Contents lists available at ScienceDirect

Journal of King Saud University – Computer and Information Sciences

journal homepage: www.sciencedirect.com

Reducing redundancy of test cases generation using code smell detection and refactoring

Rosziati Ibrahim[a,*], Maryam Ahmed[b], Richi Nayak[c], Sapiee Jamel[d]

[a] *Department of Software Engineering, Universiti Tun Hussein Onn Malaysia (UTHM), Malaysia*
[b] *College of Computer Science and Information Technology, Imam Abdulrahman Bin Faisal University (IAU), Saudi Arabia*
[c] *School of Electrical Engineering and Computer Science, Queensland University of Technology (QUT), Australia*
[d] *Department of Information Security, Universiti Tun Hussein Onn Malaysia (UTHM), Malaysia*

A R T I C L E I N F O

Article history:
Received 24 January 2018
Revised 11 June 2018
Accepted 12 June 2018
Available online 21 June 2018

Keywords:
Lazy class
Small method
Duplicate code smell
Code smell detection
Refactoring

A B S T R A C T

In software development life cycle (SDLC), the testing phase is important to test the functionalities of any software. In this phase, test cases are generated to test software functionalities. This paper presents an approach on how to detect and refactor code smells from the source codes of an Android application in order to reduce the redundancy in test case generation. Refactoring is one of the vital activities in software development and maintenance. Refactoring is a process of code alteration that aims to make structural modifications to the source code without altering any external functionality. These changes often improve software quality such as readability, execution time and maintainability. The proposed approach is then implemented and evaluated in order to determine its effectiveness in reducing the redundancy of test case generation.

1. Introduction

Software developers are given the task of analysing and detecting changes that need to be made in source codes. To ease this task, different detection and refactoring tools are developed (Moha et al., 2010; Hecht et al., 2015; Mkaouer et al., 2016; Santos et al., 2015; Sirqueira et al., 2016). In an active development environment, developers are involved in suggesting the refactoring opportunities in the source code to remove the bad smells. A code smell is a mark that there is an issue with the internal structure (code) of the system that can be corrected through code restructuring (refactoring) (Lee et al., 2016). Code smells are different from syntax errors or any other errors that can easily be detected by the compiler. Rather, in the case of code smell, the compiler cannot identify the smells because code smells do not stop the program from running but they are a symbol of bad program design. Refactoring can be defined as a practice that aims to produce improved source code quality (Bavota et al., 2015).

There are some factors to consider while deciding on which code smells to identify or detect. Detecting all smells mentioned by Fowler (1999) might not be feasible as some of these smells have been detected earlier by various researchers, while some might not be applicable in real life scenario. Some smells are easily detected in the code (these smells are known as primitive smells). Other smells can be detected from extraction of details from the code (these smells are known as extracted smells), while others can be detected by the human eye or possibly using information from the management system (these kinds of smells are known as maintenance smells). This paper focuses on the detection and refactoring of three code smells, basically extracted smells; lazy class, small method and duplicate. One of the reasons of selecting these smells is to bridge the gap between the industry and academy as mentioned by Al Dallal (2015).

The main contributions of this paper are:

1. A new approach for detecting code smells from source code and refactoring the code smells without changing a system's functionalities.
2. A framework for detecting and refactoring code smells in order to generate test cases from the source code.

* Corresponding author.
E-mail addresses: rosziati@uthm.edu.my (R. Ibrahim), mtahmed@iau.edu.sa (M. Ahmed), r.nayak@qut.edu.au (R. Nayak), sapiee@uthm.edu.my (S. Jamel).

Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

https://doi.org/10.1016/j.jksuci.2018.06.005
1319-1578/© 2018 The Authors. Production and hosting by Elsevier B.V. on behalf of King Saud University.

3. A tool that is able to reduce the generation of test cases from the source codes due to the detection and refactoring of code smells.

The rest of this paper is structured as follows. Related work for code smell and refactoring is discussed in Section 2. Section 3 presents a details discussion on the proposed approach. Section 4 presents the algorithms for code smell detection and refactoring rules. Section 5 discusses the results of implementing the proposed approach in reducing test cases generation. This paper is then concluded by a summary of the paper and brief mention of future work.

2. Related work

Fowler (Fowler, 1999) has clearly described code smells as indicators of bad planning and execution adoptions. In a few cases, some of these indicators are from activities implemented by developers while in a rush. Such instance includes executing vital patches or basically making suboptimal adoptions. In other cases, smells could be a result of some recurring, bad design solutions, also referred to as anti-patterns. Earlier studies have found that code smells obstruct code clarity (Arcoverde et al., 2011), and probably increase alteration and fault-susceptibility (Ujhelyi et al., 2015). Hence, these smells should be cautiously identified and examined and when needed, refactoring activities should be applied and executed to eliminate them. There are various methodologies that can detect smells in source code to catch the attention of system developers to their existence (Moha et al., 2010; Fowler, 1999; Fontana et al., 2015; Bavota et al., 2014; Fontana et al., 2013; Rasool and Arshad, 2017). These methodologies rely on internal or structural information mined from the source code, which could be obtained by well-defined limitations on some source code attributes. For example, some current methodologies are built on the magnitude of the source code features in terms of line of code (LOC). This includes DECOR (Moha et al., 2010) which establishes smells like LongMethod or LargeClass. Whereas some smells are based on the McCabe cyclomatic complexity such as ComplexClass, which also referred to as GodClass (Bavota et al., 2014). Some other code smells, such as Blob, tend to use more complex rules. While current methodologies exhibits worthy performances in terms of correctness and recall, it might not be suitable to identify several smells defined by Fowler (Fowler, 1999). Therefore, there is a need to consider more code smells that have not been previously studied or identified.

