Automated Testing For Anti-Patterns For Software Quality Analysing Using AOP

Dissertation submitted in partial fulfillment of requirement for award of degree of

Master of Technology
in
Computer Science and Applications

Submitted By:

Himanshu Bharti
(Roll No.601203009)

Supervised By:

Vineeta Bassi

sSMCA, Thapar University, Patiala

SCHOOL OF MATHEMATICS AND COMPUTER APPLICATIONS

THAPAR UNIVERSITY

PATIALA – 147004

July 2014

CERTIFICATE

I hereby certify that the work which is presented in the dissertation entitled, "**Automated Testing For Anti-Patterns For Software Quality Analysing Using AOP**", in partial fulfillment of the requirements for the award of degree of the **Master of Technology in Computer Science and Applications,** submitted in **School of Mathematics and Computer Applications, Thapar University, Patiala**, is an authentic record of my own work carried out under the supervision of Vineeta Bassi and refers others researcher's work which are duly listed in the reference section. The matter presented in this dissertation has not been submitted for award of any other degree of this or any other university.

(Himanshu Bharti)

601203009

This is to certify that above statement made by the candidate is correct and true to the best of my knowledge.

(Vineeta Bassi)

SMCA

Countersigned by:

(Dr. Rajesh Kumar)

Head of department

SMCA, Thapar University

Patiala

(Dr. S. K Mohapatra)

Dean (Academic Affairs)

Thapar University

Patiala

ACKNOWLEDGEMENT

First of all I would like to thank the Almighty, who has always guided me to work on the right path of the life.This work would not have been possible without the encouragement and able guidance of my supervisor **Vineeta Bassi**. I thank my supervisor for her time, patience, discussions and valuable comments. Her enthusiasm and optimism made this experience both rewarding and enjoyable. I am equally grateful to **Dr. Rajesh Kumar**, Associate Professor and Head, School of Mathematics and Computer Application, for motivation and inspiration that triggered me for the dissertation work.

I will be failing in my duty if I don't express my gratitude to **Dr. S. K. Mohapatra**, Senior Professor and Dean of Academic Affairs of University, for making provisions of infrastructure such as library facilities, immensely useful for the learners to equip themselves with the latest field.

I am also thankful to the entire faculty and staff members of School of Mathematics and Computer Applications Department for their direct-indirect help, cooperation, love and affection, which made my stay at Thapar University memorable.

Last but not least, I would like to thank my parents for their wonderful love and encouragement, without their blessings none of this would have been possible. I would also like to thank my close friends for their constant support.

ABSTRACT

This dissertation is based on the automatic detection of anti-patterns in the given code. Anti-patterns are wrong design patterns that deteriorate the quality of the software. Detection and removal of anti-patterns can enhance the software quality and will decrease the chances of the bug in the software. If we may automate the process of anti-patterns detection in the code it will reduce the time and cost of anti-pattern detection. So our goal is to automate the process of anti-pattern detection.

In the dissertation we are using an Aspect oriented based approach for detection of the anti-patterns in the code. We will use aspect which is a java enhancement of implement AOP concept. AOP separate cross cutting concerns from the main goal. That is why we are using AOP this will separate testing module from main program to be tested.

Program to be tested will be in java. First we will create UML diagram for the program, UML diagram will be created using ArgoUML. Generated UML will be converted into XMI, this XMI will be used as input for aspectJ testing module. XMI will have important information about the code to be tested and this will be used by asepctJ to get input that will be used for testing of anti-patterns. After getting the required information from the XMI, our aspectJ testing module will analyze the main code and find the anti-pattern in it. There are different type of anti-patterns, we will try to find some most common anti-patterns in the code. In the end results i.e. anti-patterns in the code will be displayed. These results can be used as reference to refactor the main code to remove anti-patterns.

LIST OF CONTENTS PAGE NO.

LIST OF FIGURES PAGE NO.

LIST OF TABLES PAGE NO.

LIST OF ABBREVIATIONS

AOP	Aspect-Oriented Programming
ASG	Abstract Syntax Graph
AST	Abstract Syntax Tree
CTL	Computational Tree Logic
LQN	Layered Queue Networks
LTL	Linear Temporal Logic
OMG	Object Management Group
SBSE	Search-Based Software Engineering
SCM	Software Configuration Management
SVG	Scalable Vector Graphics
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XPG	Extended Positional Grammer

CHAPTER 1

INTRODUCTION

1.1 ANTI-PATTERN

The study of Anti-patterns is an important research activity. The presence of ‘good’ patterns in a successful system is not enough; all also must show that some patterns are absent in unsuccessful systems. Software refactoring is a form of code modification, used to improve the software structure in support of subsequent extension and long-term maintenance. In most cases, the goal is to transform code without impacting correctness.

An anti-pattern is a frequently used, but largely ineffective solution to a problem. The term was originally used to refer to a pattern gone wrong. Just as a viable pattern describes the way from a problem to a valid solution, an anti-pattern describes the way from a problem to a poor solution. Furthermore, by adding more difficulties to the ones that originally existed, an anti-pattern may leave you in a worse position than before you started. Several researchers have published data on how to recognize and avoid adopting an anti-pattern, especially in developing computer programming. The term amelioration pattern (to ameliorate means to improve, and especially to improve a bad situation) was coined for a pattern that describes how to go from a bad solution to a better one [4].

"Anti-patterns: Refactoring Software, Architectures, and Programs in Crisis", "Anti-patterns and Patterns in Software Configuration Management(SCM)" and "Anti-Patterns in Program Management". Anti-patterns represent the latest concept in a series of revolutionary changes in computer science and software engineering thinking. As we approach the 50-year mark in developing programmable digital systems, the software industry has yet to resolve some fundamental problems in how humans translate business concepts into software applications. The emergence of design patterns has provided the most effective form of software guidance yet available, and the whole patterns movement has gone a long way in codifying a concise terminology for conveying sophisticated computer science thinking.

- Anti-patterns are negative solutions that present more problems than they address.
- Anti-patterns are a natural extension to design patterns.

- Anti-patterns bridge the gap between architectural concepts and real-world implementation.
- Understanding anti-patterns provides the knowledge to prevent or recover from them [14].

An Anti-pattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences. The Anti-pattern may be the result of a manager or developer not knowing any better, not having sufficient knowledge or experience in solving a particular type of problem, or having applied a perfectly good pattern in the wrong context. Anti-patterns provide real-world experience in recognizing recurring problems in the software industry and provide a detailed remedy for the most common predicaments. Anti-patterns highlight the most common problems that face the software industry and provide the tools to enable you to recognize these problems and to determine their underlying causes. Furthermore, Anti-patterns present a detailed plan for reversing these underlying causes and implementing productive solutions. Anti-patterns effectively describe the measures that can be taken at several levels to improve the developing of applications, the designing of software systems, and the effective management of software programs.

1.1.1 TYPES OF ANTI-PATTERN

a) Software Development Anti-Patterns

A key goal of development Anti-patterns is to describe useful forms of software refactoring. Software refactoring is a form of code modification, used to improve the software structure in support of subsequent extension and long-term maintenance. In most cases, the goal is to transform code without impacting correctness.

b) Software Architecture Anti-Patterns

Architecture anti-patterns focus on the system-level and enterprise-level structure of applications and components. Although the engineering discipline of software architecture is relatively immature, what has been determined repeatedly by software research and experience is the overarching importance of architecture in software.

c) Software Program Management Anti-Patterns

In the modern engineering profession, more than half of the job involves human communication and resolving people issues. The management Anti-patterns identify some of the key scenarios in which these issues are destructive to software processes [5].

1.2 REFACTORING

A large part of refactoring is composing methods to package code properly. Almost all the time the problems come from methods that are too long. Long methods are troublesome because they often contain lots of information, which gets buried by the complex logic that usually gets dragged in.

"*A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior"*

"Is refactoring just cleaning up code?" In a way the answer is yes, but I think refactoring goes further because it provides a technique for cleaning up code in a more efficient and controlled manner. Since I've been using refactoring, I've noticed that I clean code far more effectively than I did before. This is because I know which refactoring to use, I know how to use them in a manner that minimizes bugs, and I test at every possible opportunity [11].

1.2.1 BENEFITS OF REFACTORING

- Refactoring improves the design of software.
- Refactoring makes software easier to understand.
- Refactoring helps program faster.

1.3 UNIFIED MODELING LANGUAGE

Unified Modeling Language (UML) makes it possible to describe systems with words and pictures. It can be used to model a variety of systems: software systems, business systems, or any other system. Especially notable are the various graphical charts—use case diagrams with their stick or the widely used class diagrams. While these diagrams aren't fundamentally new, the worldwide unification of modeling languages is new with UML, which was standardized by the Object Management Group(OMG), an international association that promotes open standards for object-oriented applications [21]

UML describe it almost in its entirety. However, our experience has shown that in reality there is often a lack of time, previous knowledge, or motivation to deal with the topic with the necessary intensity. In these cases, the material can't be completely understood and put into action. We put together those parts of UML whose application has proven to be practical. With a little effort, anybody should be able to make use of UML. There are several reasons to use UML as a modeling language:

- The unification of terminology and the standardization of notation lead to a significant easing of communication for all parties involved. It facilitates the exchange of models between different departments or companies. Moreover, it eases the transfer of programs between program teams or program team members.
- UML grows as the requirements for modeling grow. Because UML is a powerful modeling language, you can start with the development of simple models or model complex systems in great detail. If the basic functionality of UML is not sufficient, you can extend it through the use of stereotypes.
- UML builds upon widely used and proven approaches.
- UML was not devised in an ivory tower but was developed mainly from real world problems and existing modeling languages. This guarantees usability and real-life functionality.
- UML is widely supported.
- UML-based bids for software systems can be compared much more easily [8].

1.4 WHAT IS A MODEL?

Models are often built in the context of business and IT systems in order to better understand existing or future systems. However, a model never fully corresponds to reality. Modeling always means emphasizing and omitting: emphasizing essential details and omitting irrelevant ones. But what is essential and what is irrelevant? There is no universal answer to this question. Rather, the answer depends on *what* the goals of the model are and *who* is viewing or reading it. In figure 1.1 Different views are formed of the objects under consideration. These views are interconnected in many ways. Generally, if one view is changed, all other views have to be adjusted as well [7].

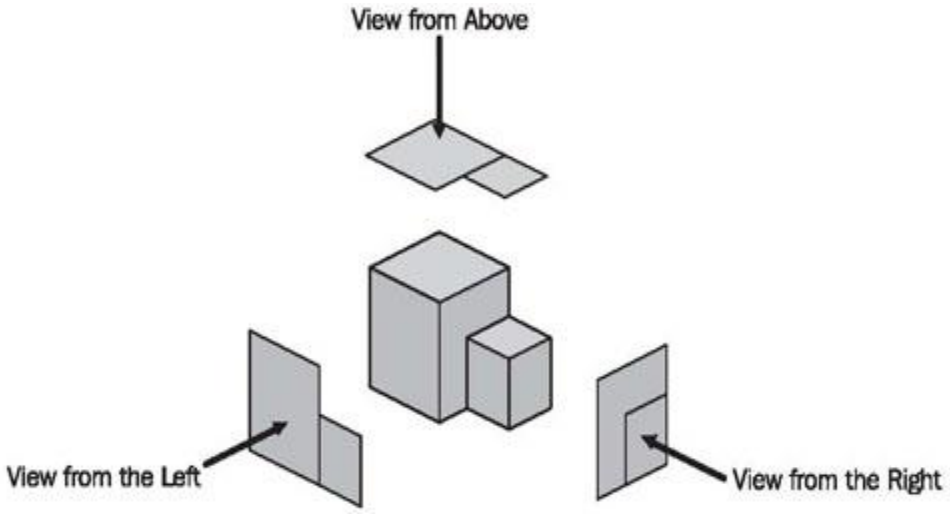

Figure1.1 Different views of an object [7]

1.5 XMI

The XML Metadata Interchange(XMI) is an Object Management Group(OMG) standard for exchanging metadata information via Extensible Markup Language (XML) [11].

