Lappeenranta University of Technology
School of Engineering Science
Degree Programme in Computer Science

Master’s Thesis

Gulshan Kundra

ENHANCING DEVELOPERS’ AWARENESS ON TEST SUITES’ QUALITY WITH TEST SMELL SUMMARIES

Examiner: Adjunct Professor (D.Sc.) Ossi Taipale
Professor Jari Porras

Supervisors: Adjunct Professor (D.Sc.) Ossi Taipale
Dr. Sebastiano Panichella
ABSTRACT

Lappeenranta University of Technology
School of Engineering Science
Degree Programme in Computer Science

Gulshan Kundra

Enhancing Developers’ Awareness on Test Suites’ Quality with Test Smell Summaries

Master’s Thesis

2018

67 pages.

Examiner: Adjunct Professor (D.Sc.) Ossi Taipale
Professor Jari Porras

Keywords: Test Smells, Test Smell Detection, Test Smell Summarization, Summary Generation, Summary Augmentation, Defect Analysis

This thesis work is a development and implementation of a part of the tool "TestSmellDescriber" which helps to improve software developers’ awareness on test suites’ quality. Producing and maintaining superior quality test suites is bug susceptible and time consuming, thus, a huge relevant part of effort and time is spent by software developers for writing and maintaining premium quality test suites, which accounts up to fifty percent of the complete software development costs. As, writing and retaining fine quality test suites is highly error prone and very often leads to introduction of test smells. In the recent empirical studies, it is reported that, presence of test smells affects and complexes both, the effectiveness of the software quality assurance and maintainability of the test code. This thesis work has improved the functionality of the tool "TestSmellDescriber" by automatically generating and augmenting test smells summaries as comments at suite and method level in the tests, thereby improving software developers’ awareness on test suites’ quality and it can also complement automated detection and removal of textual and structural smells.
ACKNOWLEDGEMENTS

I like to express my deep gratitude to Dr. Sebastiano Panichella, my thesis supervisor for giving me this great opportunity to complete my master’s thesis in Software Evolution and Architecture Lab (S.E.A.L), Department of Informatics at University of Zurich (UZH). His patient guidance, instructions, supervision, and help played a vital role for completion of this thesis successfully. It would not have been possible without his direction and advice. I like to thank to Prof. Harald Gall, for allowing me to work in the Software Evolution and Architecture Lab and make use of resources available in the lab.

I like to acknowledge Adjunct Prof. D.Sc. Ossi Taipale, Lappeenranta University of Technology, as the first examiner of this thesis and I am gratefully indebted to him for all the encouragements, valuable guidance and support, since the time I chose this opportunity for my master’s thesis.

A very special thanks goes to Ivan Taraca (Student at UZH), for all the constructive discussions, pleasant teamwork and for providing valuable recommendations during the research work.

Finally, I must express my profound gratitude to my parents and my siblings for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of successful completion of this thesis. This accomplishment would not have been possible without them. Thank you.

Lappeenranta, October 25, 2018

Gulshan Kundra
# CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTRODUCTION AND BACKGROUND</td>
<td>8</td>
</tr>
<tr>
<td>1.1</td>
<td>Introduction to the Research Topic</td>
<td>8</td>
</tr>
<tr>
<td>1.2</td>
<td>Background and Related Work</td>
<td>10</td>
</tr>
<tr>
<td>1.3</td>
<td>Problem Statement and Motivation</td>
<td>12</td>
</tr>
<tr>
<td>1.4</td>
<td>Thesis Structure</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>THEORETICAL FRAMEWORK AND LITERATURE REVIEW</td>
<td>15</td>
</tr>
<tr>
<td>2.1</td>
<td>Inception: Smell and Refactoring</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>Code Smell</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Test Smell</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Tools for Smell Detection</td>
<td>25</td>
</tr>
<tr>
<td>2.5</td>
<td>Automatic Software Summarization</td>
<td>29</td>
</tr>
<tr>
<td>3</td>
<td>APPROACH AND TOOL</td>
<td>33</td>
</tr>
<tr>
<td>3.1</td>
<td>Research Approach Overview</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Detection of Smell(s)</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Description Generation</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>Description Augmentation</td>
<td>37</td>
</tr>
<tr>
<td>4</td>
<td>STUDY AND EXPERIMENTS</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Research Work Objective</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Research Work Context</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Experimental Procedure</td>
<td>41</td>
</tr>
<tr>
<td>4.4</td>
<td>Research Method</td>
<td>43</td>
</tr>
<tr>
<td>5</td>
<td>EVALUATION</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>RQ₁: Discerned Usefulness and Quality of Smell Descriptions</td>
<td>45</td>
</tr>
<tr>
<td>5.2</td>
<td>RQ₂: Effect on Maintenance Tasks</td>
<td>47</td>
</tr>
<tr>
<td>6</td>
<td>CONCLUSION AND FUTURE WORK</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>REFERENCES</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>APPENDICES</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Appendix 1: Templates Used to Generate Smell Descriptions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Appendix 2: Emails Sent to Participants</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Appendix 3: Emails Sent to Participants Continue.</td>
<td></td>
</tr>
</tbody>
</table>
Appendix 4: Information Regarding Smells for Participants
Appendix 5: Information Regarding Smells for Participants Continue....
LIST OF ABBREVIATIONS

Advice: All symbols and abbreviations are listed on this page in the alphabetical order. Remember to introduce the abbreviation when it is used in the text for the first time.

CD Continuous Delivery
CI Continuous Integration
RQ Research Question
E-commerce Electronic Commerce
VOIP Voice Over Internet Protocol
SWUM Software Word Usage Model
LOC Lines of Code

List of Tables

Table 1 Fowler’s list of code smells ........................................ 17
Table 2 Fowler’s list of refactorings for code smells .................... 18
Table 3 Mika Mäntylä’s taxonomy of smells ............................ 20
Table 4 Test smells introduced by Van Deursen ........................ 22
Table 5 Van Deursen’s list of test smell refactorings .................... 23
Table 6 Java classes of Apache OFBiz .................................... 40
Table 7 Test cases of Apache OFBiz ...................................... 40
Table 8 Participant’s experience ............................................. 41
Table 9 Raw data collected post experiment ............................ 45
Table 10 Raw statistics concerning the evaluation of produced summaries. . 46
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>TACO’s Smell Detection Process [Palomba, 2015a]</td>
<td>27</td>
</tr>
<tr>
<td>Figure 2</td>
<td>(a) DECOR (b) DETEX Detection Technique [Moha et al., 2010]</td>
<td>28</td>
</tr>
<tr>
<td>Figure 3</td>
<td>TestSmellDescriber Overview</td>
<td>34</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Part of Test Suite Level Descriptions for <code>UtilCacheTest.java</code></td>
<td>38</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Part of Test Method Level Descriptions for <code>UtilCacheTest.assertKey</code></td>
<td>38</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Smells removed WITH and WITHOUT summaries.</td>
<td>48</td>
</tr>
<tr>
<td>Figure A1.1</td>
<td></td>
<td>63</td>
</tr>
<tr>
<td>Figure A2.1</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>Figure A3.1</td>
<td></td>
<td>65</td>
</tr>
<tr>
<td>Figure A4.1</td>
<td></td>
<td>66</td>
</tr>
<tr>
<td>Figure A5.1</td>
<td></td>
<td>67</td>
</tr>
</tbody>
</table>
1 INTRODUCTION AND BACKGROUND

Enterprise software applications are a spine of almost all types of industries ranging from health to defense to agriculture, in the present era [Andreessen, 2011]. Technological and economic shift is going on, where in, software companies are taking over massive parts of the economy. Various types of businesses are automating their processes and services with the help of software applications and delivering them as online services, such as transportation, Electronic Commerce (E-Commerce) and many more [Andreessen, 2011]. Transformation of industries with the help of software technology is a trend, and presence of software is ubiquitous. Not only on an industry level, at an individual level software has amused and captured every moment of billions of people, out of our whole population, with the help of broadband internet, smart-phones and ample other types of gadgets [Andreessen, 2011].

1.1 Introduction to the Research Topic

Software revolution is taking place in all types of industries, this includes software-based industries, quite obvious and the world is entering the Fourth Industrial Revolution [Andreessen, 2011,Baller, 2016]. Information and Communication technologies are the backbone of the revolution [Baller, 2016]. To fulfill the massive requirement, humongous amount of software is being used and produced every single day, thus development of qualitative software is an eternal challenge [Andreessen, 2011].

A very close collaboration is needed for software development in the present era of highly available and interconnected systems, among the members of software development team [Riungu-Kalliosuari et al., 2016]. Software is being used and can be updated at the same point of time, due to the change in software distribution process. Since, recurring software updates and high frequency of the releases, monitoring production systems is very significant to provide optimum and ideal user experience. Thus, software industries need to modify their practices and revamp to several alterations introduced by new concepts such as DevOps [Riungu-Kalliosuari et al., 2016]. DevOps is a set practices, aiming to decrease the time consumption between a change to a system and the change being placed into normal production, while ensuring superior quality [Bass et al., 2015]. DevOps helps to enlarge the collaboration among development and operations teams, to facilitate the management of modifications in the production environment [Riungu-Kalliosuari et al., 2016].
Software developers’ are spending a lot of time and effort to develop superior quality software products [Brooks, 1975, Moonen et al., 2008]. In order to ensure software’s quality, a very relevant and significant part of developers’ time and effort is invested into production and maintenance of test suites [Brooks, 1975, Moonen et al., 2008]. This sums up to 50 percent of the entire software development costs [Brooks, 1975, Moonen et al., 2008]. Since, writing and preserving or retaining superior quality test suites is highly bug-susceptible task as well as time consuming, which realistically, very often leads to origination of test or code smells i.e. symptoms of poor design or implementation choices [Tufano et al., 2015]. Code smells indicate portion(s) of the code, which are sources of bugs and errors. In recent empirical studies, it is reported that, the presence of code smells considerably reduces the understandability and maintainability of test code, increases the fault-proneness, and complicates the effectiveness of software quality assurance tasks [Tufano et al., 2015].

Automated test generation tools have been extensively investigated to enhance the quality of test suites [Rafi et al., 2012]. Counted number of automated test generation tools have been recommended to functionally support software developers for identification and removal of test or code smells. This thesis work is an enhancement of a proposed approach "TestSmellDescriber", which helps software developers by automatically generating description of a code portion which is affected by smell(s).

In this study a part of the tool named "TestSmellDescriber" has been developed and implemented, which automatically generates descriptions or summaries at suite level and method level, respectively. If the code is affected with textual or structural smell, "TestSmellDescriber", identifies the smell, automatically generates summaries and augment descriptions as comments. Descriptions generated by "TestSmellDescriber" also includes various methods which recommends or suggests the refactoring operations necessary for smell removal from the code, thereby supplementing software developer’s awareness about and ability to detect and fix test and code smells.

It can be argued and predicted that the enhanced approach "TestSmellDescriber" can complement current techniques available around the automated test smell detection. Upon an evaluation of this enhanced approach, which was performed with the help of a survey coupled with test case improvement tasks, it has been figured out that,

- Test smell summaries and descriptions remarkably intensify software developers’ awareness on test suite quality.
- Software developers were able to detect and fix twice as many as test smells.
Thus, the enhanced "TestSmellDescriber" approach is considered supportive as well as convenient for software developers.

1.2 Background and Related Work

A software development discipline where a software is built in such a way that it can be released to production at any point of time is referred to as Continuous Delivery (CD) [Humble and Farley, 2010]. It defines a set of practices aiming to deliver software with a greater pace to respond to market needs [Humble and Farley, 2010]. It is an agile practice, which helps us to achieve various benefits, such as: low risk releases, higher quality, lower costs, better products [Humble and Farley, 2010]. Goal of CD is to make deployments foreseeable of systems, whether the system is an application or a large-scale distributed system [Humble and Farley, 2010]. Thus, it becomes mandatory for software developers to regularly and continuously update the source code as well as elevate the level of complexity of modern software systems rapidly [Lehman, 1980]. Therefore, continuous software modifications take place under strict time pressure, consequently originating so called technical debt (All the sacrifices made by developers for releasing the software on time) [Cunningham, 1993], as well as adversely affecting maintenance and evolution tasks.