Code smells, particularly for android applications are due to the rapid development of android applications themselves. It is easy for developers to repackage android applications by adding needed functionalities to an already existing code. This simple act poses a threat to the android application market, which includes the presence of code smells (Li, 2012). Source code with the presence of smells is more prone to fault than source code with free of smells (Khomh et al., 2012; Li and Shatnawi, 2007). Other studies aimed at comprehending the effect of anti-patterns or code smells on source code comprehensibility has revealed that the existence of smells in the source code of a system does not diminish system performance, while another study on smells combination shows a considerable decrease in performance (Abbes et al., 2011; Palomba et al., 2014). Although the results of the study in (Abbes et al., 2011) indicate that the presence of smells are not harmful to the system, it also shows that smells are fairly circulated through the software development process and frequently code attributes could be affected by one or more anti-pattern. Also, there are empirical evidences that show that the amount of anti-patterns in the source code of a system escalates over time, and only in rare cases are they eliminated by means of refactoring processes (Arcoverde et al., 2011).

Most researchers (Khomh et al., 2012; Li and Shatnawi, 2007; Abbes et al., 2011; Palomba et al., 2014; Fokaefs et al., 2011; Parnin et al., 2008) recommend that it is essential for code smells and anti-pattern to be cautiously identified and examined and when required, refactoring processes should be strategized and executed to eliminate them. On the other hand, the detection and modification of design flaws in bulky and vital software systems could be extremely difficult. Hence, a new approach or method is required to support software engineers and developers in detecting anti-patterns and designing and executing refactoring activities. This is the gap for our research to design a new approach for detecting code smells and refactoring them. Therefore, this paper presents a new approach for detecting and refactoring code smells from the source codes of an Android application in order to reduce the redundancy of generating test cases from the source codes of an Android application.

3. The proposed approach – dart

Our proposed approach called DART (Detecting and Refactoring Tool) is presented in details in this section. Table 1 shows the steps involved and Fig. 1 shows the flow chart for the steps to detect and refactor three code smells in an Android application in order to reduce the redundancy in test case generation.

From Table 1, the source code of an android application is examined first to detect the code smell. Three code smells are involved namely lazy class, small method and duplicate code smell. Then these code smells are refactored in order to reduce the redundancy in test case generation. If refactoring the code smells does not change the system's functionalities, the refactored code is kept and test cases are generated. However, if refactoring changes the system's functionalities, then refactoring cannot be done and any changes need to be undone to maintain the system's functionalities. These steps are details out in Fig. 1.

Based from Table 1 and Fig. 1, our proposed methodology focuses on three code smells namely lazy class, small method and duplicate code smell. Each of these code smells are detected during the analysis of the internal structure of an Android application. For each of the detected code smells, refactoring these code smells are necessary to reduce the redundancy of generating test cases. However, our approach also checks the effect of refactoring to the system's functionalities. It is crucial not to change the system's functionalities when refactoring the code smells. Fig. 2 shows the overall framework for detecting and refactoring the code smells.

Based on Fig. 2, we develop a tool called DART (Detection and Refactoring Tool). DART is implemented using Eclipse environment based on code smell detection algorithm and refactoring algorithm. The tool is developed in Eclipse environment as a plug-in tool as recommended in Silva et al. (2015). The algorithms are being implemented in ALogcat application source code from the git

Table 1
Steps for the Proposed Approach.

Steps	Details
Step 1	Detection of Code Smell - Detect lazy class - Detect small method - Detect duplicate code smell
Step 2	Refactoring of Code Smell - Refactoring lazy class - Refactoring small method - Refactoring duplicate code smell
Step 3	Generate test cases.