1.6 ARGO UML

Argo UML is an UML diagramming application written in Java and released under the open source Eclipse Public License. By virtue of being a Java application, it is available on any platform supported by Java [24].

1.6.1 HISTORY OF ARGO UML

Argo UML was originally developed at UV Irvine by Jason E. Robbins, leading to his Ph.D. It is now an open source program hosted by. The Argo UML program now includes more than 19,000 registered users and over 150 developers.

In 2003, Argo UML won the Software Development Magazine's annual Readers' Choice Award in the “Design and Analysis Tools” category.

Argo UML development has suffered from lack of manpower. For example,undo has been a perpetually requested feature since 2003 but has not been implemented yet [7].

1.6.2 FEATURE OF ARGO UML

- All 9 UML 1.4 diagrams are supported.
- Closely follows the UML standard.
- Platform independent – Java 1.5+.
- Click and Go! Java Web Start (no setup required, starts from your web browser).
- Standard UML 1.4 Meta model.
- XMI support.
- Export diagrams as GIF, PNG, PS, EPS, PGML and SVG.
- Available in ten languages: EN, EN-GB, DE, ES, IT, RU, FR, NB, PT, ZH.
- Advanced diagram editing and zoom.
- Built-in design critics provide unobtrusive review of design and suggestions for improvements.
- Extensible modules interface.
- OCL support.
- Forward engineering (code generation supports C++ and C#, Java, PHP 4, PHP 5, Ruby and, with less mature modules, Ada, Delphi and SQL).
- Reverse engineering / JAR/class file import [31].

1.7 ASPECTJ

AspectJ, a compatible extension to the Java programming language, is one implementation of AOP. AspectJ is very widely used in a lot of Java frameworks (like Spring), but still most developers do not know AspectJ. Developers often think that AspectJ is difficult to learn or it makes your code complex, and they decide not to learn this very powerful and useful technology [21].

1.8 ASPECT-ORIENTED PROGRAMMING

Aspect Oriented Programming (AOP) is a programming paradigm which focuses on modularizing system-level concerns (like logging, transaction management, security,

performance monitoring, etc.) in the applications. In AOP language, these system-level concerns are called "crosscutting concerns" because they crosscut all the layers of the application.

Many software developers are attracted to the idea of aspect-oriented programming (AOP) but unsure about how to begin using the technology. They recognize the concept of crosscutting concerns, and know that they have had problems with the implementation of such concerns in the past. But there are many questions about how to adopt AOP into the development process.

AOP is a new technology for separating crosscutting concerns into single units called aspects. An aspect is a modular unit of crosscutting implementation. It encapsulates behaviors that affect multiple classes into reusable modules. With AOP, we start by implementing our program using our OO language (for example, Java), and then we deal separately with crosscutting concerns in our code by implementing aspects. Finally, both the code and aspects are combined into a final executable form using an aspect weaver. As a result, a single aspect can contribute to the implementation of a number of methods, modules, or objects, increasing both reusability and maintainability of the code. Figure 1.2 explains the weaving process. It should be noted that the original code doesn't need to know about any functionality the aspect has added; it needs only to be recompiled without the aspect to regain the original functionality [9].

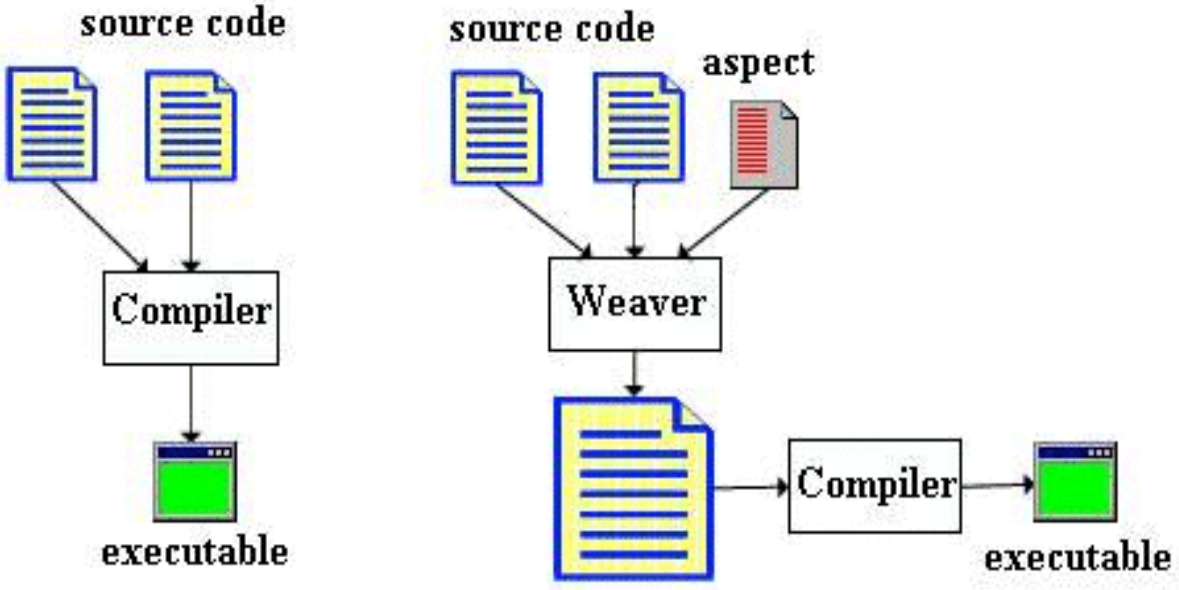

Figure1.2 Aspect weaver [21]

In that way, AOP complements object-oriented programming, not replacing it, by facilitating another type of modularity that pulls together the widespread implementation of a crosscutting concern into a single unit. These units are termed aspects, hence the name aspect oriented programming. Aspect Oriented Programming (AOP) is a promising new technology for separating crosscutting concerns that are usually hard to do in object-oriented programming. AOP is a concept, so it is not bound to a specific programming language. In fact, it can help with the shortcomings of all languages (not only OO languages) that use single, hierarchical decomposition. AOP has been implemented in different languages (for example, C++, Smalltalk, C#, C, and Java).

Of course, the language that gains a great interest of the research community is the Java language. The following is a list of tools that support AOP with Java:

- AspectJ
- AspectWerkz
- Hyper/J
- JAC
- JMangler
- MixJuice
- PROSE
- ArchJava

AspectJ, created at Xerox PARC, was proposed as an extension of the Java language for AOP [15].

CHAPTER 2

LITERATURE REVIEW

2.1 DETECTING DESIGN PATTERNS

Bieman et al. [5] examined intentional patterns in software systems. These were the patterns that the developers intentionally chose and documented. The detection of these patterns is done by performing textual search in the documentation and examining the program code to find pattern candidates. Because the textual search yielded some false positives, a manual inspection of the model was used to verify which of the candidate classes actually participate in detected Design Patterns. Manual inspection was a commonly used method to verify the accuracy of an automated or semi-automated detection approach.

Costagliola et al. [11] used UML Class Diagrams to detect the presence of patterns. The class diagrams has been generated from source or recovered from documentation. These diagrams are stores as a Scalable Vector Graphics (SVG) format. The diagram was parsed and converted to sentences that express the relations of object, called Extended Positional Grammar (XPG). The sentences of the resulting class diagram were transformed in such a way that they can be matched against templates of Design Patterns. The templates describe small class diagrams such as an inheritance relations between multiple objects. Patterns was detected by examining every class object in the class diagram and attempting to match it's inheritance structure with a template description. If a match was found, other rules are also tested for matches because a class can participate in multiple patterns.

2.2 DETECTING ANTI-PATTERNS

Similar approaches has been used to detect Anti-patterns. At the time of proposing approached, only few researches have investigated Anti-patterns. Dhambri et al. [12] examined the Blob and the Functional Decomposition Anti-patterns. These patterns were identified although no further examination was done on their effects. Anti-patterns were also mentioned to be detected by Gall et al. [18] although the Anti-patterns were not the main focus. Van Emden et al. [7] these bad patterns cannot be detected by jCosmo because they were usually on a higher level than metrics or Code Smells.

The DETEX tool was a detection program built on top of the DECOR Smell detection tool. Using domain-specific descriptions based on Code Smells called Design Smells in, Anti-pattern detection rules have been formulated and their detection algorithms can be generated by DECOR. This tool has been used to detect instances of Spaghetti Code, The Blob, Functional Decomposition and Swiss Army Knife Anti-patterns [19]

An alternative to DECOR is an approach using B-Splines. This approach used fuzzy thresholds instead of the crisp 'Yes or No' thresholds used by many tools. In general, this approach outperforms the crisp threshold approach. However, it was sensitive to the size of the training set, meaning that a small training set could severely reduce accuracy.

Some of the Anti-patterns that Stoianov et al. [2] have included in the work are regarded as Code Smells. The Anti-patterns that have been detected are Data Class, Call Super, Constant interface, The Blob, Refused Interface, Yoyo Problem and Poltergeist. Tests on various open-source projects show that some Anti-patterns do occur and can be detected.
Meyer proposed an approach for automatic refactoring of Anti-patterns. This theoretic approach describes how Anti-patterns can be detected by examining the Abstract Syntax Graph (ASG). The patterns were defined as ASG rules that describe their graph layout. These patterns can be recovered by using a graph matching algorithm. Next, transformation rules described using UML can be applied to automatically transform the Anti-patterns into normal non-Anti-pattern classes or even Design Patterns.

2.3 DETECTING BAD SMELLS

Matthew James et al. [25] built a tool that detected Bad Smells by using static analysis. This tool analyzes the Abstract Syntax Tree (AST), calculates metrics and used a rule base to detect Code Smells based on the metrics. The paper presented the results of the detection tool for two of the ten design problems that were detected. Examining the approach and conclusions, it became clear that a metric-based rulebase can be unreliable. The rules that were needed to detect the smells were open to interpretation because the design problems themselves can be described in many ways. Also the thresholds for the rules require tinkering and may not always breliable.The

problem of finding good rules and thresholds may be a threat to validity for works.

Van Emden et [31] developed the jCosmo tool. This tool parsed source code into an abstract model (similar to the Famix meta-model). It used primitive and derived smell aspects (e.g., rulebase-like) to detect the presence of smells. The meta-model was used as a basis to extract primitive smell aspects from, such as "Method M contains a switch statement". These primitive smells were then used to generate derived smell aspects such as "Class C was not use any of the methods offered by its super classes". The jCosmo tool can visualize the code layout and smell locations. The goal of this tool is to help developers assess code quality and help in refactorings. The main difference compared with other detection tools is that jCosmo tried to visualize problems by visualizing the design. Most other tools visualize the problems on a much more technical level, often at a programming instruction level.

In the following the main existing approaches for the automated generation of architectural feedback were surveyed.
In particular, here identified three principal categories of approaches are
(i) anti-pattern based approaches that make use of anti-patterns knowledge to cope with performance issues
(ii) rule-based approaches that define a set of rules to overcome performance problems
(iii) search-based approaches that explore the problem space by examining options to deal with performance flaws. In the context of the search-based process two techniques can be applied: design space exploration and meta heuristic.

