Advanced Science Letters
Vol. XXXXXXXXX

A Test Suite Reduction Approach for Software Unit Testing

Shadi Banitaan[1], Mohammad Akour[2], and Mamdouh Alenezi[3]
[1]College of Engineering & Science, University of Detroit Mercy, USA
[2]Faculty of Information Technology & Computer Sciences, Yarmouk University, Jordan
[3]College of Computer & Information Sciences, Prince Sultan University, Saudi Arabia

Software testing is usually starts with the Unit testing phase. The goal of unit testing is to reveal logic and implementation errors in each unit. The unit testing process is usually costly and time-consuming especially when the system under test is very large. In addition, because of the time pressures, the testing team may not find time to fully test the product. Therefore, identifying the units that have most of the errors helps the testing team to focus on testing them to save time and resources. In this paper, we propose an approach for unit testing that weights each method using a combination of static object-oriented metrics. The proposed approach predicts the number of test cases necessary to test system methods. It assumes that complex methods contain more errors which require executing more test cases to test the complex methods. The goal of this work is to help software developers where to dedicate their available resources when performing unit testing. The experimental results on the studied Java systems show that how small number of test cases are needed and how those test cases detect a high percentage of errors.

Keywords: Unit Testing, Test Focus Selection, Test Suite Reduction, Software Metrics.

1. INTRODUCTION

Software testing is an investigation conducted on a program with the intent of finding errors[21]. The cost of software testing is very high. It requires approximately 50% of software development cost[22]. Moreover, the distribution of bugs over different components of a software system is not uniform. Therefore, if testing resources can be concentrated on the more error prone components, then the accessible resources can be exploited more effectively, and the developed software will have a high quality with lower cost.

In this work, we focus on unit testing where individual units are tested independently. Unit testing is normally conducted by software developers because it requires deep understanding of the functional specification of the system under test. The developer usually writes test cases to test the system after he/she finishes the implementation to make sure that the system meets its design and behaves as intended. Unit testing is performed by isolating each part of the program and testing if individual parts are correct. Unit testing tries to find if the implementation of the unit satisfies the functional specification. The goal is to identify faults related to logic and implementation in each unit. If these faults are not detected, they may cause system failure when running the system.

In this work, we utilize several method level coupling and complexity metrics in order to rank software methods. Ranking methods may help software developers where to dedicate their available resources. Following are the summarized contributions in this work:

- Propose an approach to reduce the effort of unit testing in terms of the number of developed test cases using static object oriented metrics.
- Perform several experimental studies on three Java applications belong to different domains.
- Evaluate the proposed approach using mutation analysis.

The remaining parts of the paper is organized as follows: Section 2 briefly describes some background information needed in this work, Section 3 describes the proposed approach. The experimental evaluation and

*Email Address: banitash@udmercy.edu

discussion are presented in Section 4. Section 5 discusses related work. The conclusion of the paper is presented in Section 6.

2. Background

2.1 A test case

A test case is a documentation that identifies input values, expected output and the preconditions for executing the test. In other words, a test case is a sequence of steps executed on a software product, using a set of input data, expected to produce a set of outputs in a given environment[11]. A test suite is a collection of test cases with the goal of testing some specified set of functionality.

2.2 A dependency

A dependency happens when one component uses the services provided by another component. It is an association between two components such that changes to one component effect the other one. For example, if we have two components X and Y. If X depends on Y then X has an outbound dependency and Y has an inbound dependency. The presence of X requires the presence of Y. X is a dependent while Y is a dependee.

2.3 Size estimates

Size estimates of software testing is represented by number of test cases written, number of test scenarios covered, or number of configurations needed to be tested. It is the primary input for effort estimation. Effort estimation is very important since it has a direct relationship of testing cost. In this work, we consider the number of test cases to represent an effort estimation measure for unit testing[11].

3. The Proposed Test Reduction Approach

This section presents and discusses in details the proposed approach. Figure 1 depicts the proposed test reduction approach. The proposed approach in this research paper is composed of four main steps. First, the system dependencies are pulled out from Java byte-code using Dependency Finder[17]. The dependency graphs are then used to compute the dependency metrics. Second, the selected object oriented metrics are calculated. Third, for each single method the weight is measured using the chosen metrics in the prior step. Ranking methods gives an indication of which methods should be concentrated on through testing process. After that, the approach predicts the number of test cases recommended to test system methods based on its rank. In our proposed approach, we are trying to find the smallest set of methods that contain most of the errors. We are investigating whether the selected metrics can be used to identify the most error-prone methods.

