Indian Journal of Science and Technology, Vol 9(46), DOI: 10.17485/ijst/2016/v9i46/107107, December 2016
ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

A Framework to Reduce Redundancy in Android Test Suite using Refactoring

H. Omotunde[1*], R. Ibrahim[1], M. Ahmed[1], R. F. Olanrewaju[2], N. Ibrahim[1] and H. Shah[3]

[1]Department of Computer Science and Information Technology, Universiti Tun Hussein Onn Malaysia, 86400 Parit Raja, Batu Pahat, Johor, Malaysia; hi130033@siswa.uthm.edu.my
[2]International Islamic University Malaysia, 50728 Kuala Lumpur, Gombak, Malaysia
[3]College of Computer Science, King Khalid University, Abha KSA

Abstract

Objective: Test cases tend to be large in number as redundant test cases are generated due to the presence of code smells, hence the need to reduce these smells. **Methods/Statistical Analysis:** This research adopts a proactive approach of reducing test cases by detecting the lazy class code smells based on the cohesion and dependency of the code and applying the inline class refactoring practices before test case generation thereby significantly avoiding redundant test cases from being generated **Findings:** The test cases generated from the original source code is compared to test cases from the refactored code. Test cases from the refactored code show a reduction of up to 33.3% in cyclomatic complexity compared to the original source code. There was 8.2% improvement in the branch coverage of the generated test cases indicating the efficiency of the refactored code **Applications/Improvements:** From the results, refactoring is an effective technique that can reduce redundant test cases. While the focus is on test case reduction, it also improves the quality of generated test cases in terms of its branch coverage.

Keywords: Android Test Suite, Code Smell, Refactoring, Test cases, Test case redundancy

1. Introduction

Test cases are assumed to mirror the original Software Under Test (SUT). Hence, the effectiveness of test case generated can be associated to the quality of the source code of system under test[1]. Improving the quality and reliability of test cases generated improves the quality of testing[2] while an improvement in source code can enhance the quality of test cases generated. The presence of code smells in android application source code has led to many redundant and avoidable test cases being generated. This eventually increases the cost and effort in testing the application[2]. Researchers in both academy and industry are making effort in minimizing the effect of the redundant test case generated[3].

The two well-known approaches presently being looked into are, test case prioritization[4-6] and test case minimization[3,7,8]. In these two approaches, the test cases are generated which includes the redundant test cases. The two approaches are well established in the area of regression testing. There is less attention given to redundancy avoidance in a newly developed application. Hence, there is a need for a technique that averts redundant test cases from being generated in a new system. Asaithambi[9] suggested refactoring as a possible solution that can be explored to reduce the generation of redundant test cases. Refactoring the source code can eliminate code smells that could generate redundant test cases.

Refactoring is an important method of eliminating the weaknesses of a software system by applying modifications to the source code without changing the outward behaviour of the system[11]. In improving the quality of any software, it is essential that the quality of the software is maintained[11,12]. There are different types of code smell and their respective refactoring approaches as named by [13]. This paper focuses on the lazy class code smell which is

*Author for correspondence

as a result of presence of a class that is doing little to stand as a class. A lazy class can simply be defined as a class that is too small to exist. This implies that the cost of its existence is more than it's worth. A lazy class are sometimes as a result of refactoring of a large or God class. A lazy class can be refactored using the inline method of refactoring.

This study nonetheless, applies the inline class refactoring technique to refactor the android source code to eliminate lazy class code smells that could possibly lead to generating redundant test cases. This is achieved by designing the detection and refactoring rules for lazy class smell. The code smell with its detection and refactoring technique is formally specified. Manually performed refactoring is time consuming and takes lots of effort[14]. Automating the process is indispensable. Hence, a tool named Direct Attention Thinking Tools (DATT) was developed to implement the detection and refactoring rules. DATT is evaluated using alogcat application by generating branch coverage and cyclomatic complexity for the original and refactored source code using Clover for android and the results obtained were compared. Test cases are generated before and after refactoring to validate DATT.