2.4 ANTI-PATTERN BASED APPROACHES

The term Anti-pattern appeared for the first time in contrast to the trend of focus on positive and constructive solutions. Differently from patterns, anti-patterns look at the negative features of a software system and describe commonly occurring solutions to problems that generate negative consequences. Anti-patterns have been applied in different domains. For example data-flow anti-patterns help to discover errors in work flows and are formalized through the (Computational Tree Logic) CTL temporal logic. As another example, in anti-patterns help to discover multi

threading problems of Java applications and are specified through the (Linear Temporal Logic) LTL temporal logic [1]

France et al. [12] introduced a UML-based pattern specification technique. Design patterns were defined as models in terms of UML meta-model concepts: a pattern model describes the participants of a pattern and the relations between them in a graphical notation by means of roles, i.e. the properties that a UML model element must have to match the corresponding pattern occurrence.

Cortellessa et al. [10] introduced a first proposal of automated generation of feedback from the software performance analysis, where performance anti-patterns play a key role in the detection of performance flaws. However, this approach considered a restricted set of anti-patterns, and it used informal interpretation matrices as support. Performance scenarios were described (e.g. the throughput is lower than the user requirement, and the response time was greater than the user requirement) and, if needed, some actions to improve such scenarios are outlined. The main limitation of this approach was that the interpretation of performance results was only demanded to the analysis of Layered Queue Networks (LQN) a performance model. Such knowledge was not enriched with the features coming from the software architectural models, thus to hide feasible refactoring actions.

2.5 RULE BASED APPROACHES

Dobrzanski et al. [6] tackled the problem of refactoring UML models. In particular, bad smells were defined as structures that suggest possible problems in the system in terms of functional and non-functional aspects. Refactoring operations were suggested in the presence of bad smells. Rules for refactoring were formally defined, and they took into account the following features:

(i) cross integration of structure and behavior.

(ii) support for component-based development via composite structures.

(iii) integration of action semantics with behavioral constructs. However, no specific performance issue was analyzed, and refactoring was not driven by unfulfilled requirements.

Kavimandan et al. [4] presented an approach to optimize deployment and configuration decisions in the context of distributed, real time, and embedded (DRE) component based systems. Bin packing algorithms have been enhanced, and schedulability analysis have been used to make fine-grained assignments that indicate how components are allocated to different middleware containers, since they are known to impact on the system performance and resource consumption. However, the scope of this approach is limited to deployment and configuration features.

2.6 SEARCH BASED APPROACHES

A wide range of different optimization and search techniques have been introduced in the field of Search-Based Software Engineering (SBSE), i.e a software engineering discipline in which search-based optimization algorithms were used to address problems where a suitable balance between competing and potentially conflicting goals has to be found. Two key ingredients were required:

(i) the representation of the problem

(ii) the definition of a fitness function.

In fact, SBSE usually applied to problems in which there were numerous candidate solutions and where there was a fitness function that can guide the search process to locate reasonably good solutions [33].

A suitable representation of the problem allows to automatically explore the search space for the solutions that best fit the fitness function that drives towards the sequence of the refactoring steps to apply to this system (i.e. altering its architectural structure without altering its semantics).

In the software performance domain both the suitable representation of the problem and the formulation of the fitness function were not trivial tasks, since the performance analysis results were derived from many uncertainties like the workload, the operational profile, etc. that might completely modify the perception of considering candidate solutions as good ones.Some assumptions can be introduced to simplify the problem and some design options can be explicitly defined in advance to constitute the population on which search based optimization algorithms apply. We believe that in the performance domain it was of crucial relevance to find a synergy between the search techniques that involved the definition of a fitness function to

automatically capture what was required from the system, and the anti-patterns that might support such function with the knowledge of bad practices and suggest common solutions, in order to quickly converge towards performance improvements. In fact there was a mutually beneficial relationship between SBSE and predictive models. In particular eleven broad areas of open problems (e.g. balancing functional, nonfunctional properties of predictive models) in SBSE for predictive modeling are discussed, explaining how techniques emerging from the SBSE community may find potentially innovative applications in predictive modeling [25].

CHAPTER 3

PROBLEM STATEMENT

As anti-patterns in the existing code does not have any significant effect on the code, therefore most of the time testing for anti-patterns takes a back seat, as time and efforts involve does not provide any direct benefit. But that doesn't mean a code with anti-patterns should be acceptable.

A code with anti-patterns has poor quality, is hard to maintain and is not easily reusable. Moreover it may increase the risk for bugs in the future, a code with anti-pattern will not stop the code from doing what is expected but it will definitely effect the quality of your code in a negative way. A code with anti-patterns is a working code but a code without anti-patterns is for sure a quality code.

We are proposing a concept that will use aspect oriented model to search for the anti-patterns in the program and will show the anti-patterns in the program. AOP model is used because it allow us to manipulate and use the program without making any change in the source code of the program.

`

CHAPTER 4

MATERIALS & METHOD

4.1 METHOD

We are using XMI file of program, that is to be tested ,which will be based on the UML diagram, XMI provides us the required information about the program to be tested that will be used for anti-patterns testing. Further we are developing automated testing model based on AOP that will search for anti-patterns and will show all the anti-patterns in the code. Is other words AOP testing module is using to show unused data, blob and other anti-pattern problems that effects the quality of the code.

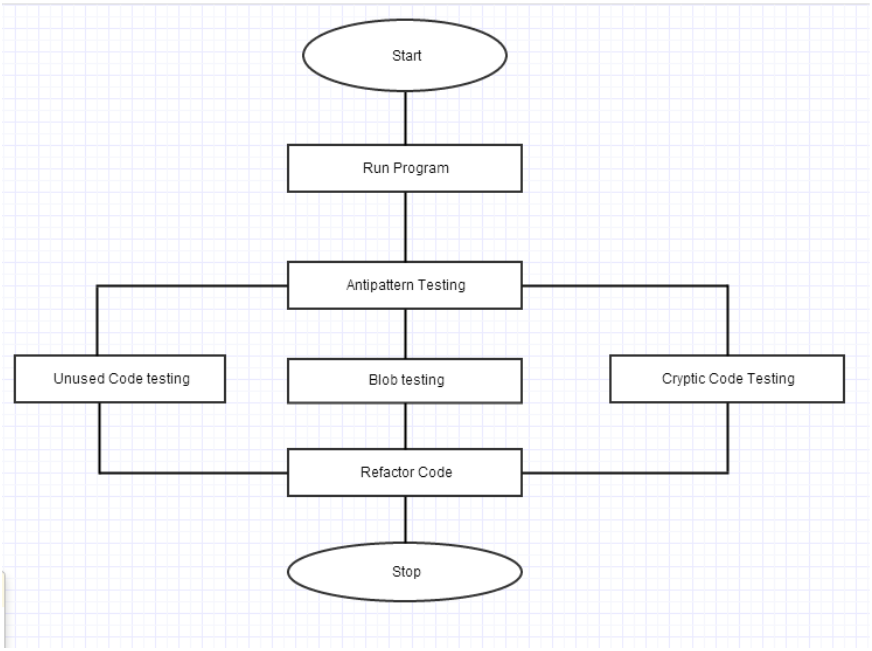

Figure4.1 Flow of anti-pattern detection and refactor [7]

4.2 TESTING PROGRAM DESCRIPTION

In the dissertation we are using XMI file as an input which will help to get the information about classes, it's methods and fields. Information got from XMI file will facilitate our testing process. Further, after getting information from the XMI, we are using this information and AOP script to find the anti-patterns with-in our code. In the dissertation we are testing the running app, here where

comes AOP, AOP allows to manipulate a java class without modifying its code, this feature of AOP make it a suitable for our dissertation where we have to test running code for anti-patterns.

The dissertation starts with creation of UML diagram for the code to be tested. We are using ArgoUml software for UML diagram creation.