Continuous Integration (CI) enabler of DevOps focuses to bring modifications into production as fast as possible, without affecting software quality [Zhao et al., 2017]. Emphasis of DevOps is on automation of developing, testing and deploying software processes [Zhao et al., 2017]. CI is potentially capable of speeding up software development processes as well as quality of the code remains unaffected, since it automatically builds and tests a project’s code base, in isolation, every time when the code gets modified i.e. push commit, pull request [Zhao et al., 2017].

Many software development companies across the globe have initiated adopting CD and CI practices, since, these practices help to produce high quality products through quick iterations [Humble and Farley, 2010, Duvall et al., 2007, Savor et al., 2016]

A single software bug can cripple a whole enterprise, in the year 2016, 1.1 trillion in assets and 4.4 billion people were impacted by software failures [Panichella, 2018]. With the Agile movement, software testing influences the effectiveness of the complete software development process [Hilton et al., 2017, Hilton et al., 2016] as well as it plays a remarkable role in ensuring the quality of massive and complex software systems for instance:
finding software bugs in various aimed environment [Fowler and Foemmel, 2006]. Software testing is the key to ensure software’s quality and it is a set of activities, intended to find bugs in a software product to enhance the quality [Saini and Rai, 2013]. Testing is performed to check whether the expected results are the actual results and the system is defect free [Saini and Rai, 2013, Van Deursen et al., 2001]. Testing is helpful in verification of complex functionality and unexpected circumstances as well as to identify the correctness and completeness of the developed software [Saini and Rai, 2013, Van Deursen et al., 2001]. As a consequence, producing and maintaining superior quality test suites is bug susceptible and time consuming, thus, a lot of effort and time is spent by software developers for writing and maintaining premium quality test suites, which accounts up to fifty percent of the complete software development costs [Brooks, 1975, Moonen et al., 2008]. As, writing and retaining fine quality test suites is highly error prone and in practice, it often leads to "poor design and implementation choices" [Tufano et al., 2015]. Additionally, Rapid releases allows quicker time-to-market and user feedback as well as on the negative side, it also implies less time developers get to test and bug fixing, hence, increased complications in the quality assurance tasks and testing scope is being narrowed down [Mäntylä et al., 2015]. Rapid release schedules are remarkably affecting software’s quality, organizations have less time to stabilize platforms and proportionally less amount of bug fixes, led to crashes of initial executions due to bug presence [Mäntylä et al., 2015]. A huge amount of software is in production, increases the monetary amount. Contrast in testing results cannot be accounted for, as manual testing can be done in various different ways [Rafi et al., 2012].

With the aims of minimizing the cost of testing activities [Fraser and Arcuri, 2015, Panichella et al., 2015, Ricca and Tonella, 2001, Tonella, 2004], to enhance the quality of available test suites [Rafi et al., 2012], and to find violations of automated oracles [Bertrand Meyer and Liu, 2007, Csallner, 2004, Fraser and Arcuri, 2015, Pacheco and Ernst, 2007, Panichella et al., 2016], extensive research has been done within automated test generation tools. Traditionally, only one or two goals were focused by automated unit test generation techniques so that test oracles can be manually added, either produce representative test suites, for instance, satisfying branch coverage or violations of automated oracles, for instance, undeclared exceptions [Fraser and Arcuri, 2015]. For object-oriented programs, automated test case generation is significantly difficult, hence, a class method’s test case includes, object creation, change of its internal state (optional) and invocation of the method being tested [Tonella, 2004]. In software development, testing activity is an important but unpleasant part of quality assurance, since it involves test case and test oracles preparation and they require a lot of effort. As these tasks are monotonous and tiresome, testing is not rigorous enough to vanish all the bugs [Bertrand Meyer and
According to the US National Institute of Standards and Technology [RTI, 2002] cost of deficient software testing in the year 2002 was $59.5 billion. The key hindrance is the involvement of large amount of manual testing activities [Bertrand Meyer and Liu, 2007]. Progress has been observed with the emergence of testing frameworks, for example, JUnit and more automation is necessary for improvements in testing activities [Bertrand Meyer and Liu, 2007].

Manually and automatically generated tests’ quality is affected by the presence of test smells [Bavota et al., 2015b, Deursen et al., 2001, Garousi and Küçük, 2018], which also affects the effectiveness of software quality assurance tasks [Bavota et al., 2015b, Panichella et al., 2016] and substantially affect the maintainability of test code [Palomba et al.,]. Due to these reasons, researchers have developed automated tools for detection of smells,(for e.g. [Tsantalis and Chatzigeorgiou, 2009b]) while continuous delivery and continuous integration processes [Bakota et al., 2014, Szoke et al., 2015].

1.3 Problem Statement and Motivation

In spite of the accessibility to automated solutions, few tools have been suggested to productively support and help software developers for identification and removal of test smells [Greiler et al., 2013, Khomh et al., 2009]. It can be argued that, since software systems are being evolved, it would be beneficial for software developers if tools help them update or evolve test suites, for instance, automatic synchronization with the new version of the system or rectifying test suites by eliminating test and code smells from test code and production code [Bavota et al., 2015b, Deursen et al., 2001, Tsantalis and Chatzigeorgiou, 2009b]. The belief is, to successfully accomplish this task, it is important that software developers should be supplied precise whereabouts of "Smells" [Fowler, 2002], for which refactoring activities are desirable.

This research study is constructed on the findings that maintainability and comprehensibility of test cases are essential elements to be optimized in association with both, automated and manually generated tests. [Bavota et al., 2015b, Daka et al., 2015, Daka et al., 2017, Deursen et al., 2001, Palomba et al.,, Panichella et al., 2016]. The belief is that, software developers will be benefited, if an approach provide aid to improve and complement available test cases by pointing at sections of the code contains bugs or errors and regions where poor design decisions are applied.

To battle with this problem, this thesis work has developed and implemented a part of
an approach "TestSmellDescriber", which, upon implementation, automatically generates test case summaries of the code region which is affected by textual [Palomba et al., 2016] or structural [Bavota et al., 2015b, Deursen et al., 2001, Tsantalis and Chatzigeorgiou, 2009b] test smell(s), at suite and method level, for every individual test. Consequently, enhancing software developers’ awareness on test suites’ quality. The generated summaries, plus methods detailing and suggesting the necessary refactoring operations are directly augmented as comments in the test code. This design acts as a code improvement enabler for software developers.

The belief is that, combining summarization techniques [Moreno and Marcus, 2017] with test/code smell(s) analysis [Deursen et al., 2001, Palomba et al., 2016, Tsantalis and Chatzigeorgiou, 2009b] helps software developers to get greater awareness of test suites’ quality as well as can be advantageous to support developers to enhance test suites maintainability. Thus, it leads to first research question (RQ):

**RQ1:** Are test case summaries enriched by test smell information considered to be useful by developers?

Available test smells analysis and detection tools are not promptly usable or practicable by software developers, since they generate metrics as outputs, which are often difficult to analyse or comprehend and to execute necessary refactoring operations. Therefore, software developers go through the assertions manually, to confirm preciseness and perhaps, update new tests to improve the whole test suite’s quality. The belief is that, this enhanced approach possesses calibre to influence developers’ ability for performing evolution and maintenance tasks. Thus, it leads to second RQ:

**RQ2:** How do test smell summaries impact developers’ ability to improve test suite’s quality?

An empirical study has been conducted to evaluate the enhanced approach "TestSmellDescriber“ and also to investigate the impact of provided summaries of the smell(s) on, firstly, software developer’s ability to improve/evolve test suites and secondly, perception of developers on test suite’s quality. The study was conducted among a blended group of 21 participants from academia and industry. The empirical study involved a survey coupled with two test suite’s improvement tasks, both must be completed by each participant based on their software developing experience. First task was to improve the test suite
without the help and presence of smell summaries and second task was to improve the test suite with the help and presence of smell summaries.

1.4 Thesis Structure

The thesis begins with an introduction to provide an overview about the research topic, followed by a related scientific work that briefly details most recent research work that has been done and outcomes. Furthermore, it is followed by a problem statement and motivation chapter, which addresses the gap in the knowledge and prior research work that this research work has attempted to complete and enhance.

The chapter titled "Theoretical framework and literature review" provides insights into the about specific and precise background information on the research topic as well as it calculates the related work done previously.

The chapter titled "Approach and Tool" illustrates the utilized approach to successfully complete the research work as well as whole procedure to detect smell and generate smell descriptions.

The chapter titled "Study and Experiments" describes ultimate objective of the research work and its context. It measures the effectiveness and usefulness of the build tool "Test Smell Descriptor" with respect to automatic smell detection and removal.

The chapter titled "Evaluation" is about results and discussions, under which all the experimental data is summarized quantitatively. It answers the key research questions regarding usefulness and quality of smell descriptions as well as effect on maintenance tasks.

Lastly, conclusions have been drawn to emphasize over, whether the research work objectives and aims have been achieved and what can be done in the future to enhance the research work.
2 THEORETICAL FRAMEWORK AND LITERATURE REVIEW

This chapter is about introduction of smell and refactoring techniques. It also explains how code smell and test smell are different from each other. Brief explanation about code smells and related refactoring approaches introduced by Fowler in his book [Fowler, 2002]. Brief explanation about test smells and related refactoring approaches introduced by Van Deursen [Deursen et al., 2001]. Lastly, it details about existing tools for smell detection.

2.1 Inception: Smell and Refactoring

The term "Code Smell" appeared in late 1990s, introduced and coined by Kent Beck in a book entitled "Refactoring: Improving the design of existing code", written by Martin Fowler [Fowler, 2002]. Fowler’s realization about knowing how to operate the mechanics of a refactoring is as important as to deciding when to initiate refactoring and when to stop [Fowler, 2002]. The desire of a solid notion instead a vague one to describe when to refactor, Martin Fowler visited Kent Beck in Zurich, where Kent Beck had come up with the notion describing "When" of refactoring in terms of "Smells" [Fowler, 2002]. Martin Fowler, in his book, also stated that, smells cannot underline the precise criteria for when a refactoring is overdue, instead smells are acting only as indications that there is trouble in the code which can be overcome by refactoring activities [Fowler, 2002].

Refactoring, as a noun referred to alter the interior structure of software to make it easier to comprehend and economical to modify without altering its observable behavior [Fowler, 2002], such as, "Extract Method" and "Pull Up Field". Usually, refactoring is a little alteration to the software but it can involve other refactorings [Fowler, 2002]. Usage of refactoring is the verb form "Refactor", referred to restructure software by applying a series of refactorings without altering its observable behavior [Fowler, 2002]. Refactoring techniques can help for cleaning up the code with efficiency and decrease the number of smells [Fowler, 2002, Abbes et al., 2011], therefore, enhancing the comprehension of code as well as simplifying development and maintenance activities.
2.2 Code Smell

The term "Bad Code Smell" or "Code Smell" or "Smell" refers to any symptom in the source code, which perhaps highlights or indicates a deeper problem [Umesh and G N, 2015, Tufano et al., 2015]. Depreciation in the software quality over a period of time is obvious due to various factors, such as inconsistent design and improper requirement analysis in the early stages of software development and software ageing [Umesh and G N, 2015]. Code smell do not restrict a software’s normal functionality, since it is neither a bug or an error nor a technically incorrect code [Umesh and G N, 2015], instead, they behave as a pointer towards weaknesses in design, that may be contributing to slow-down the development process or multiplying the risk of bugs or failures in the future. Code smells are certain code structures that proposes (sometimes they scream for) a possibility of refactoring [Fowler, 2002]. Code smell is defined by Martin Fowler as "indications to bad design and poor implementation choices" in the code and refactoring activities can be applied to overcome these code smells [Fowler, 2002].

