designed for use by applications that need to process the content of information instead of just presenting information to humans" (http://www.w3.org/TR/owl-features/).

Figure 4.13 shows an example of how to convert CodeSmell(Large Class) from a text file to an OWL file. The text file can be easily transferred from the templates defined for anti-patterns, bad code smells, and refactoring. The complete output of OWL for Large Class is shown in Appendix F.

An open ontology repository supports storing, sharing, searching, governance, and management of an ontology commonly used in the related community (http://ontolog.cim3.net/forum/oor-forum/2008-04/msg00012.html). The registry allows related community users to query the terms and properties within the ontology through web services.

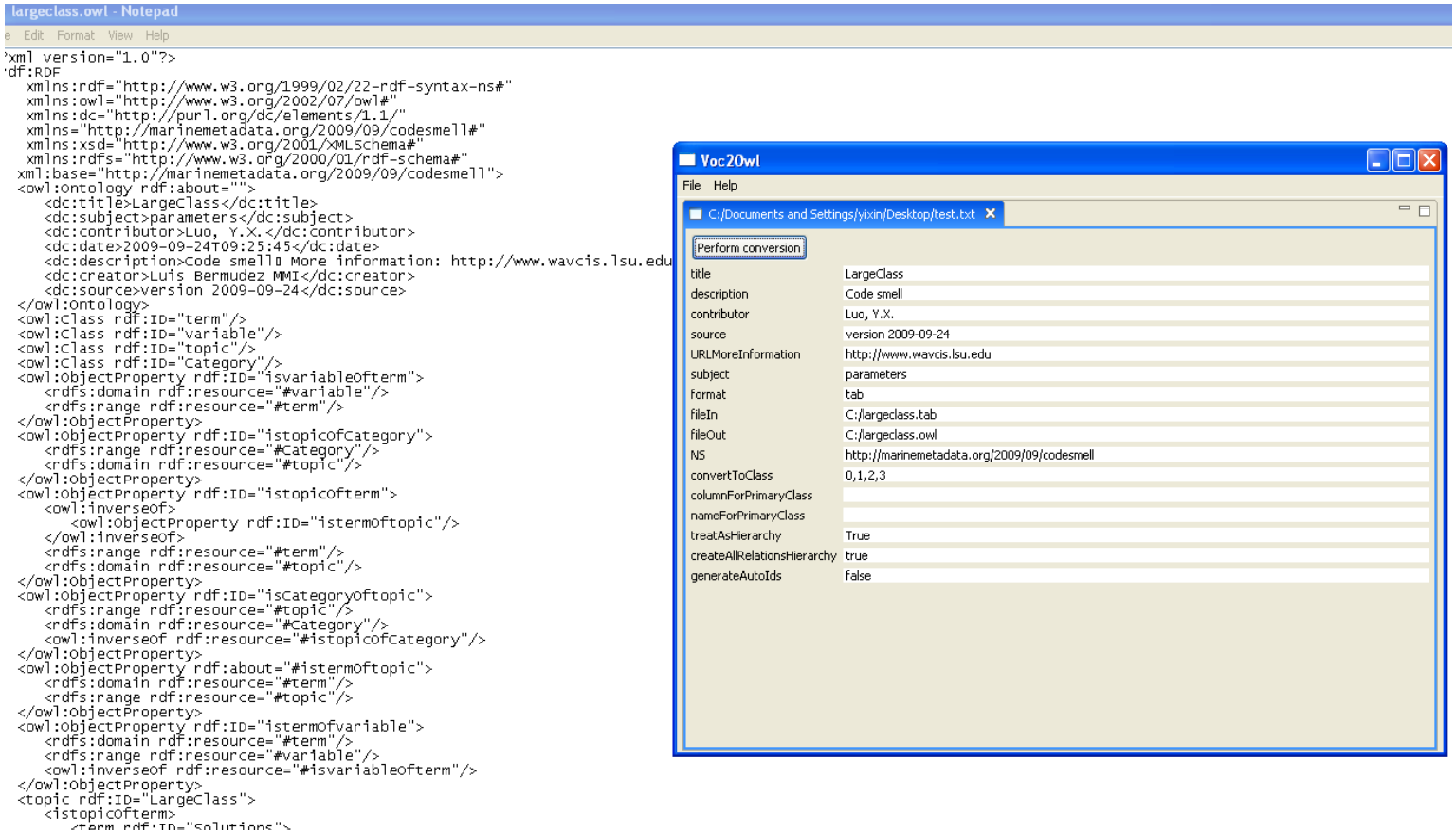
Figure 4.14 shows an example of the uploading, querying, and mapping of

Category	topic term	variable	
Code Smell	LargeClass	Symptoms	TooManyInstanceVariablesOrMethods
Code Smell	LargeClass	Solutions	ExtractClassAndExtractInterface
Code Smell	LargeClass	Identification	2.55
Code Smell	LargeClass	Remove	1.82
Code Smell	LargeClass	Impact	3.6

A. The Input File Transferred from CodeSmell (Large Class)

Figure 4.13: Example of CodeSmell(Large Class) Converted to OWL From ASCII File

(Figure 4.13 continued)

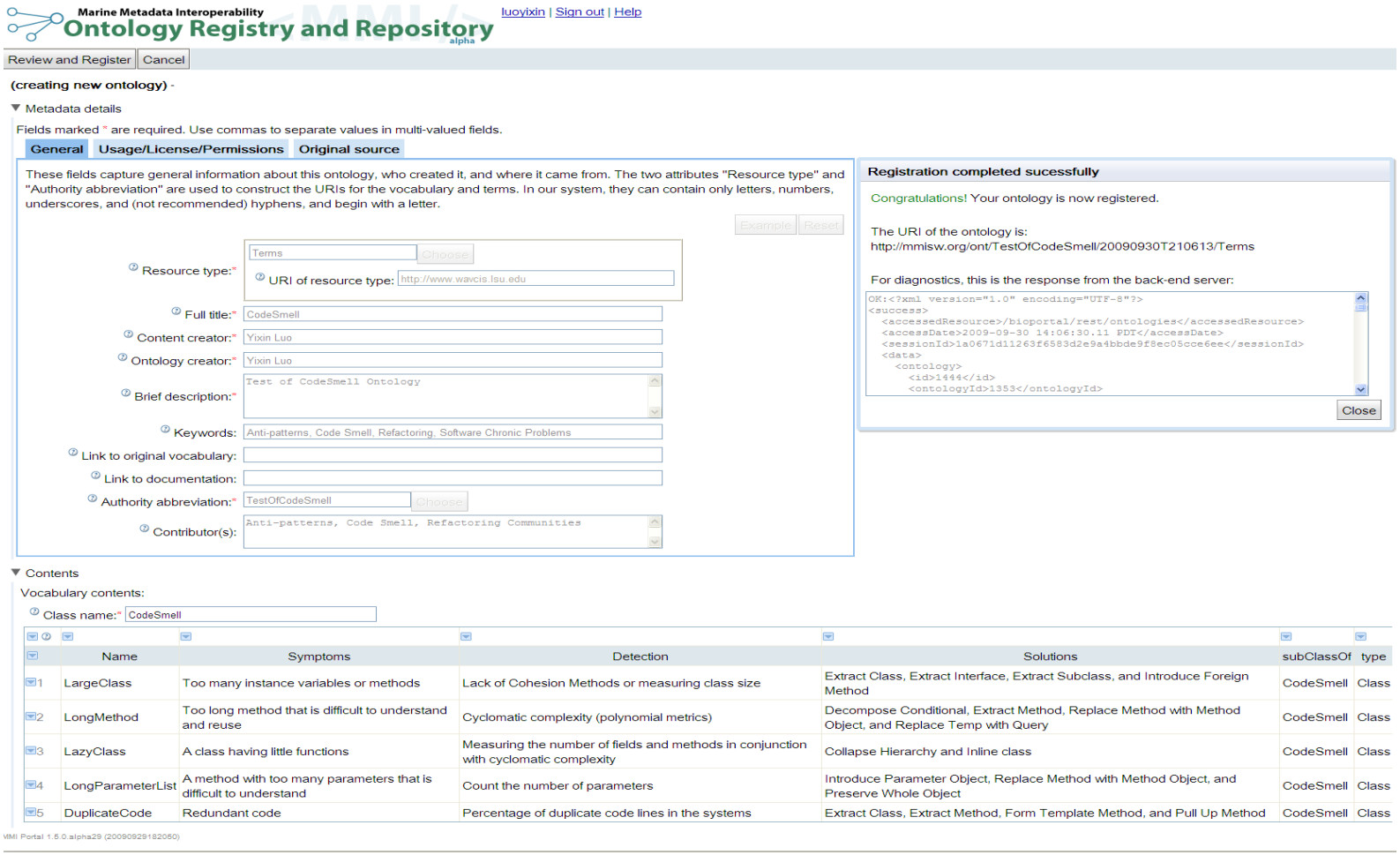


B. Ascii File Converted to OWL by Voc2OWL

(Figure 4.13 continued)

```
<?xml version="1.0"?>
<rdf:RDF
……
        <!—CodeSmell(LargeClass) - >
  <topic rdf:ID="LargeClass">
    <istopicOfterm>
      <term rdf:ID="Solutions">
        <rdfs:label>Solutions</rdfs:label>
        <istermOfvariable>
          <variable rdf:ID="ExtractClassAndExtractInterface">
            <isvariableOfterm rdf:resource="#Solutions"/>
            <rdfs:label>ExtractClassAndExtractInterface</rdfs:label>
          </variable>
        </istermOfvariable>
        <istermOftopic rdf:resource="#LargeClass"/>
      </term>
    </istopicOfterm>
    <istopicOfCategory>
      <Category rdf:ID="Code_Smell">
        <isCategoryOftopic rdf:resource="#LargeClass"/>
        <rdfs:label>Code Smell</rdfs:label>
      </Category>
    </istopicOfCategory>
    <istopicOfterm>
      <term rdf:ID="Symptoms">
        <istermOfvariable>
          <variable rdf:ID="TooManyInstanceVariablesOrMethods">
            <isvariableOfterm rdf:resource="#Symptoms"/>
            <rdfs:label>TooManyInstanceVariablesOrMethods</rdfs:label>
          </variable>
        </istermOfvariable>
……
</rdf:RDF>
```

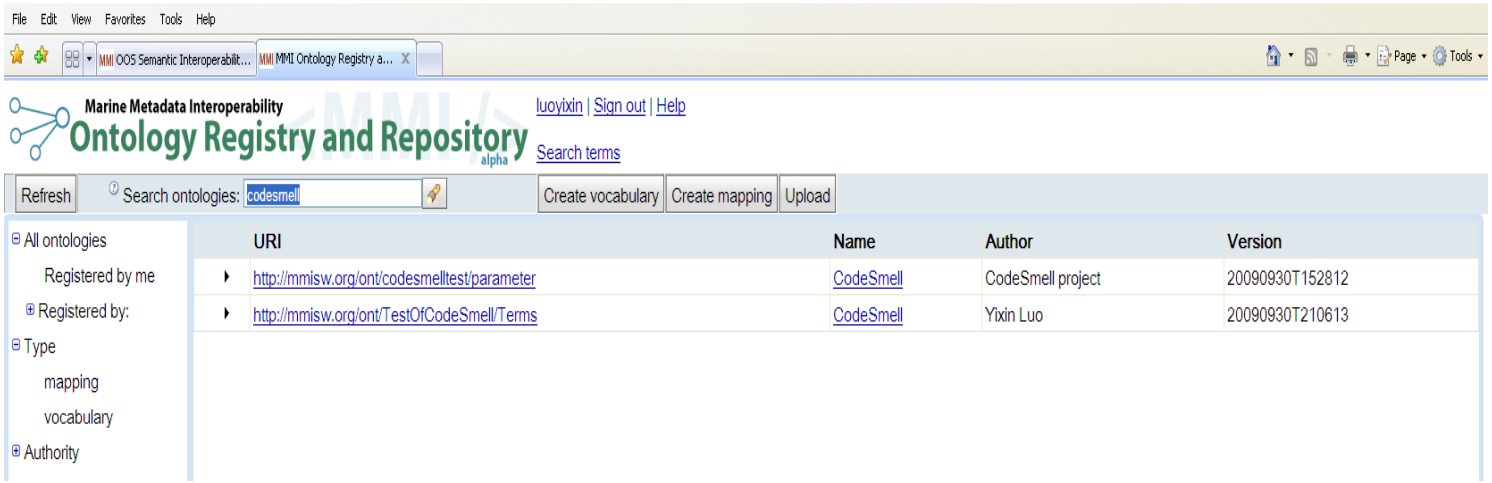
C. OWL Partial Output of CodeSmell (LargeClass)