4.3 STEPS TO RUN ARGO UML TOOL

a) Interface of argo uml

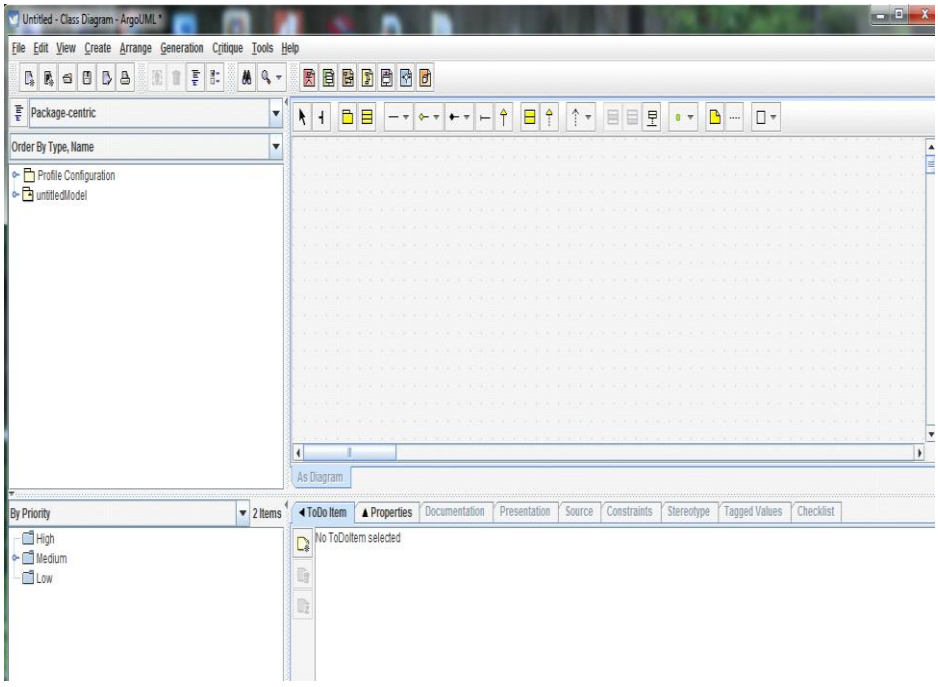

Figure4.2 Interface of argo UML

-

b) Select import sources option

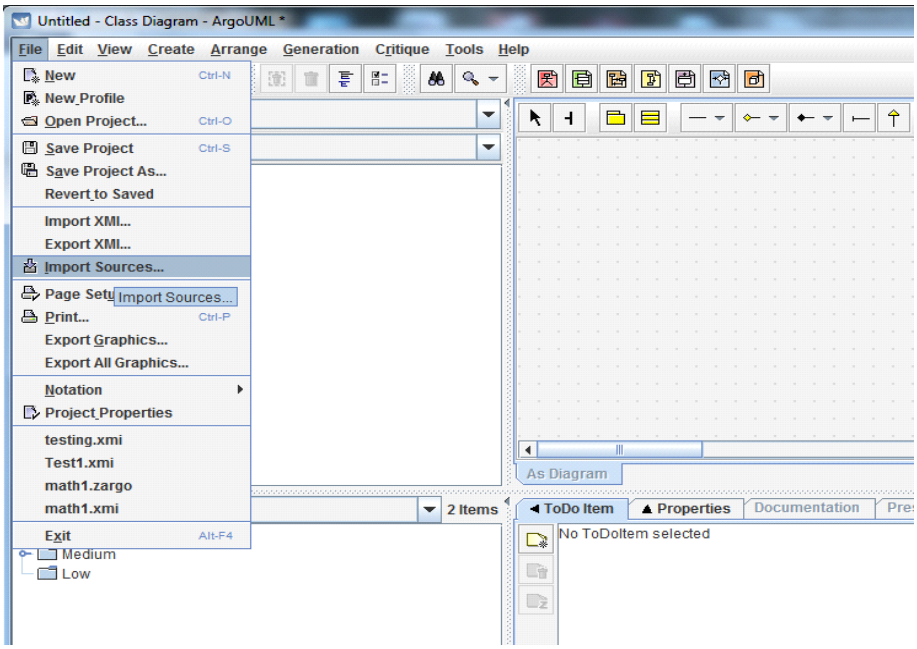

Figure4.3 Import sources in interface of agroUML tool

c) Click on untitled model on top left box

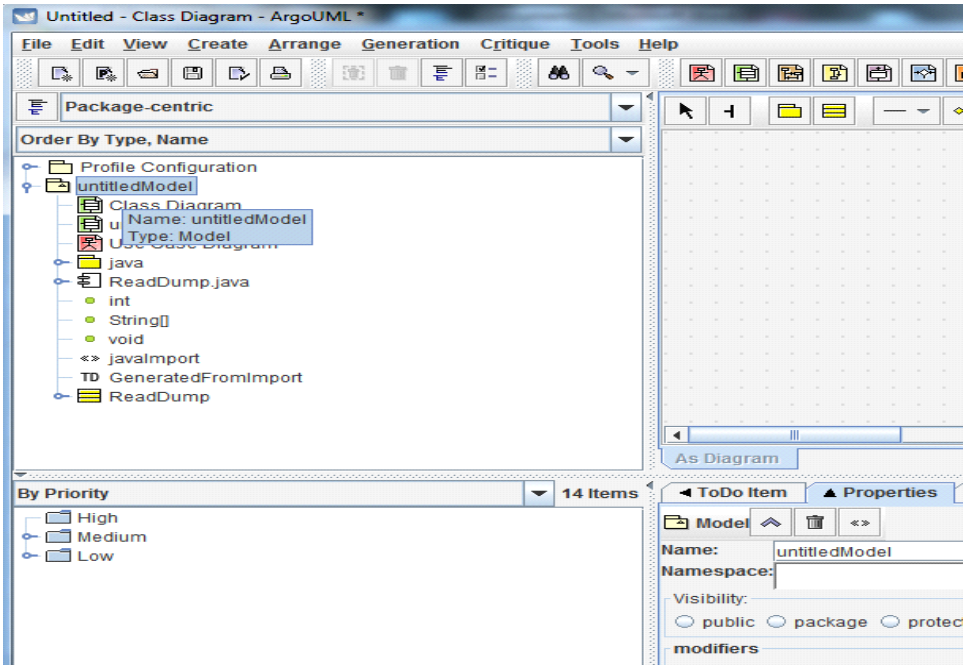

Figure 4.4 Select untitled model in agroUML tool

d) Browse source code file i.e. java file and open the file. UML diagram of the imported file will be generated

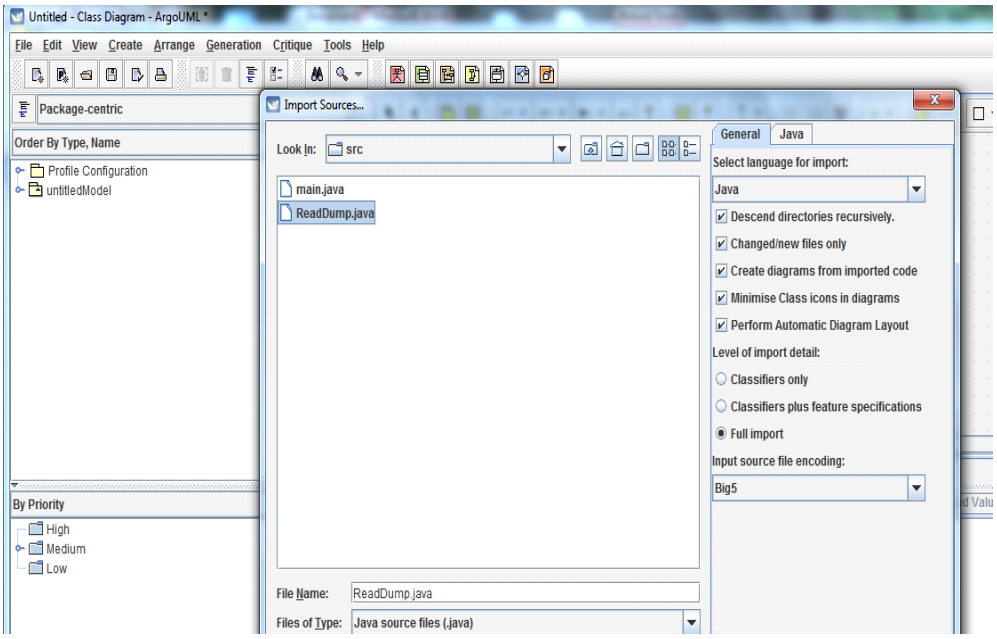

Figure4.5 Import java file in agroUML tool

e) Right click on the option and select Add to Diagram option

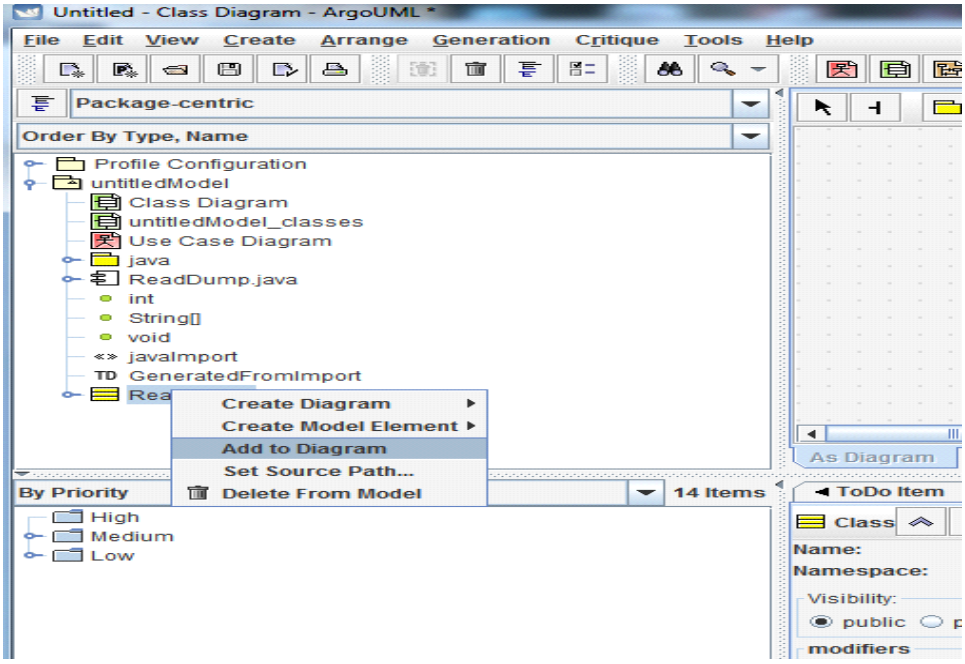

Figure4.6 Select the java file and make the UML diagram

f) Now move the mouse cursor to the right hand side empty space. That is marked with a black cross mark in the snapshot Click left mouse button. This will create a uml diagram of your class in the empty space.

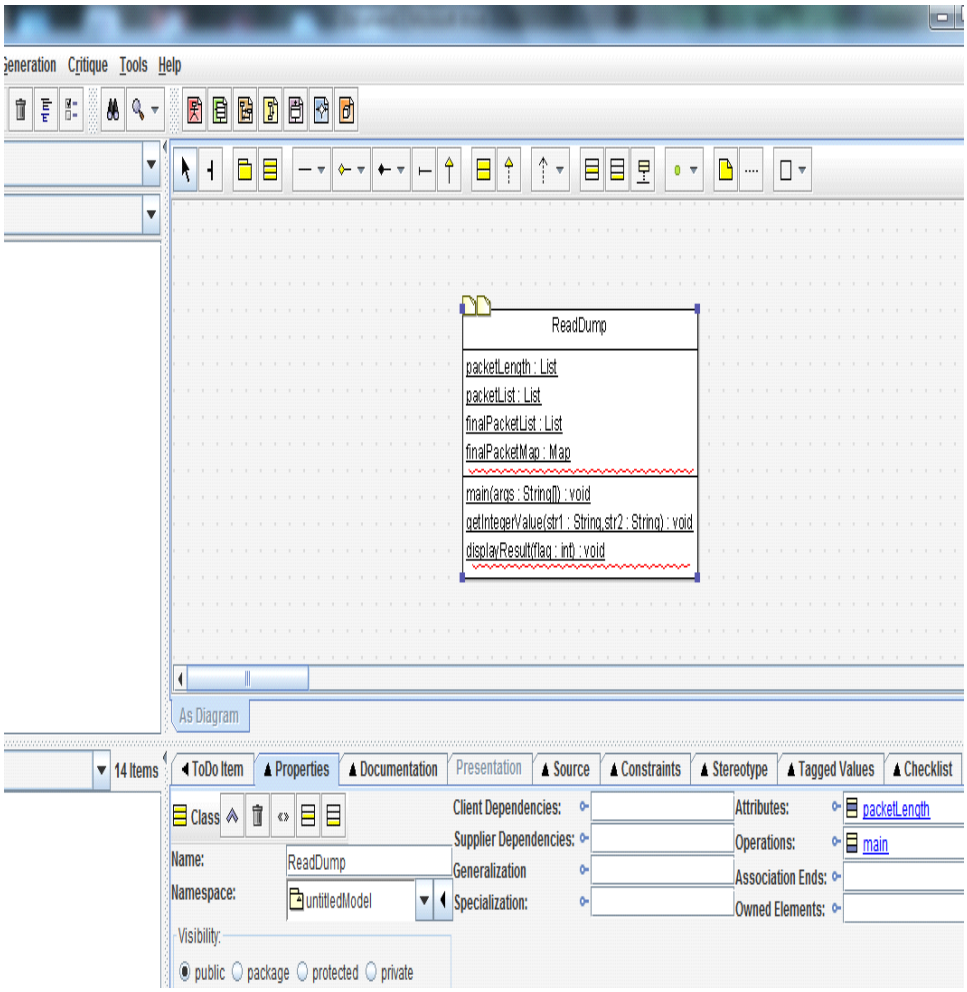

Figure4.7 Click on empty space of tool window

g) After creating uml diagram again select the file menu and select Export XMI option