Therefore, this paper is organised as follows: the next section discusses related work on test case minimization and refactoring, framework for this study is then presented. Implementation of the framework is discussed next, followed by the results and analysis. Finally, the conclusion of the study is discussed based on the findings.

2. Test Case Generation

Test case generation is costly and effort demanding, hence there is a need for an effective technique to minimize generation of unnecessary test cases. Different techniques have been proposed by researchers to generate test cases[1,15-22]. However, to reduce test case redundancy in software under test, there are basically two approaches that have been explored. The first approach is Test case minimization technique[3,4,8] by which the redundant test cases are eliminated. Another technique is the test case prioritization[4,6,23], prioritizing the test case based on coverage. In both of these approaches, it is essential that the reduced test cases can detect fault in the system while maintaining a good coverage. In[7] proposed a technique for test case minimization in regression testing for Object Oriented Programming built on innovative optimal page replacement algorithm. In their approach, the reference strings in optimal page replacement algorithm represent test cases. The modified and validated test cases are arranged and grouped using the page frame number. The algorithm then operates on the grouped chunks of test cases and generates the page fault number via optimization technique. The page fault number specifies the amount of test cases in the test suites after eliminating redundant test cases[7]. In[4] offered a new approach using a set of location-centric metrics and black-box input-guided as well as POI-aware test case prioritization techniques to reduce redundant test cases. A measurable case study to assess the success of the metrics and the modified techniques using an imperial service application as the focus was carried out[4]. The experimental result of the case study shows that the Point of Interest (POI)-aware prioritization techniques are more constant and more effective in terms of their rate of fault detection, than random ordering on different software. However, the focus of the above mentioned researches is on regression testing. There are times where it is not easy to differentiate the original code from the transformed code, in which case, test case reduction can prove abortive. The major contribution of this research is proposing a preventive approach to reduce redundant test cases. Reusability testing is another approach of reducing testing effort through test case reuse[24].Refactoring on the other hand, was primarily defined to enhance the internal structure of code and make it more maintainable and extensible, but it was later extended to include the enhancement of external features, such as performance[25,26], security[27-29] and usability[30]. As regards usability, refactoring can be a supportive approach to iteratively and analytically enhance the way users interact with a Graphical User Interface (GUI) when executing doing their day-to-day task. In an earlier study by[31], a catalogue of refactoring was defined to enhance usability, in the particular area of web application systems and business practices.

3. Framework for Test Suite Reduction

To improve the test case coverage and reduce redundancy in an android source code, DATT was designed to detect and refactor the source code prior to test case generation. This paper focuses on the lazy class smell. This particular smell is carefully chosen as a step to bridge the gap

in research and industry refactoring practice. According to[14], some of the refactoring practice found in the real world including the Inline class and method are ignored by researchers. Hence, there is a need to look into that direction. As shown in Figure 1, there are four main steps in defining the framework of DATT, namely, design of lazy class detection rules, design of refactoring rules for the detected smell, implementation of detection and refactoring rules in DATT and evaluation and validation of DATT using branch coverage, cyclomatic complexity and test cases generated from alogcat application source code. The list of activities involved in each step of the process is discussed in subsequent sections.

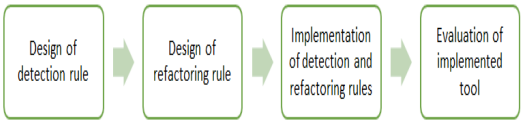

Figure 1. Research framework to detect and refactor lazy class.

3.1 Design of Detection Rules

In this step, the lazy class smell is detected using the designed detection rule. Since developers define classes based on the cohesion and dependencies[32], the detection rules are also based on class or method cohesion and dependency. Cohesion (high responsibilities of a class or method) is defined by the classes, methods or attributes calling the particular class or method in focus. It is being referred to as Caller properties in the scope of this study. While dependency is defined by the classes, methods and attributes that are being called or used by the class or method. It is referred to as callee properties in the context of this study. To detect the lazy class, the classes and methods are extracted. A distance metric is defined based on structural dependencies i.e. field accesses and method invocations between class members and cohesion. The number of lines of code is also put into consideration. Based on this information, detection rules are formed. Classes with less than three (3) calls and callers with lines of code less than fifteen (15) are regarded as lazy class. A minimum threshold of 3 is used for cohesion and dependency in this study as it has been found in most of the researches ranging from three to five[33]. Three has been selected since the output of three is included in all other alternatives. In addition,[34] in their comparison of values, concluded that a threshold of three gave the best result.