For instance, Duplicate code is a smell, it tops the list of smells and it refers to an identical code structure present in more than one place within the code [Fowler, 2002]. Unifying all the duplicate code may result better performance of the program [Fowler, 2002]. Code is tougher to debug and maintain if it is duplicated [Khomh et al., 2012]. Duplicate code smell can appear in various ways in the code, such as [Fowler, 2002],

- Presence of identical expression in more than one method of the same class and it can be refactored with the help of extract method plus invoking the code from both the places.
- Presence of identical expression in two sibling sub-classes, this can be refactored by applying extract method in both the classes, upon that pull up field.
- If more than one method produces same outcome by using different algorithms, apply substitute algorithm refactoring technique.
- Identical code in two unrelated classes

Following is the table 1, which contains names and descriptions of 22 code smells described by Martin Fowler in his book Refactoring: Improving the design of existing code [Fowler, 2002].
Table 1. Fowler’s list of code smells

<table>
<thead>
<tr>
<th>Smell Name</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duplicated Code</td>
<td>Identical expression in more than one method of the same class, in two sibling sub-classes, in two unrelated classes, and more than one method producing identical output by using different algorithm.</td>
</tr>
<tr>
<td>Long Method</td>
<td>A procedure with lots of parameters, temporary variables and too many lines of code as well as it does more than one thing.</td>
</tr>
<tr>
<td>Large Class</td>
<td>A class with too many instance variables and tries to perform too many tasks. Often also includes duplicate code.</td>
</tr>
<tr>
<td>Long Parameter List</td>
<td>A procedure which needs too many parameters.</td>
</tr>
<tr>
<td>Divergent Change</td>
<td>One class is frequently changed for various different reasons in various different ways.</td>
</tr>
<tr>
<td>Shotgun Surgery</td>
<td>If a kind of modification in a class mandates a lot of slight modifications in many different classes and all the modifications are all over the place.</td>
</tr>
<tr>
<td>Feature Envy</td>
<td>A procedure on one class is interested and needs many procedures of another class.</td>
</tr>
<tr>
<td>Data Clumps</td>
<td>Date items in clumps or bunches are often utilized jointly by a field or as parameters.</td>
</tr>
<tr>
<td>Primitive Obsession</td>
<td>Primitive types are used instead of small objects for simple tasks or a record type containing primitive types, replaces primitive types.</td>
</tr>
<tr>
<td>Switch Statements</td>
<td>When the identical switch statement is spreaded in different places in a program.</td>
</tr>
<tr>
<td>Parallel Inheritance Hierarchies</td>
<td>It is a unique case of shotgun surgery and when producing a sub-class of a class mandates the creation of a sub-class of another class.</td>
</tr>
<tr>
<td>Lazy Class</td>
<td>If a class is not doing enough to justify its existence in the code.</td>
</tr>
<tr>
<td>Speculative Generality</td>
<td>If abstract classes are not doing much or hooks and special cases are added to a class and might be required someday.</td>
</tr>
<tr>
<td>Temporary Field</td>
<td>If an instance variable is set for few definite circumstances.</td>
</tr>
</tbody>
</table>

Continuation on the next page
Continuation of Table 1

<table>
<thead>
<tr>
<th>Smell Name</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Chain</td>
<td>If a class demands one object for another object, which the class then demands for yet another object and so on. It can be seen as a long line of getThis methods.</td>
</tr>
<tr>
<td>Middle Man</td>
<td>Encapsulation often becomes delegation and if half the methods of a class’s interface delegate to another class.</td>
</tr>
<tr>
<td>Inappropriate Intimacy</td>
<td>If classes become very intimate and require too many fields and methods of each other.</td>
</tr>
<tr>
<td>Alternative Classes with Different Interfaces</td>
<td>When two classes have different procedure names but they perform the same functions.</td>
</tr>
<tr>
<td>Incomplete Library Class</td>
<td>When you require a library to have certain features and do something you would like to do.</td>
</tr>
<tr>
<td>Data Class</td>
<td>When the classes only have fields and, getters and setters methods for those fields, nothing else.</td>
</tr>
<tr>
<td>Refused Bequest</td>
<td>When sub-classes get to inherit the methods and data from super-classes, but they do not need all the inherited methods and fields.</td>
</tr>
<tr>
<td>Comments</td>
<td>Comments often lead us to smell if they are trying to cover poorly written code.</td>
</tr>
</tbody>
</table>

The smells described by Martin Fowler mentioned in the above table 1, can be terminated with the help of various refactoring techniques. The book written by Martin Fowler, Refactoring: Improving the design of existing code [Fowler, 2002], also describes and explains the use of refactoring techniques and how to apply a particular refactoring technique to remove a particular smell from the code. Martin Fowler has stated and described various occasions where smells can take place as well as explained refactoring techniques and how to apply refactoring for those occasions but not all the occasions have been defined [Fowler, 2002]. The below table 2 includes refactorings for the most usual occasions, described by Martin Fowler [Fowler, 2002].

Table 2. Fowler’s list of refactorings for code smells

<table>
<thead>
<tr>
<th>Smell Name</th>
<th>Suggested Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duplicated Code</td>
<td>Extract the duplicated procedures and invoke the code from both places.</td>
</tr>
<tr>
<td>Smell Name</td>
<td>Suggested Refactoring</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Long Method</td>
<td>Locate and extract various parts of the method that seems to go well together, into a new method.</td>
</tr>
<tr>
<td>Large Class</td>
<td>Cluster instance variables and extract those into new classes.</td>
</tr>
<tr>
<td>Long Parameter List</td>
<td>Replace parameter with method and get the data in one parameter by requesting an object you already know about or pass the whole object if they belong to one object.</td>
</tr>
<tr>
<td>Divergent Change</td>
<td>Identify and extract the parts that change for a particular cause into new classes.</td>
</tr>
<tr>
<td>Shotgun Surgery</td>
<td>With the help of extracting fields and methods from their original class, move all elements that requires modification into one entity.</td>
</tr>
<tr>
<td>Feature Envy</td>
<td>Feature envy method should be moved to the class, from which it uses the maximum data.</td>
</tr>
<tr>
<td>Data Clumps</td>
<td>New object should be formed by extracting the data fields and simplify the method call by passing the whole object.</td>
</tr>
<tr>
<td>Primitive Obsession</td>
<td>Set of primitive types should be replaced with an object that holds all the data.</td>
</tr>
<tr>
<td>Switch Statements</td>
<td>Extract the switch statement and then use move method to get it onto the class where polymorphism is required.</td>
</tr>
<tr>
<td>Parallel Inheritance Hierarchies</td>
<td>Occurrence of one hierarchy should refer to the occurrence of the other.</td>
</tr>
<tr>
<td>Lazy Class</td>
<td>Terminate the whole class or nearly useless components should be subjected to Inline class.</td>
</tr>
<tr>
<td>Speculative Generality</td>
<td>Eliminating unused parameters to remove the generality, rename abstractly named methods, eliminate abstract classes or remove the unnecessary delegation with inline classes.</td>
</tr>
<tr>
<td>Temporary Field</td>
<td>Create a new class and move the instance variable and all the code that concerns the variable into that class.</td>
</tr>
<tr>
<td>Message Chain</td>
<td>To hide the delegation, create a method or move the method that performed the message chain into the right object.</td>
</tr>
<tr>
<td>Middle Man</td>
<td>Terminate the middle man and contact the object directly.</td>
</tr>
<tr>
<td>Inappropriate Intimacy</td>
<td>Try to arrange a change bidirectional association to unidirectional. Move the method or field that requires another field or method into the respective class.</td>
</tr>
</tbody>
</table>
Continuation of Table 2

<table>
<thead>
<tr>
<th>Smell Name</th>
<th>Suggested Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternative Classes with Different Interfaces</td>
<td>Move the method into the respective class or rename the methods to make them identical.</td>
</tr>
<tr>
<td>Incomplete Library Class</td>
<td>Generate a procedure in your class with an instance of the library class as its first argument.</td>
</tr>
<tr>
<td>Data Class</td>
<td>Encapsulate public fields to hide them, or find methods or part of methods that fit better in the class and move them.</td>
</tr>
<tr>
<td>Refused Bequest</td>
<td>Generate a new sibling class and push all the unused methods to the sibling with the help of push down method and push down field.</td>
</tr>
<tr>
<td>Comments</td>
<td>Extract part of the method that needs to be commented into a new class and rename that method to reflect what the comment states. Introduce assertion if you are required to state some protocols regarding the necessary state of the system.</td>
</tr>
</tbody>
</table>

Mika Mäntylä studied and gained insights about code smells introduced by Martin Fowler and concluded that a flat list of 22 code smells, makes tough to understand all the smells, relationship between smells remains unrecognized and for each smell, larger context is unaccounted [Mäntylä et al., 2003]. He also stated that a taxonomy of smells is easier to understand and in order to address about mentioned problems he has created a taxonomy for the smells. Taxonomy is generated based on few common concepts shared by various smells within a cluster [Mäntylä et al., 2003]. The below table 3, includes categories and their description provided by Mika Mäntylä [Mäntylä et al., 2003] as well as they are mapped with 22 code smells introduced by Martin Fowler [Fowler, 2002].

Table 3. Mika Mäntylä’s taxonomy of smells

<table>
<thead>
<tr>
<th>Category</th>
<th>Category Description (Associated smells)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Bloaters</td>
<td>Whole cluster of Bloaters represents something cannot be handled effectively, since it has grown massive (Long Method, Large Class, Primitive Obsession, Long Parameter List, and Data Clumps).</td>
</tr>
</tbody>
</table>

Continuation on the next page
Continuation of Table 3

<table>
<thead>
<tr>
<th>Category</th>
<th>Category Description (Associated smells)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Object-Orientation Abusers</td>
<td>This cluster represents cases where, possibilities of object-oriented design have not been fully exploited by the solution of those cases. (<em>Switch Statements, Temporary Field, Refused Bequest, Alternate Classes with Different Interfaces, and Parallel Inheritance Hierarchies</em>).</td>
</tr>
<tr>
<td>The Change Preventers</td>
<td>Alteration or further development of software is obstructed or prevented by this bunch of smells (<em>Divergent Change and Shotgun Surgery</em>).</td>
</tr>
<tr>
<td>The Dispensables</td>
<td>They represent useless or unnecessary elements in the source code that should be removed (<em>Lazy Class, Data Class, Duplicate Code, and Speculative Generality</em>).</td>
</tr>
<tr>
<td>The Encapsulators</td>
<td>These two smells are related to and deal with data communication mechanism or encapsulation (<em>Message Chains and Middle Man</em>).</td>
</tr>
<tr>
<td>The Couplers</td>
<td>Minimal coupling between objects, a design principle of object-oriented is violated (<em>Feature Envy and Inappropriate Intimacy</em>).</td>
</tr>
<tr>
<td>Others</td>
<td>These smells have nothing in common and therefore, do not fit into any of the above mentioned categories (<em>Incomplete Library Class and Comments</em>).</td>
</tr>
</tbody>
</table>

### 2.3 Test Smell

Code smell phenomenon can also be applied to the test code, since this concept is not specific to only production code [Deursen et al., 2001]. Poorly designed tests are test smells and their existence may affect understandability and maintenance of test suites [Tufano et al., 2016]. Test smells are originated by poor design choices during the test case development [Deursen et al., 2001]. Perhaps, the process of documenting and organizing test cases into test suites, the interaction method of test cases with other test cases, production code, and external resources are all pointer to test smells [Deursen et al., 2001]. With an aim of sharing test code improvement experience with other XP practitioners, Van Deursen et al. have described a group of test smells indicating trouble in the test code [Deursen et al., 2001]. The below table 5, includes bad code smells that are specific to test code identified and explained by Van Deursen et al. in [Deursen et al., 2001]
Table 4. Test smells introduced by Van Deursen