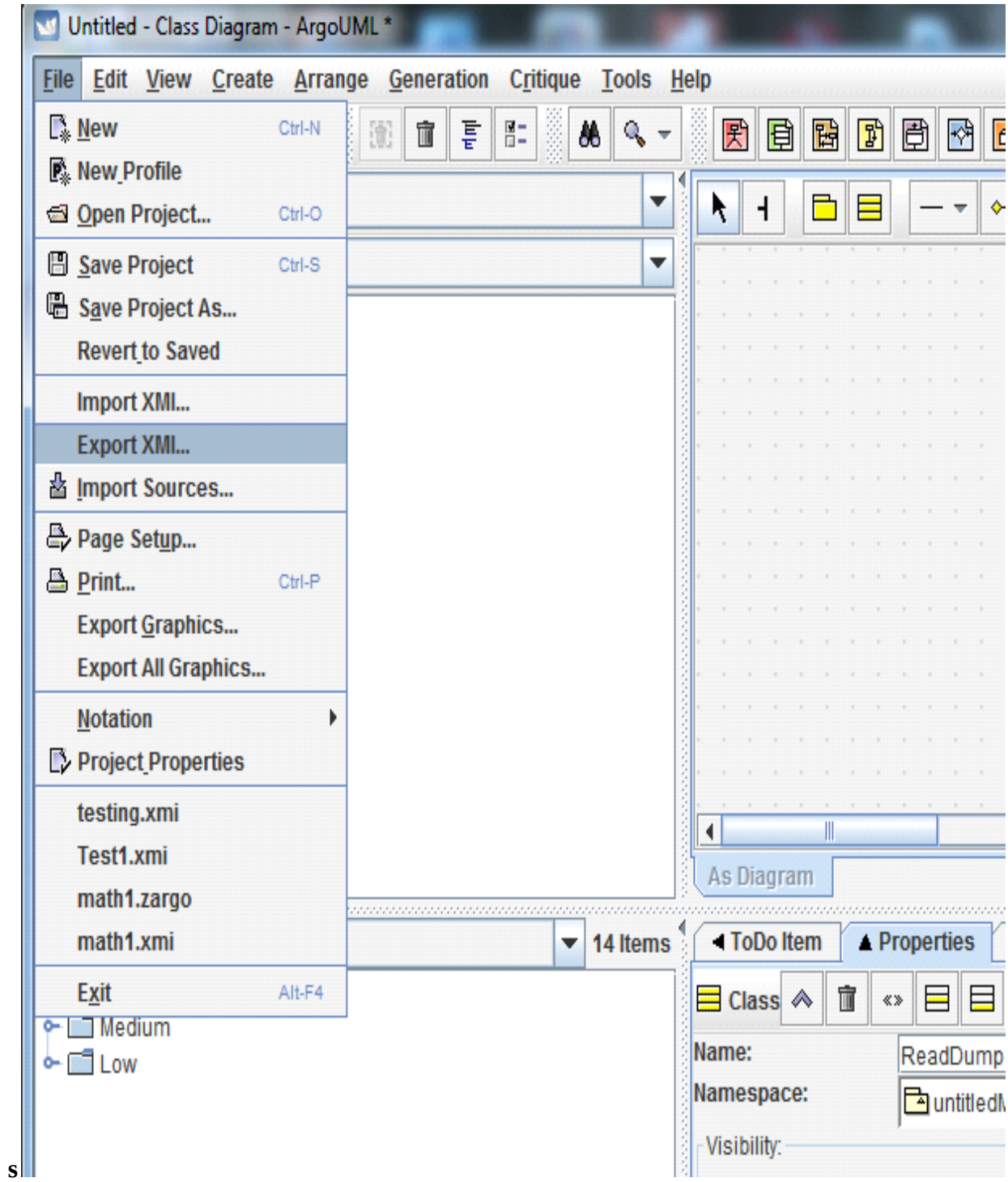

Figure4.8 Export XMI of UML diagram

h) Browse location where you want to store you XMI. You may rename your XMI if you want to. Save XMI by clicking save button

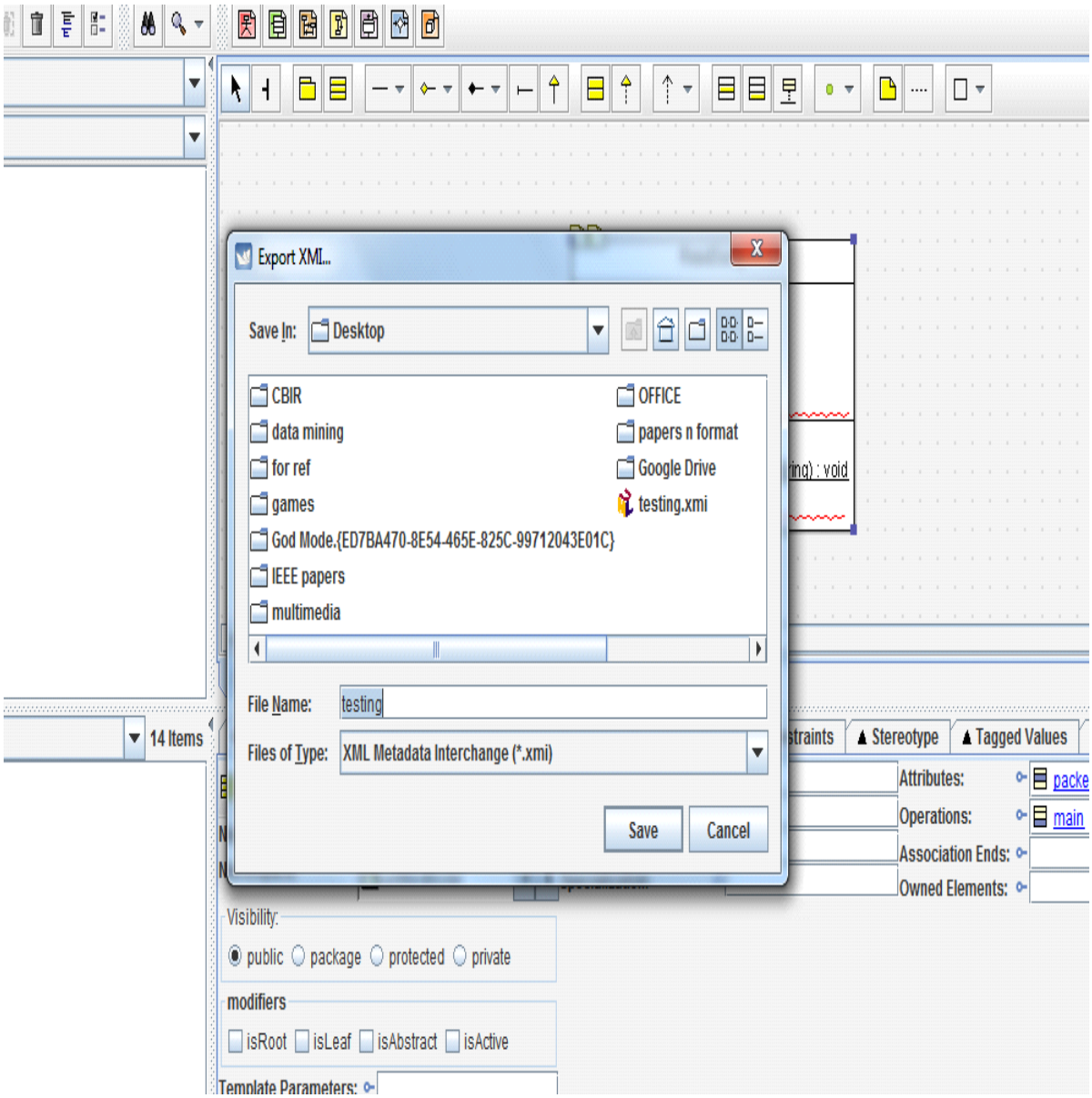

Figure4.9 Save XMI file by clicking save button in agroUML tool

i) We may open your XMI file in a web browse

After creating the UML diagram of the class we have to convert it into XMI using ArgoUml. We are using a XMI file for testing purpose. An XMI (XMI Metadata Interchange) is a standard for exchanging metadata information. First we have to give XMI file path. Once we have provided the required input we will run our program. Program will read the XMI file from the specified location. To read XMI, first we will create a DOMParser object that will parse the file. After parsing the file we will store the file in a Document object in the form of document. A Document Traversal object will be used to convert Document (XMI file is converted into document) into node.

Each node has a name and some attributes. These node names and attributes will be used to get required information. Now we pick one node at a time till all nodes are picked and we will get Node name and node attributes. After that it will be checked if node is of type class. If yes then following nodes attributes will contain information about main class. This information will be stored to use for testing.

Once all nodes are traversed, next step is to perform testing using aspect file. AOP script will initiate testing. As we will test the program we first need to run the program to test it. AOP script can capture important instances in the main code without modifying the behaviour or making any change to the main class. For testing purpose we need the objects of the classes, to solve this problem aspect will capture the instance when objects are initialized and these objects will be stored in the aspect. The study comprises of three types of anti-pattern testing.

4.4 THE BLOB TESTING

For blob testing we need the objects of the classes, to solve the problem aspect will capture the instance when objects are initialized and these objects will be stored in the aspect. In blob we need to find method call stack that means how methods are called. In a blob one object or method perform most of the task whereas it is expected that one method should perform one work. So if a single method is called again and again that is a blob. Our aspect capture each method

call. In the end by analyzing method calls we can easily find if a particular method is called again and again, and if so it will indicate a blob.

4.5 UN-USEDCODE TESTING

To find un-usedcode, that are unused fields in the code, we use Reflection (Application Programming Interface)API of java. It allow us to get the fields of an object by passing the name of the field. We get name of the fields of the class from the XMI, then these names will be passed to reflection API methods and we will get values of the fields. Fields those have data stored in them will be used fields and fields with null values will be unused fields.

4.6 CRYPTIC CODE TESTING

The cryptic code, as there is no particular standard to find what is a meaningful name for a variable, analyze that mostly variable names those are less than four character are meaningless. Therefore to detect the cryptic code our method will find all those fields that are equal or less than four character will be considered as cryptic code.

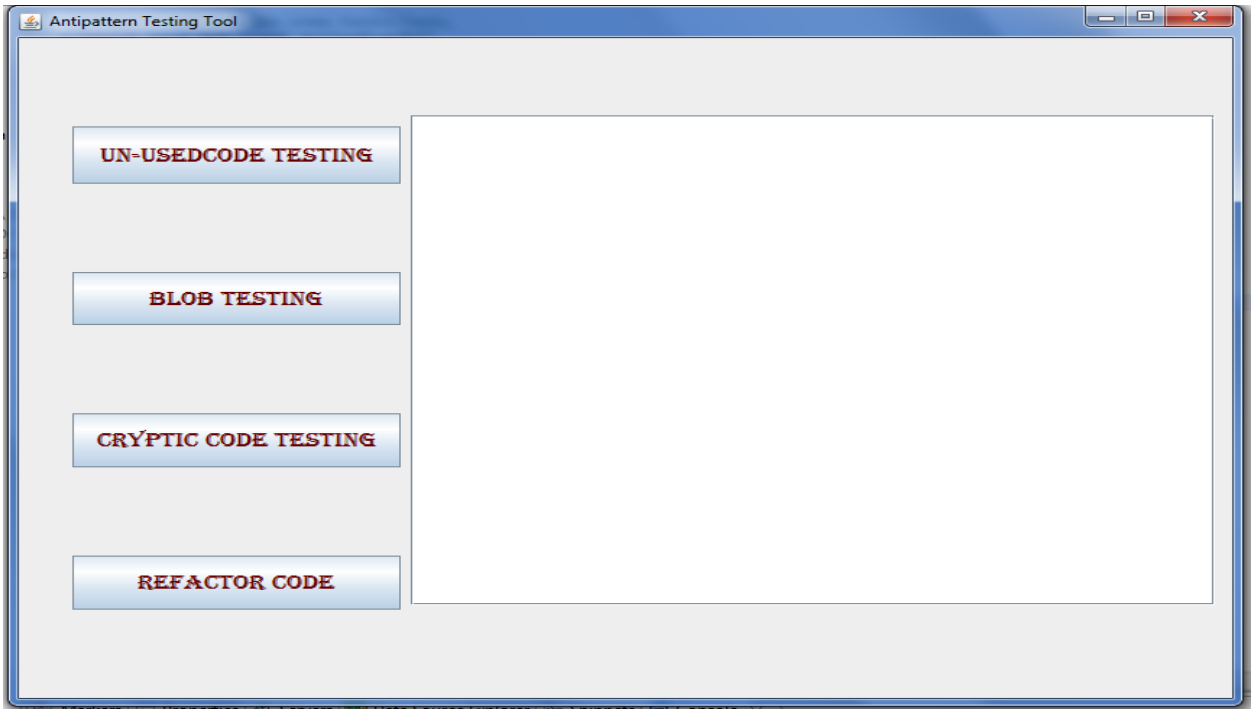

Figure4.10 Anti-pattern testing interface of source code

Further to get the information about the blobs we need to find method call stack that means how methods are called. In a blob one object or method perform most of the task whereas it is expected that one method should perform one work. So if a single method is called again and again that is a blob. Our aspect captures each method call. In the end by analyzing method calls we can easily find if a particular method is called again and again, and if so it will indicate a blob.