(A) Accessing and Storing [Bermudez]

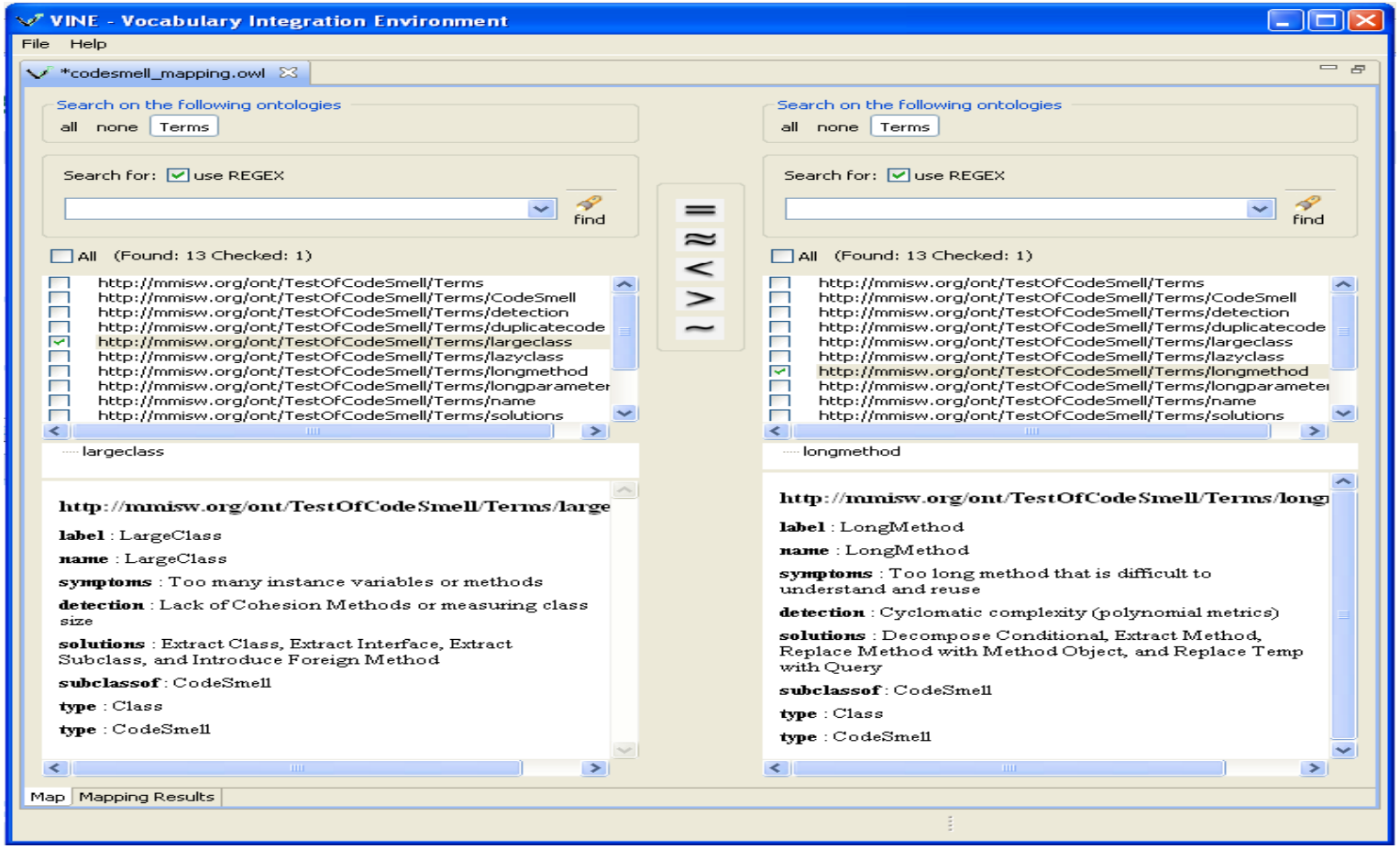
Figure 4.14: Examples of Accessing, Storing, Querying, and Mapping to OABR

(Figure 4.14 continued)



(B) Querying [Bermudez]

(Figure 4.14 continued)



(C) Mapping [Bermudez]

CodeSmell(Large Class) in the OABR registry.

Figure 4.14A shows a web page of the OABR repository describing the related information such as creator, keywords, class name, and URI about accessing OABR. The instances described by OWL could be queried by an ontological query language that supports discovery by domain, creator, terminology, and versions (Refer to Figure 4.14B). Users also could apply VINE to map vocabulary terms represented in OWL (Figure 4.14C).

Mapping is a process to describe the relations between terms that can help ontology be merged or aligned to one another [Staab et al.]. We define the mapping relations among anti-patterns or code smells as exact match, related match, and not match. Software development domain experts could compare the properties of different instances through mapping. Therefore, software developers can avoid overlapping definition by remove exact match instance, or put related match instances into new taxonomic category, or define new instance (not match with any existing instances). In Chapter 5, we will discuss more details about the relations and their applications.

We have obtained and modified several technologies and tools for implementing OABR; however, more tools and advanced technologies are needed to help it easier and more convenient for users to provide information so as to make OABR more complete and consistent.

4.10 Summary

An ontology consists of concepts, taxonomic relations and non-taxonomic relations among concepts. The OABR defines a knowledge domain model that gives a consistent definition of the properties for anti-patterns, bad code smells, and

refactoring, thus, enabling the sharing of common understanding of these concepts among software developers. The relationships between bad code smells and anti-patterns through OABR provide a new view of the related key concepts. It facilitates reliable results for properties based on ontological methods and statistical analysis. It also provides a reuse model for developing other software pattern models such as a software quality attributes model and design pattern model. We also presented tools and their applications of ontology creation, mapping, querying, and registering for OABR.

Chapter 5
Application and Evaluation of OABR

5.1 Introduction

According to [Calero], the criteria for evaluating the quality of a software onotlogy include consistency, completeness, conciseness, clarity, generality, and robustness. An ontology can be evaluated or validated by the application of the ontology and the comparison of the results with the observation or the opinions from ontology experts or the degree of acceptance from the related community [Calero]. In this research, the validation and evaluation are based on the application of OABR to identify the relations between anti-patterns and bad code smells.

We present the application of the Protégé framework to detect relations between anti-patterns and bad code smells in Section 5.2. In Section 5.3, we describes the application of metric based tools to analyze a middle- sized open source software product with different versions to detect whether the relations between anti-patterns and code smells deduced from OABR exist in the software.

5.2 Relations between Anti-patterns and Code Smells

We describe two scenarios utilizing the OABR infrastructure to identify the relations between anti-patterns and code smells. At the software design level, the OABR infrastructure can help to understand what kind of code smells related problems might occur that are caused by a given anti-pattern. On the other hand, the properties of bad code smells at the coding level may be of help tracing back to the anti-patterns at the design level that cause the problems. The information will help software developers to

prevent bad code smells by refactoring anti-patterns at the design level.

Figures 5.1A and 5.1B show examples of the OABR infrastructures that can assist software developers in understanding anti-patterns and bad code smells in two different scenarios [Luo].

Scenario 1 Anti-patterns to Bad Code Smells: The OABR infrastructure improves the understandability of the relations between anti-patterns and bad code smells in this scenario by conceptually mapping from anti-patterns to code smells to inform software developers of a code smell(s) that might result if the anti-pattern identified in a design is not resolved in the design stage A software developer would first detect an AntiPattern(Spaghetti code) by applying anti-pattern detection software such as “Analyst4j” based on the software metrics “Essential Complexity (EC_MTD)”. The software developer would utilize the OABR infrastructure to understand more about AntiPattern(Spaghetti code). Figure 5.1A displays knowledge about AntiPattern(Spaghetti code), a chronic design problem that involves applying procedural thinking in OO design. The AntiPattern(Spaghetti code)(rectangle) causes (Orange line) the instance “Many_objects_High_Coupling” (Triangle) of "“Source Code Problem”" class. The instance PoorCode(Many_objects_High_Coupling) has the symptoms described by the instance CodeSmell(Long Method). The domain of property “causes” is defined as class Anti-pattern, and the range of the “causes” property is “problems” class. For the “describes” property, the domain is the “code smell” class and the range is the “poor code” class. The OABR representation shows that the instance of AntiPattern(Spaghetti code) could be solved by instances “Code Cleanup” of subclass AP-BASED of “Refactoring” class. The

CodeSmell(LongMethod) could be removed by the instances Refactoring(Exact-method), Refactoring(Introduce-parameter-Object), Refactoring(Preserve-Whole-Object), and Refactoring(Replace-Temp-With-Query) of subclass "BS_BASED" of class "Refactoring". Through the relation between AntiPattern(SpaghettiCode) and CodeSmell(LongMethod), software developers would prevent CodeSmell(LongMethod) related problems from occurring at source code by refactoring AntiPattern(SpaghettiCode) at design level.

Scenario 2 Code Smells to Anti-patterns: The OABR representation improves the understandability of the relationship between anti-patterns and bad code smells in this scenario by conceptually mapping from code smells in existing source code to anti-patterns at the design level to assist in resolving such code smells by indicating the refactoring anti-patterns at the design level. In this scenario, software developers detect the CodeSmell(Large Class) using a metrics based code smell detection tools such as Eclipse with Check Style. Figure 5.1B provides information about CodeSmell(Large Class). CodeSmell(Large Class) is an instance of SOLUTION_BASED subclass of code smell. Its symptom property relates to the instance PoorCode(Large_class_no_cohesion) that is caused by the AntiPattern(Blob) at the design level. Therefore, the OABR infrastructure improves the understandability by showing that the anti-pattern AntiPattern(Blob) causes CodeSmell(Large Class). CodeSmell(Large Class) could be solved by refactoring(Extract Class) and refactoring(Extract Subclass) from the "BS-BASED" Refactoring category .

Figure 5.1B also implied creating a new taxonomy based on refactoring methods. CodeSmell(Large Class) and CodeSmell(Data Class) could be put in the same category

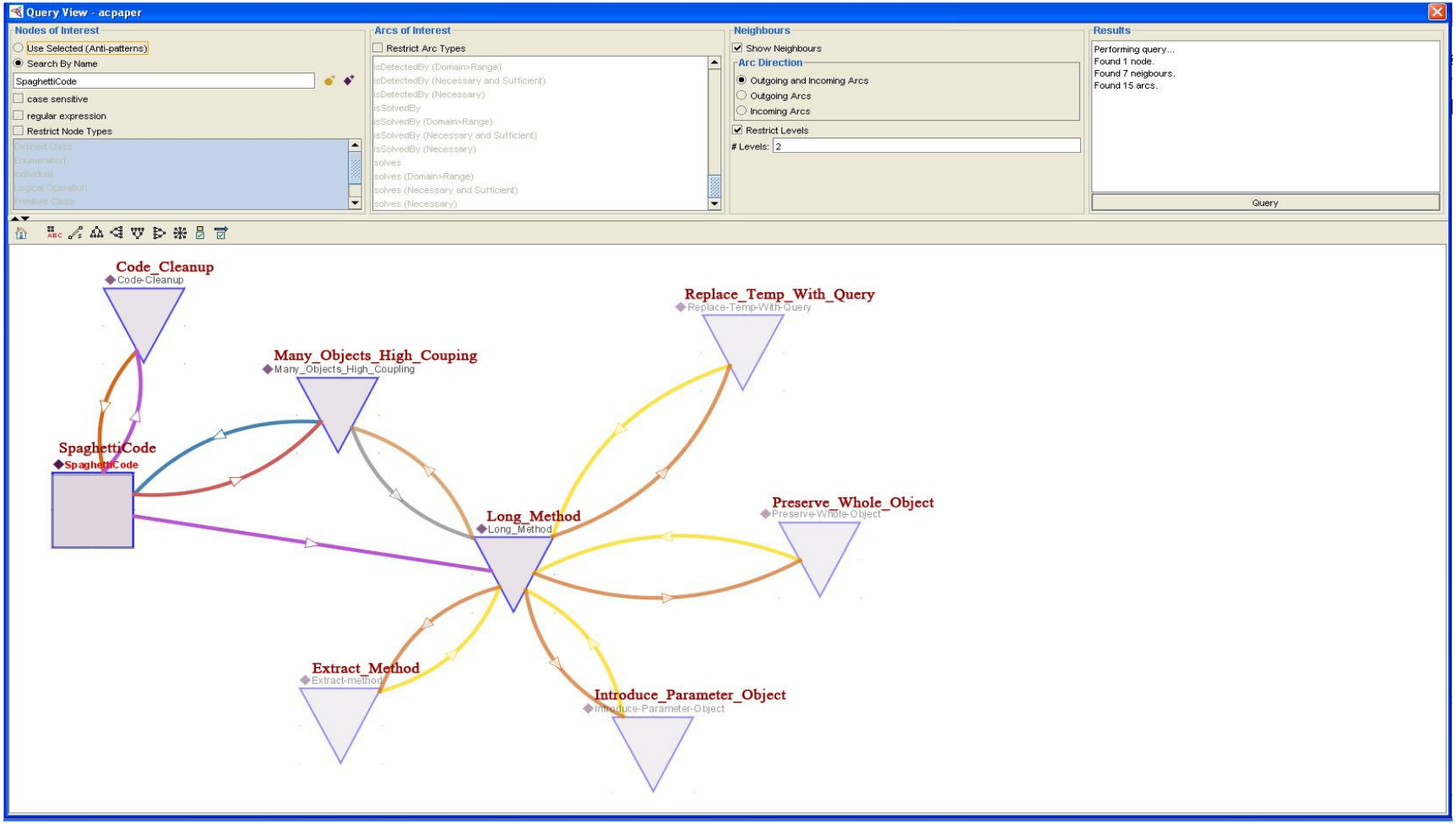
of a taxonomy as these two code smells could be solved by same refactoring methods like Refactoring(Extract Class) and Refactoring(Extract Subclass).