3.2 Design of Refactoring Rules for Lazy Class Smell

Designing the refactoring of detected smell is done at this stage. Lazy class is refactored using the inline class technique. The inline class approach implies that the methods and attributes in the lazy class are moved to the nearest class in terms of its distance metric. All references to the moved methods and attributes are updated by referencing to the host class. The process continued until the lazy class is empty. The empty lazy class is deleted from the source code of the SUT. Refactoring is not completed until the program is run to ensure no bug has been introduced into the SUT.

3.3 Implementation of Detection and Refactoring Rules in Tool Environment

Manual execution of detection and refactoring of the smells could be very tasking. Hence, this process has been automated into DATT as an eclipse plugin. DATT applies the ASTParser API of eclipse Java Development Tool to extract the code for detection of smells while it uses the ASTRewrite API for the refactoring of the code. The tool consists of two main components. They are:

- Detection of smell: To identify this smell, the detection rule is implemented in to the tool. The tool implements the agglomerative hierarchical algorithm to extract the classes and methods, and calculate the distance metric and lines of code. The agglomerative hierarchical algorithm creates an entity set of cluster for each class and method. Each entity set contains all the callee and caller class, method and attributes. The lines of code of each class visited are also measured. The process continues until all classes are checked. The tool then checks for which of the classes or methods certify the lazy class as explained earlier.
- Refactoring the source code: The refactoring option in the jdeodorant is extended to refactor the code based on the detected code smell in DATT. To refactor the lazy class, the inline class is implemented respectively using the ASTRewrite of Eclipses Java Development Toolkit (JDT). ASTRewrite rewrites the code based on the refactoring rule using the move method, move field and pull up method of refactoring. The implemented algorithms are discussed further in the next section.

3.4 Evaluation and Validation of DATT using Test Path, Branch Coverage and Test Case Generation

To evaluate the detection and refactoring rules and their implementation in DATT, number of branches (test path), branch coverage and cyclomatic complexity is generated from the source code before and after the refactoring technique using Clover for android. Cyclomatic complexity indicates the total number of stand-alone routes in a program, while the branch coverage indicates the parts of the code that is covered by the test case. One of the ways to determine the efficiency of a test case is using the branch coverage. To validate this technique, scenario based test cases are generated from the source codes. Alogcat application source code is used to verify the result for each of the code smell. Alogcat has been benchmarked by other researchers[35,36] and is an open source application on the android git repository. An overview of the classes analyzed in the Alogcat source code is shown in Table 1.

Table 1. An overview of the Alogcat aAnalyzed classes

SourceCode	Number of Methods	LinesofCode
SaveReceiver	1	8
SaveService	2	9
ShareReceiver	1	8

The result of each refactored classes and details on their respective coverage and generated test cases are discussed later in this paper.

4. Implementation of DATT

The detection and refactoring rules are implemented in a tool environment named DATT. This section presents the implementation algorithm of DATT.

4.1 Detection of the Lazy Class Code Smell

The tool identifies the classes with possibility of being moved or deleted by running a hierarchical agglomerative clustering algorithm on each class based on the Jackard distance as the distance metric and lines of code. The jackard distance is calculated using the object cluster. The object cluster of an object (attribute, method or class) comprises of every elements of the class that use or are used by the object in question. Given a class, the algorithm starts by calculating the object clusters of all objects in the class, i.e., each attribute, method and class, and putting each distinct object in a distinct collection. After that, at every stage, the algorithm chooses to combine all the clusters of a particular class together. The process stops when all classes are checked. The lines of code of each class are also measured. Algorithm 1 shows the detection algorithm. The algorithm is adopted from Jdeodorant[37] algorithm for detecting God Class smell. The rules for lazy class are adapted to the algorithm as shown in Algorithm 1.