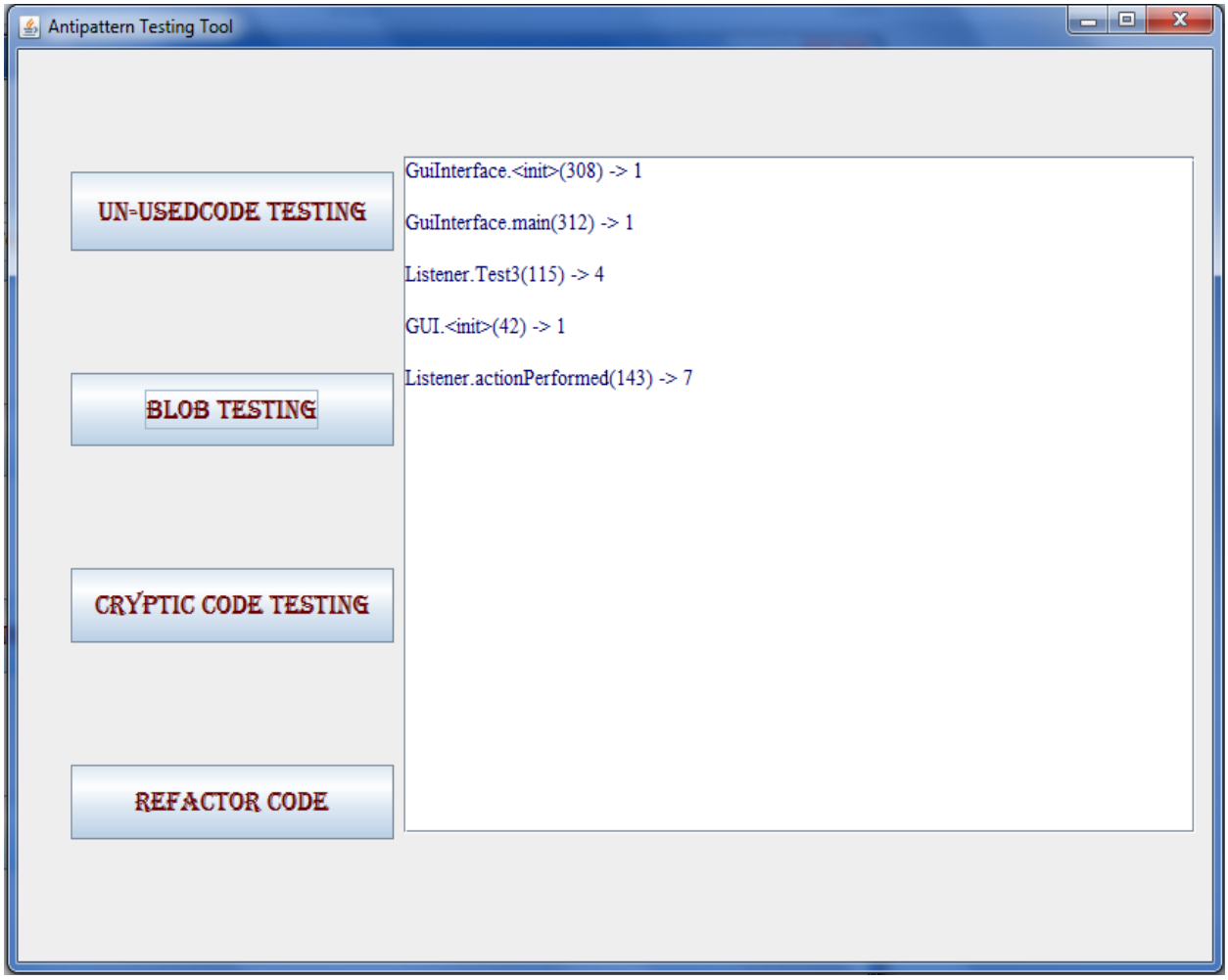

Figure4.11 Blob testing interface

To find unused code, that are unused fields in our code, in the dissertation we use Reflection API of java. It allow us to get the fields of an object by passing the name of the field. We get name of the fields of the class from the XMI, then these names will be passed to reflection API methods and we will get values of the fields. Fields those have data stored in them will be used fields and fields with null values will be unused fields.

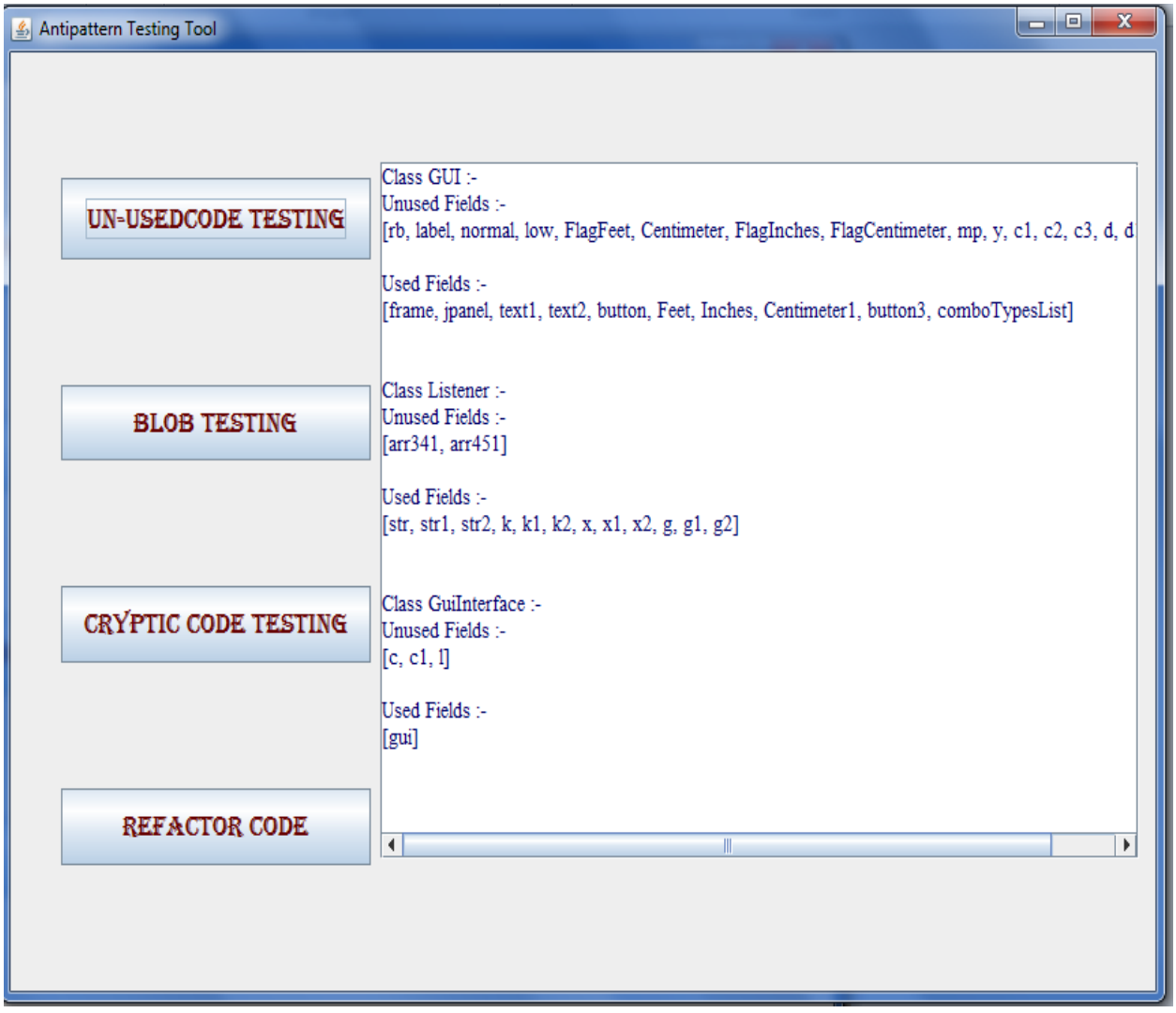

Figure4.12 Unused code testing interface

Further we test for cryptic code, as there is no particular standard to find what is a meaningful name for a variable, in our dissertation we are analysing that mostly variable names those are less than four character are meaningless. Therefore to detect the cryptic code our method will find all those fields that are equal or less than four character will be considered as cryptic code.

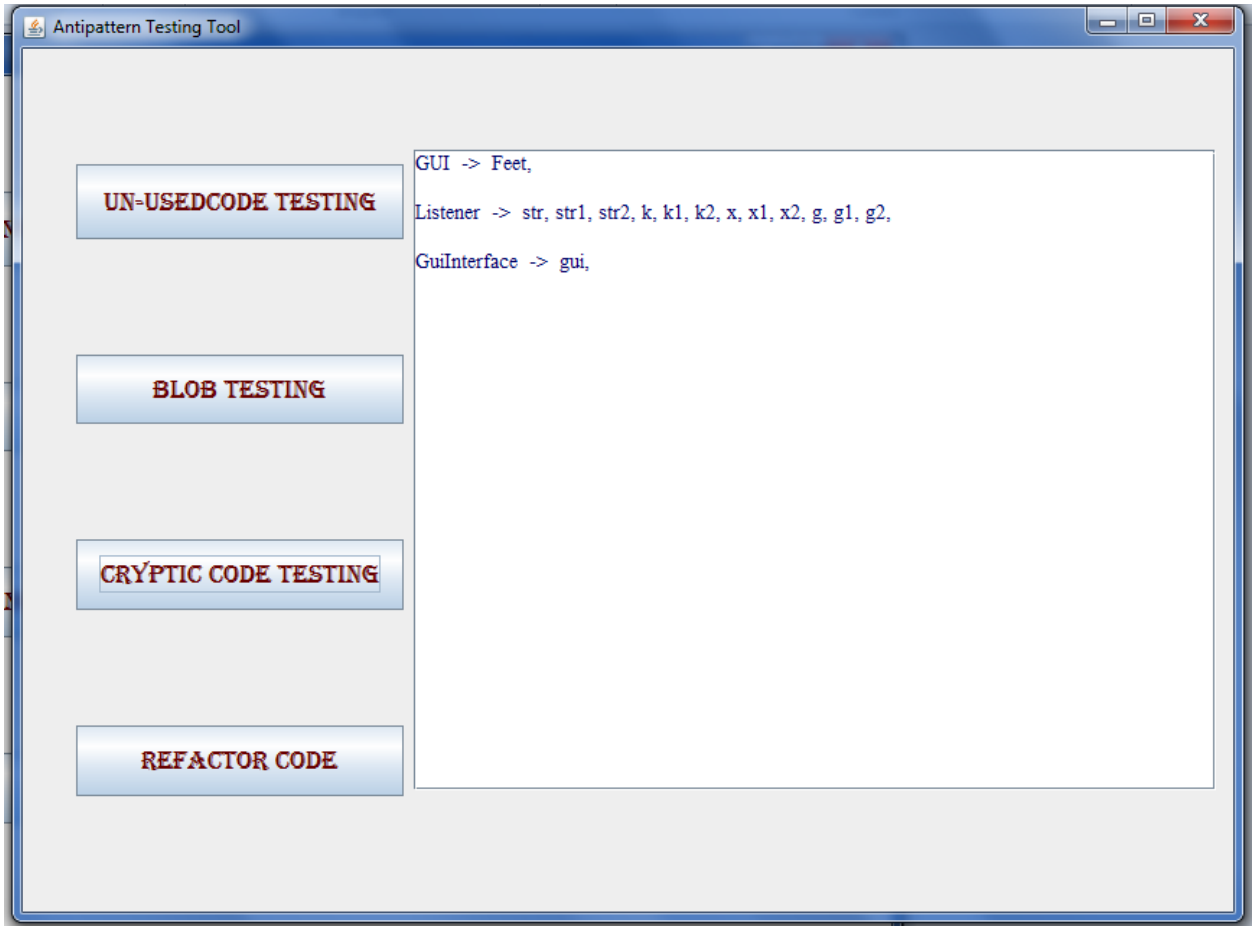

Figure4.13 Cryptic code testing interface

To make code easily maintainable we have created different classes for different purposes, and AOP script link all these classes together to make a useful testing tool. As we are performing testing on the running code, important instance of the program will be captured during execution, last phase of the dissertation

involves a user interface development. User interface development will involve various interface frame that includes main frame, that will allow user to run code and then test it by click of the buttons. And a frame that will display the result to the user, to keep the things simple and easily understandable, in the dissertation we have provided different button on the result frame, that allow user to look for a specific type of anti-pattern. User has option to check for unused code, to check for blob and check cryptic code.