The relations between anti-patterns, bad code smells and refactoring shown through the OABR infrastructure are valuable to software development. Software developers can identify the existence of an anti-pattern represented in OABR and expect the occurrence of chronic problems in the source code as shown by its associated code smell and vice versa. The OABR also provides priority information about bad code smells. Software developers could decide which code smell should be tolerated, which should be removed and when to remove it, whether refactoring at source code might be costly, or refactoring related anti-patterns at the design level is preferable. The refactoring of bad code smells and anti-patterns, especially those "strong" bad code smells, will enable software engineers to improve software quality such as maintenance and understanding in an efficient way.

5.3 Testing the Relations between Anti-patterns and Code Smells in a Software Project

In this section, we describe the test of the relations between the anti-patterns and bad code smells in an open source software project and compare them to the relations represented in OABR infrastructure shown in Section 5.2.

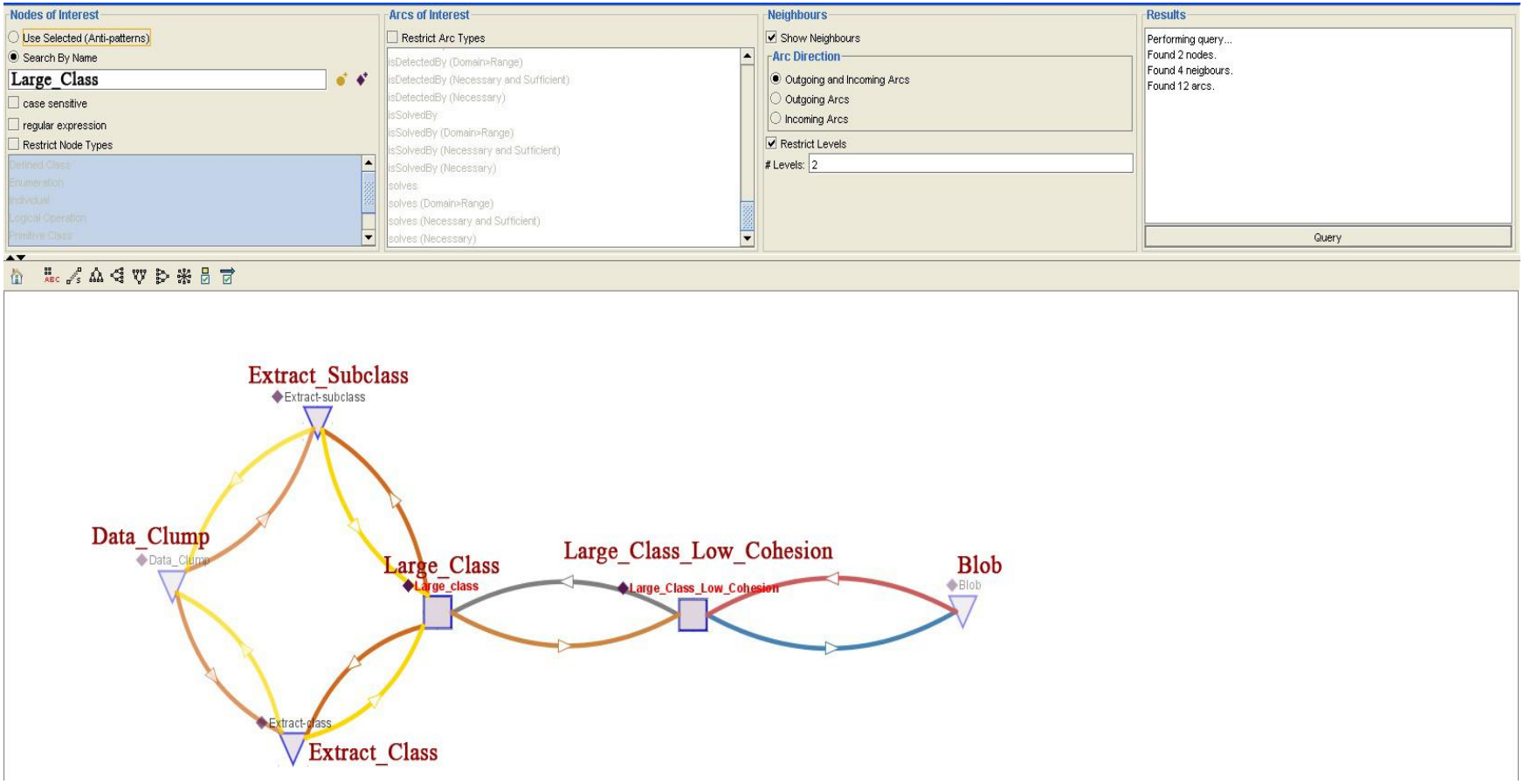
We applied metric-based tools to detect an open source software. The testing tools we used for detecting anti-pattern and bad code smells are 'Analyst4j', 'CheckStyle', and 'PMD' with Eclipse. The download and related installation documents of metric-based tools can be found at [http://www.sourceforge.net]. We use 'Analyst4j' as a plug-in for 'Eclipse' to detect anti-patterns along with 'Checkstyle' and 'PMD' as plug-ins for



A. Instance of AntiPattern(Spaghetti Code)

Figure 5.1: Instances of Anti-pattern Classes and Bad Code Smell Classes

(Figure 5.1 continued)



B. Instance of CodeSmell(Large Class)

'Eclipse' to detect bad code smells. The benefits of metrics-based tools include testing software that would be impossible to evaluate and ability to repeat processing codes automatically and systematically. The disadvantages of metric based tools are that the selection of metrics is heuristic [Wang] and they do not work for anti-patterns and bad code smells that can only be detected by heuristic analysis.

The rules for setting metrics to test related code smells are shown in Table 5.1. For a specific code smell, PMD and check style use the same metrics. Figure 5.2A to Figure 5.2C show examples of rules for detecting bad code smells in XML files for Check Style. For example, we set a value like maximum lines of code (size violation) for testing CodeSmell(Long Method). The default value for check CodeSmell(Long Method) by Check Style is 150 (Note: Some researchers believe the value should be 20 at most [http://www.ibm.com/developerworks/library/j-ap07088/]). In Figure 5.2C, we set 40 as the testing value, not including method declaration, constructor declaration or counting empty.

Based on the identification property for anti-patterns and bad code smells, we chose those that have the high or medium identification index or could be detected by metrics. The testing of anti-patterns includes AntiPattern(Spaghetti Code), AntiPattern(Swiss Knife), AntiPattern(Using Inheritance), AntiPattern(Procedure Oriented), and AntiPattern(Blob). The testing of code smells includes CodeSmell(Long Method), CodeSmell(Lazy Class)/CodeSmell(Data Class), CodeSmell(Large Class), CodeSmell(Conditional Complexity), CodeSmell(Duplicated Code) and CodeSmell(Data Class).

Table 5.1: Rules for PMD to Check Code Smell

Code Smell	Metrics	PMD Rule Names	CheckStyle Rule
Long Method	Cyclomatic Complexity, Halstead and NLOC	'ExcessiveMethodLength'	Size Violations-> Maximum Method Length
Data Class/Lazy Class	Cyclomatic Complexity and Number of fields	'CyclomaticComplexity' 'TooManyFields' 'TooManyMethods'	-
Large Class	NCLOC and Lack of Cohesion Methods	'ExcessiveClassLength'	-
Switch Statements	Conditional Complexity and NLOC	'CyclomaticComplexity'	Metrics-> Cyclomatic Complexity
Long Parameter List	Number of Parameters of Each Method	'ExcessiveParameterList'	-
Duplicate Code	-	-	Duplicates->Strict Duplicate Code

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Puppy Crawl//DTD Check Configuration
1.3//EN" "http://www.puppycrawl.com/dtds/configuration_1_3.dtd">
<!--
This configuration file was written by the eclipse-cs plugin configuration
editor
<!--
Checkstyle-Configuration: Conditional Complexity
Description:
Code Smell of "Conditional Complexity" detected!!!
-->
<module name="Checker">
<property name="severity" value="warning"/>
<module name="TreeWalker">
<module name="CyclomaticComplexity"/>
</module>
</module>
```

A. CheckStyle Rules for Detecting CodeSmell(Conditional Complexity)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Puppy Crawl//DTD Check Configuration
1.3//EN" "http://www.puppycrawl.com/dtds/configuration_1_3.dtd">
<!--
This configuration file was written by the eclipse-cs plugin configuration
editor
-->
<!--
Checkstyle-Configuration: duplicate code
Description:
Code smell of "Duplicate code" detected!
-->
<module name="Checker">
<property name="severity" value="warning"/>
<module name="StrictDuplicateCode"/>
</module>
```

B. Checkstyle Rules for Detecting CodeSmell(Duplicated Code)

Figure 5.2: Rules for Check Style to Detect Code Smells in XML

(Figure 5.2 Continued)

<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE module *(View Source for full doctype...)*>

- <!--
This configuration file was written by the eclipse-cs plugin configuration editor
--> 

<!--
Checkstyle-Configuration: Long Method
Description:
Hey, code smell of "long method" is detected here!
--> 

- <module name="Checker">

<property name="severity" value="warning" />

- <module name="TreeWalker">

- <module name="MethodLength">

<property name="max" value="40" />

<property name="countEmpty" value="false" />

</module>

</module>

</module>

C. Check Style Rules for Detecting CodeSmell(Long Method)

Table 5.2: The Output of Anti-patterns and Bad Code Smells Detected by PMD, Check Style, and Analyst4j

A. Jfreechart 0915

	Anti-patterns					**Code Smell**					
V1	Spaghetti code	Swiss Knife	Using Inheritance	Procedure Oriented	Blob	Long Method	Lazy Class/Data Class	Large Class	Conditional Complexity	Duplicated code	Long Parameter List
V2chart (34)	1	1	8	4	3	13	20	3	11	7	1
V3axis.junit (17)	0	0	0	16	1	3	1	1	0	0	0
V4plot (27)	3	3	18	1	7	36	52	6	27	27	0
V5chart.ui (12)	0	0	10	1	0	5	3	0	2	2	0
V6data (50)	0	0	23	7	1	14	28	2	15	7	1
V2time (22)	1	1	8	4	3	4	5	0	3	0	0
V3render (50)	0	0	29	15	2	47	78	1	42	26	22
V4axis (35)	0	0	19	4	7	28	38	4	17	2	0

B. Jfreechart 056

	Anti-patterns					**Code Smell**					
V1	Spaghetti code	Swiss Knife	Using Inheritance	Procedure Oriented	Blob	Long Method	Lazy Class/Data Class	Large Class	Conditional Complexity	Duplicated code	Long parameter list
V2chart (38)	0	0	17	10	2	3	10	0	6	-	10
V3axis.junit(5)	-	-	-	-	-	-	-	-	-	-	-
V4plot	-	-	-	-	-	-	-	-	-	-	-
V5chart.uil (6)	0	0	6	0	0	0	0	0	0	-	0
V6data	0	0	0	1	0	0	0	0	0	-	0
V2time	-	-	-	-	-	-	-	-	-	-	-
V3render	-	-	-	-	-	-	-	-	-	-	-
V4axis	-	-	-	-	-	-	-	-	-	-	-

