

**AUTOMATED APPROACH FOR ANTI-PATTERN DETECTION AND
REMOVAL WITH JAVA STRUCTURES**

A Dissertation

Submitted

By

Neha Nanda

To

Department of Computer Science

In partial fulfillment of the Requirement for the

Award of the Degree of

**Master of Technology in
Computer Science and Engineering**

Under the guidance of

Rohitt Sharma

(April 2015)

PAC FORM



School of: Computer Science and Engineering

DISSERTATION TOPIC APPROVAL PERFORMA

Name of the student : Neha Nanda
Batch : 2010-2015
Session : 2014-2015

Registration No : 11005541
Roll No : RK2006B27
Parent Section : K2006

Details of Supervisor:

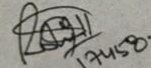
Name : Rohitt Sharma
UID : 17458

Designation : Assistant Professor
Qualification : M.Tech
Research Exp. : 1.5 year

Specialization Area: Software Engineering (pick from list of provided specialization areas by DAA)

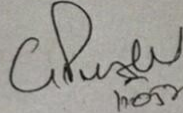
Proposed Topics:-

1. Anti-Pattern Detection
2. Software Testing
3. Software Reuse


Signature of supervisor

PAC Remarks:

Topic 1 is approved


Signature

APPROVAL OF PAC CHAIRMAN

Signature: 

Date:

*Supervision should finally encircle one topic out of three proposed topics and put up for an approval before Project Approval Committee (PAC).

*Original copy of this format after PAC approval will be retained by the student and must be attached in the Project/Dissertation final report.

*One copy to be submitted to supervisor.

ABSTRACT

Anti-patterns are poor design choices that are conjectured to make object oriented systems harder to maintain. We investigate the impact of anti-patterns on classes in object-oriented systems by studying the relation between the presence of anti-patterns and the change- and fault-proneness of the classes. Due to increased complexities in the software development, there is huge need of testing process to be carried on in better way. Due to increased complexities in the software development and increasing of anti-patterns in the software development, there is a huge need of testing process to be carried out in a better and effective way.

ACKNOWLEDGEMENT

I would like to thank everyone who has supported and guided me through my thesis dissertation. I thank them for their faith, their criticism, and their ability to boost me up when I got stuck. I would like to thank them for their useful advises which have brought a difference in my approach to successfully complete my thesis.

I would like to express my gratitude to Mr. Rohitt Sharma for his constant support, guidance and patience through the entire course of my work. He has been a constant pillar throughout as his blind faith has brought us to believe in ourselves and our work.

I would also like to thank my parents for always believing in me and helping me get throughout my M.tech. It would have not been possible without their encouragement and support .I would also like to thank my cousin for helping and guiding me, her PhD experience really counted a lot in achieving my objectives.

DECLARATION

I hereby declare that the dissertation entitled," Automated Approach For Anti-pattern Detection And Removal With Java Structures" submitted for the M.Tech Degree is entirely my original work and all ideas and references have been duly acknowledged. It does not contain any work for the award of any other degree or diploma.

Date:

NEHA NANDA

Investigator

Regno: 11005541

CERTIFICATE

This is to certify that Neha Nanda has completed M.Tech dissertation proposal titled “AUTOMATED APPROACH FOR ANTI-PATTERN DETECTION AND REMOVAL WITH JAVA STRUCTURES” under my guidance and supervision. To the best of my knowledge, the present work is the result of her original investigation and study. No part of the dissertation proposal has ever been submitted for any other degree or diploma.

The dissertation proposal is fit for the submission and the partial fulfillment of the conditions for the award of M.Tech Computer Science & Engg.

Date: _____

Signature of advisor

NAME

UID

TABLE OF CONTENTS

Chapter 1: INTRODUCTION	1
1.1 Types of Anti-patterns	3
1.1.1 Singleton Overuse	3
1.1.2 Functional Decomposition	3
1.1.3 Poltergeist	3
1.1.4 Spaghetti	4
1.1.4 Copy and Paste	4
1.1.5 Code Smells	4
1.1.6 Golden Hammer	4
1.1.7 Blob	4
1.1.8 Unused Data	4
1.1.9 Cryptic Code	5
1.2 Unified Modeling Language	5
1.2.1 Types of diagrams	5
1.3 ArgoUML	6
1.4 ECLIPSE (JAVA)	7
1.5 XMI (XML METADATA INTERCHANGE)	8
1.6 Refactoring	9
1.7 Testing Process	10
Chapter 2: REVIEW OF LITERATURE	11
Chapter 3: SCOPE OF THE STUDY	19
Chapter 4: OBJECTIVES OF THE STUDY	20
4.1 Problem Definition	20

4.2 Objectives	20
Chapter 5: METHODOLOGY	21
Chapter 6: RESULTS AND DISCUSSIONS	24
6.1 STEPS TO DEMONSTRATE METHODOLOGY	24
6.2 MEMORY AND TIME COMPARISON	36
Chapter 7: CONCLUSIONS AND FUTURE WORK	39
Chapter 8: REFERENCES	41
Chapter 9: APPENDIX	43

LIST OF FIGURES

Figure No.	Figure Name	Page No.
Figure 1	Design Pattern and Anti Pattern Concept	2
Figure 2	ArgoUML working environment	7
Figure 3	Eclipse Working Environment	8
Figure 4	Xmi Concept	9
Figure 5	Flowchart to depict methodology	24
Figure 6	ArgoUML interface	25
Figure 7	ArgoUML import sources	25
Figure 8	Selecting of java file	26
Figure 9	Selecting add to diagram	26
Figure 10	Creation of UML diagram	27
Figure 11	Export XMI	27
Figure 12	User detail interface	28
Figure 13	Enter a valid integer value in the textbox	28
Figure 14	Conversion from feet to inches	29
Figure 15	Conversion from inches to feet	29
Figure 16	Conversion from centimetre to inches	30
Figure 17	Source code checked window	30
Figure 18	Selecting the text file you wish to upload	31
Figure 19	After selecting click on upload	31
Figure 20	After uploading click on testing	32
Figure 21	Memory used message box	32
Figure 22	Antipattern testing tool window	33

Figure 23	Un-usedcode testing	34
Figure 24	Blob testing	34
Figure 25	Cryptic code testing	35
Figure 26	Refactor unused code	35
Figure 27	Refactoring of unused code	36
Figure 28	Unused code removal file	36
Figure 29	Console showing time for manual and automatic input	37
Figure 30	Execution time comparison	38
Figure 31	Memory usage comparison	38

Chapter 1

INTRODUCTION

Good guys focussing on success are known as the design patterns are known which are problem based and well defined unlike anti-patterns that are poorly defined, focus on failures and are solution based. Bad guys are known as the anti-patterns that seem to be effective and are refactored solutions but may lead to unwanted consequences. The term was coined in 1995 by Andrew Koenig, inspired by Gang of Four's book Design Patterns, which developed the concept of design patterns in the software field ^[1]. When a problem arises during coding, sometimes due to lack of knowledge, lack of experience or lack of time we conclude a solution to the problem that seems effective and easy but in turn leads more adverse problems clearly describing the concept of anti-patterns. It may not hamper the execution of the program but may lead to other issues which may go unnoticed otherwise and the ones ignored thus making detection of such anti-patterns difficult and time-consuming. These days' anti-patterns are an active research area that extends the study of design patterns into more extensive fields. There is a rising need to detect and refactor unsuccessful behaviours and thus turning the code to a better desirable solution and in turn enhancing the software quality and performance of the code that could be in terms of memory consumption, time to execute the program or cost.

Sometimes when we devise a solution to a problem it may seem useful but when applied may leave us in a condition worse than before. Applying anti-patterns may seem to be a solution to a problem and initially may seem to be a good idea but fall off when implemented. Many anti-patterns come from a corrective action gone awry ^[2]. Anti-patterns are one of the main revolutionary changes in software engineering and computer science today. But a solution not working so called the “negative solution” has attached benefits with them, they provide us with the knowledge of what does not work and what will be effective and will provide us the required solution .it serves as the software guidance much required in the evolution of a project.

Anti-patterns are new form of patterns that are accompanied by two solutions first is problematic solution that has a set of negative consequences and the second is the refactored solution which turns the anti-patterns in a healthy kind of solution.

Refactoring the code involves turning the design of external code to a more improved version without changing its external behaviour. Developers generally introduce anti-patterns in the system due to either lack of understanding, time pressure or communication skills. Source code becomes intricate to understand and thus blocks the growth and the maintenance actions. Finding anti-patterns on system's subset can lessen efforts, resources and cost overall. Different types of anti-patterns existing are swiss army knife, blob, cryptic data, unused data, message chain, lazy class, the agile charade, data last, bricks without clay etc. Various techniques are evolving that detect anti-patterns occurring in a system. The various systems available for detecting anti-patterns are eclipse, rhino, mylyn, argouml, décor, smurf etc. Yet there are some limitations to these techniques like they have limited recall and accuracy which require extensive knowledge about the anti-patterns and cannot be applied on system's subset. Difference between design patterns and anti-patterns are depicted in Figure 1.

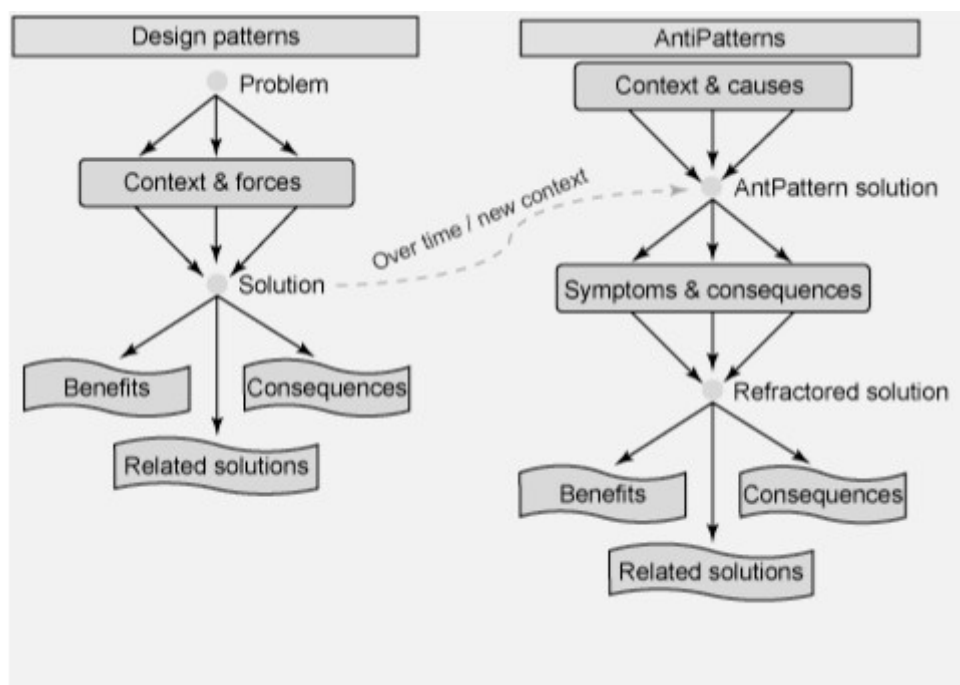


Figure 1: Design Pattern and Anti-Pattern Concept ^[3]

In our study we focus on software quality and software testing. Software quality is defined as a characteristic or an attribute of any entity. When we take attribute of any item into consideration, we refer to characteristics which are things we can compare to standards that are already known such as length, colour, malleability etc. Quality of design for any products is a set of characteristics that identify any item or product. These attributes are specified by the designers at design time and quality of conformance refers to the extent to which the design characteristics specified by the designer are followed and software testing refers to the execution of program after it has been coded to check for any errors. It has been a costly process to be carried out in comparison to other processes in the software life cycle. A lot of work is being carried out these days on testing cost and the focus remains on reducing the testing cost. Testing should be carried out in analysis and design phases in order to reduce the testing efforts .It is actually a review of the previous processes like specification, design and coding. A test case that discovers an undiscovered error is known as a successful test case whereas a test case that has a high probability of finding an undiscovered error is known as a good test case.

1.1 Types of Anti-patterns

There are different kinds of anti-patterns that act as a common vocabulary for detecting problems and gathering knowledge for their respective solutions:

1.1.1 Singleton Overuse: -It is the easiest pattern a software engineer can understand and is thought to be the good one but it violates information hiding. Larger the project more the singleton patterns. It is easy to detect a singleton pattern by looking on the class diagram and then detecting all the classes referring to themselves are the hidden singleton anti-patterns.