In the running mode, first user will be presented a starting interface of the tool. In this dissertation interface user will have three options, user may run the application that has to be tested, user may search for anti-patterns and user may exit the tool.

Figure4.14 Starting testing interface of tool

To test the application user first have to start the application, on clicking the run application button, application to be tested will execute. We have to run all functions of the application, our AOP script will capture all the important instances or points in the application, like object creation, method calls and method executions. These points will be used to find anti-patterns in the code.

Once application is executed completely, we start testing work. We start testing by clicking testing button.

Testing process will start in different phases, first on button click control will be transferred to AOP script, this script will run code to read XMI file and useful information will be stored, this information will be used to find unused code and cryptic code.

After completion of XMI reading and information retrieval phase, AOP script will run java class that will display a frame to the user, this frame will provide button to the user and each button performs a specific task. Selecting unused code button will find unused code. In unused code detection, we will find the values of fields those are extracted from XMI file, if field will be null that means it is unused, to do all this work we are using Reflection APIs. Selecting blob button will check call stack and calculate how many times a particular method is called, user will be displayed this information, by analyzing this user may find if a particular method is called again and again.

Too many calls to a particular method will indicate a blob method. Selecting cryptic code button will find variables that don't have meaning full name. Cryptic code make it difficult to understand the code as it is hard to know what kind of value a variable is storing, this reduces the readability as well as maintainability of the code. Abbreviations i.e. variable having name equal or less than 4 characters are mostly doesn't explain clearly what kind of value they are storing, we show all these in cryptic code. Once user will be displayed detected anti-patterns next major step is refactoring of code. Anti-patterns detection can only show us the problems in the code, refactoring will resolve that problems.

Therefore after finding anti-patterns we refractor the code. After refactoring we test refactored code that there is still any anti-patterns, moreover in the dissertation we compare memory consumption and time taken to execute code with and without anti-patterns. And show that code without anti-patterns decrease the memory consumption and is faster to run.

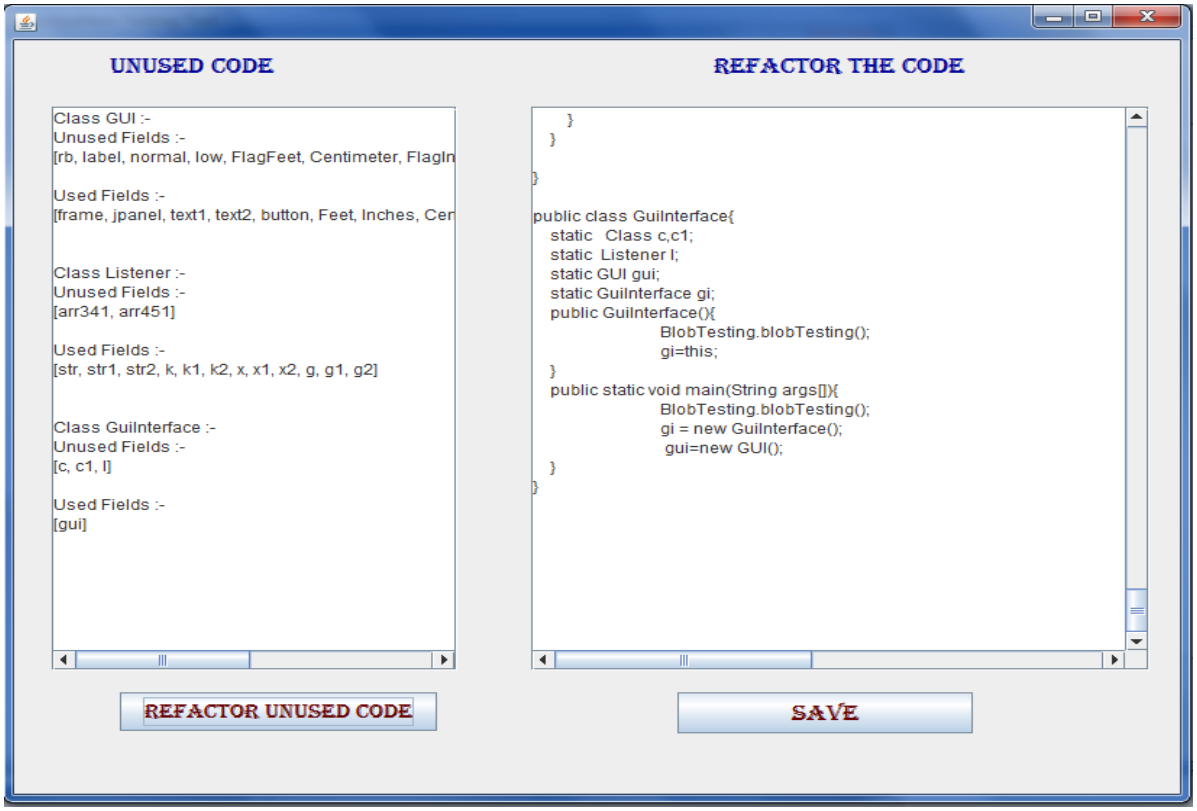

Figure4.15 Select the program and run on anti-patternn tool

For testing purpose we are using a XMI file. An XMI (XML Metadata Interchange) is a standard for exchanging metadata information. Our program working can be divided in two phases.

- In first phase we read XMI file and collect required information from it.
- In second phase we read our aspect file collect information from it and perform our testing on the bases of collected information.

3.5 WORKING OF THE PROGRAM

In the testing program we have to give three inputs.

1. We have to give XMI file path.

2. We have to give main class name of AOP based program in main class name field.

3. We have to give aspect name in the aspect field.

Once we have provided the required input we run our program. Program will read the XMI file from the specified location.

CHAPTER 5
RESULT AND DISCUSSIONS

In the dissertation we are trying to separate the results in two categories. In first category it is shown a comparison between code having anti-patterns and refactored code without anti-pattern. These results are supporting the core of our dissertation that is removing anti-patterns enhances the quality of the software. In other category we are making a comparison between a AOP based anti-pattern testing approach and a simple java based approach to detect the anti-patterns.

To make the comparison between anti-pattern code and refactored code, we are using memory and time parameters. We are testing the code number of times and those comparison results are displayed in the graph form. By analyzing these graphs we can easily conclude that removing anti-patterns are improveing the programs execution speed, moreover pruning of unused fields from the code is reducing the memory consumption of the program. Time and memory are two most important factors when we measure the software quality. Further removing anti-patterns are making code much clear, which are improving the code's readability as well as reusability.

The biggest advantage of using AOP is we can test the running code, that helped us to find the unused fields and blob easily. AOP allows us to manipulate the behavior of the java program without making an change in this dissertation and this is the motivator to use AOP in testing of anti-patterns. As we are not made any change to main code, but in other tool where AOP is not used we have to made certain changes to the main code, to capture method calls information and information about the object, that is not an acceptable approach as it is too much work and changing main code for testing purpose could alter the program. Further we found a noticeable time difference between testing using AOP and without AOP. In the dissertation results are showing that testing using AOP is faster.

5.1 TIME CONSUMPTION

In time consumption it shows a comparison between code with anti-patterns and refactored code without anti-patterns. These results are supporting the core of our dissertation that is removing anti-patterns enhances the quality of the software. We are testing the code number of times and those comparison results are displayed in the graph form. By analyzing this graph we can easily conclude that removing anti-patterns are improveing the programs execution time.

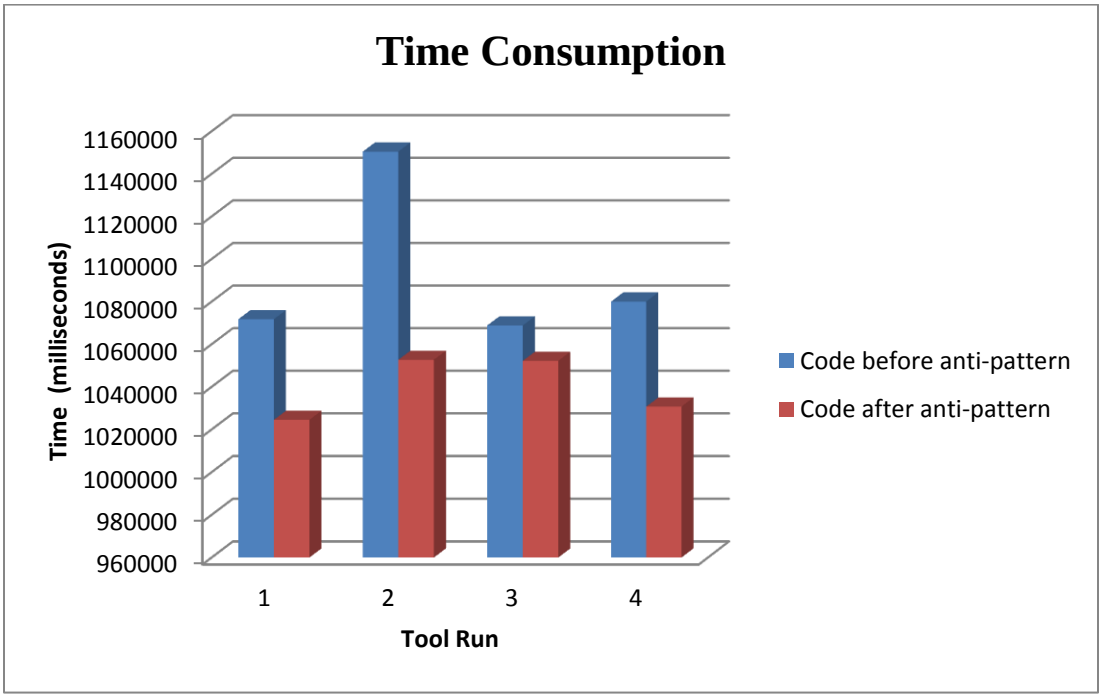

Figure5.1 "Time Consumption" graph before and after anti-pattern

TABLE 5. 1 Resultant values of "Time Consumption" before and after anti-pattern

Code before anti-pattern	Code after anti-pattern
1071748	1024483
1150430	1052688
1068816	1052199
1078055	1030695

5.2 MEMORY CONSUMPTION

In memory consumption it shows a comparison between code with anti-patterns and refactored code without anti-patterns. These results are supporting the core of our dissertation that is removing anti-patterns enhances the quality of the software. We are testing the code number of times and those comparison results are displayed in the graph form. By analyzing this graph we can easily conclude that removing anti-patterns are improving the programs execution memory.