<table>
<thead>
<tr>
<th>Smell Name</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mystery Guest</td>
<td>Consumption of external resources by a test such as, using a file which contains test data.</td>
</tr>
<tr>
<td>Resource Optimism</td>
<td>Positive assumptions made by tests about external resources can cause unpredictable test outcomes, may fail or run well.</td>
</tr>
<tr>
<td>Test Run War</td>
<td>Due to resource interference, the results are fine as long as one programmer runs it and test fails when other programmers run it.</td>
</tr>
<tr>
<td>General Fixture</td>
<td>When the setUp fixture is too general and different tests only access part of that fixture.</td>
</tr>
<tr>
<td>Eager Test</td>
<td>Many procedures of the object to be tested is checked by a test method.</td>
</tr>
<tr>
<td>Lazy Test</td>
<td>When a method is checked by several test methods using the same fixture. Thus, the tests become meaningful when they are considered together.</td>
</tr>
<tr>
<td>Assertion Roulette</td>
<td>A test method contains plenty of assertions without any explanation.</td>
</tr>
<tr>
<td>Indirect Testing</td>
<td>Methods of a test class perform tests on other objects due to references exist to them in the class to be tested.</td>
</tr>
<tr>
<td>For Testers Only</td>
<td>When methods of production class are only used by test methods.</td>
</tr>
<tr>
<td>Sensitive Equality</td>
<td>Upon computation of an actual result, it is mapped to string and compared to a string literal, the result may vary as it depends on many irrelevant details (i.e. commas, quotes, spaces, etc.)</td>
</tr>
<tr>
<td>Test Code Duplication</td>
<td>Unwanted duplication may exist in the test code, such as, duplication of code in the same test class.</td>
</tr>
</tbody>
</table>

Since, production code is adapted and refactored frequently, smells arise in production code more often than in test code [Deursen et al., 2001]. To perform complex refactorings, fresh test code is vital, thus to maintain the freshness, test code is required to be refactored [Deursen et al., 2001]. Test refactorings are alterations of test code, it does not remove or add test cases and it makes the test code better to comprehend and maintainable [Deursen et al., 2001]. Below table 5, contains several refactorings described by Van Deursen et al. in [Deursen et al., 2001].
Table 5. Van Deursen’s list of test smell refactorings

<table>
<thead>
<tr>
<th>Smell Name</th>
<th>Suggested Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mystery Guest</td>
<td>Integrate the needed resource into the test code by setting up a fixture that clutches the contents of the resource or make sure to explicitly generate or initialize the resource prior to testing and release the resource post testing.</td>
</tr>
<tr>
<td>Resource Optimism</td>
<td>Explicitly generate or initialize all the resources that are utilized.</td>
</tr>
<tr>
<td>Test Run War</td>
<td>Unique identifiers for allocated resources are a solution to find tests that do not release their resources correctly.</td>
</tr>
<tr>
<td>General Fixture</td>
<td>Extract only part of the fixture that is shared by all tests and use setUp, rest of the fixture should be kept in the method that uses it.</td>
</tr>
<tr>
<td>Eager Test</td>
<td>Split the test case into methods that test only one method of the class under test. Furthermore, allocating goal of the test cases as names to the methods.</td>
</tr>
<tr>
<td>Lazy Test</td>
<td>Cluster all the individual test cases into one test method.</td>
</tr>
<tr>
<td>Assertion Roulette</td>
<td>Add explanation to the assertion to differentiate among various assertions.</td>
</tr>
<tr>
<td>Indirect Testing</td>
<td>Such test cases can be placed into the appropriate test classes.</td>
</tr>
<tr>
<td>For Testers Only</td>
<td>Move the methods in the test code from the class to a new sub-class and perform the tests on that sub-class.</td>
</tr>
<tr>
<td>Sensitive Equality</td>
<td>Replacing toString equality checks by real equality checks. Generate real equality checks by adding an implementation for the equals methods in the object’s class and perform equality check with the help of this method.</td>
</tr>
<tr>
<td>Test Code Duplication</td>
<td>Duplication should be extracted into a new method.</td>
</tr>
</tbody>
</table>

The bad code smells described by Martin Fowler in his book [Fowler, 2002] are related to production code. About smells unique to automated test scripts was anticipated and suspected long ago [Meszaros, 2007]. To enable incremental development and delivery of software, while decreasing the amount of bugs originate as the code evolves, superior and fine quality automated unit tests are chief development practices [Meszaros, 2010]. Since, all the tests are required to be maintained over the life of the software and maintenance costs can easily override the benefits given by the tests, therefore, writing lots of tests is not enough [Meszaros, 2010]. A smell is a symptom of a problem [Meszaros,
All problems are not considered as smells and few problems can be a source cause of many smells [Meszaros, 2007]. The smell must hold us by our nose and tell "Something is wrong" is the "Sniffability Test" [Meszaros, 2007]. Over the years, this has been discovered that, at least there are two different kind of smells exist [Meszaros, 2007]:

- **Code Smell**: These smells must be identified and recognized by gazing and scanning the code. These are code-level anti-patterns, which are observed and noticed by a developer and tester during reading and writing the code.

- **Behavior Smell**: These smells affect the results of tests as they execute. Due to the presence of these smells, tests tend to fail or never pass compilation.

Code and behavior smells are mainly observed by developers while automation, maintaining and running the tests [Meszaros, 2007], and very recently, one more kind of smell has been discovered:

- **Project Smell**: These smells are mostly observed by the consumers or project managers, as they do not look at the code or project managers rarely write or run the tests [Meszaros, 2007]. Since, functionality, quality, resource and cost are focused by project managers, project smells tend to harbor around these issues [Meszaros, 2007]. These smells act as indicators to the whole health of a project.

Code, behavior and project smells are further described by Gerard Meszaros in his book xUnit Test Patterns: Refactoring Test Code [Meszaros, 2007]. Since, this thesis is more emphasized on static code or program analysis and the belief is that, to gain insights about code smells, detailed by Gerard Meszaros [Meszaros, 2007] becomes important, thus, mentioned below.

- **Obscure Test**: It is a trouble understanding what behavior a test is verifying. Obscure Test is also known as Long Test, Complex Test and Verbose Test [Meszaros, 2007]. Obscure Test makes the test tougher to understand and thus, maintain. It prevents achieving "Test as Documentation", consequently, it leads to "High Test Maintenance Cost" [Meszaros, 2007]. Obscure Test can be caused, if a test includes wrong information, smells like *Eager Test* and *Mystery Guest* can cause this or a tests that use a lot of code to do what it needs to do, Smells like *General Fixture*, *Irrelevant Information*, *Hard-Coded Test Data*, and *Indirect Testing* can cause Obscure Test [Meszaros, 2007].
• **Conditional Test Logic:** A fully automated test is a piece of code that verifies the behavior of the other code, and if it is complicated, how do we verify that it works well?, "What is this test code doing and how do we know that it is doing it correctly", a test contains code that may or may not be executed and it is also known as Indented test code [Meszaros, 2007]. Conditional Test Logic is one factor that turns the tests more complex. Smells like Flexible Test, Conditional Verification Logic, Production Logic Test, Complex Teardown, and Multiple Test Conditions can cause Conditional Test Logic [Meszaros, 2007].

• **Hard-to-Test Code:** As the name suggests, code is hard to test. Few types of code are difficult to test, such as GUI components, multi-threaded code and test code [Meszaros, 2007]. Since, code is not visible to test or very highly coupled code or due to the absence of constructors, it is hard to create object, it may be hard to get a test to compile [Meszaros, 2007]. Automated quality verification of Hard-to-Test Code is very difficult, perhaps, manual quality assessment can be done [Meszaros, 2007]. Smells like Highly Coupled Code, Asynchronous Code and Untestable Test Code can cause Hard-to-Test Code [Meszaros, 2007].

• **Test Code Duplication:** Replication of the test code many times. The necessity of performing similar things by various tests in a suite, often results in Test Code Duplication, for instance, tests may need similar fixture setup or result verification logic [Meszaros, 2007]. Smells like Cut-and-Paste Code Reuse and Reinventing the Wheel can cause Test code Duplication [Meszaros, 2007].

• **Test Logic in Production:** Logic, that should be exercised only during tests, contained by the production code [Meszaros, 2007]. Perhaps, system under test may contain logic, which cannot be utilized in a test environment [Meszaros, 2007]. It leads to a system whose behavior is entirely different in production [Meszaros, 2007]. Smells like Test Hook, For Tests Only, Test Dependency in Production, and Equality Pollution.

### 2.4 Tools for Smell Detection

Bad smell detection has caught a lot of attention by researchers and recent research to develop automatic smell detection tools, is very active, to help software developers to find smells in the code [Fontana et al., 2012]. Martin Fowler and Kent Beck, never tried to provide an accurate formula to refactor the code, since in their experience "no set of metrics rivals informed human intuition" [Fowler, 2002]. As, code smell definitions
are imperfect, informal and subjective, building automatic smell detection tools are difficult and assessing their effectiveness is equally important [Fontana et al., 2012]. Vague and informal descriptions about smells, left vast space for interpretation and several others expound a smell occurrence accurately [Tsantalis and Chatzigeorgiou, 2009b, Moha et al., 2010, Marinescu, 2004, Palomba et al., 2013]. In this field, researchers have defined and developed automatic smell detection tool to detect structural and textual smells in both production code and test code [Mazinanian et al., 2016, Moha et al., 2010, Palomba, 2015b, Bavota et al., 2014].

Ample number of tools are available for automatic code smell detection and following is a list of the tools found during the literature study about smell detection tools:

- Checkstyle ¹
- iPlasma [Marinescu et al., 2005]
- TACO [Palomba, 2015a]
- PMD ²
- StenchBlossom [Murphy-Hill and Black, 2010]
- JDeodorant ³
- JSpIRIT [Vidal et al., 2015]
- DECOR [Moha et al., 2010]
- ConQAT ⁴
- CloneDigger ⁵
- Organic ⁶
- JCosmo [Van Emden and Moonen, 2002]
- CodeVizard [Zazworka and Ackermann, 2010]

---

³JDeodorant: clone refactoring,Proceedings of the 38th International Conference on Software Engineering
⁵Clone Digger: discovers duplicate code in Python and Java [accessed 15. March 2018]
To meet certain significant requirements for building this tool, such as *to what extent available tools can be combined or integrated?*, *Can available tools detect smells in test classes or cases?*, *Can available tools detect test and code smells in production and test code?* this study concentrates over following mentioned tools:

- **TACO (Textual Analysis for Code Smell Detection)** is an independent tool and it utilizes textual analysis techniques to obtain textual information from the code [Palomba, 2015a]. Source code elements contain textual information which is assessed by TACO and calculates textual similarity amid code elements specifying a code component [Palomba, 2015a].

![Figure 1. TACO’s Smell Detection Process [Palomba, 2015a].](image)

In the figure 1, TACO’s key steps to detect a smell is represented. Eager Test and General Fixture are the two types of test smells detected by TACO [Palomba, 2015a].

- **DECOR (Detection and Correction)** analyses code structure of a system to detect smells and anti-patterns. A tool introduced by Naouel Moha *et al.* [Moha et al., 2010] for specification and detection of code and design smells\(^7\).

DECOR recognizes and detects 10 design smells/anti-patterns\(^8\) (for instance, Class Data Should Be Private or Complex Class) and 8 code smells (for instance, Long Method or Long Parameter List). DETEX is a detection technique that instantiates or implements DECOR [Moha et al., 2010].

\(^7\)Design Smell: "Certain structures in the design that indicate breach of fundamental design principles and negatively impact design quality" [Tracz, 2015].

\(^8\)Anti-Pattern: "Literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences." [Brown et al., 1998]
In the figure 2 (a) DECOR method has been compared with related work, each rectangular box represents one step and each arrow connects input or output as well as grey rectangular box represents automates steps. (b) Bold, italics and underlined input and outputs are specific to DETEX.

- An Eclipse plug-in, Organic’s goal is to gather code smells from java projects with the help of command line tools [org, 2018]. Rules published by Bavota et al. has been implemented by Organic to detect 11 types of anti-pattern/design smells (Blob Class or Spaghetti Code) and 7 types of code smells (Long Method or Lazy Class) without user interaction [Bavota et al., 2014, Bavota et al., 2015a]. The plug-in analyzes all the classes present in an existing specified directory or source folder and produces a JSON file as an output containing code smells [org, 2018].