1.1.2 Functional Decomposition: -This anti-pattern is an old one which indicates migration of old software into a new project. It can be detected in three ways first, class names seem to be like function names secondly classes perform only one thing and lastly all the attributes of a class are private but used only within a class.

1.1.3 Poltergeist: -In this anti-pattern classes appear briefly for a short period of time and then disappear. They are believed to have a very less or limited functionality

because of which programmers don't even know what these classes do. We can detect them by looking at the class names as their names end with "controller" or "manager".

1.1.4Spaghetti: -This anti-pattern is very long just like noodles. The longer the code the more difficult and error prone it becomes. You can detect them by simply looking for methods with many lines of code.

1.1.4Copy and Paste: -When you copy a code from one place to another place you introduce copy and paste anti-pattern into your code and therefore in turn duplicating the functionality. Their use must be avoided and can be reduced by using either inheritance or turning the code into a method.

1.1.5Code Smells: -This is similar to anti-patterns but not that formal. The smell of a code can be either good or it can be bad which would indicate a problem with the code. This idea was introduced by Kent Beck in the late 1990s.different tools like Chekstyle, FindBugs can be used to detect the bad smells and can be refactored to remove such odours.

1.1.6Golden Hammer: -indicates the use of the same concept or technology which has been used over and over again in many software problems. It can be resolved by expansion of the developer's knowledge through training, education and reading different books that would help expand the working to use of different approaches and technologies.

1.1.7Blob: -Also called the "God object". The key problem here is that the majority of the responsibilities are allocated to a single class. In simple terms single function performs multiple functions. It results from inappropriate responsibilities allocation. It can be removed by splitting classes into much smaller other classes.

1.1.8Unused Data: -It is the data or the part of the code included in the code but is not useful or never accessed or used. It is the unnecessary portion of the code that increases the program execution time. It leads to heaping the memory and increase the complexity of the code.

1.1.9 Cryptic Code: - Is another type of anti-patterns which defines the abbreviations used for fields instead of proper naming. This makes hard to understand what types of values the field is simulating and also makes it difficult to reuse or modify the code in future.

1.2 Unified Modeling Language

UML or Unified Modelling Language has now become a common standard for software modeling and design and in turn building object oriented software. It is a graphical language which helps in defining the visible structure of a system. It is basically used to visualize, specify, construct and document the artefacts of a software system.

1.2.1 Types of diagrams

- I.** Class Diagram: - A UML class diagram is a static view of an application. Class diagram explains the working and attributes of a class and is used to visualize, describe and document the functionality of a system and also for building of code which is executable in nature of any software application. The class diagram depicts classes, interfaces, collaborations, constraints and associations and in turn can also be known as structural diagram.
- II.** Component Diagram:-This diagram depicts a set of components which comprise of classes, interfaces or collaborations and portrays the relationships among them. These diagrams help visualize the implementation view of a system.
- III.** Deployment diagram:-They are used to represent the hardware of a system, the software that is installed on that hardware and the middleware used to connect different machines to one another. This diagram depicts a set of nodes and the respective relationships among them. This diagram is used by the deployment team.
- IV.** Object Diagram:-also known as instance diagram are used to represent real world examples of objects and the respective relationships among them and also depict the static view of a system. They help build a prototype of a system for practical purposes.

- V. Use Case Diagram:-depicts the behaviour of the target system from an external point of view and consists of a set of use cases, actors and the relationships among them. Use cases help represent functionality of a particular system and can be depicted using actors associations and use cases.
- VI. Sequence Diagram:-also known as an interaction diagram and represents the sequence of messages flow from one object to another. It performs a specific functionality and visualizes the sequence of interactions among the objects.
- VII. Collaboration diagram:-it is also an interaction diagram and portrays the structure of an organization and the messages which are sent and received. It is similar to what a sequence diagram is like but its specific functionality is to visualize the objects of an organization and the interactions among them.
- VIII. Statechart Diagram:-the internal or external events in a system are a part of any real time system and these events are responsible for changing of state of a system. These diagrams represent the state change of a system depending on the occurrence of the events.
- IX. Activity Diagrams:-they describe the flow of control of the target system and consist of activities and links which give us an idea of how the system will work if executed. Functions of any system can be known as activities of that system and the flow of such functions can concurrent, branched, or sequential.

1.3ArgoUML

It is a java based application and is a UML modelling tool available in the market and is open source in nature and which includes support for all the standard UML diagrams and is available in 10 languages.

ArgoUML support class, sequence, deployment, collaboration, use case, activity and statechart UML diagrams. Its working environment is shown in Figure 2 below.

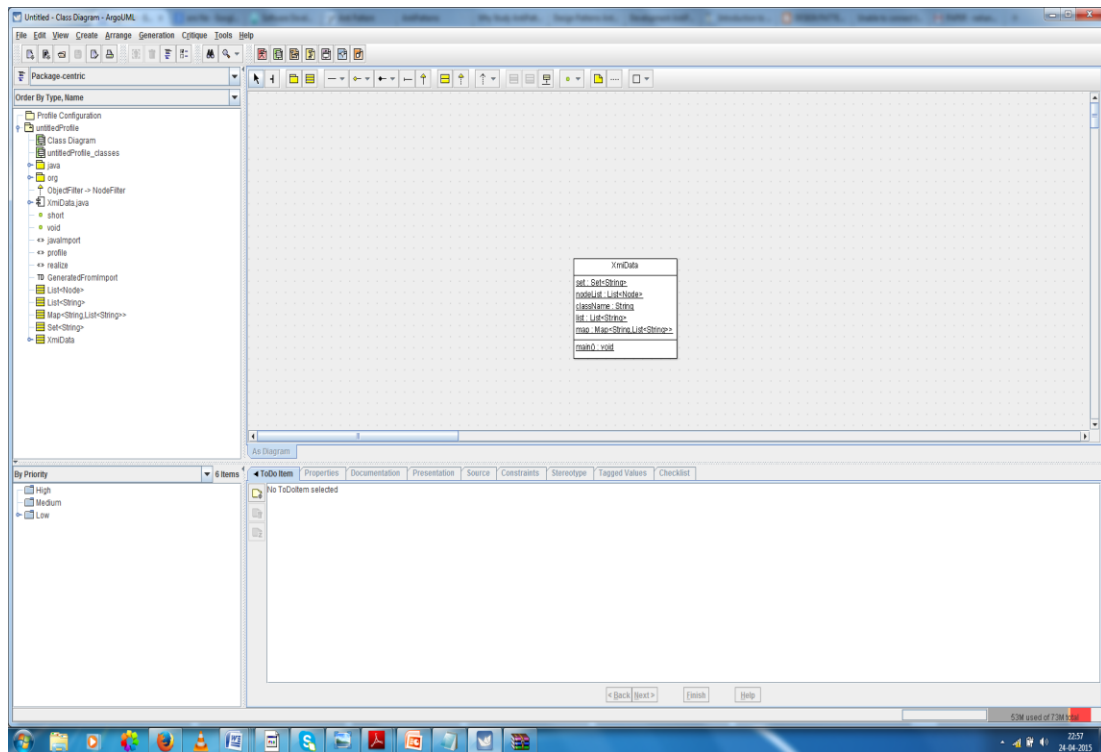


Figure 2: ArgoUML working environment

1.4 ECLIPSE (JAVA)

It is an integrated environment for project development under java and other programming languages like c, c++, PHP, ruby etc ^[4]. It was created by an open source community and can be used by android applications or java as a development environment.

Rich client applications, integrated development environment and some other tools can be built using eclipse. For any programming language having a plug-in Eclipse can work as an IDE. Its working environment is shown in Figure 3 below.

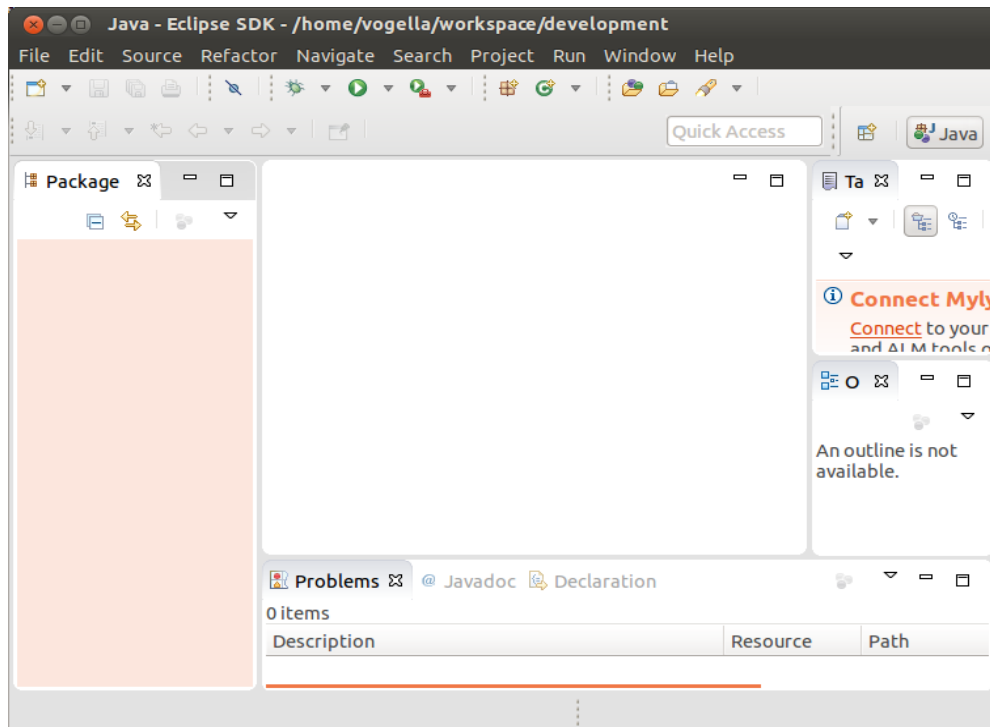


Figure 3: Eclipse Working Environment

1.5XMI (XML METADATA INTERCHANGE)

XML Metadata (data about data) interchange is a paradigm for metadata information exchange via XML (extensible markup language). It helps the programmers who work with UML (Unified Modeling Language) exchange their data models with each other. Information about data warehouses can also be exchanged among the programmers using XMI.

In short different companies which are cooperating with one another in one way or the other exchange their data whenever required using XMI.

File created in XMI format is used for exchanging UML diagrams and storing information about model design in a standardized XML format which is then used to transfer design information between software programs.

Our purpose is to read the XMI which will be produced and consists of methods, classes and variables. This XMI will be the input for eclipse testing module. XMI concept is depicted in Figure 4 below.

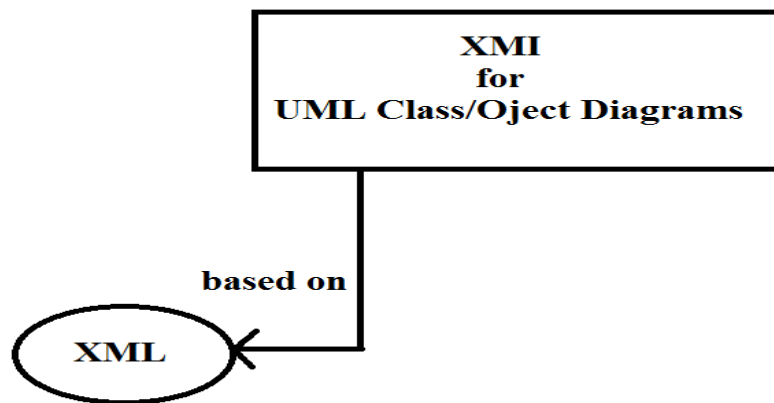


Figure 4: XMI Concept

1.6 Refactoring

Refactoring refers to changing the internal structure of the code without making any changes to its external behaviour.

It helps in changing the code so that it performs better without any previous errors.