C. Jfreechart100pre

	Anti-patterns					**Code Smell**					
V1	Spaghetti code	Swiss Knife	Using Inheritance	Procedure Oriented	Blob	Long Method	Lazy Class/Data Class	Large Class	Conditional Complexity	Duplicated code	Long parameter list
V2chart	-	-	-	-		5		-	15	-	-
V3axis.junit (21)	0	0	0	19	0	2	1	1	0	-	-
V4plot (37)	3	3	23	1	9	14	76	11	34	-	1
V5chart.uil (12)	0	0	10	1	0	2	-	-	3	-	-
V6data (19)	0	0	7	0	0	0	10	10	3	-	-
V2time (26)	0	0	22	2	5	-	12		4	-	-
V3render (10)	0	0	1	1	1	2	12	7	3	-	-
V4axis (41)	0	0	22	4	8	4	59	13	23	-	-

D. Jfreechart0920

	Anti-patterns					**Code Smell**					
V1	Spaghetti code	Swiss Knife	Using Inheritance	Procedure Oriented	Blob	Long Method	Lazy Class/Data Class	Large Class	Conditional Complexity	Duplicated code	Long parameter list
V2chart (25)	4	4	5	3	3	5	21	4	18	-	2
V3axis.junit (26)	0	0	0	25	0	0	-	2	0	-	0
V4plot (34)	3	3	22	1	9	12	48	7	40	-	1
V5chart.uil(12)	0	0	10	1	0	2	5	0	6	-	0
V6data(50)	0	0	19	6	2	2	32	2	2	-	2
V2time(23)	0	0	20	1	3	0	5	0	0	-	0
V3render(50)	0	0	34	13	4	27	86	3	85	-	27
V4axis(38)	0	0	20	4	7	5	31	5	28	-	0

Table 5.3: Testing of Correlation Coefficient R^2 about Anti-patterns and Code Smells

Correlation coefficient R^2		**Anti-patterns**				
		Spaghetti code	Swiss Knife	Using Inheritance	Procedure Oriented	Blob
Code Smell	Long Method	0.8693	-	0.432	0.1977	0.1613
	Data Class/Lazy Class	0.202	0.202	0.6811	0.06	0.2805
	Large Class	0.3489	0.3489	0.0835	0.0569	0.8415
	Conditional Complexity	0.5084	0.5327	0.4491	0.0138	0.5263
	Duplicated Code	-	-	0.4002	0.2954	0.5656
	Long Parameter List	0.1794	0.1794	0.2466	0.4024	0.0054

Table 5.4: Testing of Pearson's P-Value about Anti-patterns and Code Smells

Pearson's P-value		**Anti-patterns**				
		Spaghetti code	Swiss Knife	Using Inheritance	Procedure Oriented	Blob
Code Smell	Long Method	0.000	0.000	0.627	0.705	0.004
	Data Class/Lazy Class	0.085	0.085	0.173	0.466	0.028
	Large Class	0.831	0.831	0.514	0.681	0.009
	Conditional Complexity	0.667	0.667	0.136	0.530	0.025
	Duplicated Code	0.011	0.011	0.519	0.458	0.202
	Long Parameter List	0.441	0.441	0.767	0.862	0.553

The sample project “Jfreechart” is written in 100% JAVA language. It is open source and well documented. The “Jfreechart” project was founded in 2000 and is used by more than 40,000 developers. Currently, there are about 30 versions of ‘Jfreechart’ that can be downloaded from http://sourceforge.net/projects/jfreechart/files/. We randomly chose four versions, “056”, “0915”, “0920” and “pre100”, as the testing samples. Before the testing, we set up the configuration path and compiled the entire project.

As the number of the files processed by the free version of Check Style is limited to 50, we selected modules instead of using the whole project to test each version of ‘JFreechart’. The modules were selected randomly before the testing. The selected modules include “V2chart”, “V3axis.junit”, “V3plot”, “V5chart.ui”, “V6data”, “V2time”, “V3render”, and“V4axis”.

Table 5.2 shows the testing results of anti-patterns and bad code smells in the modules of different versions. There are missing values for some data in Table 5.2, because the testing modules do not exist in some versions. For example, version 056 does not have the V6data, V3render, and V4 axis modules, thus some code smells like CodeSmell(DuplicatedCode) could not be detected by the metric-based tools in some versions of Jfreechart.

Analysis of R squared and P values for anti-patterns and bad code smells can measure the strength of the linear relationship between anti-patterns and code smells. R squared (also called the coefficient of determination) is the proportion of variance in Y that can be accounted for by knowing X. A low p value (less than 0.05 for example) means the possibilities of the future values that are not related is quite low, thus, there is a significant relationship between two variables. The results for r squared and p value are

shown in Table 5.3 and Table 5.4, respectively. We conclude that AntiPattern(Blob) and CodeSmell(Large Class), AntiPattern(Spaghetti Code) and CodeSmell(Long Method) are positively linear correlated. The results support the finding about the relations between anti-patterns and bad code smells from OABR analysis in Section 5.2.

Another interesting finding from the results is that the testing values for AntiPattern(SpaghettiCode) and AntiPattern(SwissKnife) are the same, showing that the definition of the properties for these two anti-patterns cannot be identified from each other.

5.4 Summary

In this chapter, we showed the application of OABR in two scenarios for software development by analyzing the relations between anti-patterns and bad code smells. The tests on the anti-patterns and bad code smells by metric-based tools in different version of a real software project “Jfreechart” support the relations obtained from the OABR infrastructure.

Chapter 6
Summary and Conclusions

6.1 Summary

Anti-patterns and bad code smells describe chronic problems that affect software quality. Refactoring can help solve anti-patterns and bad code smells. In this research, we developed an ontological infrastructure, OABR, showing the relations between anti-patterns, bad code smells, and refactoring to assist in the identification and resolution of their associated problems. Focusing on the interrelationships from anti-patterns to bad code smells and from bad code smells to anti-patterns, the OABR infrastructure provides guidance to software developers in the following ways.

At the software design level, the OABR infrastructure helps to understand what kind of bad code smells related problems occur that are caused by a given anti-pattern and what refactoring for the anti-pattern should be applied. This information will help software engineers to reduce bad code smells at the design level.

At the coding level, the OABR infrastructure can help programmers to understand the bad code smells caused by a specific anti-pattern, the identify methods, and the related refactoring to remove the bad code smells.

At the design or coding level, information on the properties of bad code smells and anti-patterns can help determine which anti-pattern or bad code smell should be tolerated or removed. As removal of all the anti-patterns or bad code smells is not practical, refactoring only the selected bad code smells or anti-patterns can improve the maintainability efficiently.

6.2 Contributions

We developed the OABR infrastructure to help detect and remove software problems through the refactoring, anti-pattern, and bad code smells identification in the early stages of software development.

Significant contributions of this research to improve software quality include:

- Knowledge domain model including the software development concepts of anti-patterns, refactoring, and bad code smells. This model can facilitate the sharing of common understanding of these concepts among software communities;
- Reuse model for the development of other software pattern models such as quality models or design pattern model because OABR could be included with other ontologies like software quality attributes, software metrics, or other design patterns;
- Ontological approach including statistical analysis and more formal definitions of bad code smells and anti-patterns above and beyond the existing heuristic definitions, thereby improving their understandability and provability;
- New classification of anti-patterns, refactoring, and bad code smells that improves the clarity of related concepts; for example, bad code smells with similar causes might be resolved in similar ways;
- Consistent way to define bad code smells, anti-patterns and refactoring with templates, making it easier to identify and compare.

- New insight to the relations between anti-patterns and code smells that assist in determining whether or not to remove some anti-patterns in the early stages of software life cycle to prevent the occurrence of bad code smells.

6.3 Future Work

The ongoing challenges and future work of the research include the following:

- Obtain more inputs from the software community to expand OABR and set constraints for the class properties, given that the development of OABR is an iterative process;
- Develop OABR registries and related web services, making it easier for users to identify and test new bad code smells, anti-patterns, and refactoring.
- Expand or create a new ontology by merging or assigning OABR with other ontologies about software development such as design patterns, software metrics, and software quality attributes.

References

[Akroyd] M. Akroyd, “Anti-Patterns: Vaccinations against Object Misuse”, Proc. Object World West, 1996.

[Badder] F. Badder, “Handbook on Ontologies”, Springer, ISBN: 3-540-40834-7, 2003.

[Badder et al.] Fraanz Baader, Diego Calvanese, Deborah Mcguinness, Daniele Nardi, Peter Patel-Schneider, “The Description Logic Handbook – Theory, Implementation and Applications”, Cambridge University Presss, ISBN 0-512-78176-0, 2003.

[Bansiya & David] Bansiya & David, “A Hierarchical Model for Object-Oriented Design Quality,” IEEE Trans. Software Engineering, vol. 28, no. 1, pp. 4-17, 2002.

[Bermudez] Bermudez, L.E., Graybeal, J., and Arko, R., "A Marine Platforms Ontology: Experiences and Lessons," in Semantic Sensor Networks Workshop at the 5th International Semantic Web Conference (ISWC) 2006, Athens, GA, 2006.

[Bennett & Rajlich] K. Bennett and V. Rajlich, “Software maintenance and evolution: a roadmap”, ICSE - Future of SE Track, 2000.

[Brown et al.] W.H. Brown et al., “Anti-patterns: Refactoring Software Architectures and Projects in Crisis”, John Wiley & Sons, ISBN 0-471-19713-0, 1998.

[Buschmann & Schmidt] D. Schmidt, H. Rohnert, and F. Buschmann, “Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects,” Wiley, ISBN-13: 978-0471606956, 2000.

[Calero & Piattini] C. Calero, Francisco Ruiz, and Mario Piattini, “Ontologies for Software Engineering and Software Technology”, Springer, ISBN 3-540-34517-5, 2005.

[Cheng & Liu]Yung-pin Cheng, “An Ontology-Based Taxonomy of Bad Code Smells”, Proceeding (559) Advances in Computer Science and Technology, 2007.

[Chidamber & Kemerer] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design”, IEEE Transactions on Software Engineering, Page 476-493, 1995.

[David] Davies, J., D. Fensel, et al.., “Towards the Semantic Web: Ontology-driven Knowledge Management”, Wiley Eds., 2002.

[Emden & Moonen]Eva Van Emden and Leon Moonen, “Java Quality Assurance by Detecting Code Smells”, Proceedings of the 9th Working Conference Reverse Engineering, May 2004.

[Eden & Turner] A. Eden and R. Turner, “Toward an Ontology of Software Design: The Intension/Locality Hypothesis”, 3rd European Conf. Computing And Philosophy – ECAP, 2005

[Fowler] M. Fowler, “Refactoring: Improving the Design of Existing Code”, Addison-Wesley, ISBN 0201485672, 1999.

[Fowler] M. Fowler, “Analysis Patterns: Reusable Object Models”, Addison-Wesley Object Technology Series, ISBN-13: 978-0201895421, 1996.

[Fowler & Becker]Fowler, M. &. Becker, K. "Bad Smells in Code," in Refactoring: Improving the Design of Existing Code. pp. 75-88. Addison-Wesley, 2001.

[Franz Baader] Fraanz Baader, Diego Calvanese, Deborah Mcguinness, Daniele Nardi, Peter Patel-Schneider, “The Description Logic Handbook – Theory, Implementation and Applications”, Cambridge University Presss, ISBN 0-512-78176-0, 2003.

[Gamma & Vlissides] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, ISBN-13: 978-0201633610, 1995.

[Graybeal] Graybeal, J. and Bermudez, L.E., "When Hydrospheres Collide: Lessons in Practical Environmental Ontologies," in 7th International Conference on Hydroscience and Engineering (ICHE 2006) Philadelphia, PA, 2006

[Horridge et al.] M. Horridge, H. Hanblauch, A. Rector, R. Stevens, C. Wroe, “A Practical Guide To Building OWL Ontologies Using The Protégé-OWL Plugin and CO-ODE Tools”, The Unviersity of Manchester and Stanford University, Aug. 2004.

[Hoss] Allyson M. Hoss , “Ontology-based methodology for error detection in software design”, Louisiana State University, May 2006.

[Karpijoki] Vesa Karpijoki, “AntiPattern,” Technical Report, 2006.

[Koenig] A. Konenig, Patterns and Anti-patterns, Journal of Object-Oriented Programming, 8(10), March, 1995.

[Horridgy et al.] M. Horridge, H Knublauch, A. Rector, R. Stevens, C. Wroe, “A Practical Guide To OWL Ontologies Using the Protégé-OWL Plugin and CO-ODE Tools”, http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf, Aug. 2004.

[IEEE1990] http://standards.ieee.org/reading/ieee/std_public/description/se/index.html.

[ISO 9126] http://www.iso.org/iso/catalogue_detail.htm?csnumber=39752.

[Laplante] Phil Laplante, “Antipatterns in the Creation of Intelligent Systems”, Human - Centered Computing, Feb. 2007.