Algorithm 1 Detection of Lazy Class

procedure DETECTION OF LAZY CLASS
Get class name
If class is a super class, get sub class If class is new, add to class list
If class already exist, add to existing class counter for each class or subclass c do
read attributes, method and class call end for
If attribute call in the class is new, add to counter ++i, else return i If method call in the class is new, add to counter ++j, else return j If class call in the class is new , add to counter ++k, else return k Repeat for all classes in source code SC
for each class c do
collate the classes, methods and attributes calling say x, y, z respectively end for
for each LOC in class c do add to counter ++l
end for
If ((i,j,k,l,x,y,z 3) return true; //(i.e lazy class detected) else return false;
end procedure

4.2 Applying the Inline Class Refactoring

To apply the refactoring, ASTRewrite of Eclipses Java Development Toolkit (JDT) is used. Public attributes and methods in the lazy class are created in the existing class. The methods created should refer to the equivalent methods in the lazy class as well as the attributes. All references to the lazy class are replaced with references to the fields and methods created in the existing class. The program is tested to ensure that no error has been created. The move method and move field is used to completely transfer all functionalities to the existing class from the lazy class until the lazy class is empty. The lazy class is then deleted from the programs source code. Algorithm 2 displays the refactoring algorithm.

Algorithm 2 Refactoring of Lazy Class

procedure REFACTORING OF LAZY CLASS

Identify the movable class

If method in movable class not null

Move method content to a host class of shortest distance If attribute not null, Push Field to the host class

Remove all methods from movable class until method = null Remove all attributes from movable class until attributes = null Update method in the host class

Update field in the host class

Redirect reference calls from movable class to the host class Delete the lazy class

end procedure

5. Result and Analysis

Table 2 shows the evaluation of lazy class refactoring based on the cyclomatic complexity and branch coverage in comparison with the original alogcat source code. A test case minimization approach is incomplete if the quality of the test case is not ensured. One of the ways to do this is calculating the branch coverage of the before and after refactoring code and comparing the result. Hence, the branch coverage is calculated for each of the affected classes in the source code using the formula in [38] :

Table 2. Branch coverage results for refactored classes

	Before DATT		**After DATT**	
	Original		**Lazy Class**	
	C	**B**	**C**	**B**
SaveReceiver	1	75%	1	81.8%
ShareReceiver	1	77.7%		
SaveService	1	81.8%	1	90%

Branch Coverage =	Number of Lines of Code Covered	X 100	(1)
	Number of Total Lines of Code		

From Table 2, the CC column shows the cyclomatic complexity for each refactored classes while BC indicates the branch coverage of each class. From the CC column of original code, the total cyclomatic complexity is 3 while the total cyclomatic complexity is 2 in the refactored code. Hence, there is 33.3% reduction in the cyclomatic complexity. While for branch coverage increment, the highest increment is in SaveService.java class using small method with an increment from 81.8% to 91.2%.

Therefore, percentage reduction in cyclomatic complexity is 33.3% with a branch coverage increase up to 8.2%.

To validate the implementation of DATT in terms of test case minimization, test cases are generated from the three (3) classes of alogcat application source code before and after refactoring. Table 3 shows the test case generated from the original source code prior to refactoring while Table 4 shows the test cases generated after refactoring.

From Table 3, three (3) test cases are generated. Hence,

$$T C = \{T C 1, T C 2, T C 3\} \quad (2)$$

Of these generated test cases, TC1 and TC2 can be merged into one to give TC'1. The source code implementing this function is the same and the drop down shows Share and Save at the same time by clicking same thing. This implies that TC'1 is sufficient to stand for TC1 and TC2.

$$T C = \{T C 1, T C 2, T C 3\} \quad (3)$$

is then reduced to

$$T C' = \{T C'1, T C'2\} \quad (4)$$

The percentage decrease in number of test cases is approximately 33.3%.