Benefits of Refactoring are:-

- i. Highly reusable code
- ii. Easy up gradation an maintenance
- iii. Definitive solution for a common problem which eventually saves time
- iv. Abiding to the known code standards
- v. Trustworthy code
- vi. Number of defect counts are low
- vii. Better understand ability and readability
- viii. Automated testing can be carried out easily
- ix. Generation of efficient and effective code

1.7 Testing Process

Software technologies produces software based systems to grab and involve fresh industry needs and to provide quality, bug fixing is an important process in development. Due to the time-to- market, lack of understanding, and the developers' experience, developers cannot always follow standard designing and coding techniques, i.e., design patterns ^[5]. Test suites are often simply test cases that software engineers have previously developed, and that have been saved so that they can be used later to perform regression testing ^[6]. Re executing all the test cases require enormous amount of time thus make the testing process inefficient. Anti-patterns checking is an essential process in any software development procedure. Normally it is defined as recurring, bad designing in linking which could rise to the loose effects in software systems because it is a big difficulty in understand ability and maintainability of whole development. Due to these issues, testing of the systems is increased and hence the cost increase too. In related research consider testing cost the number of test cases that satisfy the minimal data member usage matrix (MaDUM) and studied four Java programs, Ant 1.8.3, ArgoUML 0.20, CheckStyle 4.0, and JFreeChart 1.0.13 which shows that unit testing increased due to availability of anti-patterns. Previous research also introduced some refactoring actions which applied to classes participating in anti-patterns which reduce cost of testing. In our proposed work we are enhancing the test cases for the similar work with eclipse, Web browser as additional Java programs. Effective testing of software is necessary to produce reliable systems. This is true in practice since static verification techniques have their own limitations. We are going to perform the automation testing for the Graphical User Interface. The first phase include the connecting the path of the given GUI and the XMI based on ArgoUML tool. Unified modeling language will be used for creating XMI files. Our GUI will contain the event if any of the events is clicked than the flag value will be set. Now we will convert the GUI file to the XMI code with all the flag set value for finding of anti-pattern credentials. Automation in testing for finding anti-pattern and automation of refactoring are the primary concerns which could be fulfil by judging the parameters which are responsible for anti-pattern and by providing refactoring of this anti-pattern could be avoided.

Aminata Sabané, Massimiliano Di Penta, Giuliano Antoniol, Yann-Gaël Guéhéneuc (2013) “A Study on the Relation between Antipatterns and the Cost of Class Unittesting” referred the anti-patterns as poor design choices which are recurring in nature. They affect the systems in a negative fashion in terms of maintainability and understandability. In this paper we study the anti-pattern effect on testability and test cost in particular^[7]. We consider as (upper bound) indicator of testing cost the number of test cases that satisfy the minimal data member usage matrix (MaDUM) criterion proposed by Bashir and Goel. A study was carried out on four java systems which are Ant 1.8.3, ArgoUML 0.20, CheckStyle 4.0, and JFreeChart 1.0.13. They said that unit testing requires a number of test cases a number higher than the test cases for non-antipattern classes and that anti-pattern classes should be thoroughly tested because there are more chances of defects in them. They found the number of test cases required for class unit testing in anti-patterns is higher and what is the cost benefit trade-off when we are giving priority to the testing of classes participating in anti-patterns. Their paper has discussed the estimation of cost of class unit testing based on the test cases generated by the (MaDUM)^[8] minimal data members’ usage matrix technique. This technique does not require documentation for design. This paper computed the number of test cases for four types of java systems. They described MaDUM as an $nf.nm$ matrix where nf is the number of fields and nm is the number of methods in a class. They have discussed the strategy to test classes using MaDUM by first categorizing the methods into Constructors (c): class constructors; Transformers (t): methods that alter the state of one or more fields; Reporters (r): methods that return the value of a field; others (o): methods that do not fit in the categories above^[9]. This study on this paper has been carried out for the basic purpose of estimating the cost of class unit testing for the classes participating in anti-patterns. The four java systems used in this paper have different properties and each one is used for its specific feature like argoUML is used as an open source tool for generating uml diagrams, CheckStyle is a tool used for developing java programs, it also checks that whether the code adheres to the

mentioned coding standards, JFreeChart is a java class library for generating charts in java based programs. Then they talked about refactoring and reducing the cost of testing. They said that testing cost can be reduced by performing extract method refactoring and for this they gave an example of sequence of statements being repeated in four methods with only a little variation. These repeated sequences increase the number of test cases required for testing the classes according to the MaDUM strategy. This refactoring technique in turn reduces the number of test cases of the class refactored. Overall their study concludes and supports the fact that classes participating in anti-patterns require a higher cost value to test than other classes and higher priority should be given to the classes participating in antipatterns as it is more cost effective because of the defects they contain and even refactoring should be done in a cost effective manner. Finally their future work included to extend the empirical study to more programs, feasibility for automatic detection and refactoring of antipatterns and in turn reducing the testing cost and extending the evaluation to class integration and related test integration.

A.V.K. Shanthi¹ and G. Mohan Kumar² **Research Scholar, Sathyabama University, Chennai, India.**² **Principal, Park College of Engineering, Coimbatore(2012), “Automated Test Cases Generation from UML Sequence Diagram”**, focused on testing software at early stages so that it is easier for software testers at the stages later to come. Automated testing can be carried out by creating a test case and this paper’s priority is to generate the test cases by use of UML Sequence diagram using Genetic Algorithm. They used the UML sequence diagram in this paper for specifying the design and even implemented the idea of test case automatic generation for software. They even proposed an approach for prioritization of test cases which were generated by the UML sequence diagrams using genetic algorithm and sequence dependence table (SDT). This paper gave the methodology for test case generation and described sequence diagram as an interaction kind of diagram which illustrates how different processes communicate or operate with each other and in what order. Sequence diagram in the arrangement of time sequence shows the object interactions. It also shows all the objects and classes involved in a certain program and the sequence of messages exchanged between the objects which in turn carry out the functionality of the program. Sequence diagrams are generally

associated with the use case realizations in the logical view of the system which is under development. Then they gave the idea as to how to generate the test cases using the sequence diagrams by extracting the required information from the sequence diagram and the extraction can be through a parser written in java which will extract all the information required from the file. Then based on the information extracted a (SDT) Sequence dependency table is generated. The SDT helps in generating the test path and by applying the GA the most prioritized test case is generated. Finally the conclusion was that in this paper they portrayed the development of test cases using the uml sequence diagrams using genetic algorithm. Their aim was to identify the fault in the program when implemented thus in turn reducing the testing efforts. It works at reducing the development time, improve the quality of design and find faults at an early stage. Their future scope mentions the generation of automatic tool using this approach. The automatic tool can reduce the development cost and even improve the software quality.

Benoit Baudry, Yves Le Traon, Gerson Sunyé, Testability Assessment, IWoTA Proceedings IEEE pp-70-80 (2004), “Improving the Testability of UML Class Diagrams” introduced the object-oriented testing efforts in their research work. By this they aimed at recognizing the different testability anti-patterns that deteriorate the software testability and then using uml for better testability and make implementation testable in a better way. The basic criteria they carried out in this study was to identify the key elements of the class diagram that make the testing of the part to be implemented ,difficult. In the class diagram, testability anti-patterns are identified and then worked upon for better testability and implementation. They defined all kinds of anti-patterns that could exist in uml class diagrams and then a testing criterion was carried out to see through the existence of the anti-patterns in the uml class diagram. They took the example for university library management system for the same. The class diagram they created for the same contained mainly all the features of the object –oriented programming like inheritance, abstract classes, association and usage dependency relationships. Their testing criterion detected the anti-patterns hard to find in a class diagram hindering the implementation. They said that in object-oriented systems classes are usually dependent on each other for their processing. Thus they calculated the testing effort and complexity in testing the anti-patterns

deduced from the class diagram and after detecting they worked on improving the design. They said the design can be improved in two ways .first by reducing the number of anti-patterns in the class diagram by reducing the interdependencies between various classes or either by reducing the complexity because of polymorphism. They summarized their work by saying that testing cost can be reduced by taking testing issues into account in design phase and said that doing this reduces the testing efforts in the implementation phase. They calculated testability as the effort required to build the test cases that in turn cover all interactions between the objects and finally to improve the testability they introduced UML stereotypes to add information about relationships in different class diagrams which are create, use, use_def, use_consult.

Zoltán Ujhelyi, Ákos Horváth, Dániel Varró, Norbert István Csiszár, Gábor Szoke, László Vidácsy, Rudolf Ferenc(2011), “Anti-pattern Detection with Model Queries: A Comparison of Approaches” stated that program queries have played a major role in software evolution tasks like automating the finding of anti-patterns, analysis of impact and program comprehension. They have made use of java using eclipse platform to represent program models which are processed by three model query techniques. Then they carried out comparison of the three Automation in testing for finding anti-pattern and techniques on the source code of 17 java projects by using the refactoring operations queries located in different usage profiles .They introduced refactoring which works with the goal of changing the source code of the program without making any changes to the behaviour of the program. They have conducted a detailed comparison of (1) memory usage in different ASG representations (dedicated vs. EMF) and (2) run time performance of different program query techniques. For the latter, we evaluate four essentially different solutions: (i) hand-coded visitor queries (as used in Columbus), (ii) queries implemented in native Java code over EMF models, (iii) generic model queries following a local search strategy and (iv) incremental model queries using a caching technique ^[10].they built an abstract semantic graph (ASG) from the source code as a model and then it is stored in in-memory representation. They stored the various models used in model-driven engineering (MDE) and manipulated them with accordance to the meta-modeling framework which was eclipse modelling

framework(EMF) which generated systematic API, model manipulation code, layer which is persistent in XMI, simple viewers and editors automatically from the domain metamodel. They captured anti-patterns as model queries using a high-level, declarative graph pattern based query language ^[11].they selected six types of anti-patterns and then formalized them as model queries. They then worked out the strategy by first managing the models of java programs then definition of model queries using graph patterns and in the end implementing the program queries. Their evaluation process consisted of measuring the performance using the program models from the java based projects by description of the measurements, detailed results usage profiles and evaluation of the results. Finally they concluded that they evaluated different approaches of query to find the anti-patterns for refactoring the code for java projects. They gave a way to faster implementation and a way that was easy to experiment with the queries.

Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aimeur (2012), “Support Vector Machine for Anti-Pattern Detection” gave the introduction as to about what are anti-patterns. Anti9 patterns are recurring design problems that occur when we are in the designing phase of a program. These hinder the understand ability of the code. Anti-patterns should be detected and removed at an early stage so that to reduce the cost and the time in which we deliver an error free system. This paper introduced the methods for detecting various anti-patterns in a system. They introduced an approach called SVM-detect to detect the anti patterns which is based on machine learning. They also compared the accuracy between SVMDetect and DETEX and that SVMDetect can find more anti pattern occurrences than DETEX. They also highlighted the limitations of approaches that help detect anti-pattern like they require extensive knowledge of anti-patterns, they have limited precision and recall and they cannot be applied on subsets of systems ^[12].They used SVMDetect to detect four anti-patterns: Blob, Swiss Army Knife, Spaghetti Code, and Functional Decomposition. The first step in the process was to define a training data set (TDS).It constituted the set of classes derived from object oriented system. The second step was training the SVM classifier which indicated whether a class Xi is a respective anti-pattern or not. It divided the classes into two different groups i.e. Anti-pattern or not anti-pattern e.g.

blob or not-blob. The third step was about construction a dataset DDS and the occurrences of an anti-pattern. The objects they used in their study were ArgoUML v0.19.8, Azureus v2.3.0.6, and Xerces v2.7.0 which are three open-source java systems. Their study observed that DETEX cannot detect as many ant-patterns as SVMdetect. According to their observations DETEX detected 25 anti-patterns and SVMDetect detected 40 anti-patterns in ArgoUML and in total SVMdetect detected 143 blob occurrences whereas DETEX detected 102 blob occurrences ^[13]. They concluded that anti-patterns can hinder both development as well as maintenance activities. SVMDetect could overcome the limitations previously mentioned by a technique based on support vector machine (SVM).Experiments have been shown how SVMDetect is used on three systems (ArgoUML v0.19.8, Azureus v2.3.0.6 and Xerces v2.7.0) and four anti-patterns (Blob, Swiss Army Knife, Functional Decomposition and Spaghetti code).Their future work includes the use of SVMDetect in real world environments and impact of the quality of feedback on SVMDetect results.