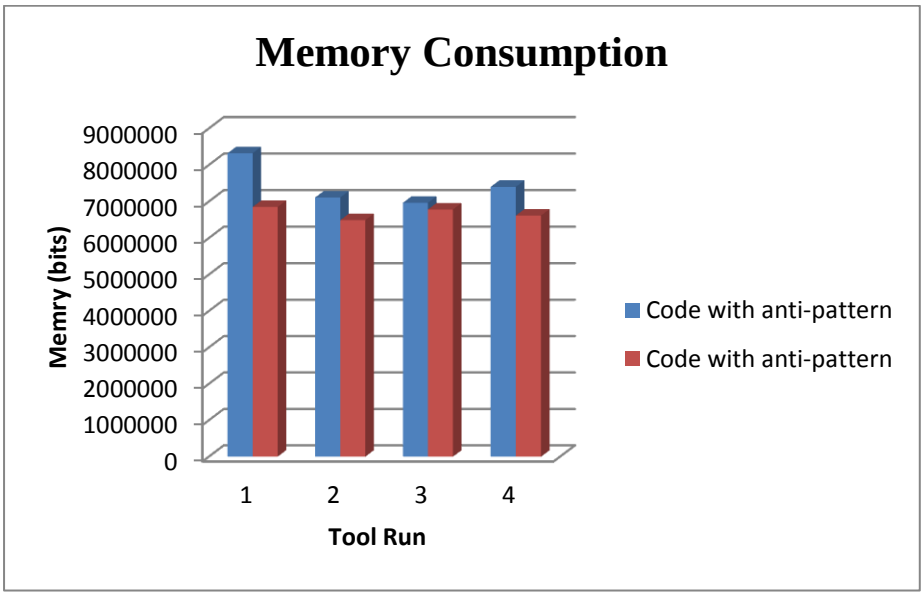

Figure5.2 "Memory Consumption" graph before and after anti-pattern

TABLE 5. 2 Resultant values of "Memory Consumption" before and after anti-pattern

Code with anti-pattern	Code with anti-pattern
8322864	6850240
7000872	6490140
6964568	6780760
7402368	6616680

5.3 TESTING TIME

In testing time we are making a comparison between a AOP based anti-pattern testing approach and a simple java based approach to detect the anti-patterns.

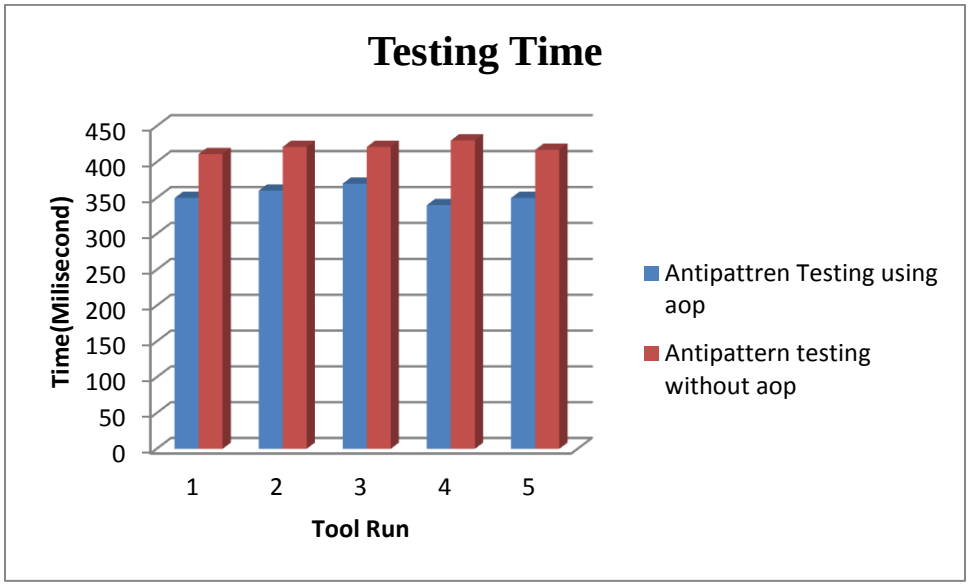

Figure5.3 "Testing Time" graph before and after anti-pattern

TABLE 5.3 Resultant values of "Testing Time" before and after anti-pattern

Anti-pattern Testing using AOP	Anti-pattern testing without AOP
345	411
350	421
370	421
340	430
346	417

CHAPTER 6
CONCLUSION AND FUTURE SCOPE

In the end after comparing we can say that a code without anti-pattern has better squality. Memory consumption is low where as refactoring of code to remove anti-pattern has made it faster. Moreover AOP provides us an approach to test anti-patterns in the code without making any modification to the code. AOP can capture the main points from the running code therefore it allow us to test a running code for anti-patterns instead of looking source code to find anti-patterns.

Further time comparison support that using AOP is a faster approach to test anti-patterns. In future we can test more anti-patterns and refactor them and make tool for generalized code.

REFERENCES

[1] Alanen, M., and Porres, I.; Difference and union of models. In UML'03 (Unified Modeling Language), San Francisco, CA, USA, 2003.

[2] Alecsandar, S., and Ioana, S.; Detecting patterns and antipatterns in software using prolog rules. In Computational Cybernetics and Technical Informatics (ICCC-CONTI), International Joint Conference on, pp. 253–258, Department of Computers, Politehnica University of Timisoara, Romania, 2010.

[3] Alur, D., Crupi, J., and Malks, D.; Core J2EE Patterns -Best Practices and Design Strategies, Sun Microsystems Press, 2001.

[4] Beyer, D., and Lewerentz, C.; Crocopat: Efficient pattern analysis in object-oriented programs. In IWPC '03: Proceedings of the 11th IEEE International Work-shop on Program Comprehension, pp. 294, Washington, DC, USA, IEEEComputer Society, 2003.

[5] Bieman, J.M., Straw, G., Wang, H., Munger, P.W., and Alexan-der, R.T.; Design patterns and change proneness: An examination of five evolving systems.In METRICS '03: Proceedings of the 9th International Symposium on Software Met-rics, page 40, Washington, DC, USA, IEEE Computer Society, 2003.

[6] Briand, L., and Labiche, Y.; A uml-based approach to system testing. *Software and Systems Modeling*, 1(1):10 – 42, 2002.

[7] Brown, W. J., Malveau, R. C., McCormick, H.W., and Mowbray, T.J.; Anti-patterns: Refactoring Software,Architectures, and Projects in Crisis, New York, John Wiley and Sons, Inc., 1998.

[8] Cheon, Y., Avila, C., Roach S., and Munoz, C.; "Checking design constraints at run-time using OCL and AspectJ," International Journal of Software Engineering, vol. 3, no. 1, pp. 5–28, 2009.

[9] Cortellessa, V., and Frittella, L.; A Framework for Automated Generation of Architectural Feedback from Software Performance Analysis. In Proceedings of the Formal Methods and Stochastic Models for Performance Evaluation, Fourth European Performance Engineering Workshop, EPEW pp. 171–185, 2007.

[10] Costagliola, G., Lucia, A.D., Deufemia, V., Gravino, C., and Risi, M.; Design pattern recovery by visual language parsing. In CSMR ’05:Proceedings of the Ninth European Conference on Software Maintenance and Reengi-neering, pp. 102–111, Washington, DC, USA, IEEE Computer Society.Computer Society, 2005.

[11] Dhambri, K., Sahraoui, H., and Poulin, P.; Visual detection of design anomalies. In CSMR ’08: Proceedings of the 2008 12th European Conference Software Maintenance and Reengineering, pages 279–283, Washington, DC, USA, IEEE Computer Society, 2008.

[12] Dietrich, J., and Elgar, C.; Towards a web of patterns. Web Semant., 5108–116, 2007.

[13] Elaasar, M., Bri, L.C., and Labiche, Y.; A metamodeling approach to pattern specification and detection, 2006.

[14] Elaasar, M., Briand, L. C., and Labiche, Y. A.; Metamodeling Approach Fixing Software Performance Problems. In International Computer Measurement Group Conference, pp. 307–320, 2002.

[15] Emden, E. V., and Moonen, L.; Java quality assurance by detecting code smells. In WCRE ’02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE’02), pp. 97, Washington, DC, USA, IEEE Computer Society, 2002.

[16] Fowler, M.; Refactoring: Improving the Design of Existing Code, Reading, MA, Addison-Wesley Longman, 1999.

[17] Gall, H., Jazayeri, M., and Krajewski, J.; Cvs release history data for detecting logical couplings. In IWPSE ’03: Proceedings of the 6th International Workshop on Principles of Software Evolution, page 13, Washington, DC, USA, IEEE Computer Society, 2003.

[18] Gamma, E., Helm, R., Johnson, R., and Vlissides J.; Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA, Addison-Wesley, 1995.

[19] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G., “An overview of AspectJ,” in ECOOP 2001— Object-Oriented Programming 15th European Conference, Budapest Hungary, ser. LNCS, Knudsen, J. L., Ed. Berlin:Springer-Verlag, vol. 2072, pp. 327–353, 2001.

[20] Lanza, M., and Marinescu, R.; Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer, 2006.

[21] Lee, H., Youn, H., and Lee, E.; A design pattern detection technique that aids reverse engineering, 2008.

[22] Meyer, M.; Pattern-based reengineering of software systems. In WCRE ’06: Proceedings of the 13th Working Conference on Reverse Engineering, pp. 305–306, Washington, DC, USA, IEEE Computer Society, 2006.

[23] Moha, N., Yann, G., Duchien, L., and Meur, A.F.; Decor: A method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng., 36:20–36, 2010.

[24] Munro, M.J.; Product metrics for automatic identification of “bad smell” design problems in java source-code. In METRICS ’05: Proceedings of the 11th IEEE International Software Metrics Symposium, pp. 15, Washington, DC, USA, IEEE Computer Society, 2005.

[25] Niere, J., Wilhelm, S., Wadsack, J.P., Wendehals, L., and Welsh, Jim.; Towards pattern-based design recovery. In ICSE ’02: Proceedings of the 24th International Conference on Software Engineering, pages 338–348, New York, NY, USA, 2002.

[26] Oliveto, R., Khomh, F., Antoniol, G., and Gueheneuc, Y.; Numerical signatures of anti-patterns: An approach based on b-splines. Software Maintenance and Reengineering, European Conference on, pp. 248–251, 2010.

[27] Petriu, D., and Somadder, G.; "A Pattern Language For Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers," Proceedings of EuroPLoP, Kloster Irsee, Germany, 1997.

[28] Richters, M., and Gogolla, M.; “Aspect-oriented monitoring of UML and OCL constraints,” in The 4th AOSD Modeling with UML Workshop, San Francisco, CA, co-located with UML, 2003.

[29] Shi, N., and Ronald, A.; Olsson. Reverse engineering of design patterns from java source code. In ASE’06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, pp. 123–134, Washington, DC, USA, IEEE Computer Society, 2006.

[30] Smith, C. U., and Willams, L.G.; Software Performance Anti-patterns; Common Performance Problems and their Solutions. In *International Computer Measurement Group Conference*, pp. 797–806, 2001.

[31] Smith, C.U.; Performance Engineering of Software systems, Reading, MA, Addison-Wesley, 1990.

[32] Stotts, D., Smith, J.M., and Jason, Mcc.; Spqr: Flexible automated design pattern extraction from source code. In In 18th IEEE Intl Conf on Automated Software Engineering, pp. 215–224. IEEE Computer Society Press, 2003.

[33] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S.T.; Design pattern detection using similarity scoring. IEEE Trans. Softw. Eng., 32:896–909, 2006.