- JDeodorant is an Eclipse plug-in developed by Tsantalis et al. to detect design problems as well as it proposes suitable refactorings to resolve the detected design problems [jde, 2018]. Refactorings are automatically executed upon the user acceptance [Mazinanian et al., 2016]. JDeodorant locates several refactoring opportunities in regards to various design problems below mentioned [Mazinanian et al., 2016, Tsantalis and Chatzigeorgiou, 2011, Tsantalis and Chatzigeorgiou, 2009a, Tsantalis and Chatzigeorgiou, 2009b, Fokaefs et al., 2007, jde, 2018].
  - Feature Envy problem can be rectified by applying suitable Move Method refactoring.
  - Type Checking problem can be rectified by applying suitable Replace Conditional with Polymorphism refactoring.
– State Checking problem can be rectified by applying suitable Replace Type code with State/Strategy refactoring.

– Long Method problem can be rectified by applying suitable Extract Method refactoring.

– God Class or Large Class problem can be rectified by suitable Extract Class refactoring.

– Duplicated Code problem can be resolved by suitable Extract Clone refactoring.

### 2.5 Automatic Software Summarization

Continuous delivery and continuous integration practices are adopted by various software companies across the globe to build high quality products due to the advantage of rapid iterations [Humble and Farley, 2010, Duvall et al., 2007, Savor et al., 2016]. In the similar context, inexperienced [Panichella, 2015] and experienced [Dagenais et al., 2010, Zhou and Mockus, 2012] software developers’ active contribution to a software project [Steinmacher et al., 2015] requires to handle software data, for e.g., build logs and test code and technologies, for e.g., versioning and issue tracking systems [Nazar et al., 2016, Ponzanelli et al., 2015]. For software developers’ massive amount of information is available to observe, analyze and comprehend while executing release cycle tasks, to decrease software developers’ load different strategies are needed [Meyer et al., 2017].

Software maintenance and testing tasks have goals of detecting software bugs and defects as early as possible [Shahin et al., 2017]. Since software maintenance and testing tasks are labor-intensive and costly, many researchers in the field of Software Engineering, invested their time and effort to invent tools with a goal of increasing software developers’ productivity during these activities [Shahin et al., 2017]. Nonetheless, more research is still required in formulating prototypes which can complement or support software developers’ productivity while performing these activities [Shahin et al., 2017].

Software maintenance and evolution tasks are important and to support these tasks automatic software summarization techniques have been introduced and utilized, lately, these techniques produces natural language descriptions or summaries of various kinds of software artifacts [Haiduc et al., 2010, Moreno et al., 2013, Moreno et al., 2015, Murphy, 1996, Panichella et al., 2016, Rastkar et al., 2010, Sorbo et al., 2016]. Automatic software summarization is appearing and growing in software engineering research. It is encour-
aged and influenced by automatic text summarization [Moreno and Marcus, 2017]. A
process of automatic text summarization is "a reductive transformation of source text to
summary text through content reduction by selection and/or generalization on what is
important in the source" described by Spärck Jones [Jones, 1998], while it is defined by
Mani and Maybury as "the process of distilling the most important information from a
source(s) to produce an abridged version for a particular user(s) and task(s)" [Mani and
Maybury, 1999]. When automatically summarizing text, various factors becomes vital
and obligatory, such as purpose of the summary or target audience etc [Moreno and Mar-
cus, 2017].

Defining summarizing techniques to support software engineering activities demands
analysis of diverse and complex software artifacts [Moreno and Marcus, 2017]. Auto-
matic software summarization is a process of creating brief depiction of one or many
software artifacts [Moreno and Marcus, 2017]. This generated brief depiction is inform-
ation, required to execute any particular software engineering task by a software stake-
holder [Moreno and Marcus, 2017]. Software artifacts summarization includes analysis
and processing of source code, occasionally, which makes it different than text summa-
ration [Moreno and Marcus, 2017]. Since, the definition is based on several factor, such
as input, output and aim, existing summarization techniques can be classified accord-
ingly [Moreno and Marcus, 2017]

- Text-to-Text Summarization techniques produce text-based summaries from textual
  software architect [Rastkar et al., 2010, Sorbo et al., 2016].

- Code-to-Text Summarization techniques produce text-based summaries from source
code artifacts [Moreno et al., 2013, Panichella et al., 2016, Sridhara et al., 2010].

- Code-to-Code Summarization techniques produce source-based summaries from
  source code artifacts [Ying and Robillard, 2013, Moreno et al., 2015].

- Mixed Artifact Summarization techniques produce summaries for miscellaneous
  software artifacts [Ponzanelli et al., 2015].

Automatic software summarization approaches are not only affected by factors such as
input and output, as well as affected by purpose factors, such as relation to source [Moreno
et al., 2015, Panichella et al., 2016, Ying and Robillard, 2013] or summary type [Rastkar
et al., 2010, Sorbo et al., 2016, Sridhara et al., 2010] and more [Moreno et al., 2017, Rastkar
et al., 2010, Ying and Robillard, 2013].
Software developers invest more time to read and navigate through code when compared to writing code [LaToza et al., 2006, Ko et al., 2006]. During maintenance task, it becomes almost impossible to go through entire code of any large system for software developers, thus, it becomes necessary for them to understand source code entities such as packages and methods [Haiduc et al., 2010]. Moreover, several times, a class name or a method name fail to describe its purpose, while going through the complete code takes way long [Haiduc et al., 2010]. Very often, software developers only scans the source code during maintenance task and if the source code is well structured, the scanning activity only helps to determine code’s relevance as well as it can lead to misinterpretation or misapprehension [Haiduc et al., 2010]. Commonly, software developers have to go through the entire source code, since method headers are not useful and reading complete source code takes massive amount of time [Haiduc et al., 2010]. Source code summaries can be offered to software developers, which includes significantly brief and most important unambiguous natural language descriptions, to reduce the time and achieve better source code comprehension consequently make precise decision [Haiduc et al., 2010].

Panichella et al. proposed an approach known as TestScribe [Panichella et al., 2016], which is built to automatically generate summaries of the code run by every test case to provide an active view of each class going through the test [Panichella et al., 2016]. By generating natural language description, also called as summaries, help developers to comprehend code going through test better without analysing the complete code [McBurney and McMillan, 2014]. An empirical evaluation containing 30 software developers have revealed that, TestScribe helps to fix bugs and enhance software developers’ bug fixing performance with the help of automatically generated summaries [Panichella et al., 2016].

TestScribe’s working process includes four key phases [Panichella et al., 2016]:

- **Test Suite Generation:** With the help of a tool known as Evosuite [Fraser and Arcuri, 2013], automatic generation of JUnit test cases takes place. To generate test cases, TestScribe requires production code and Evosuite. Test suites can also be provided directly [Panichella et al., 2016].

- **Test Coverage Analysis:** After the generation of test cases, TestScribe utilizes Cobertura ⁹, to find out about statements and branches tested by each test case [Panichella et al., 2016]. TestScrib further to form the key textual corpus, which is required to generate summaries, extracts keywords from identifiers’ names and

---

⁹http://cobertura.github.io/cobertura/
to perform this task, a self-constructed parser based on *JavaParser*[^10] is built. This phase generates fine-grained code elements and lines of code covered by each test case as an output [Panichella et al., 2016].

- **Summary Generation**: Forming a higher-level view of code under test, is the ultimate aim of this phase [Panichella et al., 2016]. To achieve this aim, TestScribe implements *Software Word Usage Model* (SWUM), introduced by Hill *et al.* [Hill *et al.*, 2009]. It helps to extract natural language phrases from the statements. Split the identifiers’ names and enlarge the abbreviations of identifiers and type names. TestScribe uses *Language Tool*[^11] a Part-of-Speech tagger to differentiate and obtain verbs, nouns and adjectives. These outputs are used to determine if the name of a method or an attribute should be treated as noun phrase, verb phrase or prepositional phrase [Panichella et al., 2016]. For generation of summaries, TestScribe utilizes template-based technique. This technique accepts the output of SWUM to fill-up incomplete natural language sentences, present in the pre-defined templates. Summaries are formed at various levels of abstractions, such as, 1. class level summaries, includes general description of the class tested by JUnit test, 2. short summary of structural code coverage, and 3. fine summary of every statement included in test cases [Panichella et al., 2016].

- **Summary Aggregation**: TestScribe’s information aggregator supplements JUnit test class with generate summaries supplied by the summary generator.

The tool ”TestSmellDescriber” is built to automatically generate summaries of a portion of code affected by textual and structural smells along with refactoring suggestions to remove smells, thus, helping software developers’ to improve quality of tests and enhancing their awareness on test suites’ quality.

[^10]: https://github.com/javaparser/javaparser
3 APPROACH AND TOOL

This chapter is about the approach adopted to execute the research work and generate solution for research questions. It describes the process of detecting smell(s) and how the detected smell(s) can be removed.

3.1 Research Approach Overview

In regards to code and text summarization [Lucia et al., 2012, McBurney and McMillan, 2014, Moreno et al., 2013, Panichella et al., 2016, Sridhara et al., 2010], this has been observed and analyzed that, existing tools and approaches produce static summaries or descriptions of the source or test code, without considering particular sections of the code affected by structural or textual smells.

The approach "TestSmellDescriber" is intended to compose test case summaries [Moreno and Marcus, 2017, Panichella et al., 2016] of each section of the code belong to every individual test, which is suffering from textual [Palomba et al., 2016] and structural [Bavota et al., 2015b, Deursen et al., 2001, Tsantalis and Chatzigeorgiou, 2009b] smells. The "TestSmellDescriber" approach includes three phases in order to automatically generate test case summaries represented in figure 3:

1. Smell Detection
2. Summary Generation
3. Description Augmentation

The tool "TestSmellDescriber" go through the test case to check if it is affected by smells or not, with the help of incorporated two smell detection tools, DECOR [Moha et al., 2010] and TACO [Palomba, 2015a].

In the second phase it produces descriptions at test suite and test method level depending on the results of the previous phase. The test suite level summary consists of a general summary about test and code smells affecting the test case, a summary recommending refactoring alternatives for detected smells and a quantitative summary of the smells affecting the actual test class in the context of complete project. The method level summary consists of a brief explanation which helps to localize the smell and a refactoring information which helps for the removal of the smell(s).
In the last phase, tool bundles-up all the produced description and augment the complete description into the test class.

### 3.2 Detection of Smell(s)

This phase discusses about the automatic smell detection process. Test code and production code belongs to any project is supplied as an input to this phase, incorporated tools DECOR and TACO detects all the smells affecting the analyzed project. The supplied input to incorporated tools could be all JAR files or all test class files of a chosen project. After the input is supplied in the required format, DECOR navigates through the all the classes and investigates the model for anti-patterns by utilizing a set of structural rules and metrics [Moha et al., 2010].

As structural level analysis of the selected project is done by DECOR on the contrary, TACO identifies or detects smells in the code by taking the complete advantage of techniques based on textual analysis. TACO identifies smells by assessing textual information accommodated in various elements of the source code and computing textual similarity among such code elements. Very similar to any other information retrieval technique, TACO performs a sequence of pre-processing steps of the content of code elements as a first step [Oliveto et al., 2010]. With the help of Java camel case convention, specifically, it splits the terms in software artifacts or elements, which means splitting the terms based
on capital letters, underscores and digits. Remaining set of terms removal of special char-
acters, common English stop words and keywords related to programming takes place. Using term weighting strategy, the normalized terms are weighted by evaluating their oc-
currences within various software artifacts [Oliveto et al., 2010], and resulting elements modeled as vectors. Hence, the cosine of the angle between the corresponding vectors is measured to measure textual similarity among code elements [Oliveto et al., 2010]. This is noteworthy to state that this tool focuses on the production of summaries related to two types of smells at this stage [Moha et al., 2010]:

• Type 1: LongParameterList a method containing more than 3 or 4 parameters. It may exist due to combining many types of algorithms in a single method and can be fixed using refactoring, such as: ReplaceParameterWithMethodCall or IntroduceParameterObject [Fowler, 2002].

• Type 2: LongMethod a method includes several lines of code. A method with more than 10 lines of code considered as a symptom of bad design choice and the same can be fixed using refactoring such as, extract method or IntroduceParameterObject [Fowler, 2002].