Zhaogang Han, Peng Gong, Li Zhang, Jimin Ling, Wenqing Huang (2013), “Definition and Detection of Control-flow Anti-Patterns in Process Models “introduced the antipatterns as “design flaw that lead to errors for a process model” ^[14]. They also introduced an approach that dealt with detection of anti-patterns which could be both system designed as well as user defined. Firstly they converted the process model into a refined process structure tree (RPST) by an algorithm which is known as cycle equivalence algorithm. Then they designed a control- flow anti-pattern description language (CAPDL) is defined and an algorithm regarding CAPDL is proposed. They observed that control- flow anti pattern was the most common anti-pattern in process models which introduced errors like lack of synchronization and deadlock. The first step they introduced in the process was “process model pre-processing” which is further divided into two steps: The control flow structure of a process model is converted into a workflow graph and then the cycle equivalence algorithm is used to convert the workflow graph into RPST.Worflow graph used BPMN^[15] notations like parallel branch, exclusive choice,synchronization,simple merge etc,to represent elements in the workflow graph and symbols “AND-Split”, “AND-Join”,”XOR-Split” and “XOR-Join” are used respectively. Then a “Refined

Process Structure Tree” is formed which is used to enhance the efficiency of control flow anti-pattern detection. This approach was composed of nodes and canonical single entrance single exit process blocks in workflow graph. Leaf nodes in RPST corresponded to nodes in the workflow graph and non- leaf nodes corresponded to process blocks. Then a language was introduced known as CAPDL (Control-Flow Antipattern description language) which worked on a basic concept of predicates. Various rules were defined which were composed of series of predicates and a concept with a higher level than a rule was an anti-pattern because anti-pattern could be formed from one or more rules. Concept of module was also introduced to CAPDL so as to reuse existing definition of antipattern and rule. Therefore by using predicate, rule, anti-pattern and module, CAPDL has introduced concise control- flow anti-pattern definition approach. In order to verify the effectiveness of the CAPDL an anti-pattern detection experiment based on 278 process models was initialized in which four frequently occurring anti-patterns were detected. Finally the concluded that researches on control- flow anti pattern is still in its preliminary stage and anti patterns description approaches need to support more anti-patterns. Their future work would include the extensibility of the ability of the workflow graph and CAPDL and improvement in the existing algorithm to enhance the recall rate, search efficiency and precision.

Abdou Maiga¹, Nasir Ali¹, Neelesh Bhattacharya, Aminata Saban´e, Yann-Ga¨el Gu´eh´eneuc, and Esma Aimeur(2012), “SMURF: A SVM-based Incremental Antipattern Detection Approach” introduced anti-pattern occurrence reason as lack of understanding, time pressure, lack of communication or lack of skills making the source code difficult to understand. They introduced SMURF that could be used both in intra-system and inter system configurations. Anti-patterns are generally result of misuse of object oriented or design patterns. They introduced spaghetti code related to classes without object-oriented structure so they do not exploit polymorphism and inheritance so cannot be used by developers and blob as a large class that controls that controls the behaviour of a system. They also introduced the four limitations of approaches used to detect anti-patterns which are they require extensive knowledge of anti-patterns, they have limited precision and recall, they are not incremental and they cannot be applied on subsets of a system ^[16]. This paper proposed their approach

SMURF to detect anti-patterns using SVM and practitioner's feedback. They studied four anti-patterns which are blob, spaghetti, functional decomposition and Swiss army knife and perform more than 300 experiments to compare the results of DETEX AND BDTEX with the results of SMURF. Their observation concluded that SMURF performed better than DETEX as DETEX could detect 102 blob occurrences whereas SMURF detected 143 blob occurrences. Then they compared SMURF and BDTEX. BDTEX is a probabilistic approach which provides that a class is an occurrence of anti-pattern ^[17]. This comparison concluded that BDTEX contains a high level of uncertainty thus BDTEX has high recall but bad precision. SMURF overcame all the four limitations of previous approaches used to detect anti-patterns. They also showed that accuracy of SMURF was greater than that of DETEX and BDTEX when detecting anti-patterns on a set of classes or on the entire system. Finally they said that SMURF accuracy improves when using practitioner's feedback. Their future work included using SMURF in real world environments and further they would use other systems and anti-patterns. Another study would include evaluation of the impact of feedback on SMURF results.

Chapter 3

SCOPE OF THE STUDY

This study aims at automatic detection of anti-patterns which are recurring design problems which deteriorate the quality of the system .They increase the overall cost of the system and in our study we work at reducing the overall cost and time of the execution of a system.Antipattern detection has been a major concern in the programming world because it directly affects the cost and the time of the execution process and various techniques have been devised for detecting and removing the anti-patterns. This study helps in investigating the impact of anti-patterns on classes in object-oriented systems and as a result helps in carrying out the testing process in a better way. The study also helps in bug fixing, provide quality product, cost cutting, automate testing for GUI and carrying out faster execution of code.

4.1 Problem Definition

Our concept 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 and aims at optimized unit testing for anti-pattern detection.

4.2 Objectives

Our research will start with study of anti pattern detection mechanisms like smurf and decor. We have processed detection systems to find techniques with better accuracy and quality. Some of the objectives which need to be fulfilled are given below.

- 1.) Find optimized testing technique for detection of anti patterns.
- 2.) Providing solution for anti pattern by testing approaches.
- 3.) Detecting the anti-pattern using testing approaches.
- 4.) Refactoring the code and making it anti-pattern free.

Chapter 5

METHODOLOGY

In our research, we focused on the automatic testing for finding of anti-patterns. Whole process is divided in two parts. Part one consists of XMI read and collects required information from it and second part is used to collect information and a java based tool will use this information to test given code for anti-patterns. An additional feature added to the program is uploading a text file and converting the values in it. First of all UML diagram is created for the program, which is created using Argo UML. The created UML is transformed into XMI; eclipse testing module uses this XMI as an input. Then we give file path to XMI. We run our program when we have finally provided the required input. XMI file is read by the program from the particular location. A DOM Parser object is created in order to read XMI which is used to parse the file. After the file is parsed, the file is stored in a Document object in the structure of a document.

Once we get the required knowledge from the XMI, eclipse (java) testing module goes through the major code and locates the anti-patterns in it. Different types of anti-patterns exist, and in our study we have tried to discover some most familiar anti-patterns which are unused data, blob and cryptic code anti-patterns that deteriorate the quality of the code. We created a tool that will automate the process of finding the mentioned anti-patterns. The information we got from XMI file is used for the detection of unused code and cryptic code.

When the code is executed there are certain fields which were defined but not used, these types of anti-patterns are well-known as unused code anti-patterns. This anti-pattern exceeds the program execution time limit and in turn consumes needless memory space which may deteriorate the quality of the code. XMI file's data supplies us with the information about the fields defined in the class. Our program looks up these fields to locate their values, if the field is found idle during the execution of program, that field is struck as unused. The program searches the whole code and provides the entire list of unused fields in the program.

When multiple functions are performed by a single function or multiple responsibilities are assigned to single function, blob anti-patterns are identified. It is also famous as God object. Method call stack is printed to locate blob anti-pattern. Method call stack displays the calling situation in the program. It is expected that a single function performs one single task. If some method is called time and again, it is most probably a blob pattern. Inspection of the call stack is necessary to detect blob kind of anti-patterns.

Cryptic code is a type of anti-pattern which uses abbreviations for declarations or fields instead of suitable naming. This makes intricate to identify what types of values the field is referring to and makes it complex to modify or reuse the code in future. This makes it hard and burdensome to reuse the code or alter the code in future. To locate cryptic code our program describes the least length required for an ideal naming of the field. The fields having characters in a lesser limit than the precise given length are considered as cryptic code anti-patterns and after searching the whole program the detected anti-patterns are added to the list which will print them.

After locating the anti-patterns, code is refactored to eliminate the anti-patterns from the code. This makes the operation of the code faster and will enhance quality of the code. Refactoring of the code is carried out in three different stages. During each stage one of the anti-patterns is isolated from the code.

After the refactoring process is complete, refactored code again goes through the testing tool to ensure that there are no anti-patterns present. Now we compare the implementation time and memory usage of refactored and non-refactored code. The following flowchart in Figure 5 depicts the methodology briefly.

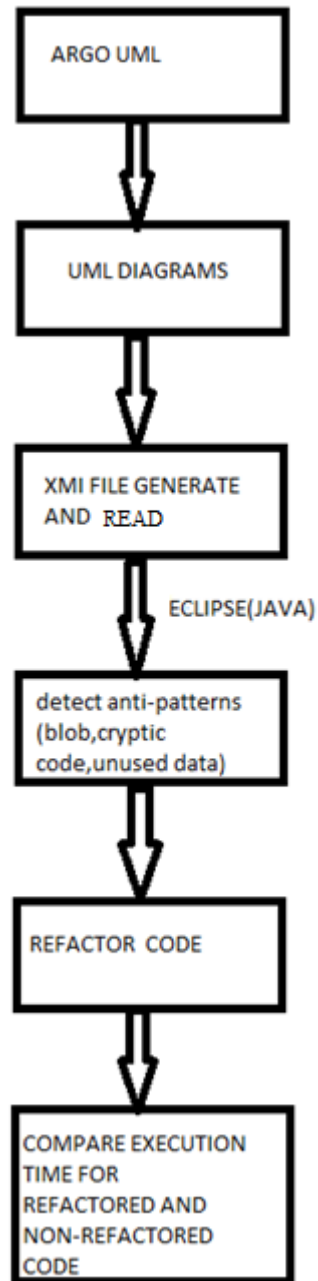


Figure 5: Flowchart to depict methodology

6.1 STEPS TO DEMONSTRATE METHODOLOGY

- i. First of all we will install ArgoUML and open it as shown in Figure 6.

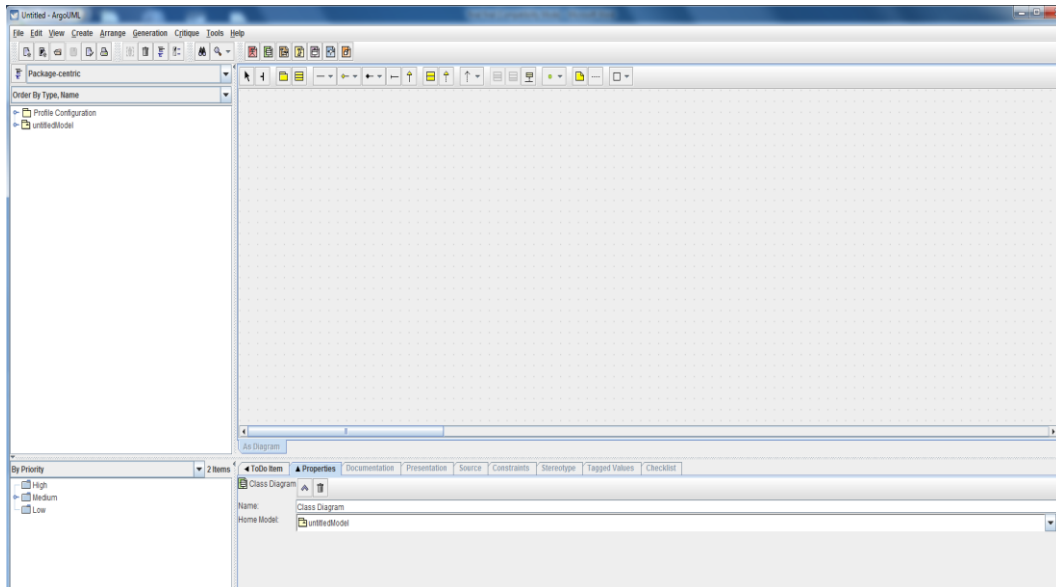


Figure 6: ArgoUML interface

- ii. Go to file menu and click on import sources option as shown in Figure 7.