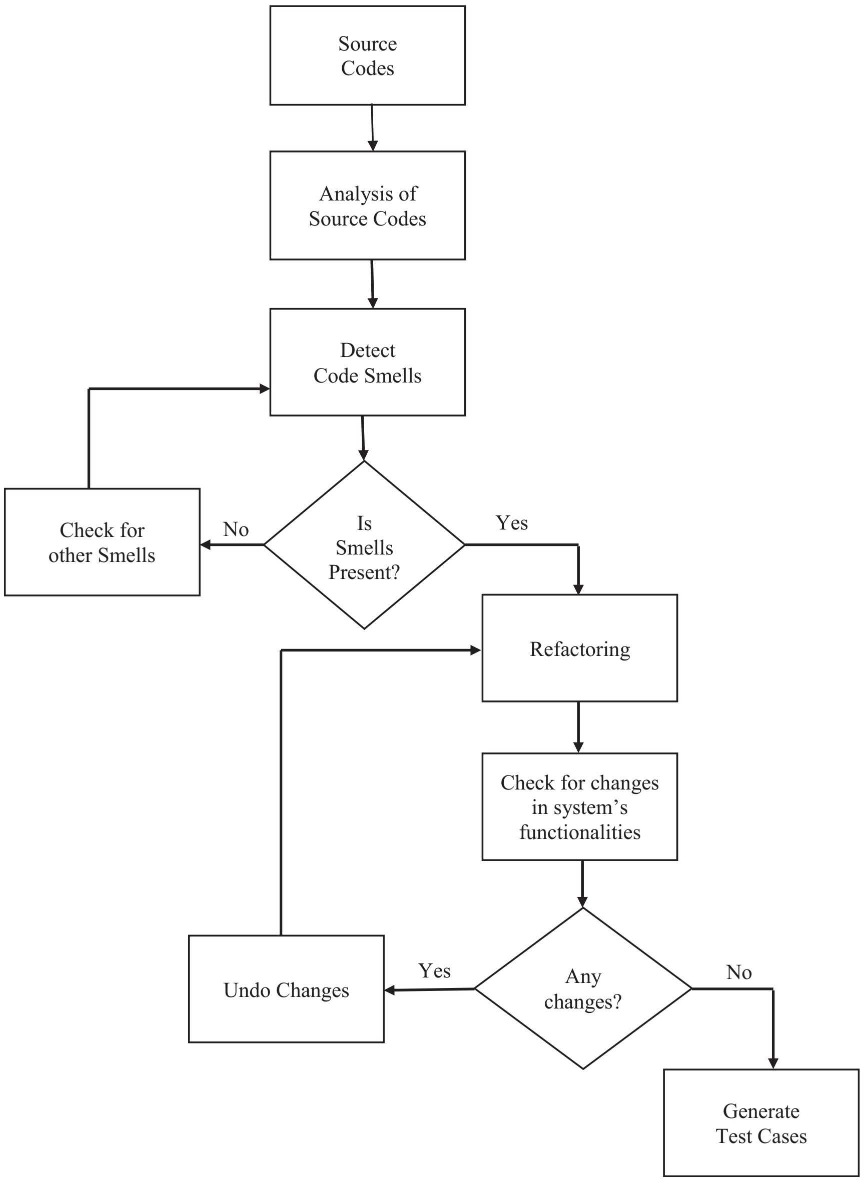


Fig. 1. Flowchart of the Proposed Approach.

repository (git repository, 2016). Both algorithms are discussed in Section 4.

4. Code smell detection algorithm and refactoring algorithm

Our proposed approach extends the definitions used in (Bavota et al., 2014) and extract classes as proposed in (Fokaefs et al., 2011). Detection of code smells based on their source code involves static analysis of the source code of an Android application as described in Fig. 1. Since software developers and system analysts classify classes using cohesion and dependencies attributes (Bavota et al., 2014), we use the same approach in the detection rules for detecting code smells in the source code and classify them using the cohesion and dependency property of a class or method. Cohesion (highly needed class or method) can be described using classes, methods or the attributes calling the class or method in focus. These properties are known as the caller properties as defined in (Bavota et al., 2014), while the dependency attribute is described using the number of classes, methods and attributes that are being called or used by the class or method in focus. These properties are referred to as callee properties in (Bavota et al., 2014). To detect the lazy class and small method, the classes and methods are identified. A distance metric is then calculated based on internal structure dependencies which include access to the field and method calling between class members. The cohesion property is also considered. In addition, the number of lines of code (LOC) is put into consideration. Duplicate smells are detected by matching strings to compare the LOC of same and different classes. Using the information above, detection rules are generated. Classes with fewer than three (3) callees and callers with lines of code (LOC) fewer than fifteen (15) are said to be lazy class. Methods that do not have up to three (3) callees and callers with lines of code fewer than five (5) and attribute not up to three (3) are identified as small methods. Likewise, duplicate codes are defined based on similarity in at least ten (10) lines of code of different or same classes.

In our proposed approach, three codes smells are used during the detection steps as described in Fig. 2. Android application source codes can contain one or more of the following smells: lazy