3.3 Description Generation

My Contribution

To generated natural language summaries about the code affected by smells TestSmellDe-
scriber implements an approach influenced by Software Word Usage Model (SWUM) in-
troduced by Hill et al. [Hill et al., 2009]. The bottom line idea of SWUM is that, using any random section of test and production code, it can derive actions, themes and secondary arguments and utilize to link linguistic information to programming language semantics and structure. For example, a method signature typically contains verbs, noun and prepositional phrases and the same can be inflated in order to produce natural language sentences, like SWUM consider verbs existing in method names as actions meanwhile rest of the name contains theme.

"TestSmellDescriber" a new approach, at an abstract level uses detected smells as main actions meanwhile, analysis and extraction of main elements of the source code is to find theme. Incorporated SWUM helps to produce descriptions at various levels of abstraction, as represented in figure 3: brief and elaborated method description, brief and elaborated refactoring suggestion, as well as smell description with respect to total project. Natural
language templates are used to produce descriptions [Haiduc et al., 2010], and augmented in the code all together with the information collected about smells while smell detection process.

*Description of Smell(s)* produced by the approach "TestSmellDescriber" is formulated on the basis of smell’s specification and categorization given by Fowler [Fowler, 2002], Van Deursen [Deursen et al., 2001], Mäntylä [Mäntylä et al., 2003], and Meszaros [Meszaros, 2010]. Elaborated description comments at class level while brief description comments at method level. Smell descriptions serves the purpose being a pointer to design issues for the developers. This can help software developers to locate the smell as well as to assess problems produced by smell. Furthermore, to assist with the location of a cause of smell, brief method descriptions are available.

*Description to Refactor* provides information for removal of smell(s). It is a type of recommendation for software developers regarding, how to apply an appropriate refactor technique to remove a particular smell from the code, and these refactoring recommendations are given by Fowler et al. [Fowler, 2002]. At class level, to improve class’s overall design, refactoring explanation provides a type of summary about potential refactoring techniques which can be applied on the whole test class. At method level, test methods suffering from smell(s), includes refactoring explanation regarding actual refactoring to execute for smell removal. Smell descriptions and refactoring recommendations at class and methods levels are clustered together. Furthermore, at method level, descriptions are given to help software developers with linking, elaborated refactoring recommendations with the test suite level refactoring recommendations.

*Quantitative Descriptions* are generated and provided to software developers on the basis of smell occurrences in a given project. At first place, TestSmellDescriber details about how influential a kind of smell is in a test class, when compared to all kinds of smell detected in the same class, this is achieved using the succeeding equation:

$$D_{\text{smell}} = 100 \times \frac{\text{smellOccurrencesOf-typeA}}{\text{allSmellOccurences}}$$

It provides information about how frequent a type of smell occurrence is, while comparing all other smells present in a given project, using the succeeding equation:

$$F_{\text{smell}} = 100 \times \frac{\text{smellOccurencesInProject}}{\text{allSmellOccurencesInProject}}$$

It also provides information about how frequently a type of smell occurs in a test class, when compared to all the smell occurrences in the whole project:
\[ C_{\text{smell}} = 100 \times \frac{\text{smellOccurrencesOfTypeAInClass}}{\text{smellOccurencesOfTypeAInProject}} \]

Succeeding example is a template, used by "TestSmellDescriber" to represent quantitative description to help software developers.

"This method accounts/These methods account for <D_{\text{smell}} > \% of all found problems in this test class. This smell represents <F_{\text{smell}} > \% of all found problems in the project with <C_{\text{smell}} > \% occurring in this test."

Quantitative descriptions are significantly helpful, since software developers gets an opportunity to evaluate troublesome regions in an analyzed test case and in the complete project.

### 3.4 Description Augmentation

**My Contribution**

In this phase "TestSmellDescriber" deals with supplementing original JUnit test classes with previously generated descriptions. It augments the descriptions with test method and test class.

*Test Suite Level Descriptions* are composed of the following elements:

- Description related to detected smell(s).
- An elaborated description of the smell(s).
- A description explaining how to apply a refactoring technique to remove the smell.
- A quantitative description about smell’s frequency in the test class and in the complete project.

For example, following figure 4, represents a section of smell descriptions produced by "TestSmellDescriber", for a class UtilCacheTest from a project Apache OFBiz. Various description elements mentioned above are highlighted using different colors and consistent with figure 3.
Test Method Level Descriptions are used to funnel down to the root cause of the problem. These summaries are produced for method smells, i.e. a method is a source of a smell. Test Method Level Descriptions are composed of the following elements:

- Brief summary of the smell
- Brief refactoring recommendation (it refers to the elaborated refactoring explanation present in the test suite level).

Above mentioned elements can be noticed in the figure 5, which represents a method level summary of a method `UtilCacheTest::assertKey()` from a project Apache OFBiz.

Numbering for the refactoring recommendation can be observed in the figure 4 and 5, which links the test method level refactoring summary with the test suit level.
4 STUDY AND EXPERIMENTS

This chapter is about measuring the effectiveness of the build tool "Test Smell Descriptor" with regards to automatic smell detection and removal.

4.1 Research Work Objective

Objective of this research work is to scrutinize to what magnitude, descriptions produced by the tool or approach "TestSmellDescriber", Impact software developers’ potential to recognize and fix code or test smells and Improve software developers’ awareness about test suites’ quality. Given the background, exploring the impact with regards to software maintenance and evolution outline, under which a Java class is developed and to test the same class, manual test cases are formulated (purpose is to locate errors or bugs in the code) and software developers are interested to assess, enhance, and mature the standard or quality of such test cases. Software developers’ awareness or perception of test case’s quality is the key focus of quality parameter in this study, when test cases are supplemented with test case summaries, compared to test cases without summaries. Precisely, test case’s comprehensibility and maintainability is the focal point, when the quality is concern.

Study perspective is, estimating how effective automatic test case summarization techniques are, when applied a real time maintenance and evolution tasks with an aim of enhancing the test suites’ quality, is of researchers’ interest, thus this research work answers succeeding RQs:

RQ₁ Are test case summaries enriched by test smell information considered useful by developers? First target of this study is to scrutinize, whether software developers consider test smell summaries useful to understand test case’s quality better as well as to aid evolution and maintenance tasks.

RQ₂ How do test smell summaries impact developers’ ability to improve test suite quality? Second target of this study is, if software developers are interested to modify test cases when summaries are present. Furthermore, if the test cases are supplemented with test smell summaries, validate whether software developers can locate and fix a greater number of test or code smells, this varies and depends over software developers possess different experience.
4.2 Research Work Context

The context of this research work is composed of Objects and Participants.

- Java classes and test cases, selected from a Java open-source project are treated as objects.
- Student, researchers and professional software developers are treated as Participants, who scrutinize the selected objects.

Apache OFBiz \(^{12}\) is the selected project or object system and furthermore, four selected classes from the project are (1) FlexibleStringExpander, class enlarges String values which includes Unified Expression Language syntax, (2) TimeDuration, it implements an inflexible representation of a time period, (3) FlexibleMapAccessor, it is used to access map values, (4) UtilCache, it is made up of generalized cache utility. Following are the related test cases: FlexibleStringExpanderTests, TimeDurationTests, UtilCacheTests and FlexibleMapAccessorTest. Above mentioned Java classes and test cases are non-trivial as well as it is practical to scrutinize all of them within 30 minutes. Furthermore above mentioned classes does not require to scan many other classes in the project. Below drawn table 6 and table 7 elaborate the characteristics of the Java classes and test cases.

<table>
<thead>
<tr>
<th>Table 6. Java classes of Apache OFBiz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name of Java Class</strong></td>
</tr>
<tr>
<td>------------------------</td>
</tr>
<tr>
<td>FlexibleStringExpander</td>
</tr>
<tr>
<td>TimeDuration</td>
</tr>
<tr>
<td>UtilCache</td>
</tr>
<tr>
<td>FlexibleMapAccessor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 7. Test cases of Apache OFBiz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test Case Name</strong></td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>FlexibleStringExpanderTests</td>
</tr>
<tr>
<td>TimeDurationTests</td>
</tr>
<tr>
<td>UtilCacheTests</td>
</tr>
<tr>
<td>FlexibleMapAccessorTest</td>
</tr>
</tbody>
</table>

In both the above mentioned tables, Lines of code (LOC) refers to number of lines of code in the respective Java class and test case. Methods in table 6 refers to number of methods

\(^{12}\)https://ofbiz.apache.org/
present in every Java class. Each test case is influenced by minimum one smell, either LongParameterList or LongMethod, one of the reason to consider them for this research work.

To involve participants to successfully carry out this research work, in total 53 email invitations were sent out, 25 to researchers and 28 to software developers. Below drawn table 8 reports that, among all the invitations sent out, 21 agreed and participated to perform the task, out of which 8 were professional software developers and rest 13 were a blend of 5 master students, 6 PhD students and 2 senior researchers. Most of the participants possess at least 2 to 5 years of professional software testing and Java programming experience.

<table>
<thead>
<tr>
<th>Programming Exp.</th>
<th>Absolute #</th>
<th>Testing Exp.</th>
<th>Absolute #</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 months-2 years</td>
<td>3 (14.30%)</td>
<td>5 months-2 years</td>
<td>6 (28.60%)</td>
</tr>
<tr>
<td>2-5 years</td>
<td>8 (38.10%)</td>
<td>2-5 years</td>
<td>6 (28.60%)</td>
</tr>
<tr>
<td>5-7 years</td>
<td>2 (9.50%)</td>
<td>5-7 years</td>
<td>3 (14.30%)</td>
</tr>
<tr>
<td>7-10 years</td>
<td>6 (28.60%)</td>
<td>7-10 years</td>
<td>4 (19.00%)</td>
</tr>
<tr>
<td>&gt;10 years</td>
<td>2 (9.50%)</td>
<td>&gt;10 years</td>
<td>2 (9.50%)</td>
</tr>
<tr>
<td>Σ</td>
<td>9 (100%)</td>
<td>Σ</td>
<td>9 (100%)</td>
</tr>
</tbody>
</table>

### 4.3 Experimental Procedure

Research work experiment was conducted offline. Emails including mandatory experimental material and instructions about task’s execution were sent out to all the participants. Google Forms \(^{13}\), were used as a medium to guide each participant from beginning to the end of the task as well as to gather information about the performed activities. Every participant was sent an experimental package via email which was composed of (i) a link to google form pre-task survey, purpose of this is to collect information basic information about the particular participant, for instance, profile and experience, (ii) a link to google form task survey with instructions and materials to perform the task and (iii) a link to google form post-task survey, to gather information regarding task and usefulness of the task. Furthermore, prior to the experiment we provided a description about the anticipated task i.e. this experiment involves two maintenance and evolution tasks and every task contains two pairs of Java classes and test cases.

**Tasks Assignment:** Every candidate received two tasks to perform, every task contains two Java classes and two corresponding JUnit test cases and one of them in each task is

\(^{13}\) https://docs.google.com/forms
supplemented with smell summaries. With an aim to assess the usefulness of the tool "TestSmellDescriber" in each task the participants were provided balanced set of smell summaries as follows:

- **Group A:** This group is made up of 10 participants, who received the first test case supplemented with summaries, while the second test case was without the summaries.

- **Group B:** This group is made up of 11 participants, who received the first test case without summaries while the second one was enriched with test case summaries.

**Detailed Task Description:** Prior to initiating the experiment, every participant had to fill in a pre-task questionnaire, which helped to gather information about every candidate’s programming and software testing experience. Upon completion of the pre-task questionnaire, candidate is directed to execute the first task as per the specification and instructions provided via email as a work-space, consist of, the required project data, such as Java classes and test cases. The ultimate stated objective of the research work were (a) *To scrutinize the test cases* (b) *Detect and fix existing smells*, with a key concentration on the detection and removal of two smells specifically, LongParameterList and LongMethod. To supplement the smooth execution of the tasks a natural language text document was added in the experimental package, which explains about test and code smells, several types of smells potentially affecting the test cases and about refactoring operations, which supports smell removal.