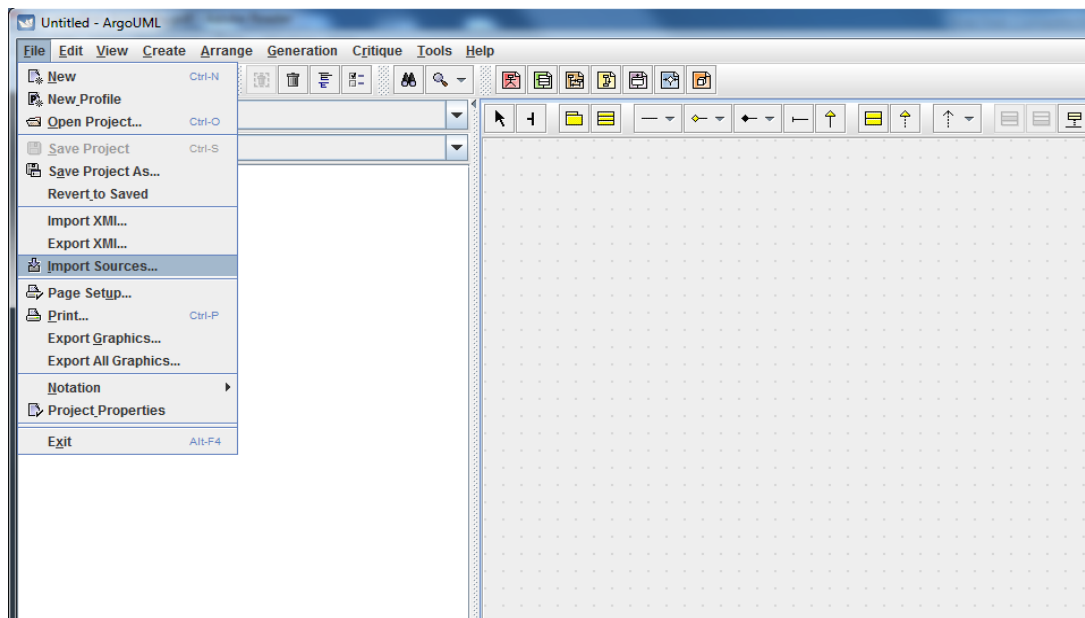


Figure 7: ArgoUML import sources

- iii. Import sources window appears. Browse the xmi.java file from the location where your java xml metadata exchange file exists as depicted in Figure 8.

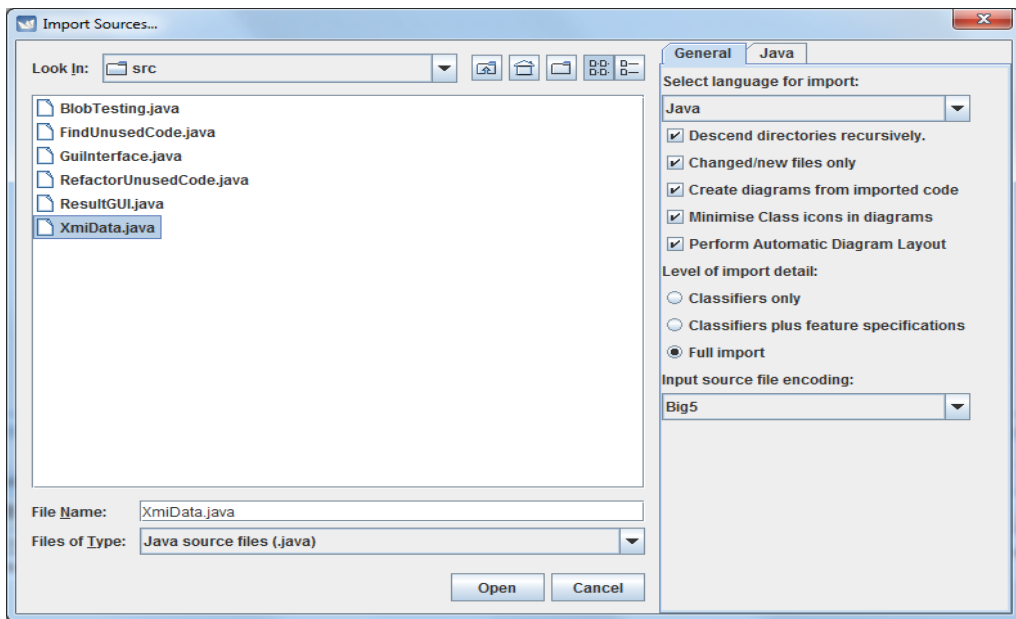


Figure 8: Selecting of java file

- iv. Click on Untitled model on the left side of the window in the configuration section and right click on xmi data class type.
- v. Click on add to diagram option as shown in Figure 9.

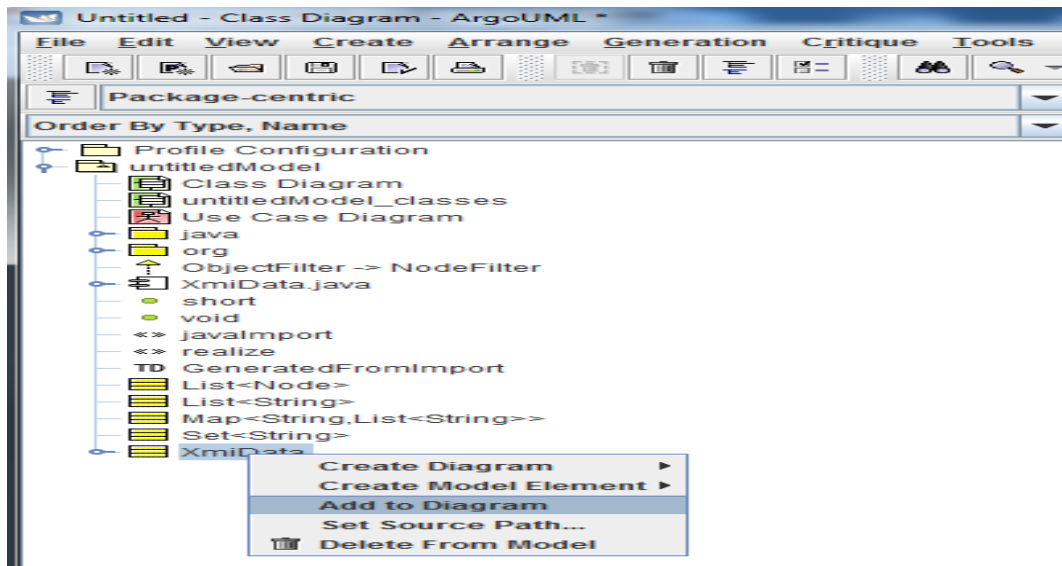


Figure 9: Selecting add to diagram

- vi. Move the cursor to the right where UML diagrams are created a black cross sign appears.
- vii. Left click once and the UML class diagram for xmi data is created as shown in Figure 10.

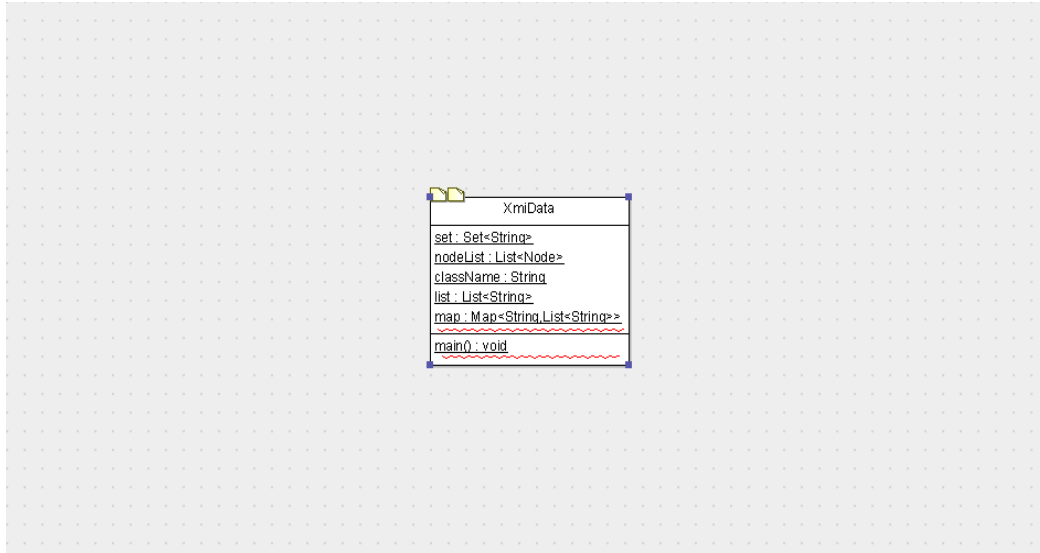


Figure 10: Creation of Uml diagram

- viii. Now again go to the file menu and click on export xmi option.
- ix. Save the xmi file created at any location you desire for further access as shown in Figure 11.

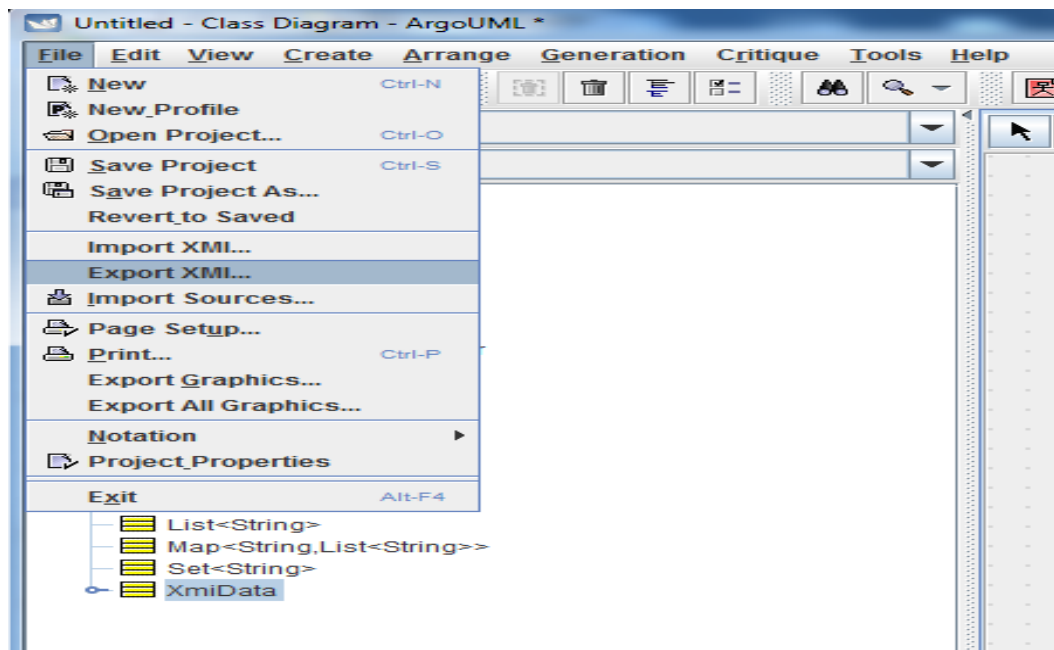


Figure 11: Export XMI

- x. Now we finally implement our methodology and run our program in eclipse.
- xi. First of all a user interface appears which asks for an input number for conversion from either feet to inches, inches to feet or centimetre to inches as shown in the Figure 12.

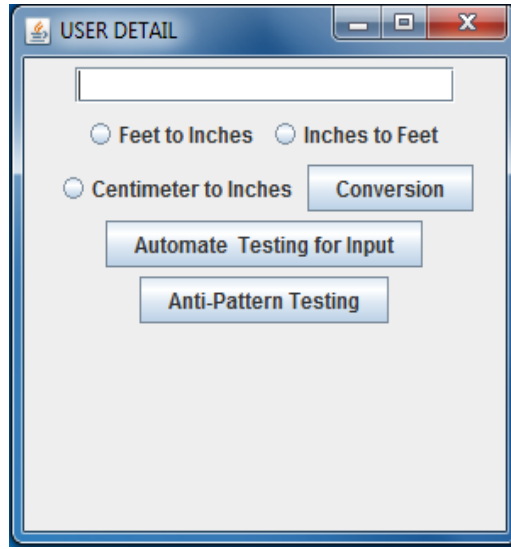


Figure 12: User detail interface

- xii. We can enter any numeral we wish to enter in the text box provided and choose any of the three radio buttons according to the type of conversion we want as shown in Figure 13.

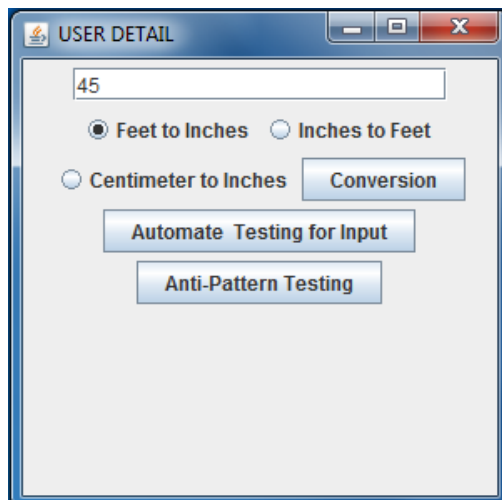


Figure 13: enter a valid integer value in the textbox

xiii. Feet to inches conversion. This is depicted below in Figure 14.