6. Conclusion

This paper presents a pre-emptive framework to reduce test cases in android applications. The framework is in four steps: design of rules for detection of the lazy class

Table 3. Test cases generated from original Alogcat source code

Name of Classes	**Test Cases**	**Scenarios**	**Input**	**Expected Output**	**Actual Output**
SaveReceiver	TC1	Create intent for SaveService	Click on drop down	Access to save	Access to save
ShareReceiver	TC2	Create intent for ShareService	Click on drop down	Access to share	Access to share
SaveService	TC3	Handle intent for LogSaver	Click on save	Open the save dialog	Save dialog opened

Table 4. Test cases generated from refactored Alogcat sourcecode

Name of Classes	Test Cases	Scenarios	Input	Expected Output	Actual Output
Request Receiver	TC'1	Create intent for SaveService and Share Service	Click on drop down	Access to Save and Share	Access to Save And Share
SaveService	TC'2	Handle intent for Log Saver	Click on save	Open the save Dialog	Save dialog opened

code smells, refactoring of the detected smells based on designed refactoring rule R1, R2 and R3 respectively, implementing the detection and refactoring rules in DATT tool and evaluating DATT by comparing generated number of branches and cyclomatic complexity of original code and the refactored codes of an android application. The outcome shows that refactoring the source code prior to test case generation has reduced the test cases by 33.3% and increased the branch coverage up to 9.2%. Hence, the approach is a prospective test case reduction technique. This implies that the cost and effort in testing can be reduced by eliminating the code smells prior to test case generation. However, there is a need to look into the effect of other refactoring measures on test case generation and other benchmark parameters such as the fault detection capability.

7. References

1. Anand S, Burke EK, Chen TY, Clark J, Cohen MB, Grieskamp W, et al. An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software. 2013; 86(8):1978–2001.
2. Chen L, Li Q. Automated test case generation from use case: A model based approach. 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT).IEEE; 2010.
3. Asaithambi SPR, Jarzabek S. Pragmatic Approach to Test Case Reuse-A Case Study in Android OS BiDiTests Library. Software Reuse for Dynamic Systems in the Cloud and Beyond. Springer; 2014. p.122–38.
4. Ke Z, Bo J, Chan WK. Prioritizing Test Cases for Regression Testing of Location-Based Services: Metrics, Techniques, and Case Study. IEEE Transactions on Services Computing.. 2014; 7(1):54–67.
5. Papadakis M, Malevris N. Mutation based test case generation via a path selection strategy. Information and Software Technology. 2012; 54(9):915–32.
6. Zhang C, Groce A, Alipour MA, editors. Using test case reduction and prioritization to improve symbolic execution. Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM; 2014.
7. Mondal SK, Tahbildar H. Regression Test Cases Minimization for Object Oriented Programming using New Optimal Page Replacement Algorithm. International Journal of Software Engineering and Its Applications. 2014; 8(6):253–64.
8. Zhang W, Zhao D. Reuse-Oriented Test Case Management Framework. International Conference on Computer Sciences and Applications (CSA).IEEE; 2013.
9. Asaithambi S, Jarzabek S. Towards Test Case Reuse: A Study of Redundancies in Android Platform Test Libraries,Berlin Heidelberg. Springer; 2013. p. 49–64.
10. Fowler M. Refactoring: Improving the design of existing code. India: Pearson Education;; 1999.
11. Lashari SA, Ibrahim R, Senan N. Fuzzy Soft Set based Classification for Mammogram Images. International Journal of Computer Information Systems and Industrial Management Applications. 2015; 7:66–73.
12. Ahmed M, Ibrahim R, Ibrahim N. An Adaptation Model for Android Application Testing with Refactoring. Growth. 2015; 9(10):65–74.
13. Fowler M. Refactoring: Improving the Design of Existing Code. 1997. Available from: http://www.martinfowler.com/books/refactoring.html.
14. Al Dallal J. Identifying refactoring opportunities in object-oriented code: A systematic literature review. Information and Software Technology. 2015; 58:231–49.
15. Jena SKSAK, Mohapatra DP. A Novel Approach for Test Case Generation from UML Activity Diagram. 2014.
16. Ibrahim R, Saringat MZ, Ibrahim N, Ismail N. An Automatic Tool for Generating Test Cases from the System's Requirements. 2007;861–6.
17. Nguyen CD, Marchetto A, Tonella P, editors. Combining model-based and combinatorial testing for effective test case generation. Proceedings of the 2012 International Symposium on Software Testing and Analysis, ACM; 2012.

