International Journal of Applied Engineering Research ISSN 0973-4562 Volume 14, Number 7 (2019) pp. 1599-1603

Search-Based Software Engineering Approach for Detecting Code-Smells with Development of Unified Model for Test Prioritization Strategies

N.Kamaraj[1], A.V.Ramani[2],

[1] *Assistant Professor and Head of Department of Information Technology, Sri Ramakrishna Mission Vidyalaya College of Arts and Science, Coimbatore-641 020, India.*

[2] *Associate Processor & Head of Department of Computer Science Department, Sri Ramakrishna Mission Vidyalaya College of Arts and Science, Coimbatore-641 020, India.*

Abstract
Source code of large systems is iteratively refined, restructured and evolved due to many reasons such as correcting errors in design, modifying a design to accommodate changes in requirements, and modifying a design to enhance existing features. This huge cost may potentially be greatly lowered by providing automatic or semi-automatic solutions to improve their understandability, adaptability and extensibility to eliminate bad-practices. Many studies related that these software maintenance activities utilise up to 90 percent of the total cost of a normal software project. This study considers code-smells detection a main software development problem. The idea is that different methods are combined during the detection process to find a consensus regarding the detection of code-smells. Different adaptations (zero parameter, one, two and more parameter functions) are found out, to solve a common goal which is the detection of code-smells. In addition, this work provides the first single model that is generic enough to study GUI and Web applications together to generate test cases. Use this model to define general error validating criteria that are applicable to both GUI and Web applications. The ultimate goal is to evolve the model and use it to develop a unified theory of how all Event Driver Software should be tested.

Keywords: Search-Based Software Engineering, Code-Smells, Software Quality, Test-Suite Prioritization.

1. INTRODUCTION

The current definition of software engineering is still being debated by practitioners today as they struggle to come up with ways to produce software that is "cheaper, better, faster". Cost reduction has been a primary focus of the IT industry since the 1990s. Total cost of ownership represents the costs of more than just acquisition. It includes things like productivity impediments, upkeep efforts, and resources needed to support infrastructure.

The Software Engineering Institute offers certifications on specific topic like Security, Process improvement and Software architecture.

Apple, IBM, Microsoft and other companies also sponsor their own certification examinations. Many IT certification programmes are oriented toward specific technologies, and managed by the vendors of these technologies. These certification programs are tailored to the institutions that would employ people who use these technologies.

In many existing systems, rules are manually defined to identify the key symptoms that characterize a code-smell using combinations of mainly quantitative (metrics), structural, and lexical information [1]. However, in an exhaustive scenario, the number of possible code-smells to manually characterize with rules can be large [2]. For each code-smell, rules that are expressed in terms of metric combinations need substantial calibration efforts to find the right threshold value for each metric [3]. Another important issue is that translating symptoms into rules is not obvious because there is no consensual symptom-based definition of code-smells [4]. When consensus exists, the same symptom could be associated to many code-smells types, which may compromise the precise identification of code-smell types.

To address these issues, this study proposed an approach to generate detection rules from examples of code-smells using structural metrics [5]. However, the quality of the generated rules depends on the coverage of the different suspicious behaviours of code-smells, and it is difficult to ensure such coverage. Thus, there are still some uncertainties regarding the detected code-smells due to the difficulty to evaluate the coverage of the base of code-smell examples.

In another recent work, an approach is proposed based on an artificial immune system metaphor to detect code-smells by deviation with examples of good code-practices and well-designed systems [6].

Some of the detected code fragments that are different from well-designed code were not code-smells but just new good-practice behavior. Thus, it is believed that an efficient approach will be used to combine both detection algorithms to find better consensus when detecting code-smells.

This paper also focuses on the first challenge i.e., to develop a single model for GUI and Web application testing. To provide focus is controlled the model on test prioritization techniques.

This allows the user to tailor the model to prioritization-specific issues as well as to recast the prioritization criteria in a form that is general enough to leverage the single model.

In future, the model will be extended to other testing problems that are shared commonly by GUI and Web applications. The ultimate goal is to simplify the model and to build up a theory of how EDS should be tested.