[Laplante et al.] Phillip Laplante and Colin Neill, “Antipatterns: Identification, Refactoring, and Management”, ISBN 0-8493-2994-9, Auerbach Publications, 2009.

[Lee & Meier] D. Lee and R. Meier, “Primary-Context Model and Ontology:
A Combined Approach for Pervasive Transportation Services”, Pervasive Computing and Communications Workshops, March 2007.

[Luo] Yixin Luo, Allyson Hoss, and Doris Carver, “Ontological Analysis of Anti-Patterns and Code Smells”, IEEE Aerospace 2010 Big Sky, Montana.

[Marinescu] R. Marinescu, “Detecting design flaws via metrics in object oriented systems”, In Proceedings of TOOLS, 2001.

[Marinescu et al.] Radu Marinescu and Danie Ratiu, "Quantifying the quality of object-oriented design: The factor-strategy model", In Proceedings of WCRE on Software Maintenance, Page 350-359, 2005

[Mens] T. Mens and T. Tourwe, "A Survey of Software Refactoring", IEEE Trans. Software Engineering, 30(20):126-139, Feb. 2004.

[Mika] Mika Mantyla, "Bad Smells in Software - a Taxonomy and an Empirical Study", Dissertation, Helsinki University of Technology, 2003.

[Mika & Lassenius] Mika Mantyla and C. L, "Bad Smells - Humans as Code Critics", Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004.

[Moha et al.] Naouel Moha, Jihene Rezgui, Yann-Ga Gueuc, Petko Valtchev, Ghizlane El Boussaidi. "Using FCA to Suggest Refactorings to Correct Design Defects", Proceedings of the 4th International Conference On Concept Lattices and Their Applications (CLA 2006), pp. 297-302, In S. Ben Yahia & E. Mephu Nguifo Eds, October 30-November 1st, 2006, Hammamet, Tunisia.

[Moha] Naouel Moha, Yann-Ga Gueuc, Pierre Leduc. "Automatic Generation of Detection Algorithms for Design Defects." Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE 2006), pp. 297-300, September 18-22, 2006, Tokyo, Japan.

[Moha 2] Naouel Moha. "Detection and Correction of Design Defects in Object-Oriented Architectures", Doctoral Symposium, 20th edition of the European Conference on Object-Oriented Programming (ECOOP 2006), July 3-7, 2006, Nantes, France

[Noy & McGuinness] N. Noy and D. McGuinness, "Ontology Development 101: A guide to Creating Your First Ontology", Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, March 2001.

[Ratiu & Marinescu] Daniel Ratiu and Radu Marinescu, "Using History Information to Improve Design Flaws Detection", In Proceedings of CSMR, May 2003.

[Reinout] Reinout Van Rees, "Clarity in the usage of the terms ontology, taxonomy and classification." In Robert Amor, editor, Construction IT bridging the distance, W78 international conference. CIB-W78, University of Auckland, 2003.

[Slinger] Stefan Slinger, "Code Smell Detection in Eclipse", Thesis, Delft University of Technology, 2001.

[Smith & Lioyd] Connie Smith and Lioyd Williams, "New Software Performance Anti-Patterns: More Ways to Shoot Yourself in the Foot", Performance Engineering Services and Software Engineering Research, Sept. 2006.

[Smith et al.] Connie Smith and Lloyd Williams, "Software Performance Anti-patterns?" Proceedings 2nd International Workshop on Software and Performance, Dec. 2005.

[Staab et al.] S. Staab and R. Studer, " Handbook on Ontology", Springer, ISBN 3-540-

40834-7, 2004.

[Topquadrant] Topquadrant white paper, “Ontology Myth or Magic? Toward the Practical Application of Ontology-enabled Knowledge Solutions”, <http://www.topquadrant.com/documents/TQ0403_Semantic%20Solutions-Getting%20Started.pdf>, USA, 2003

[Zhang] X.P. Zhang, “Design and Implementation of an Ocean Observing System: WAVCIS (Wave-Current-Surge Information System) and Its Application to the Louisiana Coast”, Dissertation, Louisiana State University, July 2003.

Appendix A: Bad Code Smell Examples

DATA CLUMPS
Name: Data Clumps Symptoms: Data is always coherent with each other. Detection: If one value is removed, the data set will be meaningless. Relationship: Magic Numbers/Magic String Solutions: Extract Class, Introduce Parameter Object, and Preserve Whole Object Identifications: Medium Removal: difficult Impact: Medium

DEAD CODE
Name: Dead Code Symptoms: code never process at running time Detection: No reference to a method or a class Relationship: Boat Anchor Solutions: Collapse Hierarchy, Inline Class, Rename Method, and Remove Parameter Identifications: Easy Removal: Easy Impact: Medium

DUPLICATE CODE
Name: Duplicate Code Symptoms: Redundant code Detection: Percentage of duplicate code lines in the systems Relationship: Cut and Paste Solutions: Extract Class, Extract Method, Form Template Method, and Pull Up Method Identifications: Easy Removal: Medium Impact: Medium

FEATURE ENVY
Name: Feature Envy Symptoms: A method is more tightly coupled to the other class than to the local one. Detection: Measuring couplings Relationship: N/A Solutions: Extract Method, Move Field, and Move Method Identifications: Difficult Removal: Medium Impact: Strong

INAPPROPRIATE INTIMACY
Name: Inappropriate Intimacy Symptoms: Tightly coupled classes Detection: Measuring couplings Relationship: N/A Solutions: Change Bidirectional Association to Unidirectional, Hide Delegate, Move Field, Move Method, and Replace Inheritance Identifications: Medium Removal: Difficult Impact: Strong

INCOMPLETE LIBRARY CLASS
Name: Incomplete Library Class Symptoms: not complete library Detection: N/A Relationship: N/A Solutions: Introduce Foreign Method and Introduce Local Extension Identifications: Medium Removal: Medium Impact: Strong

LARGE CLASS
Name: Large Class Symptoms: Too many instance variables or methods Detection: Lack of Cohesion Methods or measuring class size Relationship: Blob/God Object Solutions: Extract Class, Extract Interface, Extract Subclass, and Introduce Foreign Method Identifications: Medium Removal: Difficult Impact: Strong

LAZY CLASS
Name: Lazy Class Symptoms: A class having little functions Detection: Measuring the number of fields and methods in conjunction with cyclomatic complexity. Relationship: Poltergeist/Lava flow Solutions: Collapse Hierarchy and Inline class Identifications: Medium Removal: Easy Impact: Weak

LONG METHOD
Name: Long Method Symptoms: Too long method that is difficult to understand and reuse Detection: Cyclomatic complexity (polynomial metrics) Relationship: N/A Solutions: Decompose Conditional, Extract Method, Replace Method with Method Object, and Replace Temp with Query Identifications: Medium Removal: Difficult Impact: Strong

LONG PARAMETER LIST
Name: Long Parameter List Symptoms: A method with too many parameters that is difficult to understand Detection: Count the number of parameters Relationship: N/A Solutions: Introduce Parameter Object, Replace Method with Method Object, and Preserve Whole Object Identifications: Medium Removal: Medium Impact: Medium

MESSAGE CHAINS
Name: Message Chain Symptoms: classes asking object from one to another Detection: Measuring the couplings of a method Relationship: N/A Solutions: Hide Delegate Identifications: Medium Removal: Medium Impact: Strong

MIDDLE MAN
Name: Middle Man Symptoms: A class delegating most of its tasks to subsequent classes Detection: Many methods coupled to one class with a low cyclomatic complexity Relationship: N/A Solutions: Inline Methods, Replace Delegation with Inheritance, and Remove Middleman Identifications: Medium Removal: Difficult Impact: Strong

PARALLEL INHERITANCE HIERARCHIES
Name: Parallel Inheritance Hierarchies Symptoms: Existing parallel class hierarchies Detection: N/A Relationship: N/A Solutions: Move Field and Move Method Identifications: Medium Removal: Medium Impact: Strong

PRIMITIVE OBSESSION
Name: Primitive Obsession Symptoms: Using primitive instead of small classes Detection: N/A Relationship: N/A Solutions: Extract Class, Introduce Parameter Object, Replace Array with Object, Replace Data Value with Object, Replace Type Code with Subclass/State/strategy Identifications: Medium Removal: Medium Impact: Strong

REFUSED BEQUEST
Name: Refused Bequest Symptoms: A class could not support its inherited methods or inherited data Detection: N/A Relationship: N/A Solutions: Replace Inheritance with Delegation Identifications: Medium Removal: Medium Impact: Medium

SHORTGUN SURGERY
Name: Shortgun Surgery Symptoms: A small change affecting several classes Detection: N/A Relationship: N/A Solutions: Inline Class, Move Field, and Replace Parameter with Explicit Method Identifications: Medium Removal: Difficult Impact: Strong

SPECULATIVE GENERALITY
Name: Speculative Generality Symptoms: Unnecessary code created in anticipating the future changes Detection: Similar to Dead Code Relationship: N/A Solutions: Collapse Hierarchy, Inline Class, Remove Parameter, and Rename Method Identifications: Medium Removal: Medium Impact: Medium

SWITCH STATEMENTS
Name: Switch Statements Symptoms: Replacing polymorphism with type codes or runtime class type detection Detection: runtime detection Relationship: N/A Solutions: Introduce Null Object, Replace Conditional with Polymorphism, Replace Method with Explicit Method, Replace Type Code with Subclass/State/Strategy Identifications: Medium Removal: Medium Impact: Weak

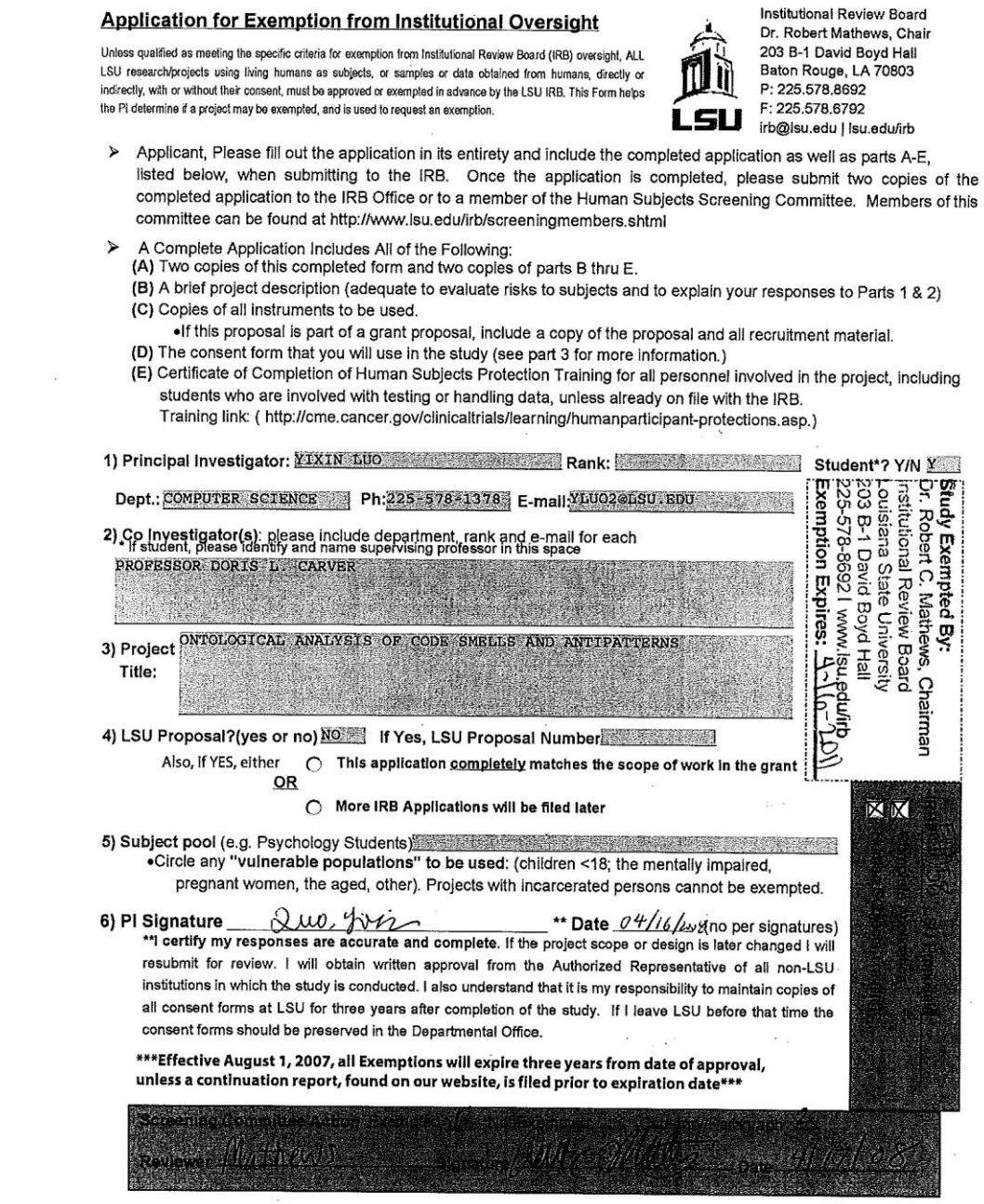
TEMPORARY FIELD
Name: Temporary Field Symptoms: A class has a variable that is only used in some situations. Detection: Comparing different methods that access each field Relationship: N/A Solutions: Extract Class and Introduce Null Object. Identifications: Medium Removal: Medium Impact: Weak

Appendix B: Refactoring Examples

CHANGE BIDIRECTIONAL ASSOCIATION TO UNIDIRECTIONAL
Name: Change Bidirectional Association to Unidirectional Scenario: Two-way association between classes needing just one associate Mechanics: 1. Check the fields that hold pointers 2. Remove reader, updates to the field and remove the field **3.** Compile and test

COLLAPSE HIERARCHY
Name: Collapse Hierarchy Scenario: Sub-class and parent class is similar Mechanics: 1. Select the class to be removed 2. Merge the class **3.** Adjust references and remove the empty class

Appendix C: Application for Exemption from Institutional Oversight



Application for Exemption from Institutional Oversight

Unless qualified as meeting the specific criteria for exemption from Institutional Review Board (IRB) oversight, ALL LSU research/projects using living humans as subjects, or samples or data obtained from humans, directly or indirectly, with or without their consent, must be approved or exempted in advance by the LSU IRB. This Form helps the PI determine if a project may be exempted, and is used to request an exemption.