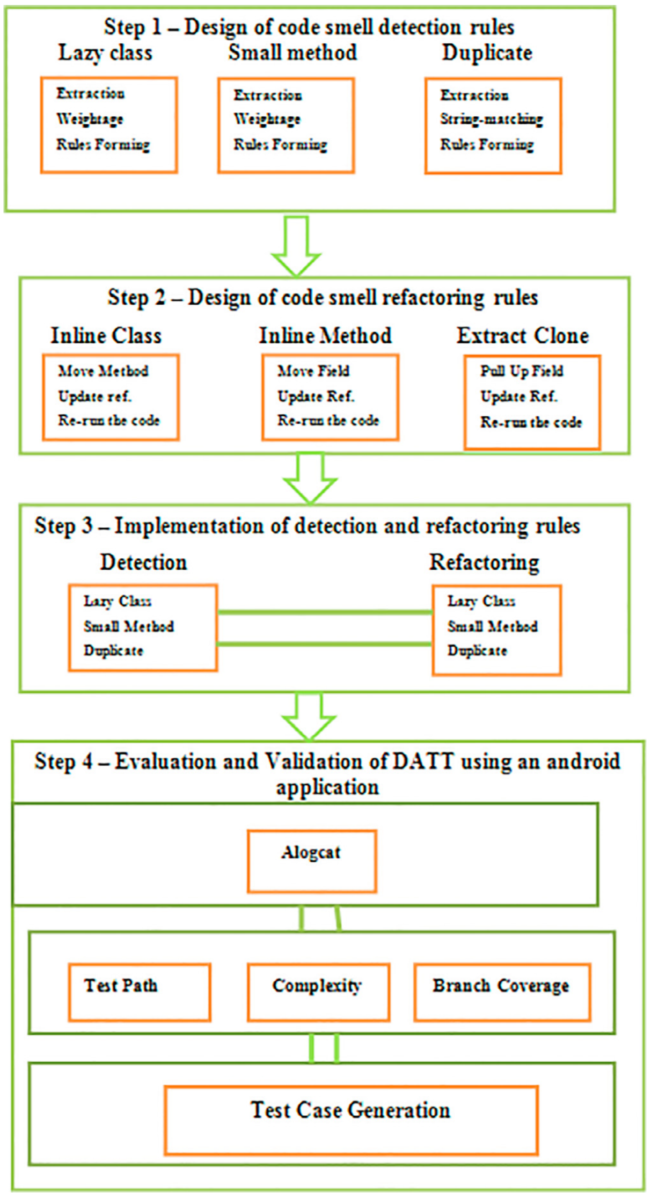


Fig. 2. The Framework for the Detection and Refactoring of Code Smells.

class, small method and duplicate. A threshold of 3 is used to be the least number for cohesion and dependency in this research as it has been established that most research work ranges from three to five (Palomba et al., 2014). Three has been chosen as the outcome of three is incorporated in all other feasible options. Moreover, (Palomba et al., 2014) in their assessment of minimum threshold, established that a threshold of three has the best outcome.

It is essential to eliminate the code smells detected in order to enhance the quality of testing by minimizing the test path and cyclomatic complexity. Lazy class is refactored using the inline class technique. The inline class method involves the movement of methods and attributes in the lazy class to the closest class based on its distance metric. All associations or families (method and attributes) of the moved class are updated by redirecting it to the host class. This process continues until the detected lazy class is empty. The emptied lazy class is then removed from the source code. Using a similar approach, the small method is refactored using the inline method technique. The attributes and method in the small method are moved to the most related existing class in terms of its distance metric. The process continues until the small method is empty and then deleted. All references to the old small method are transferred to the new class hosting the attributes and methods. The extract clone technique is used to refactor duplicate smells by extracting similarities found in the program. If the extracted line of code is a class, a single superclass is created to maintain the functionalities of the two similar classes. If the extraction is found in a method either in constructors or a series of conditional statements, a method is created that can be easily understood and similar codes are placed outside the condition. Refactoring is incomplete until the program is executed to ensure no bugs have been initiated into the source code.

5. Reduction in redundancy of generating test cases

A case study is presented in this section in order to show how the proposed approach can reduce the redundancy of test cases generated from an Android application. The evaluation of DART is done using Alogcat application source codes obtained from the git repository (git repository, 2016) where DART is evaluated using branch coverage, cyclomatic complexity and test cases generated from the Alogcat application source code. An overview of the classes analyzed in the Alogcat source code is shown in Table 2.

ALogcat is a well-known developer tool logcat, in the form of an Android application (git repository, 2016). It views color-coded, scrolling (tailed) Android device (logcat) logs directly from the phone. It does not require USB, adb, or email service. An overview of Alogcat is shown in Fig. 3.

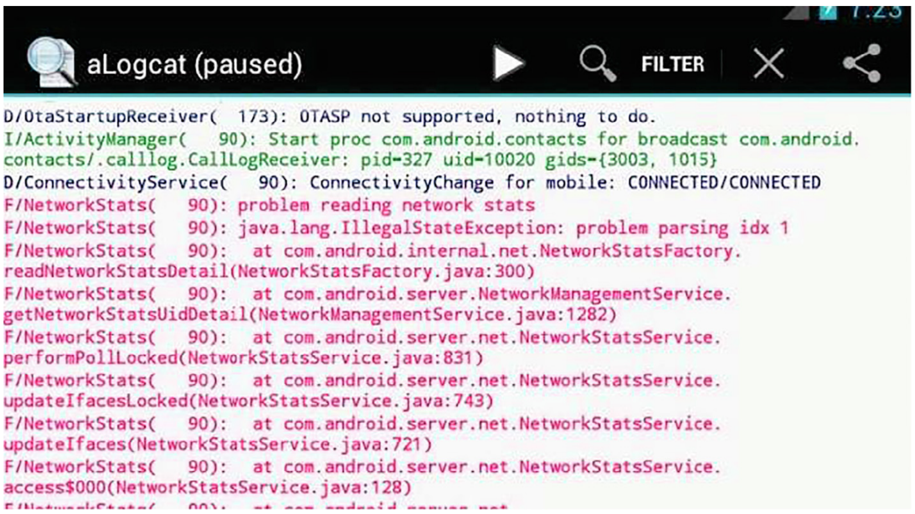
The analysis of the resultant source code is done using clover for Android to generate the number of branches and cyclomatic complexity. Based on the framework in Fig. 2, three parameters are used for the evaluation of our proposed approach. They are test path, complexity and branch coverage. The branch coverage is calculated manually to validate the detection and refactoring rules implemented in DART. An analysis of the Alogcat in detecting the lazy class is presented in Table 3.

From Table 3, four (4) lazy classes can be detected: Alogcat Application, Save Receiver, Save Service and Share Receiver. Alogcat Application.java has no callee or caller with just two (2) attributes and twelve (12) lines of code. Save Receiver has three (3) callees, no caller, one (1) attribute and eight (8) lines of code. Save Service has two (2) callees, one (1) caller, two (2) attributes and nine (9) lines of code. Share receiver has three (3) callees, no caller, one (1) attribute and eight (8) lines of code. The features of each of the afore-mentioned classes characterize them as lazy classes.

