A GUI based Unit Testing Technique for Anti-pattern Identification

Harvinder Kaur
Department of Information Technology,
University Institute of Engineering & Technology
Panjab University, Chandigarh, India
binny.mavi18@gmail.com

Puneet Jai Kaur
Department of Information Technology,
University Institute of Engineering & Technology
Panjab University, Chandigarh, India
puneetkaur79@yahoo.co.in

***Abstract*—Anti-patterns are considered as deficient programming practices that are unacceptable as a solution. They can be thought of as certain patterns in software development that are undesirable in comparison to design patterns which are acceptable solutions formalized for a common problem. From the study on anti-pattern classes involved in object-oriented systems, we examined the impact of presence of anti-patterns on the system. There are certain issues with the testing strategies used for finding bugs in the software systems such as objectives of testing, procedure used to test new functions, maximum resources and time needed to execute the project and the environment in which testing needs to be carried out. There is an enormous requirement to carry out testing process in a more suitable way due to the enlarged complications in the development of software systems. Anti-pattern testing is concerned with the reduction of testing cost for different modules. In this paper, we propose a new GUI based testing technique for the identification of anti patterns in object-oriented systems through automation testing.**

Keywords—Object-oriented, Anti-pattern, Detection Techniques, Testing, Software systems

I. INTRODUCTION

Anti-patterns inspection is a vital activity in any software development approach. Due to the availability of anti-patterns in the system, it becomes very burdensome to maintain such systems. Anti-patterns connect the architectural concepts and real world implementations. Understanding anti-patterns provides the expertise to prevent or recover from them.

The systems containing anti-patterns are less testable. It implies that there are more chances that these types of systems include bugs. These bugs can result in uncertainty of the system. So, it becomes necessary to test such system. In our research work, we will pass our application through regression testing to ensure that the results are not affected due to any changes made in the application. This will make sure that the system is bug free. Automation improves the performance of a system making it easy to use and it also allows for less consumption. The previous approaches which have been proposed by researchers for anti-pattern detection are either semi-automated or manual. Only some of them are automated which are uncertain.

In this paper we have discussed the methodology by following which we will develop the fully automated GUI based anti-pattern application which can be used for detection purpose and will produce desirable results. The testing of anti-pattern applications requires a lot of expenses which can be reduced by refactoring the application. Refactoring will also result in better performance and certainty. Our research will focus on the automation of both detection and refactoring.

II. LITERATURE SURVEY

This section reviews related literature pertaining to Anti-patterns and their detection, testability of classes and unit testing approaches in Object-oriented systems.

Dhambri et al. [1] (Manual approach) and Simon et al. [2] proposed some visualisation techniques to determine a trade-off between manual inspections (time-consuming and non-representative) and fully automated approaches (systematic and productive).

Munro [3] used metric-based heuristics for the identification of anti-patterns. A template is also proposed by him for the systematic characterization of code smells, so that limitation of text-based descriptions can be avoided. The template is composed of three things: name of code smell, text-based descriptions of its attributes, and heuristics to detect them.

Lanza and Marinescu [4] and van Emden and Moonen [5] performed fully automated detection of anti-patterns but they do not tackle uncertainty and long lists. Their methodologies represented the results of detection through visualisation techniques. The quality analysts' interpretation and thresholds are not taken care of while implementing this approach.

Kundu et.al [6] presented an approach by applying UML 2:0 syntax and with use case scope to bring out test cases from activity diagrams. They also proposed an activity path coverage criterion which is a test coverage criterion. The major objective is to cover more faults like synchronization

978-1-4799-4236-7/14/$31.00 ©2014 IEEE

faults, faults in a loop than the existing work, from the generated test suite that follows the *activity path* coverage criterion. At a time, they have concentrated on only activity diagram of a single use case. Their work was found to be contained in the following three steps: First, augmentation of the activity diagram with needed test information. Second, Conversion of the activity diagram into an activity graph. Third, Generation of test cases from the activity graph.