18. Swain R, Panthi V, Behera PK, Mohapatra DP. Automatic test case generation from UML state chart diagram. International Journal of Computer Applications. 2012; 42(7):26–36.
19. Khan SUR, Lee SP, Ahmad RW, Akhunzada A, Chang V. A Survey on Test Suite Reduction Frameworks and Tools. International Journal of Information Management. 2016; 36(6): Part A, 963–75.
20. Jatana N, Suri B, Kumar P, Wadhwa B. Test Suite Reduction by Mutation Testing Mapped to Set Cover Problem. Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies, ACM; 2016.
21. Gupta A, Gupta A, Kushwaha DS, editors. Test Case Reduction Using Decision Table for Requirements Specifications. Proceedings of the International Congress on Information and Communication Technology, Springer; 2016.
22. Alipour MA, Shi A, Gopinath R, Marinov D, Groce A. Evaluating non-adequate test-case reduction. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ACM. 2016.
23. Shahid M, Ibrahim S. A New Code Based Test Case Prioritization Technique. International Journal of Software Engineering and its Applications. 2014; 8(6): 31–8.
24. Ahmed M, Ibrahim R. Improving effectiveness of testing using reusability factor. International Conference on Computer and Information Sciences (ICCOINS).IEEE; 2014.
25. Dig D. A refactoring approach to parallelism. IEEE Software. 2011; 28(1):17–22.
26. Rieger M, Van Rompaey B, Du Bois B, Meijfroidt K, Olievier P. Refactoring for performance: An experience report. Proc. Software Evolution. 2007; 2(9):1–9.
27. Yoshioka N, Washizaki H, Maruyama K. A survey on security patterns. Progress in informatics. 2008; 5(5):35–47.
28. Omotunde H, Ibrahim R. Mitigating SQL Injection Attacks via Hybrid Threat Modelling. 2nd International Conference on Information Science and Security (ICISS).IEEE; 2015.
29. Omotunde H, Ibrahim R. A Review of threat modelling and ITS hybrid approaches to software security testing. Research gate. 2006.
30. Garrido A, Rossi G, Distante D. Refactoring for usability in web applications. IEEE Software. 2011; 28(3):60.
31. Al Dallal J. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. Information and Software Technology. 2012; 54(10):1125–41.
32. Bavota G, De Lucia A, Marcus A, Oliveto R. Automating extract class refactoring: an improved method and its evaluation. Empirical Software Engineering. 2014; 19(6):1617–64.
33. Palomba F, Bavota G, Oliveto R, De Lucia A. Anti-Pattern Detection: Methods, Challenges, and Open Issues. Advances in Computers. 2014; 95:201–38.
34. Silva D, Terra R, Valente MT. JExtract: An Eclipse Plug-in for Recommending Automated Extract Method Refactorings. Federal University of Minas Gerais: Brazil. 2014, pp.1-8.
35. Dennis B. Repatterning: Improving the Reliability of Android Applications with an Adaptation of Refactoring [PhD thesis]https://etd.auburn.edu/handle/10415/4219.2014.
36. Fokaefs M, Tsantalis N, Stroulia E, Chatzigeorgiou A. JDeodorant: Identification and application of extract class refactorings. Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA.ACM; 2011. p. 1037–9.
37. Yoo S, Harman M. Pareto efficient multi-objective test case selection. Proceedings of the 2007 International symposium on Software testing and analysis, London, United Kingdom. ACM; 2007. P. 140–50.