Fig. 4 displays the clover view for the original Alogcat application source code. The branches and complexity of refactored Alogcat based on the lazy class detected by DART is presented in Fig. 5. Based on Fig. 5, the lazy class has same number of branches and average method complexity as the original Alogcat code (as in Fig. 4) of 158 and 1.8 respectively with a reduction in total complexity of 253 from 254 (in Fig. 4).

Table 2
Analyzed Classes in Alogcat.

Source Code	Number of Methods	Lines of Code
LogActivity	38	455
LogCat	8	147
LogDumper	2	62
LogEntry	5	39
LogSaver	3	50
SaveReceiver	1	8
SaveService	2	9
ShareReceiver	1	8
ShareService	2	27



```
D/OtaStartupReceiver(  173): OTASP not supported, nothing to do.
I/ActivityManager(   90): Start proc com.android.contacts for broadcast com.android.
contacts/.calllog.CallLogReceiver: pid=327 uid=10020 gids={3003, 1015}
D/ConnectivityService(   90): ConnectivityChange for mobile: CONNECTED/CONNECTED
F/NetworkStats(   90): problem reading network stats
F/NetworkStats(   90): java.lang.IllegalStateException: problem parsing idx 1
F/NetworkStats(   90):   at com.android.internal.net.NetworkStatsFactory.
readNetworkStatsDetail(NetworkStatsFactory.java:300)
F/NetworkStats(   90):   at com.android.server.NetworkManagementService.
getNetworkStatsUidDetail(NetworkManagementService.java:1282)
F/NetworkStats(   90):   at com.android.server.net.NetworkStatsService.
performPollLocked(NetworkStatsService.java:831)
F/NetworkStats(   90):   at com.android.server.net.NetworkStatsService.
updateIfacesLocked(NetworkStatsService.java:743)
F/NetworkStats(   90):   at com.android.server.net.NetworkStatsService.
updateIfaces(NetworkStatsService.java:721)
F/NetworkStats(   90):   at com.android.server.net.NetworkStatsService.
access$000(NetworkStatsService.java:128)
```

Fig. 3. Overview of Alogcat Application.

Table 3
Analysis of alogcat to detect lazy class.

[HTML]C0C0C0 No	Name of Classes	Callees	Callers	Number of Attributes	Lines of code
1	Alogcat Application			2	12
2	Background Color		9,15	12	>15
3	Buffer		10,11 and 16	14	>15
4	Filter Dialog	9,15	9,15	>15	>15
5	Format		9,10,11,15	>15	>15
6	Intent	7,15	9,15,17,19	9	>15
7	Level		6,9,10,11,12.15,16	>15	>15
8	Lock		17,18,19,20	3	>15
9	LogActivity	2,4,5,6,7,10,12,			
13,14,15,16	4,10,20	>15	>15		
10	LogCat	3,5,7,9,15	9,18	>15	>15
11	LogDumper	5,7,15	14,20	3	>15
12	LogEntry	7	9,13	8	>15
13	LogEntry Adapter	12,15,21	9	10	>15
14	LogSaver	11,15	9	8	>15
15	Prefs	2,3,4,5,6,7,21	4,6,9,10,11,13, 14,16,20	>15	>15
16	PrefsActivity	3,7,15,21	9	>15	>15
17	SaveReceiver	6,8,18		1	8
18	SaveService	8,10	17	2	9
19	ShareReceiver	6,8,20		1	8
20	ShareService	8,9,11,15	19	2	>15
21	Textsize		13,15,16	>15	>15

Refactoring of small method and duplicated code smell is also done for the number of branches (test path) and cyclomatic complexity in Alogcat. Table 4 shows the comparison of results for Alogcat without using DART and using DART for refactoring of lazy class, small method and duplicated code smell.

Based on Table 4, it can be seen that the total complexity is reduced after refactoring with DART and the number of branches are also reduced after refactoring with DART (about 5% reduction).

We also test the branch coverage before and after the refactoring process and compare the results. The branch coverage is calculated for each of the affected classes in the source code using the following formula:

$$Branch\ Coverage = \frac{\text{Number of Lines of Code Covered}}{\text{Number of Total Lines of Code}} \times 100 \quad (1)$$

Table 5 shows the results of the branch coverage results before and after refactoring using DART.

From Table 5, the CC column shows the cyclomatic complexity (CC) for each refactored class while BC indicates the branch coverage (BC) of each class. From the CC column of the original code, the total cyclomatic complexity is calculated as:

$$\text{Total Complexity before refactoring}$$
$$= 1 + 1 + 77 + 7 + 2 + 28 + 15 + 1 + 11 = 142 \quad (2)$$