Sabane et.al [7] studied the impact of anti-patterns on testability and the cost of testing. They recognized the number of test cases, which comply with the minimal data member usage matrix (MaDUM) specification proposed by Bashir and Goel, as a measure of testing cost. They also reported that unit testing of anti-pattern classes' demands a larger number of test cases than that required for non-antipattern unit testing. Anti-pattern classes are more defect prone than the non-antipattern classes. Therefore the testing process of anti-pattern classes must be carried out deliberately. By applying various refactoring actions, the testing cost of anti-patterns can be reduced.

Fowler [8] suggested the developers about the appropriate place in code where refactoring can be applied by defining 22 distinct code smells and antipattern symptoms.

Brown et al. [9] introduced and illustrated 40 antipatterns, including the well-known Blob, Functional Decomposition, and Spaghetti Code.

Marinescu [10] described a rule based approach to specify and detect antipatterns.

III. Limitations of Previous Approaches

The basic idea of proposing this technique came after studying and observing the limitations of all the previous approaches used for anti-pattern detection.

- Rule based and language dependent
- Complex and Uncertain
- Not fully automated
- Expensive to implement
- Time Consuming
- Very difficult to test

By keeping in mind all these flaws, we will be implement the technique and will provide a user friendly interface for anti-pattern detection through automation testing. Our application can be tested easily as we will be applying regression testing on our application and will prove that it is bug free even after making certain changes.

IV. Identification Of Different Types of Anti-patterns

There are different types of existing anti-patterns which can be detected by detection techniques proposed by various researchers. In our research we will be working with three types of anti-patterns which are discussed below:

BLOB:

Blob is a well-known anti-pattern that is also popular from the name of 'The God Class' and 'Winnebago'. A class that has been assigned a lot of responsibilities is called as Blob. The main causes for the occurrence of this type of anti-pattern are: Sloth and Haste. Due to the existence of Blob in software applications, it becomes difficult to manage the fields such as: Functionality, Performance and Complexity of an application.

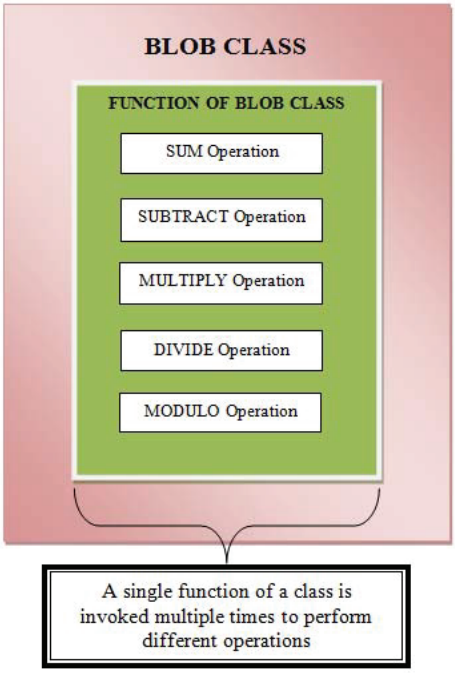

Figure 1: Example showing Blob

An example showing blob is shown in figure 1.

UNUSED CODE:

Unused code is another type of anti-pattern that is recognized as Lava Flow. These are considered to be frozen code that is not used within the entire system. The main causes for the occurrence of this type of anti-pattern are: Sloth, Avarice and Greed. Due to the existence of Dead code in software applications, it becomes difficult to manage the fields such as: Functionality, Performance and Complexity of an application. This type of anti-pattern can be removed by Architectural Configuration Management .It is very expensive to analyze, verify and test unused code.

CRYPTIC CODE:

The use of abbreviations instead of proper names is the root cause for the existence of cryptic code. This type of code can be removed or refactored by designing a system that recognizes proper naming for the defined fields.

V. Research Methodology

A GUI based automated approach for anti-pattern identification is proposed here. This approach will make use of testing process. The proposed technique can be defined as an optimized unit testing for the detection of anti-patterns.