It was precisely explained in the given instructions that, as per the provided notion about test and code smells in the text document, JUnit test cases are required to be updated and maintained. Therefore, all the participants were suggested to scan the available test suite and to modify the test cases(detect and remove test and code smells). It was mandatory for every participant to use no more than 30 minutes for each pair of Java class and test case. The anticipated time duration to complete the whole assignment was two hours, covering completion of two tasks and fill-up all the questionnaires. Candidates had a possibility to complete before time, only if they feel and believe that all smells are located and removed. Participants were asked to complete a post-task survey, which was aimed to gather qualitative insights and feedback.
4.4 Research Method

Each task was coupled with a post-task survey, completed by candidates and upon completion of the whole experiment, every candidate had to complete a post-experiment questionnaire, which supported this experiment by generating information about perceived effectiveness, applicability and impact of the provided smell summaries during the execution of tasks. This generated information is utilized effectively to address the RQ$_1$, furthermore, the questionnaire was formulated in such a way that it also demanded to assess the general quality about the supplemented smell summaries (feedback to improve), according to the widely accepted and adopted metrics [Moreno et al., 2013, Panichella et al., 2016, Sridhara et al., 2010]...

- **CONTENT ADEQUACY:** When considering only the content of the produced summaries or descriptions present in the JUnit test cases, important details about the detected smell is explained precisely in the summary.

- **CONCISENESS:** When considering only the content of the produced summaries or descriptions present in the JUnit test cases, if there any presence of irrelevant information in the description.

- **EXPRESSIVENESS:** When considering only the representation of the summaries or descriptions of JUnit test cases, how clearly readable and comprehensive are the description.

Information regarding applied modifications to the test cases was extracted from the post-experiment questionnaire completed by the participants and this information is utilized to count the number of removed detected and removed smell(s) by every candidate. This helps to evaluate to what extent, generated summaries impact software developers’ ability to detect and remove code or test smells.

Statistical tests have been performed to authenticate whether any significant statistical difference is present among the number of detected and removed smells by candidates when depending on test cases with and without smell summaries. With the help of non-parametric Wilcoxon Rank Sum Test with a $p$-value threshold of 0.05. Significant $p$-value specifies that there is a statistical difference present among the number of removed smell achieved by candidates utilizing test cases with or without smell summaries. For more clear and precise interpretation of the achieved results, the effect size of the observed
differences using Vargha-Delaney ($\hat{A}_{12}$) statistic is computed as well [Vargha and Delaney, 2000]. To be specific, the Vargha-Delaney ($\hat{A}_{12}$) statistic categorizes the acquired effect size values into four different levels, namely, *negligible, small, medium and large*. Also, verifying whether some *co-factors*, such as programming experience, impact the number of fixed smells is important too. In order to achieve this, two-way permutation test [Baker, 1995] is used with 1,000,000 number of iterations of the permutation test procedure to guarantee that results should not vary over multiple executions of the test procedure.
5 EVALUATION

5.1 RQ₁: Discerned Usefulness and Quality of Smell Descriptions

Discerned descriptions’ usefulness and comprehensibility. Towards the termination of the conducted research experimental work, all the participants were questioned with a very specific objective to scrutinize discerned comprehensibility and usefulness of provided summaries during the execution of maintenance and evolution tasks. It is significant to state that, 95% of the candidates found that the tasks are reasonably tough to perform but also 71.40% of the candidates had ample amount of time to complete the given tasks. Below drawn table 9 contains post-experiment raw data about the tool TestSmellDescriber.

<table>
<thead>
<tr>
<th>Questions</th>
<th>Disagree</th>
<th>No Strong Opinion</th>
<th>Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1: Do you easily understand and relate the generated descriptions with the code?</td>
<td>0%</td>
<td>9.50%</td>
<td>9.5%</td>
</tr>
<tr>
<td>Q2: Is it difficult to understand the test method level descriptions?</td>
<td>57.10%</td>
<td>33.30%</td>
<td>0%</td>
</tr>
<tr>
<td>Q3: Is it difficult to understand the test suite level descriptions?</td>
<td>61.90%</td>
<td>33.30%</td>
<td>0%</td>
</tr>
<tr>
<td>Q4: Are the generated Test Smell Summaries useful to be more aware of the general test quality?</td>
<td>4.80%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Q5: Modifying the test code as suggested by the descriptions provided by TSD is helpful?</td>
<td>4.80%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Q6: The task without the generated comments or descriptions is prohibitively difficult?</td>
<td>4.80%</td>
<td>0%</td>
<td>9.50%</td>
</tr>
</tbody>
</table>

Table 9 reports that, candidates assessed the comprehensibility of summaries provided by TestSmellDescriber with the help of Likert scale intensity from very low to very high in Q₁ - 3. Outcomes of Q₁ emphasizes that 81% of candidates find provided descriptions easy to read and comprehend. Furthermore, upon asking the identical question in regards to descriptions provided at test suite level and test method level in Q₂ - 3, the discerned readability of descriptions is high or very high for 90% to 95% of all. Observe, Q₄ - 5 in the table 9, approximately 95% of participants accept, available smell summaries as a
relevant source of information to perform the tasks (Q5) and increased awareness about the analyzed test suite quality (Q5). Additionally, executing the tasks without generated and supplemented summaries would be prohibitively difficult, approximately 85% of all the participants believe that.

\[ RQ_1: \text{According to human judgments, produced smell descriptions are (i) easy to understand and are (ii) perceived as a useful source of information to perform maintenance and evolution tasks aimed at improving the test suite quality.} \]

The about mentioned is a general finding and Various participants has passed feedback as a confirmation of this general finding, such as "The combination of class and method descriptions are useful" and "The descriptions at class level provide a good overview of the test suite problems".

**Summaries’ Quality:** Nonetheless, the participants were notified to evaluate the overall quality of the provided smell summaries, as per the three popular and well established dimensions: CONTENT ADEQUACY, CONCISENESS, and EXPRESSIVENESS as detailed in the above subsection "4.4:Research Method" belongs to the section 4. The outcomes are summarized in the below drawn table 10.

<table>
<thead>
<tr>
<th>Table 10. Raw statistics concerning the evaluation of produced summaries.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Content adequacy</strong></td>
</tr>
<tr>
<td>Response category</td>
</tr>
<tr>
<td>Is not missing any information.</td>
</tr>
<tr>
<td>Missing some information.</td>
</tr>
<tr>
<td>Missing some very important information.</td>
</tr>
<tr>
<td><strong>Conciseness</strong></td>
</tr>
<tr>
<td>Response category</td>
</tr>
<tr>
<td>Has no unnecessary information.</td>
</tr>
<tr>
<td>Has some unnecessary information.</td>
</tr>
<tr>
<td>Has a lot of unnecessary information.</td>
</tr>
<tr>
<td><strong>Expressiveness</strong></td>
</tr>
<tr>
<td>Response category</td>
</tr>
<tr>
<td>Is easy to read and understand.</td>
</tr>
<tr>
<td>Is somewhat readable and understandable.</td>
</tr>
<tr>
<td>Is hard to read and understand.</td>
</tr>
</tbody>
</table>
Table 10 highlights the maximum number of candidates consider that produced summaries are concise and easy to comprehend. Precisely, (a) 90.50% of all the candidates believe that generated descriptions do not miss out on very important information, i.e. they are adequate; (b) approximately, 71.40% of all participants consider the supplemented descriptions are concise while rest 28.60% believe that, descriptions contain some unnecessary information; and (c) all participants consider that the produced summaries are easy to read and understand or somewhat readable and understandable. Although, overall feedback was positive, several suggestions were also made by participants for improvement.

- **Relevance of the Test Suite and Test Method Level Summaries**
  Participants trust that it is "useful to have the comment in the actual place where the smells are detected" and "the descriptions at both levels serve important purposes". Furthermore, candidates all feel that "it was a bit of a nuisance having to scroll back to the top to go through the suggestion".

- **Irrelevant and Redundant Information**
  Software developers involved in this research work were concerned by the fact that "the descriptions are bit redundant in general" and also in few other cases "description of the method arguments is unnecessary".

- **Information to Incorporate into Descriptions**
  Few participants suggested as a feedback to "leverage the extracted static information and descriptions for guiding to fix with potential patches" and to supply "Suggestions on how to split the code to reduce the size of the method, i.e. if some parameter values are redundant and may be deduced from other parameter values".

5.2 **RQ₂: Effect on Maintenance Tasks**

From RQ₁, observation can be made that participants discerned the produced smell descriptions as simple or easy to comprehend as well as a useful source for performing maintenance and evolution tasks on test cases. With the help of RQ₂, it can be verified whether participants’ ability is increased to identify and fix more number of test or code smells, when relying over test cases supplemented with smell summaries. Furthermore, how this increased ability vary depending on participants’ software programming and testing experience?
Impact on participants’ smell detection and removal potential. Below figure 6 displays the number of smells detected and removed by participants WITH and WITHOUT the help of produced descriptions.

By scanning figure 6, it can be discovered that the average number of fixed smells is higher for three test cases out of four, when the descriptions were provided. The only test case "UtilCacheTests" exists with no average evident improvement. Particularly, with test cases "FlexibleStringExpanderTests" and "FlexibleMapAccessorTest" a high improvement with regards to smell detection and removal of smells can be observed, i.e. double the number of smells were fixed when descriptions were provided. Furthermore, with the test case "TimeDurationTests", participants’ smell detection and removal potential is increased by 33%. Therefore, upon considering outcomes of all test cases, participants WITHOUT the help of TestSmellDescriber’s descriptions, were able to locate and fix on average only 0.83 smells out of 2(maximum available in each test case). Whereas, when the TestSmellDescriber’s descriptions were present, the median number of detected and fixed smells are 1.64, by the same participants. This represents a significant improvement (twice as many as number of smells were fixed by participants), while considering the amount of time allocated was 60 minutes per task, for both the scenarios (WITH or WITHOUT descriptions).

Statistically, this finding of the research work is moderately supported by the outcomes of Wilcoxon test, which emphasizes that the use of descriptions, produced by TestSmellDe-
scriber, noteworthy enhanced the smell detection and removal performance of the involved participants, by accomplishing significant \( p \)-values (lesser than the significance level of 0.05) for "FlexibleStringExpanderTests" and "FlexibleMapAccessorTest" test cases, i.e., of 0.0004 and 0.0003 respectively. On the other hand, "UtilCacheTests" and "TimeDurationTests" test cases do not display any statistical significant \( p \)-values (0.72 and 0.11 respectively). The Vargha-Delaney \( \hat{A}_{12} \) statistic also confirms that immensity of improvement is bigger for both "FlexibleStringExpanderTests" with effect size 0.94 and "FlexibleMapAccessorTest" with effect size equals to 0.94.

**Scrutiny of potential co-factors influencing the accomplished results.** With the help of two-way permutation test, it has been verified, whether the number of removed smells among the two studied scenarios (WITH and WITHOUT descriptions) are influenced by participants’ programming or testing experience, which also represents two relevant co-factors in this research work. The two-way permutation tests spotlight the number of detected and removed smells is not importantly influenced by the participants’ programming experience for FlexibleStringExpanderTests, and TimeDurationTests test cases, \( (p\text{-values} \in \{0.34, 0.38\}) \), while for UtilCacheTests and FlexibleMapAccessorTest test cases, participants’ programming experience seems to play a vital role for detection of smell outcomes \( (p\text{-values} \in \{0.04, 0.05\}) \). For all tasks, candidates’ software testing experience do not importantly influenced \( (p\text{-values} \in \{0.20, 0.32, 0.66, 0.87\}) \) the number of removed smells.