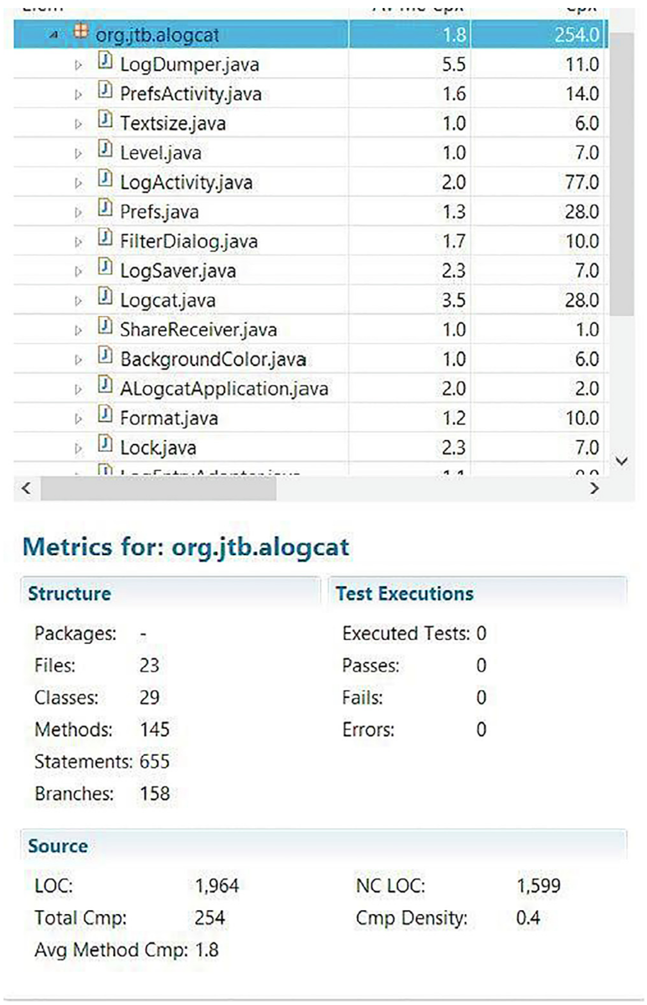
After refactoring, the total complexity is:

$$\text{Total Complexity after Refactoring}$$
$$= 1 + 71 + 8 + 4 + 23 + 14 + 1 + 9 = 131 \quad (3)$$

The reduction in cyclomatic complexity is calculated using the following formula:

$$Reduction\ in\ Cyclomatic\ Complexity = \frac{\text{NCCO} - \text{NCCR}}{\text{NCCO}} \times 100 \quad (4)$$

where
NCCR = Number of Cyclomatic Complexity after refactoring and
NCCO = Number of Cyclomatic Complexity before refactoring



org.jtb.alogcat	1.8	254.0
LogDumper.java	5.5	11.0
PrefsActivity.java	1.6	14.0
Textsize.java	1.0	6.0
Level.java	1.0	7.0
LogActivity.java	2.0	77.0
Prefs.java	1.3	28.0
FilterDialog.java	1.7	10.0
LogSaver.java	2.3	7.0
Logcat.java	3.5	28.0
ShareReceiver.java	1.0	1.0
BackgroundColor.java	1.0	6.0
ALogcatApplication.java	2.0	2.0
Format.java	1.2	10.0
Lock.java	2.3	7.0

Metrics for: org.jtb.alogcat

Structure

Packages: -
Files: 23
Classes: 29
Methods: 145
Statements: 655
Branches: 158

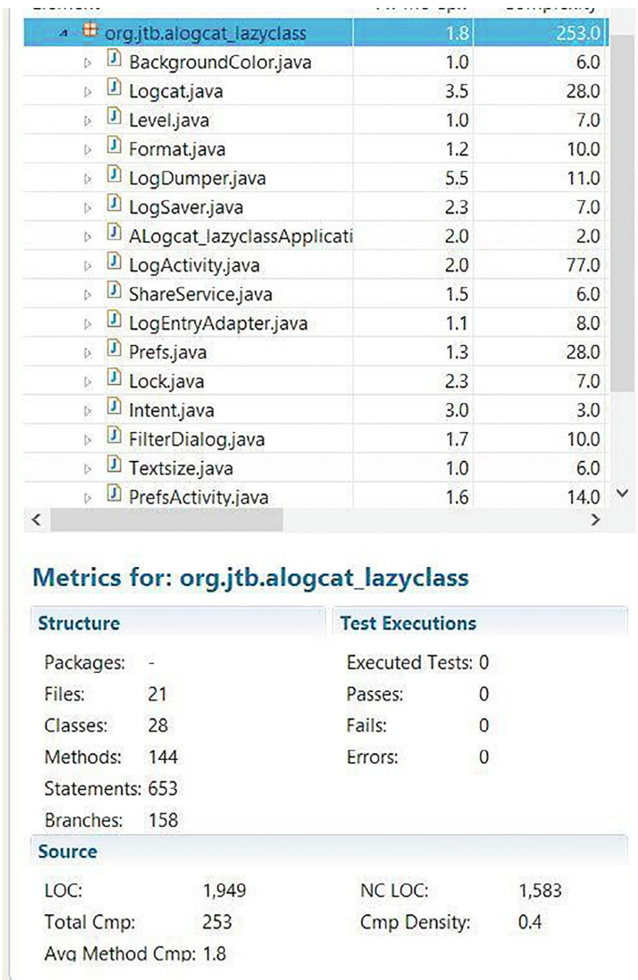
Test Executions

Executed Tests: 0
Passes: 0
Fails: 0
Errors: 0

Source

LOC: 1,964
Total Cmp: 254
Avg Method Cmp: 1.8
NC LOC: 1,599
Cmp Density: 0.4

Fig. 4. Branch and Complexity in Alogcat.