The algorithmic approach adopted to attain the presented work is shown in figure 2.

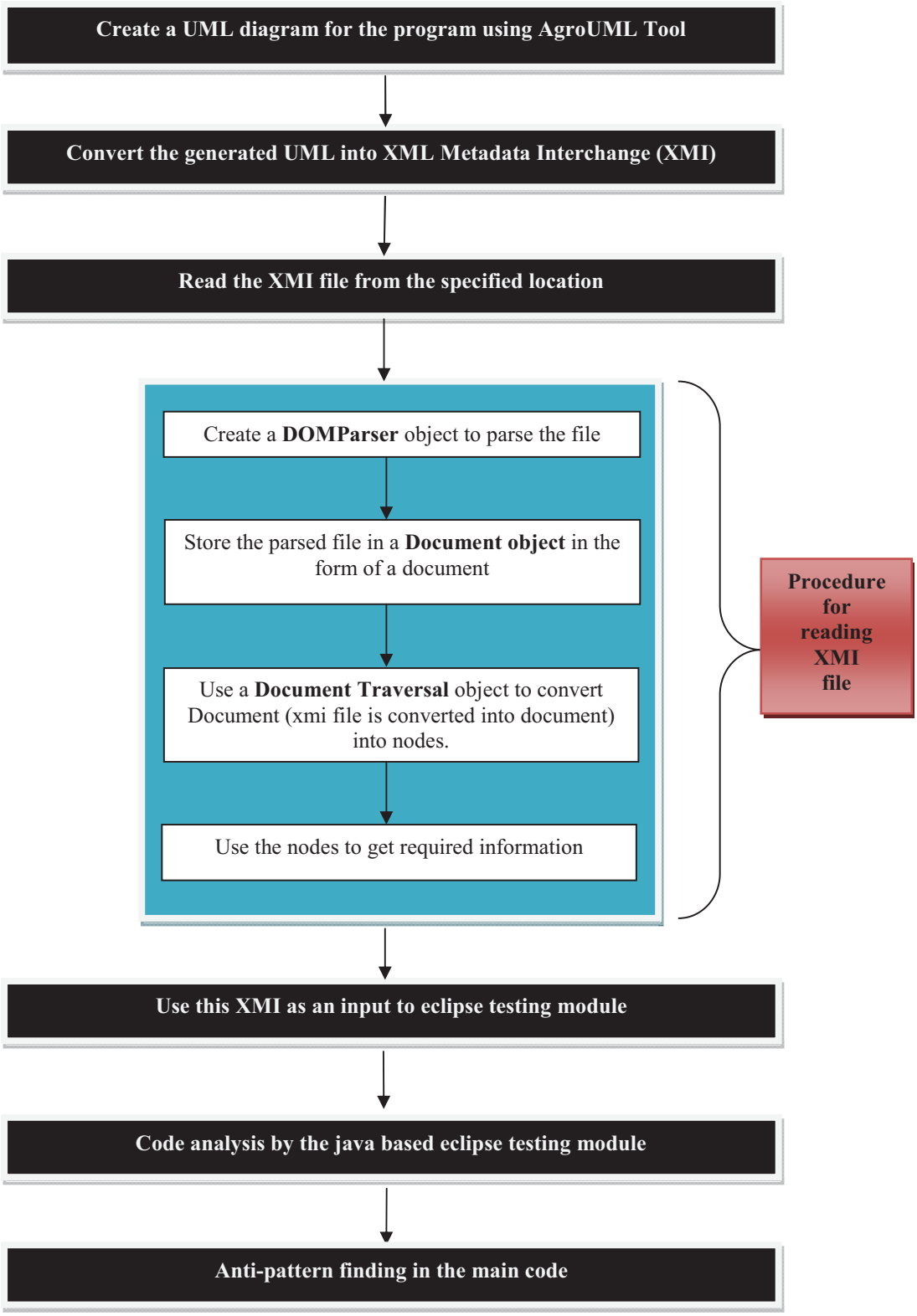

Figure 2: Approach used to implement the presented work

The modules will be tested one by one for anti-pattern occurrences. To produce well grounded systems which are persistently superior in quality, adequate testing of software system is obligatory.