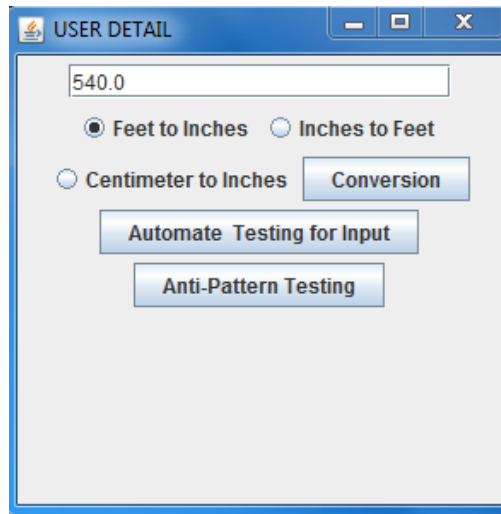


Figure 14: Conversion from feet to inches

xiv. Inches to feet conversion. This is shown below in Figure 15.

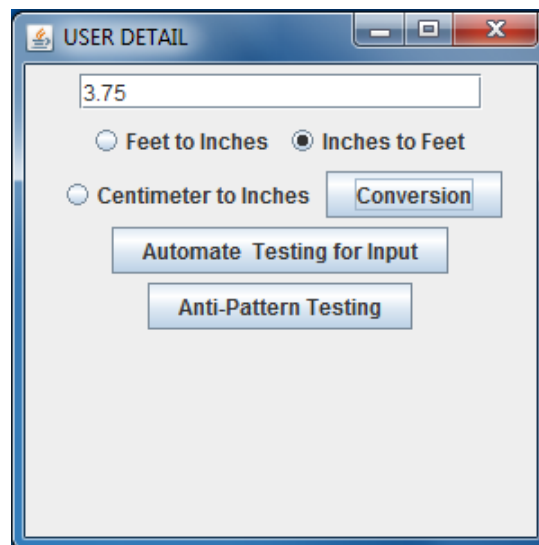


Figure 15: Conversion from inches to feet

- xv. Centimetre to inches conversion. This is depicted below in Figure 16.

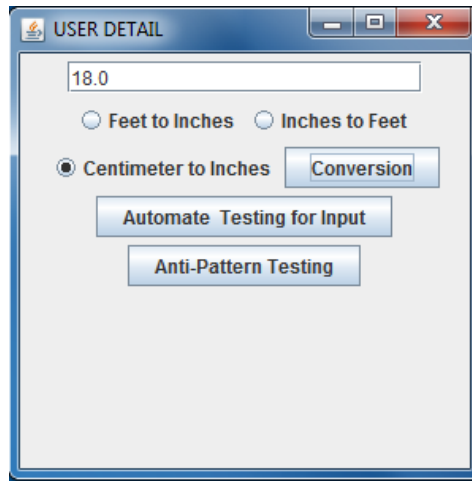


Figure 16: Conversion from centimetre to inches

- xvi. When we click on Automate testing for input command button a new dialog box appears “Source Code Checked” as shown in Figure 17.

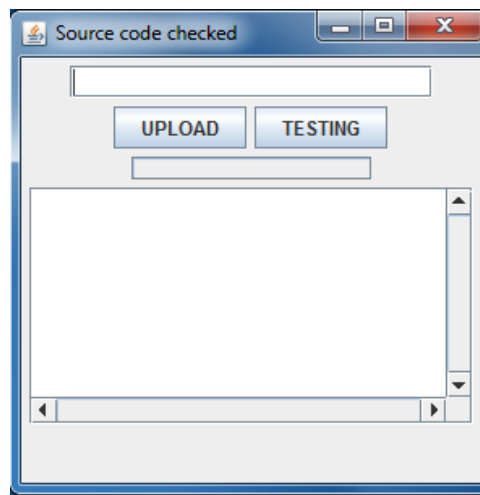


Figure 17: Source code checked window

- xvii. We can upload any text file containing the numeral you want to test the conversion for. In this program a file named “upload” is created which is located on the desktop containing the numeral for conversion.
- xviii. By clicking on Upload combo box a window appears named open where we can select the file we want to test i.e. upload file located on the desktop.

xix. After selecting the file click on open as shown in Figure 18.

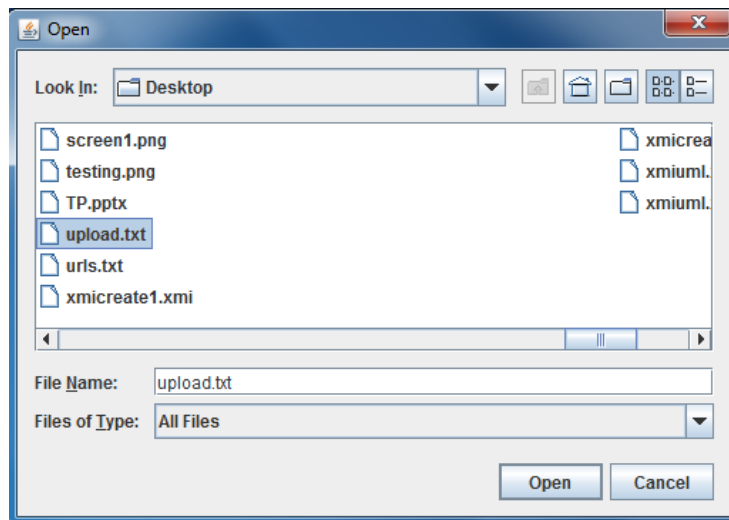


Figure 18: Select the text file you wish to upload

xx. When we click on open the path of the file selected appears in the empty text box on the source code checked dialog box as shown in Figure 19.

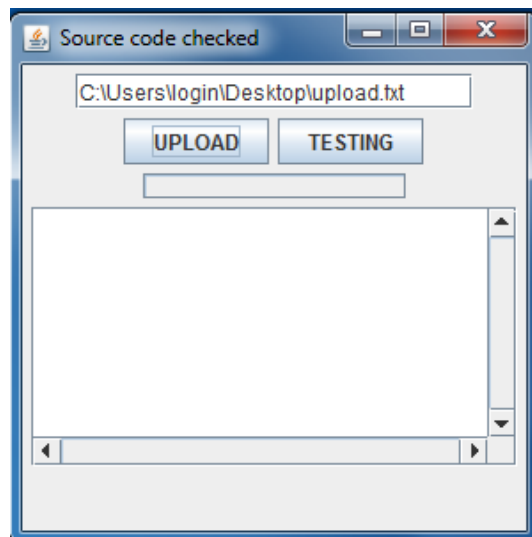


Figure 19: After selecting click on upload

- xxi. Next click on testing to automatically generate the three conversions done previously as shown in Figure 20.

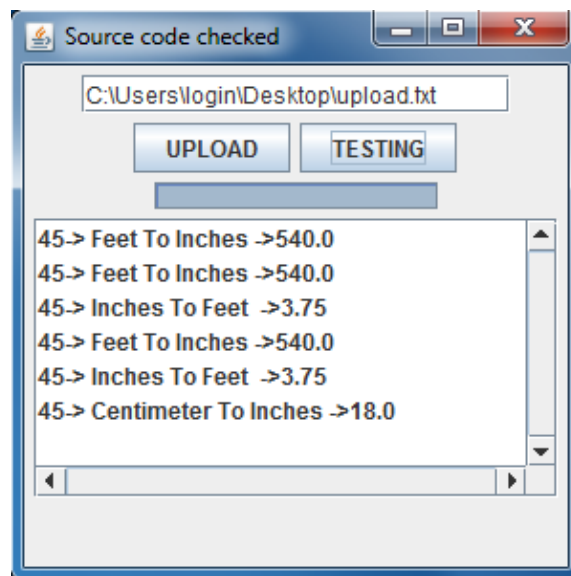


Figure 20: After uploading click on testing

- xxii. Now close the source code checked dialog box and on the “user detail” dialog box click on Anti-pattern testing combo box.
- xxiii. A message box will appear which will tell you about the memory used by the entire conversion program containing anti-patterns as demonstrated in Figure 21.

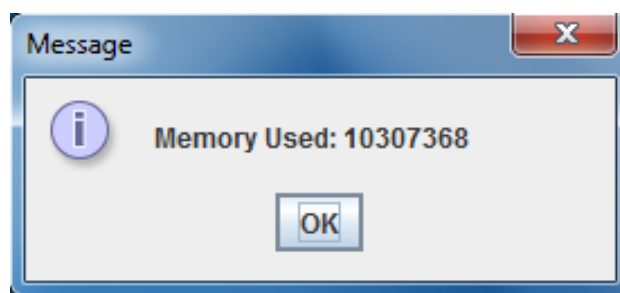


Figure 21: Memory Used message box.

xxiv. When we click on ok a new window “Antipattern testing tool” appears as shown in Figure 22.

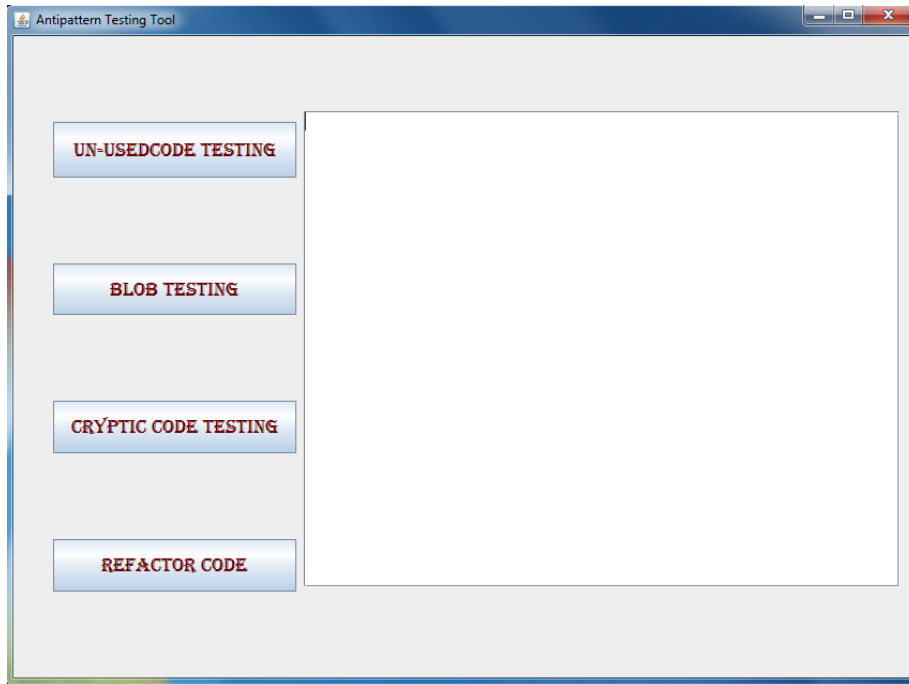


Figure 22: Antipattern testing tool window

xxv. Now we can individually check for each type of anti-patterns among the three anti-patterns namely blob, unused code and cryptic code.

xxvi. When we click on UN-USED CODE TESTING, all the unused variables and parameters get listed along with the ones used as shown in Figure 23. The listed data includes:-

- a) Used variables in listener class
- b) Unused variables in listener class
- c) Used variables in GUI class
- d) Unused variables in GUI class
- e) Used variables in GUI Interface class
- f) Unused variables in GUI Interface class

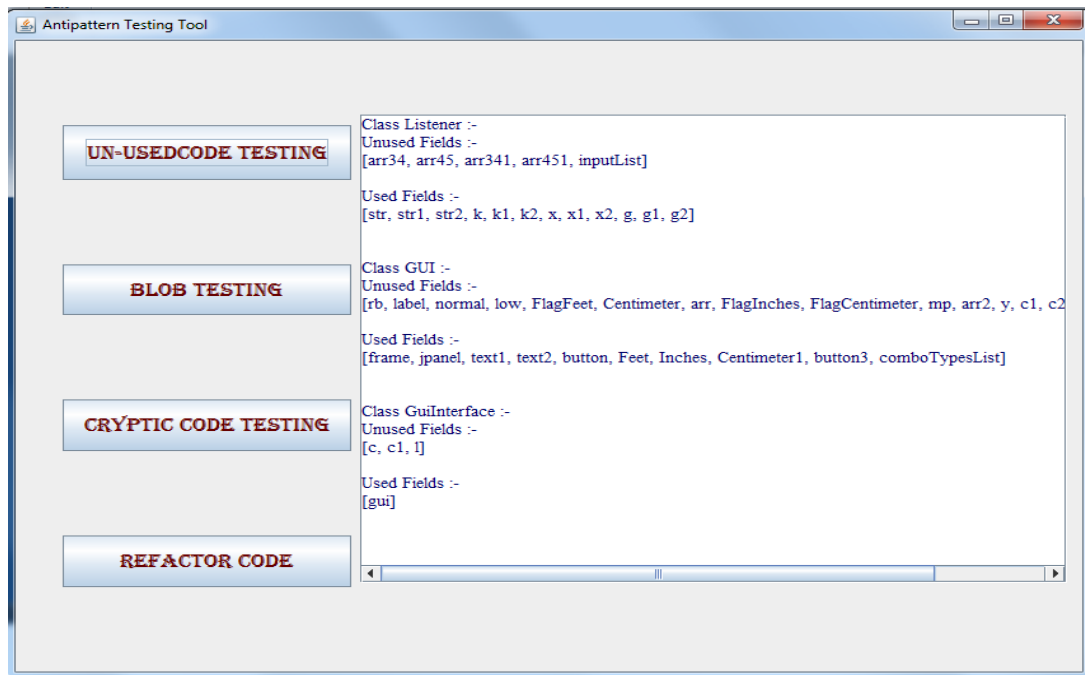


Figure 23: Un-usedcode testing

- xxvii. By clicking Blob Testing combo Box all the classes having more than 3 responsibilities or a single function performing multiple functions is depicted. All the classes and the number of responsibilities they perform are shown in the blank text box alongside as shown in Figure 24.