org.jtb.alogcat_lazyclass	1.8	253.0
BackgroundColor.java	1.0	6.0
Logcat.java	3.5	28.0
Level.java	1.0	7.0
Format.java	1.2	10.0
LogDumper.java	5.5	11.0
LogSaver.java	2.3	7.0
ALogcat_lazyclassApplicati	2.0	2.0
LogActivity.java	2.0	77.0
ShareService.java	1.5	6.0
LogEntryAdapter.java	1.1	8.0
Prefs.java	1.3	28.0
Lock.java	2.3	7.0
Intent.java	3.0	3.0
FilterDialog.java	1.7	10.0
Textsize.java	1.0	6.0
PrefsActivity.java	1.6	14.0

Metrics for: org.jtb.alogcat_lazyclass

Structure

Packages: -
Files: 21
Classes: 28
Methods: 144
Statements: 653
Branches: 158

Test Executions

Executed Tests: 0
Passes: 0
Fails: 0
Errors: 0

Source

LOC: 1,949
Total Cmp: 253
Avg Method Cmp: 1.8
NC LOC: 1,583
Cmp Density: 0.4

Fig. 5. Branch and Complexity in Alogcat – Refactoring of lazy class.

Table 4
Comparison of result for Alogcat before and after refactoring.

	Before DART	After Refactoring with DART		
	Original	Lazy Class	Small Method	Duplicate
Number of Classes	29	28	29	20
Number of Methods	145	144	140	146
Average Complexity	1.8	1.8	1.8	1.7
Total Complexity	254	253	248	249
Number of Branches	158	158	156	150

Table 5
Branch coverage results for refactored classes.

Classes	Before DART		After Refactoring with DART					
	Original		Lazy Class		Small Method		Duplicate	
	CC	BC	CC	BC	CC	BC	CC	BC
SaveReceiver	1	75%	1	81.8%	–	–	1	81.8%
ShareReceiver	1	77.7%			–	–		
LogActivity	77	93.3%	–	–	–	–	71	98.3%
LogSaver	7	84.4%	–	–	–	–	8	88.3%
ShareService	2	93.1%	–	–	–	–	4	93.1%
LogCat	28	85.3%	–	–	23	88%	–	–
LogEntry	15	88.6%	–	–	14	88.6%	–	–
SaveService	1	81.8%	1	90%	1	91.2%	–	–
LogDumper	11	95.1%	–	–	9	95.1%	–	–

Table 6
Generated Test cases 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 ondrop down	Access to Share	Access to share
LogActivity	TC3	Save to non existing path	Input a non-existing path	Error saving log	Error saving log
	TC4	Share HTML log	Click on share	Open the share dialog	Share dialog opened
	TC5	Access to menu item	Click on menu	Show menu Item	Menu item shown
LogSaver	TC6	Save to non existing path	Input a non-existing path	Error saving log	Error saving log
	TC7	Save to a path	Click on save	Open the save Dialog	Open the save dialog
	TC8	Save to existing path	Input an existing path	Buffer writes to path	Buffer wrote to path
ShareService	TC9	Share HTML log	Click on share	Open the share Dialog	Share dialog opened
	TC10	Start intent service	Click on share	Share dialog will open	Opened share dialog
SaveService	TC11	Handle intent for LogSaver	Click on save	Open the save Dialog	Save dialog opened

Table 7
Generated Test cases from refactored Alogcat source code.

Name of Classes	Test Cases	Scenario	Input	Expected Output	Actual Output
RequestReceiver	TC'1	Create intent for SaveReceiver andShareReceiver	Click on drop down	Access to save and share	Access to save and share
LogActivity	TC'2	Save to non-existing path	Input a non-existing path	Error saving log	Error saving log
	TC'3	Access to menu item	Click on menu	Show menu Item	Menu Item Shown
LogSaver	TC'4	Handle Intent for LogSaver	Click on save	Open the save dialog	Save dialog opened
	TC'5	Save to a path	Click on save	Open the save dialog	Open the save dialog
	TC'6	Save to existing path	Input an existing path	Buffer writes to path	Buffer writes to path
ShareService	TC'7	Share HTML log	Click on share	Open the share dialog	Share dialog opened
	TC'8	Start Intent service	Click on share	Share dialog will open	Opened share dialog

Based on the formula in (4), the percentage reduction in cyclomatic complexity is calculated using the results from Table 5, which shows that the cyclomatic complexity is reduced by 7.75%.

To validate the implementation of DART in terms of reducing the generation of test cases, test cases are generated from six (6) classes using Alogcat application source code before and after refactoring. Table 6 shows the test case generated from the original source code prior to refactoring while Table 7 shows the test cases generated after refactoring.

Based on Table 6, eleven test cases are generated from the six classes in Alogcat. However after refactoring, the test cases generated has reduced to eight test cases only Table 7 as follows:

$$TC = \{TC1, TC2, TC3, TC4, TC5, TC6, TC7, TC8, TC9, TC10, TC11\} \tag{5}$$

is reduced to

$$TC' = \{TC'1, TC'2, TC'3, TC'4, TC'5, TC'6, TC'7, TC'8\} \tag{6}$$

Therefore, the percentage reduction in number of test cases generation is approximately 28%.

6. Conclusion

This paper has discussed an approach to detect and refactor three code smells, which are lazy class, small method and duplicate. Code smell detection and refactoring are basic practices in the software development and maintenance environment, whereas refactoring has been established to improve the maintainability of software system. DART has been implemented in the Eclipse environment as a plug-in tool. The case study discussed in this paper shows that DART is able to reduce the generation of test cases from Alogcat application in an Android application. The results show that the number of generated test cases was reduced by 28% with an improvement of up to 5.0% in branch coverage. This implies that DART is able to reduce the redundancy in test cases, while there is a tendency to improve the efficiency of generated test cases. For future works, other code smells can be used in order to reduce the redundancy of generating test cases while maintaining or improving the branch coverage for test cases.