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.

The related research in this area also suggested refactoring strategies which reduced the cost of testing when exercised with the classes involved in anti-patterns.

The proposed approach will be comprised of two stages:

Stage 1: In the first stage, the path of developed GUI application and the XMI will be connected on the basis of ArgoUML tool.

Stage 2: In the second stage, a java based tool will make use of the collected to test the anti-pattern occurrences in the given code.

For the generation of XMI files, Unified modeling language will be practiced. A set of events will be contained in the GUI. The flag will be assigned a value, if the user will tick any event. For all the flags whose value has been fixed, the GUI file will be transformed to the XMI code for the detection of anti-patterns. The java programs such as Eclipse and web browsers will be provided as an input to Matrix data member to observe the modifications that influences the classes associated with anti-patterns.

The principal interests associated with our research includes: Automation in testing for anti-pattern identification and automation of refactoring. These two concerns can be accomplished by finding out the main causes responsible for the occurrence of anti-patterns. After that we can refactor the code so as to improve the function of the developed application which involves anti-patterns. By providing automated detection and automated refactoring of anti-patterns, the effort in terms of time or other parameters can be reduced to an extent and therefore the application's function can be improved. Regression testing will be used for reduction of test cases cost by test case priority in testing and further introduction of refactoring for the considered programs.

VI. Conclusion And Future Work

This paper explains the proposed process for anti pattern detection. In our continuous research, we are working on anti pattern detection in eclipse system which is based on AgroUMLtool for testing unified modeling language diagram bugs. Our target for research is to have a durable and powerful anti-pattern detection system with unit testing process. In our continuous research we are proceeding with testing of ArgoUML 0.20, CheckStyle 4.0, and JFreeChart 1.0.13 based tools.

In future we will commence our research with simple testing scenarios such as testing the whole code without using any test case for achieving our objectives. After that we will search for simple test case scenario for the detection of anti patterns which will redeem resources and enhance the capability of bug finding also. Finally we will practice the set of test cases formulated so far with possible solutions for anti patterns so as to provide more suitable procedure for the selection of test cases.

References

[1] K. Dhambri, H. Sahraoui, P. Poulin, "Visual detection of design anomalies," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*.IEEE Computer Society, 2008, pp. 279–283.

[2] F. Simon, F. Steinbrückner, C. Lewerentz, "Metrics based refactoring," in *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR'01)*. IEEE Computer Society ,2001, pp. 30-38.

[3] M.J. Munro, Product metrics for automatic identification of "bad smell" design problems in java source-code, in *Proceedings of the 11th International Software Metrics Symposium*.IEEE Computer Society Press, Sept. 19-22, 2005, pp.15.

[4] M. Lanza, R. Marinescu, *Object-Oriented Metrics in Practice*. Springer Berlin Heidelberg*, 2006.*

[5] E. van Emden, L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*.IEEE Computer Society Press, 2002.

[6] D. Kundu, D. Samanta, "A Novel Approach to Generate Test Cases from UML Activity Diagrams,". Journal of Object Technology, vol. 8, no. 3, May-June 2009, pp. 65-83.

[7] A. Sabane, M. D. Penta, G. Antoniol, Y. –G. Gueheneuc, "A Study on the Relation Between Antipatterns and the Cost of Class Unit Testing," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering.* IEEE Computer Society ,July 2013.

[8] Fowler, M.: Refactoring – Improving the Design of Existing Code.1st edn. Addison-Wesley (1999)

[9] W.J. Brown, R.C. Malveau, W.H. Brown, H.W. McCormick III, T.J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*.1st ed. John Wiley and Sons, March 1998.

[10] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *Proceedings of the 20th International Conference on Software Maintenance*.IEEE Computer Society Press, Sept. 11-14, 2004, pp. 350–359.