2. LITERATURE REVIEW

In this section, previous works regarding code smells are reviewed. Caroline et al. in [7] investigated a new heterogeneous method that dynamically sets the migration period of a distributed Genetic Algorithm (dGA). Each island GA of this multi population technique self-adapts the period for exchanging information with the other islands regarding the local evolution process.

Thus, the different islands can develop different migration settings behaving like a heterogeneous dGA. The proposed algorithm is tested on a large set of instances of the Max-Cut problem, and it can be easily applied to other optimisation problems. The results of this heterogeneous dGA are competitive with the best existing algorithms, with the added advantage of avoiding time consuming preliminary tests for tuning the algorithm.

They have presented a new heterogeneous dGA to solve the Max-Cut problem, which uses different migration periods in each subpopulation following an adaptive, automatic, and problem- matching strategy. Migration periods are dynamically set based on the changes observed in the average fitness population, resulting in an inexpensive self-adaptation procedure.

Naouel Moha et.al stated that code and design smells are poor solutions to recurring implementation and design problems. They may hinder the evolution of a system by making it hard for software engineers to carry out changes [8].

They proposed three contributions to the research field related to code and design smells: (1) DECOR, a method that embodies and defines all the steps necessary for the specification and detection of code and design smells (2) DETEX, a detection technique that instantiates this method and (3) an empirical validation in terms of precision and recall of DETEX. The originality of DETEX stems from the ability for software engineers to specify smells at a high-level of abstraction using a consistent vocabulary and domain-specific language for automatically generating detection algorithms.

Using DETEX, they specified four well-known design smells. The antipatterns Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, and their 15 underlying code smells, and are generated automatically their detection algorithms. The researcher applies and validates the detection algorithms in terms of precision and recalls on XERCES v2.7.0, and discusses the precision of these algorithms on 11 open source systems.

The detection of smells is important to improve the quality of software systems, to facilitate their evolution, and thus to reduce the overall cost of their development and maintenance.

They proposed the following improvements to previous work. First, they introduced DECOR, a method that embodies the entire step necessary to define detection techniques. Second, they casted their detection technique, that is called DETEX, in the context of the DECOR method.

DETEX now plays the role of reference instantiation of our method. It is supported by a DSL for specifying smells using high-level abstractions, taking into account the context of the analyzed systems, and resulting from a thorough domain analysis of the text-based descriptions of the smells. Third, they applied DETEX on four design smells and their 15 underlying code smells and discussed its usefulness, precision and recall.

This is the first such extensive validation of a smell detection technique. Their detection technique and the inputs, outputs, processes, and implementations defined in each step can be generalized to other smells.

2.1 Cooperative Parallel Model Scheme

Nowadays, real life optimisation problems are more and more complex. Consequently, their resource requirements are ever increasing. Optimisation problems are often hard and expensive from a CPU time and/or memory viewpoint [9]. The use of meta-heuristics, such as Evolutionary Algorithms and Particle Swarm Optimization (PSO), allows reducing the computational complexity of the search process.

However, the latter remains computationally costly in different application domains where the objective functions and the constraints are resource intensive and the size of the search space is huge. In addition, to the problem of complexity, find today resource-expensive search methods such as hybrid meta-heuristics and multi-objective ones.

The rapid technology development in terms of processors, networks and data storage tools renders parallel distributed computing very interesting to use. This fact has motivated researchers to focus more on designing and implementing parallel meta-heuristics, in order to solve more complex optimization problems [10].

2.2 Challenges and Open Problems

In the following, some code-smells' detection issues and challenges will be introduced. Later, in Section 3, these issues will be discussed in more detail with respect to the approach. Overall, there is no general consensus on how to decide, if a particular design violates a quality heuristic.

In fact, there is a difference between detecting symptoms and asserting that the detected situation is an actual code-smell. For example, an object-oriented programme with a hundred classes from which one class implements all the behavior and all the

other classes are only classes with attributes and accessors. No doubt, people are in presence of a Blob.

Unfortunately, in real-life systems, one can find many large classes, each one using some data classes and some regular classes. Deciding which classes are Blob, candidates heavily depend on the interpretation of each analyst. In some contexts, an apparent violation of a design principle may be consensually accepted as normal practice.