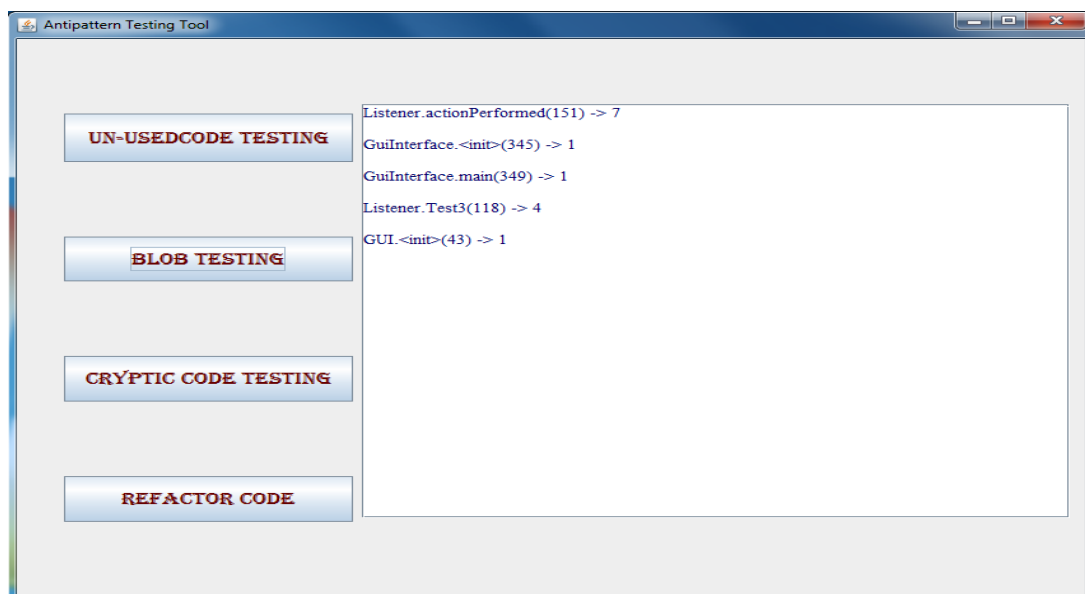


Figure 24: Blob testing

- xxviii. Then by clicking on cryptic code testing combo box the variables names using abbreviations not understandable by the user are highlighted as shown in Figure 25.

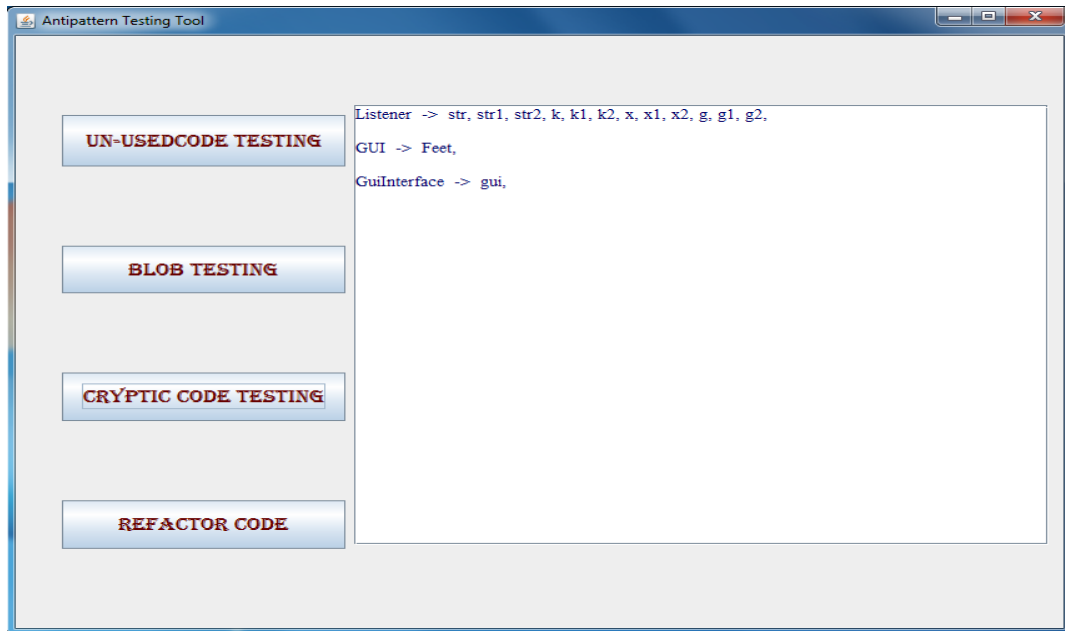


Figure 25: Cryptic code testing

- xxix. Once the anti-pattern detection is over we can move on to refactoring of code by clicking on the refactor combo box.
- xxx. Left click on refactor code combo box.
- xxxi. A new window appears where we can refactor and improve our code design as shown in Figure 26.

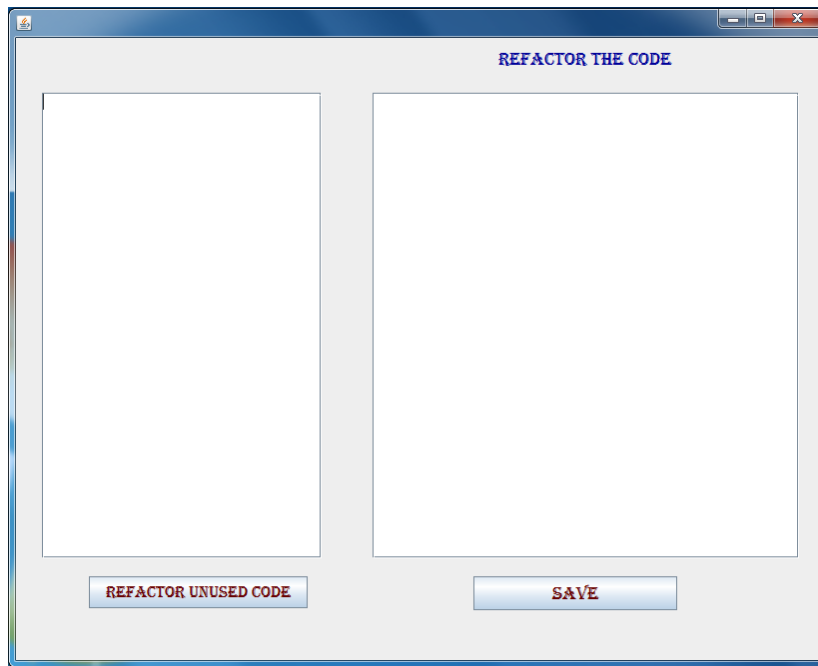


Figure 26: Refactor unused code

- xxxii. Click on Refactor unused code combo box.
- xxxiii. Automatic refactoring of unused code is done.
- xxxiv. On the left side of the window unused code is shown and on the right side of the window automatic refactoring of unused code is shown in Figure 27.

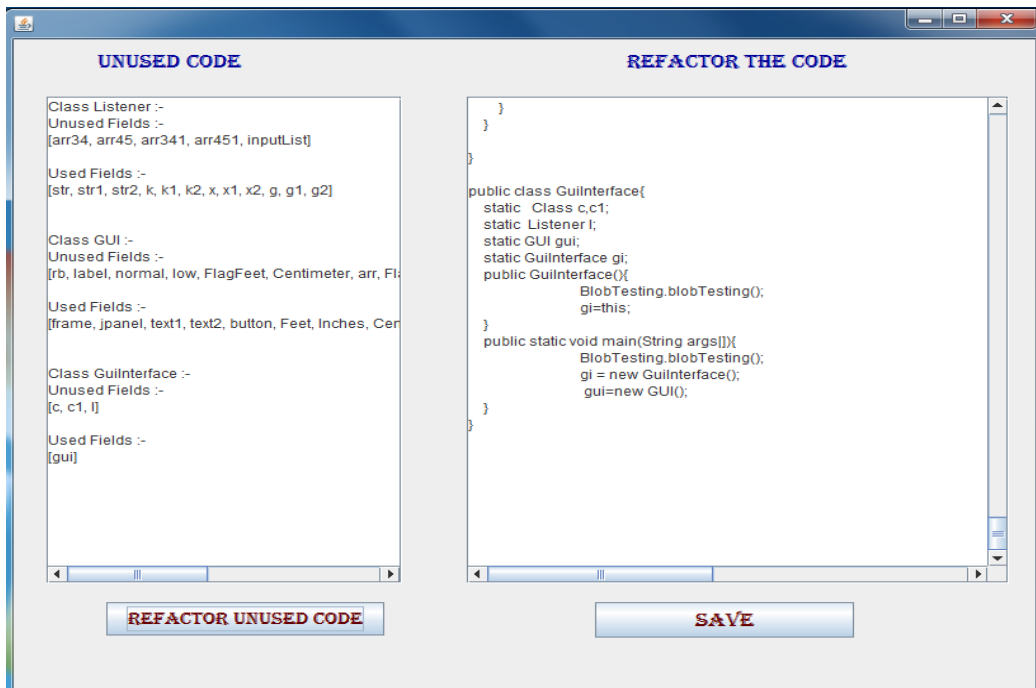


Figure 27: Refactoring of unused code

- xxxv. Click on save button and a copy of the refactored code is saved in the location specified in your code as shown in Figure 28.

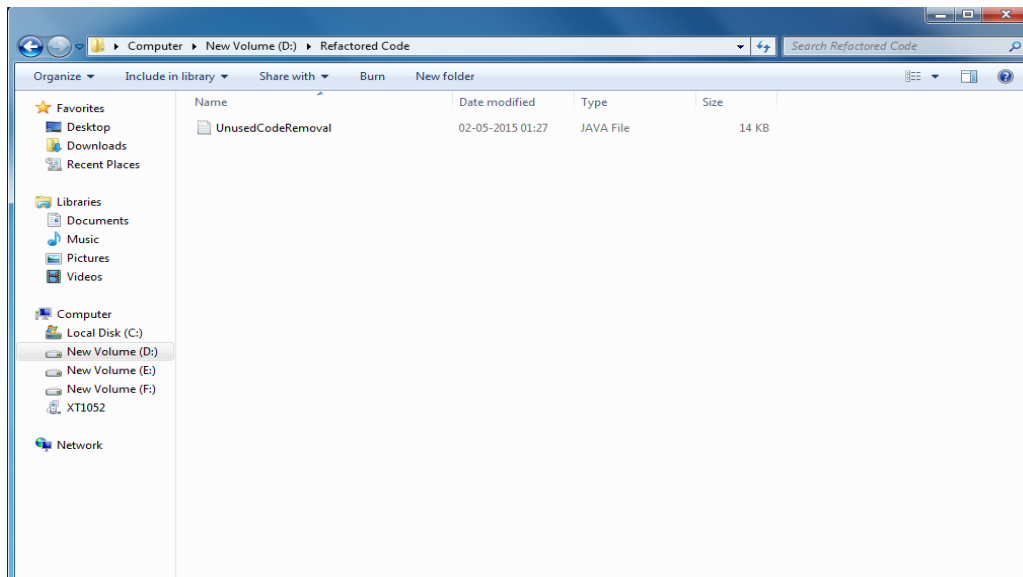


Figure 28: Unused code removal file

- xxxvi. However the refactoring of cryptic code and blob anti-patterns has to be done manually.
- xxxvii. In case of cryptic code anti-pattern we have to replace the detected abbreviations with proper, easily understandable names.
- xxxviii. In case of blob anti-pattern we have to manually allocate or distribute responsibilities of functions or classes having more than 3 responsibilities to other classes or functions with less number of responsibilities to maintain a balance.