LSU

Institutional Review Board
Dr. Robert Mathews, Chair
203 B-1 David Boyd Hall
Baton Rouge, LA 70803
P: 225.578.8692
F: 225.578.6792
irb@lsu.edu | lsu.edu/irb

- Applicant, Please fill out the application in its entirety and include the completed application as well as parts A-E, listed below, when submitting to the IRB. Once the application is completed, please submit two copies of the completed application to the IRB Office or to a member of the Human Subjects Screening Committee. Members of this committee can be found at http://www.lsu.edu/irb/screeningmembers.shtml
- A Complete Application Includes All of the Following:
  - **(A)** Two copies of this completed form and two copies of parts B thru E.
  - **(B)** A brief project description (adequate to evaluate risks to subjects and to explain your responses to Parts 1 & 2)
  - **(C)** Copies of all instruments to be used.
    - If this proposal is part of a grant proposal, include a copy of the proposal and all recruitment material.
  - **(D)** The consent form that you will use in the study (see part 3 for more information.)
  - **(E)** Certificate of Completion of Human Subjects Protection Training for all personnel involved in the project, including students who are involved with testing or handling data, unless already on file with the IRB. Training link: (http://cme.cancer.gov/clinicaltrials/learning/humanparticipant-protections.asp.)

1) Principal Investigator: YIXIN LUO **Rank:** **Student*? Y/N** Y

Dept.: COMPUTER SCIENCE **Ph:** 225-578-1378 **E-mail:** YLUO2@LSU.EDU

2) Co Investigator(s): please include department, rank and e-mail for each
* If student, please identify and name supervising professor in this space
PROFESSOR DORIS L. CARVER

3) Project Title: ONTOLOGICAL ANALYSIS OF CODE SMELLS AND ANTIPATTERNS

4) LSU Proposal?(yes or no) NO **If Yes, LSU Proposal Number**

Also, If YES, either ○ **This application completely matches the scope of work in the grant**
OR
○ **More IRB Applications will be filed later**

5) Subject pool (e.g. Psychology Students)
- Circle any **"vulnerable populations" to be used:** (children <18; the mentally impaired, pregnant women, the aged, other). Projects with incarcerated persons cannot be exempted.

6) PI Signature Luo, Yixin **** Date** 04/16/2008 (no per signatures)

****I certify my responses are accurate and complete.** If the project scope or design is later changed I will resubmit for review. I will obtain written approval from the Authorized Representative of all non-LSU institutions in which the study is conducted. I also understand that it is my responsibility to maintain copies of all consent forms at LSU for three years after completion of the study. If I leave LSU before that time the consent forms should be preserved in the Departmental Office.

*****Effective August 1, 2007, all Exemptions will expire three years from date of approval, unless a continuation report, found on our website, is filed prior to expiration date*****

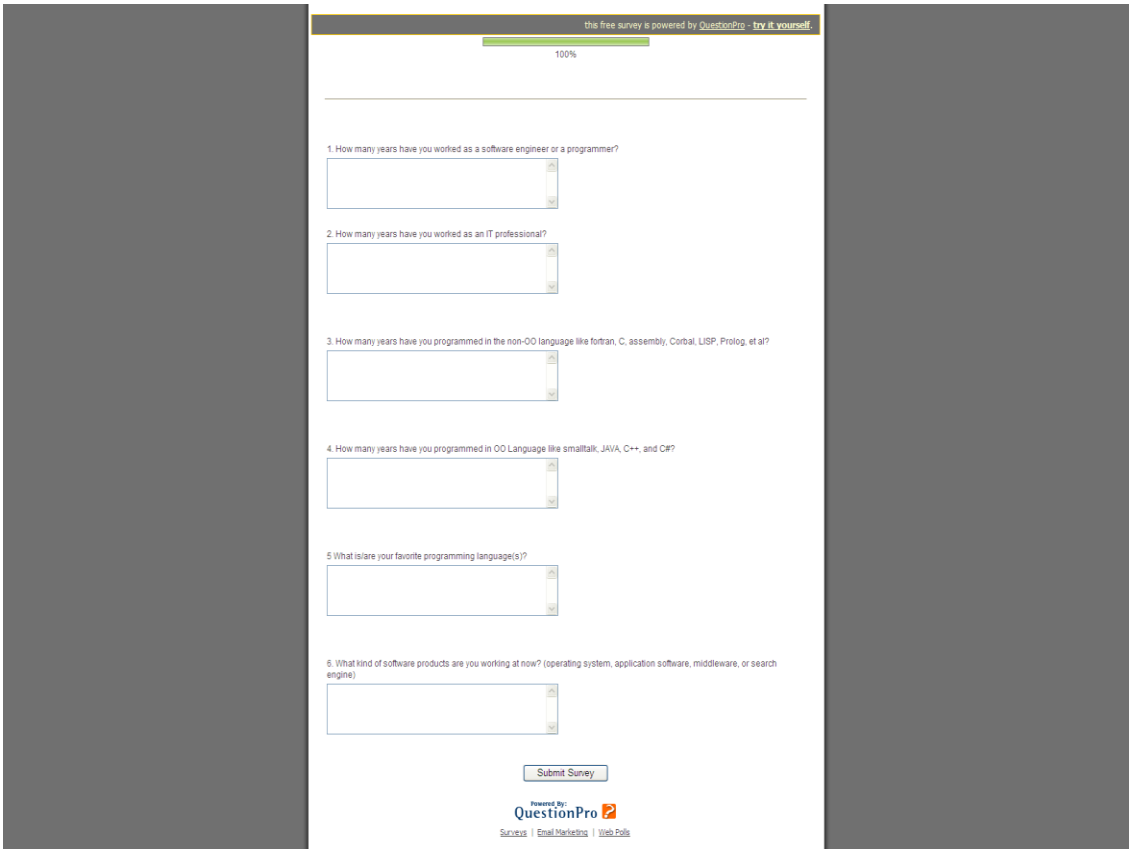
Study Exempted By:
Dr. Robert C. Mathews, Chairman
Institutional Review Board
Louisiana State University
203 B-1 David Boyd Hall
225-578-8692 | www.lsu.edu/irb
Exemption Expires: 4-16-2011

☒ ☒

Screening Committee Action: Exempt / Non-exempt / Expedited / Full Board

Reviewer: [illegible] Signature: [illegible] Date: 4/17/08

\


this free survey is powered by QuestionPro - try it yourself.
100%
1. How many years have you worked as a software engineer or a programmer?
2. How many years have you worked as an IT professional?
3. How many years have you programmed in the non-OO language like fortran, C, assembly, Corbal, LISP, Prolog, et al?
4. How many years have you programmed in OO Language like smalltalk, JAVA, C++, and C#?
5 What is/are your favorite programming language(s)?
6. What kind of software products are you working at now? (operating system, application software, middleware, or search engine)
Submit Survey
Powered By:
QuestionPro
Surveys | Email Marketing | Web Polls


Test Survey - Windows Internet Explorer
Test Survey
this free survey is powered by QuestionPro - try it yourself.
5%
Long Method is a method that is too long, so it is difficult to understand, change, or extend.
IDENTIFICATION(0 - hardest to identify 5 - easiest to identify -999 - don't know)
0 1 2 3 4 5 -999
Removal (0 - hardest to remove 5 - easiest to remove -999 - don't know)
0 1 2 3 4 5 -999
Impact (0 - no impact 5 - biggest impact -999 - don't know)
0 1 2 3 4 5 -999
Continue
Please contact LUOYIXIN@NETSCAPE.NET if you have any questions regarding this survey.
Powered By:
QuestionPro
Surveys | Email Marketing | Web Polls
Done
Internet
100%

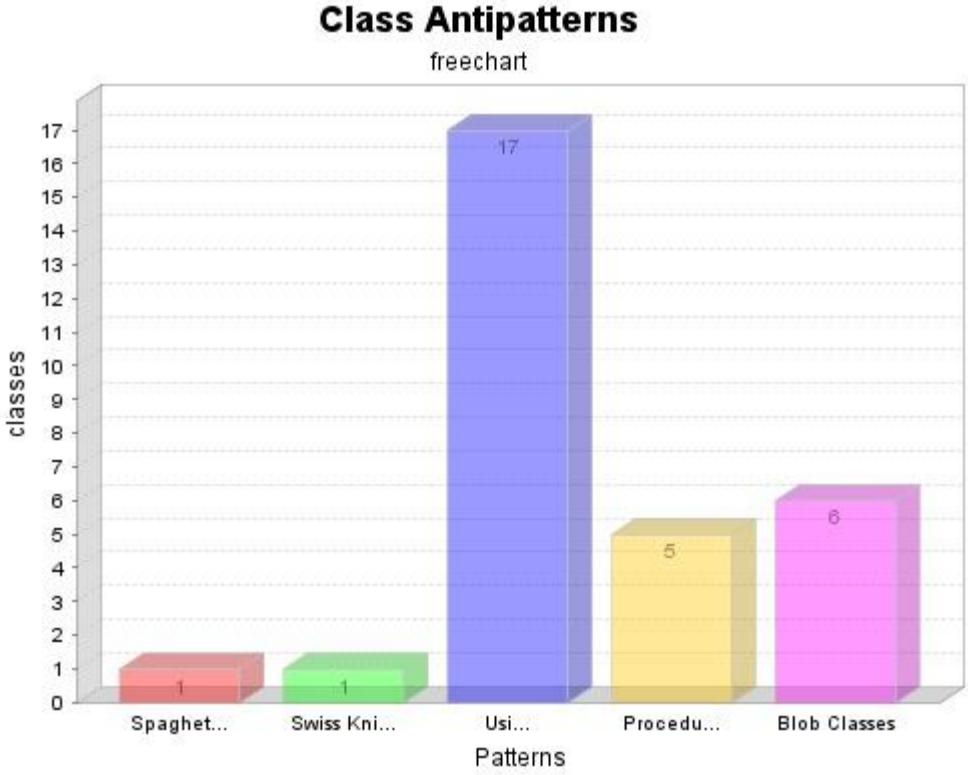
Appendix D: Examples of Using DL to Express OABR

Symbol
≡ (concept equivalence/definition) ∃ (existential restriction) ¬ (negation) ∀ (universal restriction) ∩ (intersection or conjunction of concepts) ∪ (union or disjunction of concepts) ( ) (Concept/role assertion)
Atomic Concepts
SoftwareProblems, SoftwareChronicalProblems, SourcecodeProblems, Solution, Bad solution, Refactoring
TBox
DesignProblems≡∀hasDesignProblem.softwareChronicalProblems Sourcecodeproblems≡∀hasSourcecodeProblem.softwarechronicalProblems Badsolution ≡ ¬ goodsolution DesignPattern≡∀hasDesignProblem.softwareChronicalProblems ∪ Solution AntiPattern≡DesignProbelms ∪ BadSolution CodeSmell ≡ ∃showSymptoms.SourceCodeProblems MediumCodeSmell≡((∃hasImpact.CodeSmell) ∩(≥2 hasImpact ∩ ≤3 hasImpact)) ∩((∃hasIdentification.CodeSmell) ∩ (≥2 hasIdentification ∩ ≤3 hasIdentification)) ∩ ((∃hasRemoval.CodeSmell) ∩(≥2 hasRemoval ∩ ≤3 hasRemoval))