It has been revealed by two-way permutation test that a related co-factor impacting the number of detected smells is the actual target pair of Java classes and test cases \( (p\text{-values}= 0) \). If the observation is not made about statistical significant outcome for all Java classes and test cases, this result is not astonishing at all. This finding was also mirrored in the outcomes of two-way permutation test, among the number of fixed smells and programming experience, where significant \( p \)-values of UtilCacheTests and FlexibleMapAccessorTest were observed. It means that an individual’s programming experience has an effect on the number of detected smells only for a specific Java class and test case, hence, the essential factor impacting the outcomes is represented by the target pair of Java classes and test cases. This finding is specifically important since it spotlight the use of descriptions produced by TestSmellDescriber is beneficial for almost each participant and this outcome is independent from their software programming and testing experience. Distinctly, a replication research work including more number of participants would be required to further investigate this aspect on other several projects, hence verifying what characteristics of test and production code (e.g. its overall complexity) impacted the accomplished outcomes.
It can be concluded that:

**RQ2**: TestSmellDescriber’s descriptions significantly help software developers to locate and remove more smells (twice as many at the same time). This result is not impacted by the software developers’ programming and testing experience, but it can be affected by any specific characteristics (e.g., the complexity) of the target Java classes/test cases.
6 CONCLUSION AND FUTURE WORK

Contemporary research work demonstrated that the quality of automatically and manually generated test cases are affected by the presence of smells [Bavota et al., 2015b, Van Deursen et al., 2001, Garousi and Küçük, 2018]. It is also outlined by contemporary empirical studies that, smells substantially complicates the maintainability of test code [Palomba et al., ] and the effectiveness of the software quality assurance tasks [Bavota et al., 2015b, Panichella et al., 2016].

This research work contributes towards handling the obstacle of maintainability of manually and automatically generated test cases by supplying the following:

- An approach is developed, called as "TESTSMELLDSCRIBER" to help software developers to improve accessible test cases, by spotlighting potential regions and sections of the code, where errors and bugs are present or poor design decisions are applied. This tool "TestSmellDescriber" is designed and build to automatically produce test case descriptions (i.e. natural language descriptions) of every portion of the code affected by textual and structural smells, for every individual test, by that enhancing software developers’ awareness about test suite’s quality. Furthermore, for smell removal, refactoring techniques are suggested and recommended to software developers as additional comments in the test cases, this enables them to improve the code.

- To assess the developed tool "TestSmellDescriber" and its efficiency, an empirical study has been conducted with a blend of 21 participants belongs to industry and academia. An investigation had also taken place, to check whether smell summaries are considered useful by participants to better and clearly understand test case quality and to smooth up the evolution and maintenance tasks. Lastly, verification had done, whether software developers are able to locate and fix more structural and textual smells, when test cases are supplemented with smell summaries or descriptions as well as how does this vary depending on participants programming and testing experience.

Outcomes of the research work specify that, according to the involved participants, (RQ1) produced test smell summaries are discerned as a useful source of information to execute maintenance and evolution tasks with an objective of enhancing the test suites’ quality. Furthermore, outcomes also spotlight that (RQ2) The tool TestSmellDescriber significantly helps software developers to locate and remove more number of smells (double
the fixed number of smells) lowering maintenance effort. This outcome is majorly affected or impacted by specific characteristics (for example: complexity) of targeted Java classes or test cases (and not particularly affected by participant’s programming or testing experience).

As future work, there is a plan to further enhance the descriptions produced by the tool "TestSmellDescriber" via (a) Leveraging historical data and information to supply to software developers information about the impact of smells in the history of the project; (b) Complementing the produced descriptions with static information to facilitate and guide about removal of smells with possible patches, for e.g. Suggestion about splitting the code to decrease the size of a test method; (c) rework on the research work with a wider set of test cases and Java classes from other projects, scrutinizing the advantages of utilizing this approach on automatically generated test cases.
REFERENCES


RTI (2002). The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, National Institute of Standards and Technology, Gaithersburg, MD.


Appendix 1. Templates Used to Generate Smell Descriptions

Figure A1.1
Appendix 2. Emails Sent to Participants

Dear <Participant Name>,

First of all, we would like to thank you for being a part of this research and helping us to achieve our research goals.

Who we are:
Since you agreed to participate in this survey you probably know one or more of us already. We are Dr. Sebastiano Panichella, Senior Research Associate at the S.E.A.L. group at the University of Zurich (UZH), Ivan Taraca, Master Student at the UZH, and Gulshan Kundra, Master Student at the Lappeenranta University of Technology (LUT), and currently working on his master thesis at the UZH.

Our Research:
Our group is performing a study aimed at investigating the usefulness of a tool called TestSmellsDescriptor (TSD), which aims at supporting developers to understand the characteristics of test suites TSD is used to automatically detect “test and code smells” by leveraging information retrieval techniques and static analysis tools. Information about the detected smells are used by TSD to generate detailed as well as general descriptions about the presence of the smells in a software project’s test suite, along with recommendations to improve test quality, e.g., by refactoring operations.

Timeline:
The deadline to complete the whole survey is approximately one-week (maximum 8 days), since the time you receive this email. We greatly appreciate early submissions :)

Survey:
Our goal of this survey is to observe how TSD impacts the overall “Developer’s Awareness about Test Suite Quality” and how this helps them to perform appropriate changes on test or source code. For this we constructed a survey, which we split up into four parts for your convenience. At the start of each survey we ask you for your e-mail address. This is solely to assist us in connecting the four split up parts into one.

1. Pre-task survey:
This short preliminary survey helps us to gather background and context information about you, e.g. about your experience with Java. You can find the survey under the following link:
Link: https://docs.google.com/forms/d/e/1FAIpQLSfo7yu0mmpOowaWf-hq0lNHr9v09ZRFt5SnMANN4-Rq2T-A/viewform?usp=sf_link

2. Test smell tasks:
The second part of our survey consists of two refactoring tasks coupled with a survey. For each task we will provide you with the source code of a selected Apache project, from which we additionally selected two test classes (1 project, 2 tasks, 2 classes to analyze per each project).
Goal: These classes harbor test smells. The goal of your task is to identify any smells that you are able to find within the test classes and to improve the code by removing the smells to the best of your knowledge.
Timeline: We ask you to spend a maximum amount of thirty minutes on each test class. Please note down the time it took you to complete each task. The task for each project should be completed in a single run, but you can safely perform both tasks at different times (e.g., on two different days).
Resources: To aid you in your search we share in this email a PDF containing information regarding test smells and their refactoring. This helps you to refresh up your knowledge on smells to prepare you for the tasks. To start the task, download and unzip the source code of the projects from the following provided Dropbox link:
Link: https://www.dropbox.com/s/sdlufobztktcsf8u/Workspacen1.zin?d=0
Before starting the task and the time you can make yourself familiar with the source code.

Figure A2.1
Appendix 3. Emails Sent to Participants Continue....

a) First Task:
To start the first task, open up the survey under the following link, where you can find the selected test classes. After completing the task directly fill out the survey. Reminder: Don't forget to note down the time it took you to complete the code part.

Link: https://docs.google.com/forms/d/e/1FAIpQLSdfn2Y4EClv90UTZtm51-ucG2ZRT_220fU3Km11R4hRl5vTQ/viewform?usp=sf_link

b) Second Task:
The second task can be started by opening the following link. Again, please fill out the survey after completing the code part of this task.

Link: https://docs.google.com/forms/d/e/1FAIpQLScn4aEVQSOAxX_EatbeRljiGc7OMWASaQX_E-PWlPM1S0Y7wbo/viewform?usp=sf_link

3. Post-task survey:
One more survey to go! You almost made it. Thank you again for participating. Once you have completed above tasks you can fill out the survey which can be found under the following link:

Link: https://docs.google.com/forms/d/e/1FAIpQLSdfEWk0mcnKJ0K3J4Ec7evNMvKl3_bkkvUc5f5eCQeAMpzw/viewform?usp=sf_link

If you have any confusions/questions related to the tasks or surveys, please don't hesitate to contact any of us under the following email addresses.
- panichella@ifi.uzh.ch
- ivan.taraca@uzh.ch
- guishan.kundra@student.lut.fi

Thank you again for participating and by aiding us in our research!

Best Regards,
Sebastiano Panichella
Ivan Taraca
Guishan Kundra

Figure A3.1
Appendix 4. Information Regarding Smells for Participants

Smell Information and Refactoring

What is code smell?

Code smells as "indicators that there is trouble" in the code, which can be solved by refactoring. Code smells indicate areas, which potentially house the root of bug or error-causing code, as they are "symptoms of poor design and implementation choices". Code smells have many implications and consequences. The smell Duplicated Code denotes a code portion that can be found in more than one place. This duplication makes the code harder to maintain and debug. Changes to the code in one part draw changes in the duplicated version with it. This impediment can easily introduce bugs into the code leading to errors.

What is refactoring?

The method to remove smells and to thereby improve the code is performed by refactoring. So, can duplicated code in two methods be refactored by extracting the duplicated code into a method, which will be invoked from both previous places. Refactoring will ease the future development and present maintainability, as a combination of smells significantly reduces the comprehension of the code.

Code Smells and Refactoring by Martin Fowler

There are various code smells and refactoring described by Fowler in his book. If you would like to go through all the smells he has described, follow the below mentioned link.

https://www.csie.ntu.edu.tw/~ntucis/Refactoring_improving_the_design_of_existing_code.pdf

List of code smells that we have considered in our study described by Fowler, is mentioned below:

<table>
<thead>
<tr>
<th>ID</th>
<th>Smell Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>Long Method</td>
<td>A Long Method occurs if a function does more than one thing.</td>
</tr>
<tr>
<td>F3</td>
<td>Large Class</td>
<td>The class is doing too much. A Large Class often leads to duplicated code and chaos.</td>
</tr>
<tr>
<td>F4</td>
<td>Long Parameter List</td>
<td>A Long Parameter List smell is a method that requires too many parameters. A long list of parameters is hard to understand, may becomes inconsistent, difficult to use and is susceptible to constant changes as you need more data.</td>
</tr>
</tbody>
</table>

Fowler has further defined the ways to eliminate the smells. Fowler first defined several refactoring methods. He then described ways to remove smells by applying those refactoring on the smelly code. There are many instances and scenarios where smells can take place.

Figure A4.1
Appendix 5. Information Regarding Smells for Participants Continue....

List of code smells refactoring that can be applied to remove the smells affecting the test cases in our study, as described by Fowler:

<table>
<thead>
<tr>
<th>ID</th>
<th>Smell Name</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>Long Method</td>
<td>Find parts of the method that go together and extract them into a new method.</td>
</tr>
<tr>
<td>F3</td>
<td>Large Class</td>
<td>Bundle instance variables together and extract those into new classes.</td>
</tr>
<tr>
<td>F4</td>
<td>Long Parameter List</td>
<td>Replace the parameter with a new class that holds all requested data, or if they already belong to one object, pass the whole object.</td>
</tr>
</tbody>
</table>

Code Smells and Refactoring by Van Deursen

There are various code smells and refactoring described by Van Deursen in this paper. The code smells indicate trouble in test code, i.e., test smells. Code smells are not specific to production code and can also be applied to test code, but van Deursen et al. have acknowledged that refactoring test code requires additional test-specific refactoring.

If you would like to go through all the smells he has described, follow the below mentioned link:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.29.486&rep=rep1&type=pdf

List of code smells that we have considered in our study described by Van Deursen, is mentioned below:

<table>
<thead>
<tr>
<th>ID</th>
<th>Smell Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D4</td>
<td>General Fixture</td>
<td>The fixture is too general. Individual test cases only access and require part of the provided fixture.</td>
</tr>
<tr>
<td>D5</td>
<td>Eager Test</td>
<td>The test checks too much functionality/methods of the object under test in a single test case.</td>
</tr>
<tr>
<td>D6</td>
<td>Lazy Test</td>
<td>Several test methods check the same method using the same fixture. The tests only have meaning when they are considered together.</td>
</tr>
</tbody>
</table>

List of code smells refactoring that can be applied to remove the smells affecting the test cases in our study, as described by Van Deursen:

<table>
<thead>
<tr>
<th>ID</th>
<th>Smell Name</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>D4</td>
<td>General Fixture</td>
<td>Extract the parts of the fixture that are not required by all methods into the methods that require it.</td>
</tr>
<tr>
<td>D5</td>
<td>Eager Test</td>
<td>Separate the test case into methods that test only one method of the class under test. Additionally, assign meaningful names to the methods describing the goal of the test cases.</td>
</tr>
<tr>
<td>D6</td>
<td>Lazy Test</td>
<td>Combine the individual test cases into one test method.</td>
</tr>
</tbody>
</table>

Figure A5.1