6.2 MEMORY AND TIME COMPARISON

- i. After completion of the implementation we can compare memory and time consumption before and after the removal of the anti-patterns.
- ii. Time comparison as depicted on the eclipse console is shown in Figure 29 for input value 12.

```

GuiInterface [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (02-May-2015 6:42:45 pm)
final time for manual input is: 24028158
final time for manual input is: 24903581

final time for manual input is: 25779004

Size of array:1
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
final time for file input is: 894444

Size of array:2
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
final time for file input is: 1280754

Size of array:3
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
final time for file input is: 1814703

```

Figure 29: Console showing time for manual and automatic input

iii. The graph in Figure 30 shows how the execution time decreases after the removal of anti-patterns.

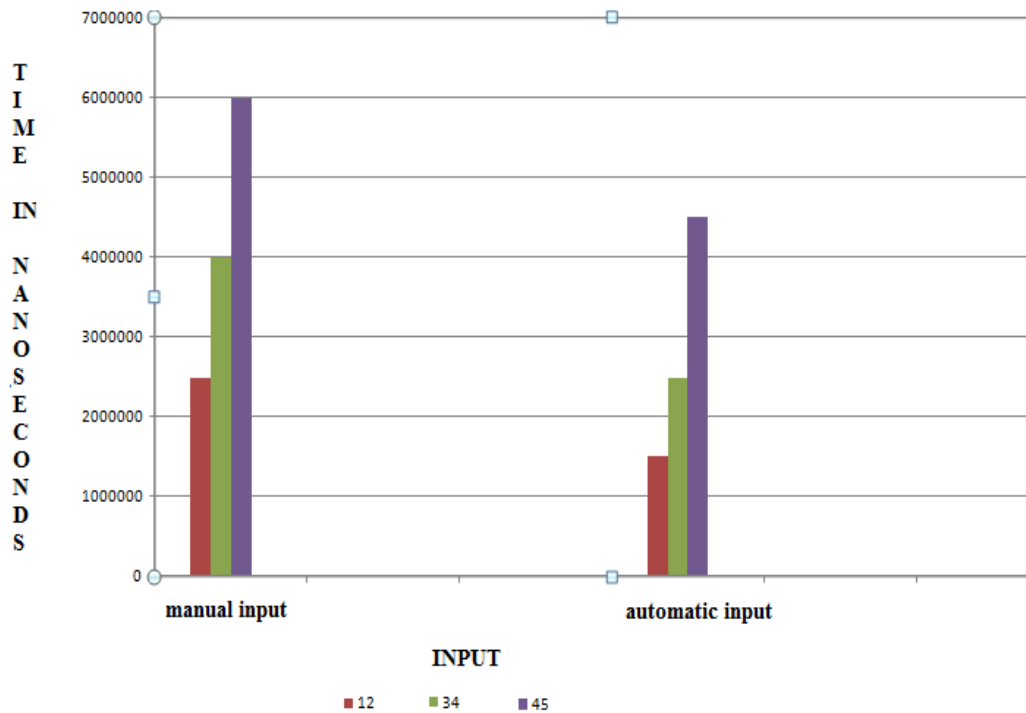


Figure 30: Execution time comparison

- iv. Comparison for memory before and after the anti-pattern removal in depicted in graph Figure 31.

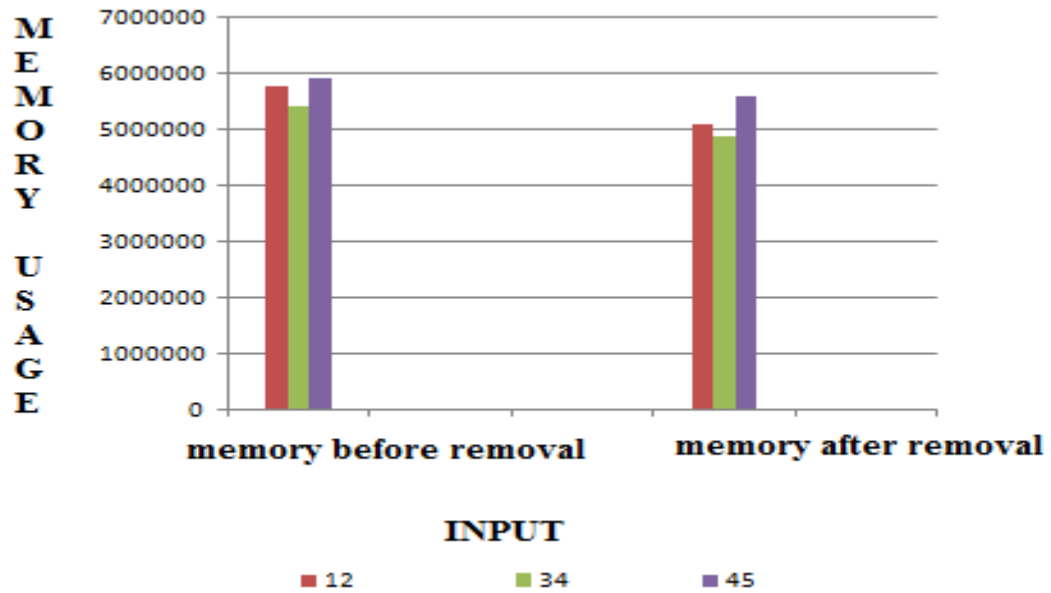


Figure 31: Memory usage comparison

CONCLUSIONS AND FUTURE WORK

Poor design choices that are induced to make object oriented systems harder to maintain are identified as anti-patterns which amplify the programming cost, increase the execution time and diminish the understanding of any program. It makes the source code complicated to understand and hamper the progress and the maintenance activities. Detecting anti-patterns on system's subset can lessen cost, labour and assets to be used. Various techniques are budding that discover anti-patterns stirring in a system. Locating and elimination of anti-patterns is a vital process in any software development system.

In related research consider testing cost the number of test cases that satisfy the minimal data member usage matrix (MaDUM) and studied four Java programs, Ant 1.8.3, ArgoUML 0.20, CheckStyle 4.0, and JFreeChart 1.0.13 which shows that unit testing increased due to availability of anti-patterns. Previous research also introduced some refactoring actions which applied to classes participating in anti-patterns which reduce cost of testing. In our proposed work we are enhancing the test cases for the similar work with eclipse, Web browser as additional Java programs.

We aim at removal of anti-patterns which can enhance the software quality and will decrease the chances of the bug in the software. Automation in testing for finding anti-pattern is the primary concerns of our study. The main objective is to find testing technique for detection of anti patterns and providing solution for anti pattern by testing approaches. We concentrated on finding three types of anti-patterns which are blob, cryptic code and unused data anti-patterns.

When multiple functions are performed by a single function or multiple responsibilities are assigned to single function, blob anti-patterns are identified. Method call stack is printed to locate blob anti-pattern. Method call stack displays the calling situation in the program.

Cryptic code is a type of anti-pattern which uses abbreviations for declarations or fields instead of suitable naming. This makes intricate to identify what types of values the field is referring to. To locate cryptic code our program describes the least length

required for an ideal naming of the field. The fields having characters in a lesser limit than the precise given length will be considered as cryptic code anti-pattern.

When the code is executed there are certain fields which were defined but not used, these types of anti-patterns are well-known as unused code anti-patterns. XMI file's data supplies us information about the fields defined in the class. Our program looks up these fields to locate their values, if the field is found idle during the execution of program, that field is struck as unused. The program searches the whole code and will provide the entire list of unused fields in the program.

After locating the anti-patterns, code is refactored to eliminate the anti-patterns from the code. This makes the operation of the code faster and enhances quality of the code. Refactoring of the code is carried out in three different stages. During each stage one of the anti-patterns is isolated from the code.

After the refactoring process is complete, refactored code again goes through the testing tool to ensure that there are no anti-patterns present. Then we have compared the implementation time and memory usage of refactored and non-refactored code.

Future work can include repeating the same research using different anti-patterns, automatic refactoring of cryptic code and blob anti-patterns or we can implement the same work on some other platform and compare as to which tool is better.

Chapter 8

REFERENCES

- [1] www.omg.org/technology/readingroom/Anti-Pattern.htm
- [2] www.antipatterns.com/EdJs_Paper/Antipatterns.html
- [3] www.ijcse.net/docs/IJCSE15-04-02-036.pdf
- [4] www.tutorialspoint.com/eclipse/
- [5] M. Fowler, "Refactoring – Improving the Design of Existing Code", 1st ed. Addison-Wesley, June 1999.
- [6] Aminata Sabane, "A Study on the Relation between Antipatterns and the Cost of Class Unit Testing", European Conference on Software Maintenance and Reengineering, July 2013.
- [7] Li Bao-Lin, Li Zhi-shu, Li Qing, Chen Yan Hong, "Test Case automate Generation from UML Sequence diagram and OCL Expression", International Conference on Computational Intelligence and Security 2007, pp 1048-52
- [8] I. Bashir and A. L. Goel, Testing Object-Oriented Software: Life-Cycle Solutions, 1st ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2000.
- [9] A Study on the Relation between Antipatterns and the Cost of Class Unit Testing Aminata Sabane¹, 2, Massimiliano Di Penta³, Giuliano Antoniol², Yann-Gaël Guéhéneuc¹.
- [10] Abdou Maiga¹, Nasir Ali¹, Neelesh Bhattacharya¹, Aminata Sabane¹, Yann-Gaël Guéhéneuc, and Esma Aimeur "SMURF: A SVM-based Incremental Anti-pattern Detection Approach", 2012 19th Working Conference on Reverse Engineering, pp-466-475.

- [11] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esmá Aimeur (2012), “Support Vector Machine For Anti Pattern Detection”, pp 1-4.
- [12] Zhaogang Han, Peng Gong, Li Zhang, Jimin Ling, Wenqing Huang “Definition and Detection of Control- flow Anti-Patterns in Process Models”, 2013 IEEE 37th Annual Computer Software and Applications Conference Workshops, pp-433-438.
- [13] S. R. Chidamber, “A metrics suite for object oriented design,” IEEE Trans. Softw. Eng., 1994.
- [14] Zoltán Ujhelyi, Ákos Horváth, Dániel Varró, Norbert István Csiszár, Gábor Szoke, László Vidácsy, Rudolf Ferenc(2011)” Anti-pattern Detection with Model Queries: A Comparison of Approaches”
- [15] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró, “A graph query language for EMF models,” in Theory and Practice of Model Transformations, ser. Lecture Notes in Computer Science, J. Cabot and E. Visser, Eds. Springer Berlin / Heidelberg, 2011, vol. 6707, pp. 167–182.
- [16] Abdou Maiga¹, Nasir Ali¹, Neelesh Bhattacharya¹, Aminata Sabané¹, Yann-Gaël Guéhéneuc, and Esmá Aimeur “SMURF: A SVM-based Incremental Anti-pattern Detection Approach”, 2012 19th Working Conference on Reverse Engineering, pp-466-475.
- [17] 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.
- [18] www.medium.com/things-developers-care-about/what-is-quality-code-4c07a0a3653
- [19] www.blog.codeclimate.com/blog/2014/04/01/launching-today-automated-refactoring/
- [20] www.thinkandgrowentrepreneur.com/2014/09/refactoring-and-building-next-big-thing.html

GUI	Graphical User Interface
XMI	Extensible Markup Language
MaDUM	Minimum Data Members' Usage Matrix
UML	Unified Modelling Language
SDT	Sequence Dependence Table
GA	Genetic algorithm
SVM	Support vector machine
TDS	Training data set
CAPDL	Control-Flow Anti-pattern Description Language
DOM	Document Object Model
RPST	Refined Process Structure Tree
ASG	Abstract Semantic Graph
EMF	Eclipse Modelling Framework
MDE	Model-Driven Engineering
API	Application Programming Interface