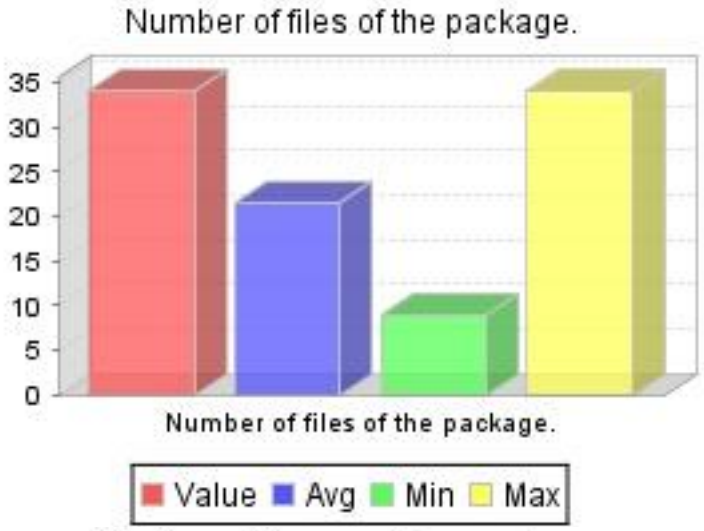
StrongCodeSmell≡≡((∃hasImpact.CodeSmell) ∩ (3≤ hasImpact)) ∪ ((∃hasIdentification.CodeSmell) (≥3 hasIdentification ∩ ≤5 hasIdentification)) ∪ ((∃hasRemoval.CodeSmell) (≥3 hasRemoval ∩ ≤5 hasRemoval))) WeakCodeSmell≡((∃hasImpact.CodeSmell) ∩ (2≥ hasImpact)) ∪ ((∃hasIdentification.CodeSmell) ∩ (2≥ hasIdentification)) ∪ ((∃hasRemoval.CodeSmell) ∩(2≥ hasRemoval))
Role links
Ans(antipattern cause problems show the symptoms by code smells) ≡∃causeProblems.(software-chronicalProblems∪BadSolution).haveSymptoms ∩ showSyptoms.sourcecodeProblems anti-patterns ⊃ design-patterns design-patterns≡O-O design patterns ∪ GOF patterns ∪ micro-architecture and system patterns ∪ concurrency patterns ∪ Process Patterns ∪ Anti-patterns Anti-patterns ≡ Organizational anti-patterns ∪ Project management anti-patterns ∪ Team-management ∪ Analysis ∪ general design ∪ O-O design ∪ programming ∪ methodological ∪ Configuration
ABox (Concept Assertions)
CodeSmell(LongMethod), CodeSmell(LargeClass), CodeSmell(PrimitiveObsession), CodeSmell(LongParameterList), CodeSmell(DataClumps), CodeSmell(SwitchStatemetns), CodeSmell(TemporaryField), CodeSmell(RefusedBequest), CodeSmell(AlternativeClasseswithDifferentInterfaces), CodeSmell(ParallelInheritanceHierarchies), CodeSmell(LazyClass), CodeSmell(DataClass), CodeSmell(DuplicateCode), CodeSmell(SpeculativeGenerality), CodeSmell(MessageChains), CodeSmell(MiddleMan), CodeSmell(FeatureEnvy), CodeSmell(InappropriateIntimacy),

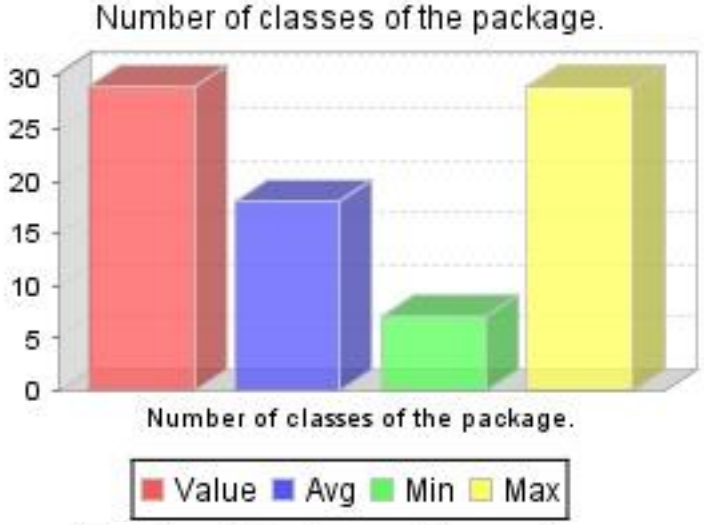
CodeSmell(DivergentChange), CodeSmell(ShotgunSurgery), CodeSmell(IncompleteLibraryClass), CodeSmell(Comments) AntiPattern(Blob), AntiPattern(StovepipeSystem), AntiPattern(GasFactory), AntiPattern(LavaFlow), AntiPattern(AmbiguousViewpoint), AntiPattern(FunctionalDecomposition), AntiPattern(Poltergeists), AntiPattern(BoatAnchor), AntiPattern(GoldenHammer), AntiPattern(SpaghettiCode), AntiPattern(InputKludge), AntiPattern(CutAndPaste), AntiPattern(StoveppipeEnterpise), AntiPattern(SwissAmryKnife) Refactoring(ExtractClass), Refactoring(InlineClass), Refactoring(ForeignMethod), Refactoring(RenameMethod), Refactoring(ExtractMethod), Refactoring(PullUpMethod), Refactoring(MoveMethod), Refactoring(MoveField), Refactoring(CollapseHierachy), Refactoring(DecomposeConditional), Refactoring(ReplaceParameterWithExplicitMethods), Refactoring(IntroduceNullObject), Refactoring(ReplaceConditionalWithPolymorphism) hasSolution(LargeClass, ExtractClass)
ABox (Role Assertions)
hasSymptoms, showSymptoms, hasDesignProblems, hasSourcecodeProblems, causeProblems, hasRefactorings, hasImpact, hasRemoval, hasIdentification, hasProblem, hasContext, hasConsequences, hasRoot-Cause, hasSolution

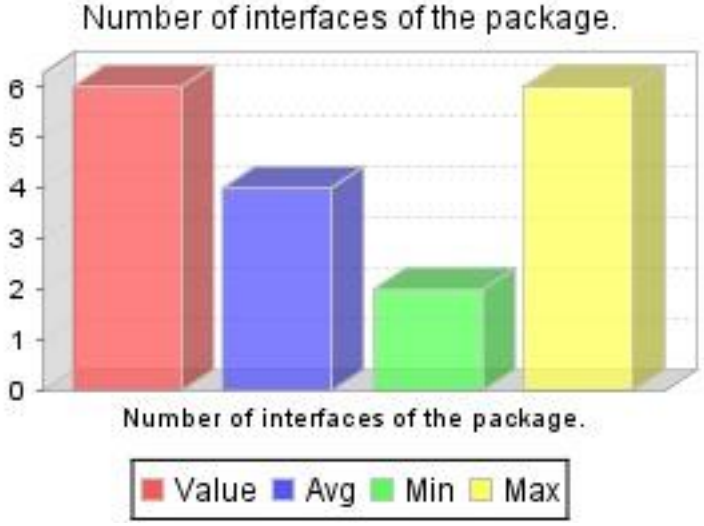
Appendix E: Sample Tests From Metric-based Tools Such As Check Style, PMD, and Analyst4j



Metrics


Number of files of the package.
35
30
25
20
15
10
5
0
Number of files of the package.
Value
Avg
Min
Max


Number of classes of the package.
30
25
20
15
10
5
0
Number of classes of the package.
Value
Avg
Min
Max

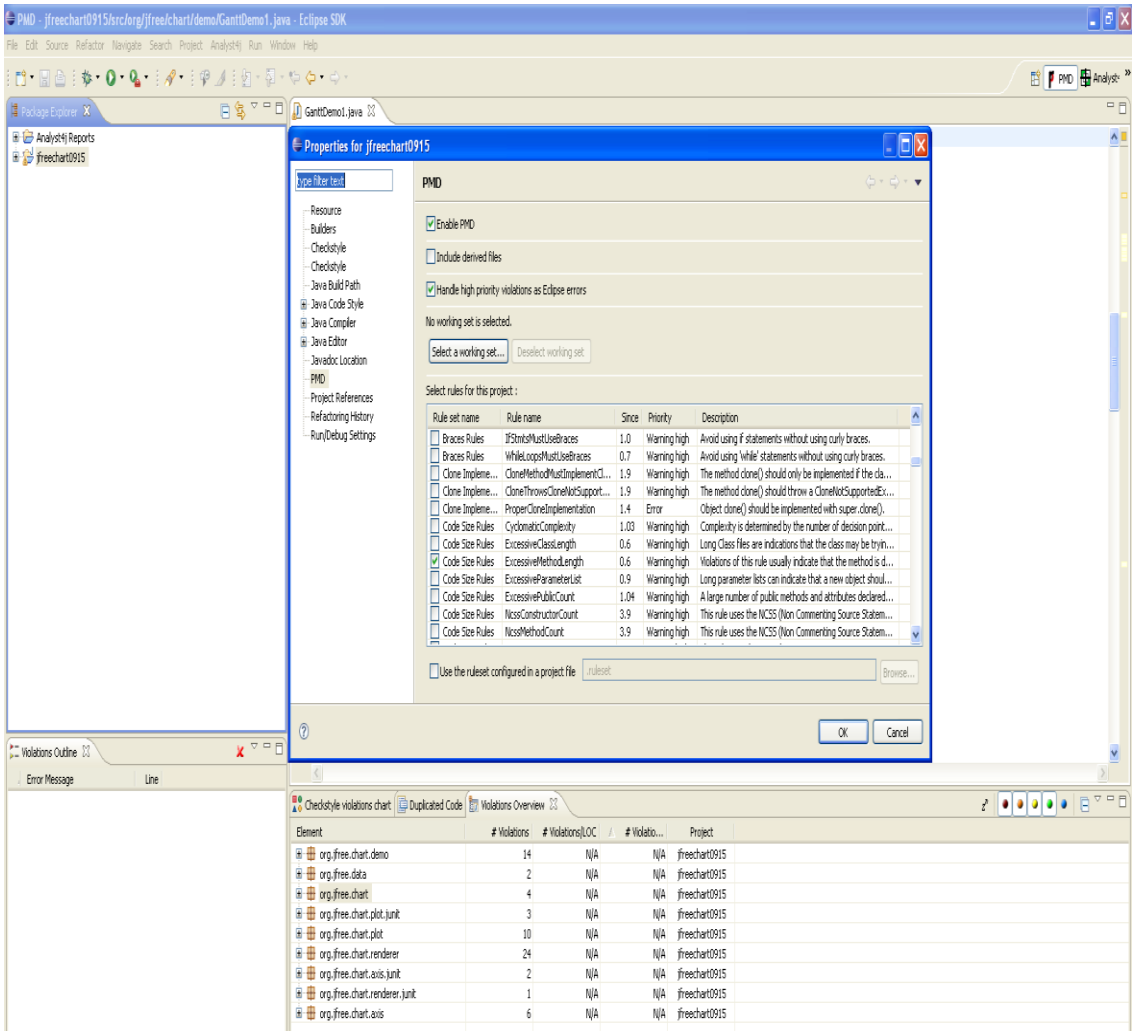

Number of interfaces of the package.
6
5
4
3
2
1
0
Number of interfaces of the package.
Value
Avg
Min
Max

Metrics - Peer Comparison

Metric	Value	Avg	Min	Max
Number of files of the package.	34.0	21.5	9.0	34.0
Number of classes of the package.	29.0	18.0	7.0	29.0
Number of interfaces of the package.	6.0	4.0	2.0	6.0
Number of commented lines of the package.	5534.0	3210.5	887.0	5534.0
Number of lines of code of the package.	5133.0	2878.0	623.0	5133.0
Average cyclomatic complexity of method in the package.	2.17	1.87	1.57	2.17
Average number of anonymous classes of the method in the package.	0.0	0.0	0.0	0.0
Average inner classes of a class in the package.	0.0	0.0	0.0	0.0
Average number of lines of code of a class in the package.	136.0	96.72	57.44	136.0
Average weighted method of a class in the package.	12.86	10.26	7.67	12.86
Average weighted method complexity of a class in the package.	33.69	24.56	15.43	33.69
Average response for class in the package.	56.0	42.5	29.0	56.0
Average lack of cohesion of methods of a class in the	0.7	0.61	0.51	0.7

package.				
Average coupling between objects of a class in the package.	10.62	9.1	7.57	10.62
Average inheritence depth of a class in the package.	1.37	1.35	1.33	1.37
Average halstead effort of a file in the package.	-	-	-	-
Average halstead volume of the package.	3696.94	2757.92	1818.89	-
Average maintainability index of a file in the package.	109.75	112.53	109.75	115.31
Average number of children of a class in the package.	0.34	0.67	0.34	1.0
Average number of lines of code of a file in the package.	150.97	110.1	69.22	150.97
Average number of conditional statements of the method in the package.	0.87	0.56	0.26	0.87
Average number of statements of the method in the package.	5.19	4.4	3.61	5.19
Average number of unused parameters of a method in the package.	0.06	0.03	0.0	0.06
Average number of unused variables of a method in the package.	0.0	0.0	0.0	0.0
Average essential complexity of a method in the package.	1.33	1.43	1.33	1.54