For example, a "Log" class is responsible for maintaining a log of events in a program, used by a large number of classes that is a common and acceptable practice. However, from a strict code-smell definition, it can be considered as a class with an abnormally large coupling.

3. TEST PRIORITIZATION STRATEGIES

3.1 Code-Smell

Code-smell also called design anomalies or design defects, refer to design situations that adversely affect the software maintenance [11]. As stated by [12], bad-smells are unlikely to cause failures directly, but may do it indirectly. In general, they make a system difficult to change, which may in turn introduce bugs. Different types of code-smells, presenting a variety of symptoms, have been studied in the intent of facilitating their detection and suggesting improvement solutions.

In [11], Fowler et al. define 22 sets of symptoms of code smells. These include large classes, feature envy, long parameter lists (LPLs), and lazy classes (LCs). Each code-smell type is accompanied by refactoring suggestions to remove it. Brown et al. [13] define another category of code-smells that are documented in the literature, and named anti-patterns. In the approach, the authors focused on the eight following code-smell types:

–Blob. It is found in designs where one large class monopolies the behaviour of a system (or part of it), and the other classes primarily encapsulate data. – Spaghetti code (SC). It is a code with a complex and tangled control structure. –Functional decomposition (FD). It occurs when a class is designed with the intent of performing a single function. This is found in code produced by non-experienced object-oriented developers. – Feature envy (FE).

It occurs when a method is more interested in the features of other classes than its own. In general, it is a method that invokes several times accessor methods of another class. – Data class (DC). It is a class with all data and no behavior. It is a class that passively stores data. – Lazy class. A class that isn't doing enough to pay for itself. – Long parameter list. Methods with numerous parameters are a challenge to maintain, especially if most of them share the same data-type. – Shotgun surgery (SS).

It occurs when a method has a large number of external operations calling it, and these operations are spread over a significant number of different classes. As a result, the impact of a change in this method will be large and widespread.

The researcher has chosen these code-smell types in our experiments because they are the most frequent and hard to detect and fix based on a recent empirical study [2].

3.2 Code-Smells Detection:

The code-smells' detection process consists in finding code fragments that violate structure or semantic properties such as the ones related to coupling and complexity. In this setting, internal attributes used to define these properties, are captured through software metrics and properties are expressed in terms of valid values for these metrics [14].

This follows a long tradition of using software metrics to evaluate the quality of the design including the detection of code-smells [1].

The most widely used metrics are the ones defined by Chidamber and Kemerer [14]. These metrics include: (1) Depth of Inheritance Tree (DIT), (2) Weighted Methods per Class (WMC), and (3) Coupling Between Objects (CBO).

In this paper, variations of these metrics and adaptations of procedural ones as well, e.g. the number of lines of code in a class (LOCCLASS), number of lines of code in a method (LOCMETHOD), number of attributes in a class (NAD), number of methods (NMD), lack of cohesion in methods (LCOM5), number of accessors (NACC), and number of private fields (NPRIVFIELD) have been used.

3.3 Test Prioritization of GUI and Web Applications

Due to their user-centric nature, GUI and Web systems routinely undergo changes as part of their maintenance process. New versions of the applications are often created as a result of bug fixes or requirements modification [15]. In such situations, a large number of test cases may be available from testing previous versions of the application which are often reused to test the new version of the application.

However, running such tests may take a significant amount of time. Rothermel et al. report an example for which it takes weeks to execute all of the test cases from a previous version [18]. Due to time constraints, a tester must often select and execute a subset of these test cases. Test case prioritization is the process of scheduling the execution of test cases according to some criterion to satisfy a performance goal.

3.4 Modeling Test Cases

A test case is modeled as a sequence of actions. For each action, a user sets a value for one or more parameters. Examples of test cases for both GUI and Web applications have been provided. Table 1a provides a sample test case for a GUI application called TerpWord.