Acknowledgements

The authors would like to thank Malaysian Ministry of Higher Education (MoHE) for supporting this research under the Fundamental Research Grant Schema (FRGS).

References

Abbes, M., Khomh, F., Gueheneuc, Y.-G., Antoniol, G., 2011. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: 15th European Conference on Software Maintenance and Reengineering (CSMR2011), Oldenburg, 2011, 181–190.

Al Dallal, J., 2015. Identifying refactoring opportunities in object-oriented code: a systematic literature review. Inf. Softw. Technol. 58, 231–249.

Arcoverde, R., Garcia, A., Figueiredo, E., 2011. Understanding the longevity of code smells: preliminary results of an explanatory survey. In: Proceedings of the 4th Workshop on Refactoring Tools (WRT2011), Honolulu, USA, May 2011, 33–36.

Bavota, G., De Lucia, A., Marcus, A., Oliveto, R., 2014. Automating extract class refactoring: an improved method and its evaluation. Empirical Software Eng. 19, 1617–1664.

Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., Palomba, F., 2015. An experimental investigation on the innate relationship between quality and refactoring. J. Syst. Softw. 107, 1–14.

Fokaefs, M., Tsantalis, N., Stroulia, E., Chatzigeorgiou, A., 2011. JDeodorant: Identification and Application of Extract Class Refactorings. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE2011), Honolulu, USA, May 2011, 1037–1039.

Fontana, F.A., Zanoni, M., Marino, A., Mäntylä, M.V., 2013. Code Smell Detection: Towards a Machine Learning-Based Approach. In: 2013 IEEE International Conference on Software Maintenance (ICSM2013). Eindhoven, 2013, September 2013, 396–399.

Fontana, F.A., Mangiacavalli, M., Pochiero, D., Zanoni, M., 2015. On experimenting refactoring tools to remove code smells. In ACM Scientific Workshop Proceeding (XP2015), Helsinki, Finland, May 2015.

Fowler, M., 1999. Refactoring: improving the design of existing code. Pearson Education India. ISBN-13: 978-0201485677.

git repository, 2016. https://git-scm.com (Retrieved date: September 2016).

Hecht, G., Rouvoy, R., Moha, N., Duchien, L., 2015. Detecting antipatterns in android apps. In: Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft2015), Florence, Italy, May 2015, 148–149.

Khomh, F., Di Penta, M., Guéhéneuc, Y.-G., Antoniol, G., 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empirical Software Eng. 17, 243–275.

Lee, S.J., Lo, L.H., Chen, Y.C., Shen, S.M., 2016. Co-changing code volume prediction through association rule mining and linear regression model. Expert Syst. Appl. 45, 185–194.

Li, W., Shatnawi, R., 2007. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J. Syst. Softw. 80, 1120–1128.

Li, S., 2012. Juxtapp and DStruct: detection of similarity among android applications. Technical Report No. UCB/EECS-2012-111. University of California at Berkeley, May 2012. Available Online: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-111.html.

Mkaouer, M.W., Kessentini, M., Cinnéide, M.O., Hayashi, S., Deb, K., 2016. A robust multi-objective approach to balance severity and importance of refactoring opportunities. Empirical Software Eng. 22 (2), 894–927.

Moha, N., Gueheneuc, Y.G., Duchien, L., Le Meur, A.F., 2010. DECOR: a method for the specification and detection of code and design smells. IEEE Trans. Software Eng. 36, 20–36.

Palomba, F., Bavota, G., Oliveto, R., De Lucia, A., 2014. Anti-pattern detection: methods, challenges, and open issues. Adv. Comput. 95, 201–238.

Parnin, C., Görg, C., Nnadi, O., 2008. A catalogue of lightweight visualizations to support code smell inspection. In: Proceedings of the 4th ACM Symposium on Software Visuallization - SoftVis '08. https://doi.org/10.1145/1409720.1409733.

Rasool, G., Arshad, Z., 2017. A Lightweight Approach for Detection of Code Smells. Arabian J. Sci. Eng. 42 (2), 483–506.

SantosNeto, B.F.D., Ribeiro, M., Da Silva, V.T., Braga, C., De Lucena, C.J.P., De Barros Costa, E., 2015. AutoRefactoring: a platform to build refactoring agents. Expert Syst. Appl. 42, 1652–1664.

Silva, D., Terra R., Valente, M.T., 2015. JExtract: An Eclipse Plug-in for Recommending Automated Extract Method Refactorings. In: Brazilian Conference on Software: Theory and Practice (Tools Track), 2015, 1–8.

Sirqueira, T.F.M., Brandl, A.H.M., Pedro, E.J.P., Silva, R.D.S., Araújo, M.A.P., 2016. Code smell analyzer: a tool to teaching support of refactoring techniques source code. IEEE Lat. Am. Trans. 14, 877–884.

Ujhelyi, Z., Szőke, G., Horváth, A., Csiszár, N.I., Vidács, L., Varró, D., 2015. Performance comparison of query-based techniques for anti-pattern detection. Inf. Softw. Technol. 65, 147–165.