Average number of recursive calls of the method in the package.	0.0	0.0	0.0	0.0
Percentage of comments of the package.	52.01	55.38	52.01	58.74
Dependency Inversion Principle of a package.	45.71	61.75	45.71	77.78
Instability of a package.	0.0	0.0	0.0	0.0
Abstractness of a package.	0.29	0.25	0.22	0.29



Appendix F: Examples of OWL for Code Smells

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns="http://www.wavcis.lsu.edu/codesmell#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xml:base="http://www.wavcis.lsu.edu/codesmell">
   <owl:Ontology rdf:about="">
      <dc:source>version 2009-10-01</dc:source>
      <dc:title>Code Smell </dc:title>
      <dc:contributor>Yixin Luo</dc:contributor>
      <dc:description>Transfer code smells from ascii to OWL
 More                                                    information:
http://www.wavcis.lsu.edu/codesmellhttp://www.wavcis.lsu.edu/codesmell</dc:descrip
tion>
      <dc:date>2009-09-30T02:54:00</dc:date>
      <dc:subject>Parameter</dc:subject>
      <dc:creator>Luis Bermudez MMI</dc:creator>
   </owl:Ontology>
   <owl:Class rdf:ID="Solutions"/>
   <owl:Class rdf:ID="Name"/>
   <owl:Class rdf:ID="Detection"/>
   <owl:Class rdf:ID="subClassOf"/>
   <owl:Class rdf:about="#">
      <rdfs:label></rdfs:label>
   </owl:Class>
   <owl:Class rdf:ID="type"/>
   <owl:ObjectProperty rdf:ID="isNameOf">
      <rdfs:domain rdf:resource="#Name"/>
      <rdfs:range rdf:resource="#"/>
      <owl:inverseOf>
         <owl:ObjectProperty rdf:ID="hasName"/>
```

```
  </owl:inverseOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="issubClassOfOf">
  <rdfs:range rdf:resource="#"/>
  <rdfs:domain rdf:resource="#subClassOf"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="hassubClassOf"/>
  </owl:inverseOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasSolutions">
  <rdfs:range rdf:resource="#Solutions"/>
  <rdfs:domain rdf:resource="#"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="istypeOf">
  <rdfs:domain rdf:resource="#type"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="hastype"/>
  </owl:inverseOf>
  <rdfs:range rdf:resource="#"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hastype">
  <rdfs:range rdf:resource="#type"/>
  <rdfs:domain rdf:resource="#"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="isDetectionOf">
  <rdfs:range rdf:resource="#"/>
  <rdfs:domain rdf:resource="#Detection"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="hasDetection"/>
  </owl:inverseOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasDetection">
  <rdfs:range rdf:resource="#Detection"/>
  <rdfs:domain rdf:resource="#"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hassubClassOf">
```

<rdfs:domain rdf:resource="#"/>
<rdfs:range rdf:resource="#subClassOf"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="isSolutionsOf">
<owl:inverseOf rdf:resource="#hasSolutions"/>
<rdfs:range rdf:resource="#"/>
<rdfs:domain rdf:resource="#Solutions"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasName">
<rdfs:domain rdf:resource="#"/>
<rdfs:range rdf:resource="#Name"/>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="symptoms">
<rdfs:label>Symptoms</rdfs:label>
<rdfs:domain rdf:resource="#"/>
</owl:DatatypeProperty>
<Detection
rdf:ID="Measuring_the_number_of_fields_and_methods_in_conjunction_with_cyclomatic_complexity">
<rdfs:label>Measuring the number of fields and methods in conjunction with cyclomatic complexity</rdfs:label>
<isDetectionOf>
<rdf:Description rdf:ID="A_class_having_little_functions">
<hasSolutions>
<Solutions rdf:ID="Collapse_Hierarchy_and_Inline_class">
<isSolutionsOf rdf:resource="#A_class_having_little_functions"/>
<rdfs:label>Collapse Hierarchy and Inline class</rdfs:label>
</Solutions>
</hasSolutions>
<symptoms>A class having little functions</symptoms>
<hasDetection
rdf:resource="#Measuring_the_number_of_fields_and_methods_in_conjunction_with_cyclomatic_complexity"/>
<hastype>
<type rdf:ID="Class">
<istypeOf rdf:resource="#A_class_having_little_functions"/>

```
            <istypeOf>
              <rdf:Description rdf:ID="Too_many_instance_variables_or_methods">
                <hasName>
                  <Name rdf:ID="LargeClass">
                    <rdfs:label>LargeClass</rdfs:label>
                    <isNameOf
rdf:resource="#Too_many_instance_variables_or_methods"/>
                  </Name>
                </hasName>
                <hassubClassOf>
                  <subClassOf rdf:ID="CodeSmell">
                    <issubClassOfOf rdf:resource="#A_class_having_little_functions"/>
                    <issubClassOfOf
rdf:resource="#Too_many_instance_variables_or_methods"/>
                    <rdfs:label>CodeSmell</rdfs:label>
                    <issubClassOfOf>
                      <rdf:Description
rdf:ID="Too_long_method_that_is_difficult_to_understand_and_reuse">
                        <hastype rdf:resource="#Class"/>
                        <hasDetection>
                          <Detection
rdf:ID="Cyclomatic_complexity_polynomial_metrics">
                            <rdfs:label>Cyclomatic complexity (polynomial
metrics)</rdfs:label>
                            <isDetectionOf
rdf:resource="#Too_long_method_that_is_difficult_to_understand_and_reuse"/>
                          </Detection>
                        </hasDetection>
                        <symptoms>Too long method that is difficult to understand and
reuse</symptoms>
                        <hassubClassOf rdf:resource="#CodeSmell"/>
                        <rdfs:label>Too long method that is difficult to understand and
reuse</rdfs:label>
                        <hasSolutions>
                          <Solutions
rdf:ID="Decompose_Conditional_Extract_Method_Replace_Method_with_Method_O
```

bject_and_Replace_Temp_with_Query">
<isSolutionsOf
rdf:resource="#Too_long_method_that_is_difficult_to_understand_and_reuse"/>
<rdfs:label>Decompose Conditional, Extract Method, Replace Method with Method Object, and Replace Temp with Query</rdfs:label>
</Solutions>
</hasSolutions>
<hasName>
<Name rdf:ID="LongMethod">
<rdfs:label>LongMethod</rdfs:label>
<isNameOf
rdf:resource="#Too_long_method_that_is_difficult_to_understand_and_reuse"/>
</Name>
</hasName>
<rdf:type rdf:resource="#"/>
</rdf:Description>
</issubClassOfOf>
<issubClassOfOf>
<rdf:Description rdf:ID="Redundant_code">
<hassubClassOf rdf:resource="#CodeSmell"/>
<symptoms>Redundant code</symptoms>
<hasSolutions>
<Solutions
rdf:ID="Extract_Class_Extract_Method_Form_Template_Method_and_Pull_Up_Meth od">
<isSolutionsOf rdf:resource="#Redundant_code"/>
<rdfs:label>Extract Class, Extract Method, Form Template Method, and Pull Up Method</rdfs:label>
</Solutions>
</hasSolutions>
<rdf:type rdf:resource="#"/>
<hasDetection>
<Detection
rdf:ID="Percentage_of_duplicate_code_lines_in_the_systems">
<isDetectionOf rdf:resource="#Redundant_code"/>
<rdfs:label>Percentage of duplicate code lines in the

systems</rdfs:label>
</Detection>
</hasDetection>
<hastype rdf:resource="#Class"/>
<hasName>
<Name rdf:ID="DuplicateCode">
<isNameOf rdf:resource="#Redundant_code"/>
<rdfs:label>DuplicateCode</rdfs:label>
</Name>
</hasName>
<rdfs:label>Redundant code</rdfs:label>
</rdf:Description>
</issubClassOfOf>
<issubClassOfOf>
<rdf:Description rdf:ID="A_method_with_too_many_parameters_that_is_difficult_to_understand">
<hasSolutions>
<Solutions rdf:ID="Introduce_Parameter_Object_Replace_Method_with_Method_Object_and_Preserve_Whole_Object">
<rdfs:label>Introduce Parameter Object, Replace Method with Method Object, and Preserve Whole Object</rdfs:label>
<isSolutionsOf rdf:resource="#A_method_with_too_many_parameters_that_is_difficult_to_understand"/>
</Solutions>
</hasSolutions>
<hastype rdf:resource="#Class"/>
<hasDetection>
<Detection rdf:ID="Count_the_number_of_parameters">
<isDetectionOf rdf:resource="#A_method_with_too_many_parameters_that_is_difficult_to_understand"/>
<rdfs:label>Count the number of parameters</rdfs:label>
</Detection>
</hasDetection>

<hasName>
<Name rdf:ID="LongParameterList">
<rdfs:label>LongParameterList</rdfs:label>
<isNameOf
rdf:resource="#A_method_with_too_many_parameters_that_is_difficult_to_understan
d"/>
</Name>
</hasName>
<symptoms>A method with too many parameters that is difficult
to understand</symptoms>
<rdfs:label>A method with too many parameters that is difficult
to understand</rdfs:label>
<rdf:type rdf:resource="#"/>
<hassubClassOf rdf:resource="#CodeSmell"/>
</rdf:Description>
</issubClassOfOf>
</subClassOf>
</hassubClassOf>
<rdf:type rdf:resource="#"/>
<symptoms>Too many instance variables or methods</symptoms>
<rdfs:label>Too many instance variables or methods</rdfs:label>
<hasDetection>
<Detection
rdf:ID="Lack_of_Cohesion_Methods_or_measuring_class_size">
<rdfs:label>Lack of Cohesion Methods or measuring class
size</rdfs:label>
<isDetectionOf
rdf:resource="#Too_many_instance_variables_or_methods"/>
</Detection>
</hasDetection>
<hasSolutions>
<Solutions
rdf:ID="Extract_Class_Extract_Interface_Extract_Subclass_and_Introduce_Foreign_M
ethod">
<isSolutionsOf
rdf:resource="#Too_many_instance_variables_or_methods"/>

<rdfs:label>Extract Class, Extract Interface, Extract Subclass, and Introduce Foreign Method</rdfs:label>
</Solutions>
</hasSolutions>
<hastype rdf:resource="#Class"/>
</rdf:Description>
</istypeOf>
<istypeOf rdf:resource="#Redundant_code"/>
<istypeOf rdf:resource="#Too_long_method_that_is_difficult_to_understand_and_reuse"/>
<rdfs:label>Class</rdfs:label>
<istypeOf rdf:resource="#A_method_with_too_many_parameters_that_is_difficult_to_understand"/>
</type>
</hastype>
<hasName>
<Name rdf:ID="LazyClass">
<rdfs:label>LazyClass</rdfs:label>
<isNameOf rdf:resource="#A_class_having_little_functions"/>
</Name>
</hasName>
<rdf:type rdf:resource="#"/>
<rdfs:label>A class having little functions</rdfs:label>
<hassubClassOf rdf:resource="#CodeSmell"/>
</rdf:Description>
</isDetectionOf>
</Detection>
</rdf:RDF>

Vita

Yixin Luo is a doctoral candidate in the Department of Computer Science at Louisiana State University. He received his Bachelor of Science degree in materials engineering from Wuhan University of Technology. He received his Master of Science degree in physics and computer science from Southern University at Baton Rouge in 2001. His research interests are in software security, software development, high performance computing, and software ontology. He is a student member of ACM and SAGE and a member of AGU.

He worked as IT analyst for four years at Coastal Studies Institute (CSI) of Louisiana State University. He maintained and developed software products for an ocean observation system at Gulf of Mexico. He is currently employed by Center for Computation and Technology (CCT) of LSU, and his work is related to Teragrid development and maintenance. The degree of Doctor of Philosophy will be awarded at the May 2010 commencement at Louisiana State University.