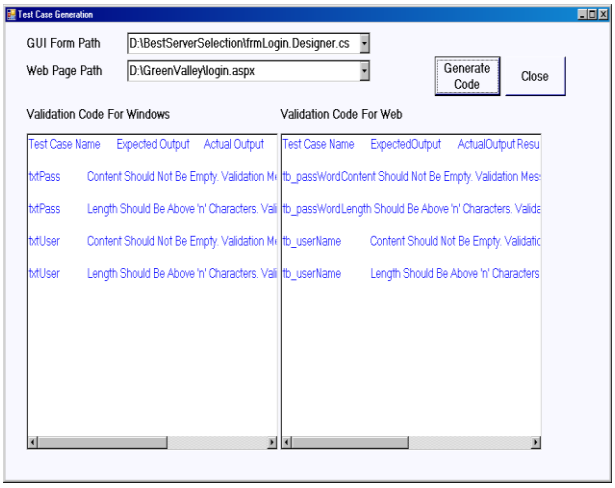

Table -1
Example GUI Test Case

Start of Test Case Number of Actions	<Test Case> <Length>4</Length>
Action 1	<Menu> <Window>TerpWord</Window> <Nonterminal>File</Nonterminal> </Menu> <Menu> <Window>TerpWord</Window> <Nonterminal>Save</Nonterminal> </Menu>
Action 2	<Component> <Window>Save</Window> <Nonterminal>File Name text field</Nonterminal> <Eventtype>SETTEXT</Eventtype> <Eventvalue>examplefile</Eventvalue> </Component> <Component> <Window>Save</Window> <Nonterminal>Files of Type drop – down box</Nonterminal> <Eventtype>LEFTCLICK SELECT</Eventtype> <Eventvalue>Plain Text File (*.txt)</Eventvalue> </Component> <Component> <Window>Save</Window> <Nonterminal>OKbutton</Nonterminal> <Eventtype>LEFTCLICK</Eventtype> </Component>
Action 3	<Menu> <Window>TerpWord</Window> <Nonterminal>Edit</Nonterminal> </Menu> <Menu> <Window>TerpWord</Window> <Nonterminal>Find….</Nonterminal> </Menu>
Action 4	<Component> <Window>Find</Window> <Nonterminal>Find what drop – box</Nonterminal> <Eventtype>SETTEXT</Eventtype> <Eventvalue>software defect</Eventvalue> </Component> <Component> <Window>Find</Window> <Nonterminal>Find Next button</Nonterminal> <Eventtype>LEFTCLICK</Eventtype> </Component>
End of Test Case	</Testcase>

(a) Sample GUI Test Case

The test case sets nine parameters to values and visits three unique windows. The test includes visits to the TerpWord main window, Save, and Find windows. An action occurs when a user sets values to one or more parameters on a window before visiting a different window.

Window Name	P-V No.	P-V description (<parameter, value>)
TerpWord	PV.1 PV.2	<File, Null> <Save, Null>
Save	PV.3 PV.4 PV.5	<File name text field, SELECT = "example File"> <Files of Type drop-down box, LEFTCLICK SELECT="Plain Text File (*.txt)"> <OK button, LEFTCLICK>
TerpWord	PV.6 PV.7	<Edit, Null> <Find…., Null>
Find	PV.8 PV.9	<Find what drop box, SETTEXT="software defect"> <FindNext button, LEFTCLICK>

(b) Windows and User interactions in test case.

From Table 1a, it is revealed that in Action 1, the user selects File->Save from the TerpWord main menu. The parameter values associated with this action are shown in the first two rows of Table 1b. The parameter-values set in Action 2 occur on the Save Window to set the file name to "exampleFile," select the file type as plain text, and click the OK button. The user sets parameter-values in Action 3 on the TerpWord main window by selecting Edit->Find. Action 4 involves parameter-values on the "Find" window.

The user sets the text of the "Find what drop-box" to "software defect" and then executes a "left-click" on the Find Next button. Table 1a summarises the windows, parameters, and values in this test case and assigns unique numbers to each window and action. Table 2a shows a sample user-session (test case) from a Book application that contains four actions.

Table 2
Example Web Test Case

No. of actions in Test Case	Action
Action 1	GET Default.jsp
Action 2	GET Login.jsp
Action 3	POST Login.jsp? Password = guest &FormName=Login &FormAction=login &Login=guest
Action 4	GET BookDetail.jsp? item_id =22

(a) Sample user session-based Web test case

Table 2b shows the important data that is parsed from the testcase. From the testcase in Table 2a, it is clear that the Login page is accessed with the parameter-values <Password, guest>, <FormName, Login>, <FormAction, login>, and <Login, guest>. For example test case in Table 2a, all of the parameter values are shown in Table 2b.

Window Name	P-V No.	P-V description (<parameter, value>)
GET Default.jsp	PV.1	<null, null>
GET Login.jsp	PV.2	<null, null>
POST Login.jsp	PV.3 PV.4 PV.5 PV.6	<Password, guest> <Formname, Login> <FormAction,login> <Login, guest>
GETBookDetail.jsp	PV.7	<item_id , 22>

(b) Windows and parameters value in test case

4. CONCLUSION

Through this study, the problem of finding defects in programming constructs is solved. The projects find zero, one and multiple function written classes are found out, and also long parameter list functions are also tracked. In function calling most of the functions of other classes are also found out. In addition, the test case generation is also taken place. The application is tested well so that the end users use this software for their whole operations. It is believed that almost all the system objectives that have been planned at the commencements of the software development have been met with and the implementation process of the project has been completed. A trial run of the system has been made and is giving good results. The procedures for processing are simple and regular order. The process of preparing plans has been missed out which might be considered for further modification of the application.

REFERENCES

[1] N. Fenton and S. L. Pfleeger, Software Metrics: A Rigorous and Practical Approach. Int. Thomson Comput. Press, London, UK, 1997.

[2] N. Moha, Y. G. Gu eh eneuc, L. Duchien, and A. F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," IEEE Trans. Softw. Eng., vol. 36, no. 1, pp. 20– 36, Jan./Feb. 2010.

[3] N. Moha and Y. G. Gueheneuc, "Decor: A tool for the detection of design defects," in Proc. Int. Conf. Autom. Softw. Eng., 2007, pp. 527–528.

[4] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in Proc. Int. Conf. Softw. Maintenance, 2004, pp. 350–359.

[5] M. Kessentini, W. Kessentini, H. A. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in Proc. IEEE 19th Int. Conf. Program Comprehension, 2011, pp. 81–90.

[6] M.Kessentini, S.Vaucher, and H.A.Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code " in Proc. Int. Conf. Autom. Softw.Eng., 2010, pp.113–122.

[7] C. Salto and E. Alba, "Designing heterogeneous distributed GAs by efficiently self-adapting the migration period," Appl. Intell., vol. 36, no. 4, pp. 800–808, 2012.

[8] N. Moha, Y. G. Gu eh eneuc, L. Duchien, and A. F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," IEEE Trans. Softw. Eng., vol. 36, no. 1, pp. 20– 36, Jan./Feb. 2010.

[9] P. Siarry and Z. Michalewicz, Advances in Metaheuristics for Hard Optimization (Natural Computing Series). New York, NY, USA: Springer, 2008.

[10] E. Alba,Parallel Metaheuristics: A New Classof Algorithms. Hoboken, NJ,USA:Wiley,2005.

[11] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code. Reading, MA,USA: Addison Wesley, 1999.

[12] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning. Reading, MA,USA: Addison Wesley, 1989.

[13] W. J. Brown, R. C. Malveau, W. H. Brown, and T. J. Mowbray, Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis. Hoboken, NJ, USA: Wiley, 1998.

[14] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," IEEE Trans. Softw. Eng., vol. 20, no. 6, pp. 293–318, Jun. 1994.

[15] K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression Testing in an Industrial Environment," Comm. ACM, vol. 41, no. 5, pp. 81-86, May 1988.

[16] B. Cleary, C. Exton, J. Buckley, and M. English, "An Empirical Analysis of Information Retrieval Based Concept Location Techniques in Software Comprehension," Empirical Software Eng., vol. 14, no. 1, pp. 93-130, 2008.

[17] M. Revelle, B. Dit, and D. Poshyvanyk, "Using Data Fusion and Web Mining to Support Feature Location in Software," Proc. 18th Int'l Conf. Program Comprehension, pp. 14-23, 2010.

[18] A.K. McCallum, "Mallet: A Machine Learning for Language Toolkit," http://mallet.cs.umass.edu, 2002.

[19] Z. Harris, "Distributional Structure," Word, vol. 10, pp. 146-162, 1954.