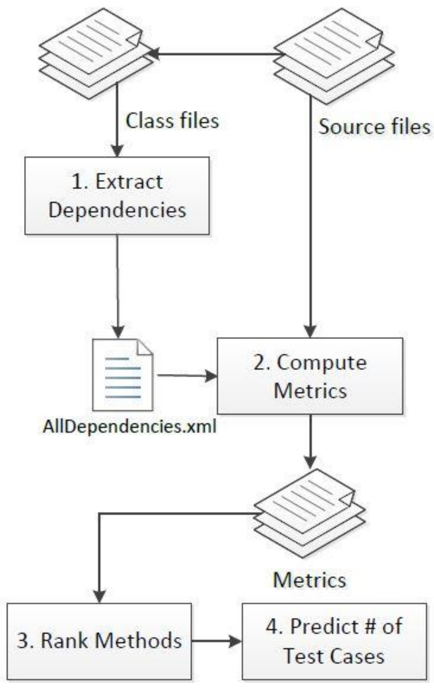

Fig.1. The proposed test reduction approach.

In this work, we use the following metrics:

- Inbound All Method Dependencies (IM): The set of methods that depend on a given method.
- Outbound All Meth Dependencies (OM): The set of methods that a given method depends on.
- Outbound All Field Dependencies (OF): The set of fields that a given method depends on.
- Local Variables (LVAR): The number of local variables used by a given method.
- PAR: The number of input parameters in a given method.
- NOCMP: The number of complex input parameters in a given method. The input parameter is considered complex if it is not a primitive data type.
- Maximum Nesting Depth (MND): The maximum depth of nesting for a given method.

The metrics are used to calculate and assign a weight for each method. The weight of method is calculated below:

$$weight(m_i|c_k) = \frac{IC_{mi} \times (IM_{mi} + OM_{mi} + OF_{mi})^2}{\sum_{y \epsilon M(c_k)} IC_y \times (IM_y + OM_y + OF_y)^2}$$

Where $M(c_k)$ denotes all methods in the c_k class. The denominator is used for normalization. ICm_i is the internal complexity of method m_i and it is measured as follows:
$ICm_i. = MNDm_i + PARm_i + NOCMPm_i + LVARm_i$

After calculating the weight for each method, the approach gives a recommendation about the number of test cases that are required to test each method based on methods weights. In this research paper, we suppose the preliminary number of required test cases to test a software system S is M(S)/2 where M(S) represents the total number of methods in the system. The weight for each method is measured using the aforementioned equation. The class weight is computed by calculating the summation of all method weights that associated with the targeted class. We create a tool to calculate the metrics automatically. The tool is developed using R language version 2.15.0[23].

4. Experimental Evaluation

How good are our test cases? We can answer this question by applying mutation testing. In mutation testing, mutants (artificial defects) are injected into software. Mutants are usually created using an automated mutation tool. In our work, mutation operators are automatically generated using MuJava tool. We create mutants using both intra-method and intra-class operators. MuJava tool is widely used to perform mutation analysis[21, 27]. After creating mutation operators, test cases are run against both the original program and the faulty programs. We say that the mutant is killed if test cases differentiate the output of the original program from the mutant programs; otherwise the mutant is still alive. Previous work found that mutation testing is a reliable way of assessing the fault-finding effectiveness of test cases and the generated mutants are similar to real faults[2, 3]. Lyu et al.[20] conducted an experiment which engaged 34 development teams to develop independent versions of a program to measure the effectiveness of coverage testing versus mutation testing. They concluded that mutation testing is a more truthful indicator of testing quality.

For evaluation purposes, we pick three free open source applications that are implemented using Java programming language. Table 1 presents a summary of the software applications under study. PureMVC is a framework for building applications based on the Model, View and Controller concept. The Cinema application is a management system that is responsible for movie tickets and movie schedules. ApacheCli is a library that provides an API for parsing command line options passed to programs.

Table.1. A summary of the applications.

Project	# of classes	# of methods	Source
PureMVC	22	139	http://puremvc.org/
Cinema	10	106	http://alarcos.esi.uclm.es
ApachiCLI	20	207	http://commons.apache.org/cli/

The selected systems contain unit test cases developed using the JUnit testing framework. The JUnit framework is an open-source testing framework. JUnit is usually used for developing unit tests where each single method is tested in isolation. Tables 2, 3, and 4 show the results of applying mutation testing on the selected applications. For the PureMVC application, 30 mutants are injected and 26 of them are killed. For the Cinema application, 1545 mutants are injected and all of them are killed. For the ApacheCLI application, 1924 mutants are injected and 1562 of them are killed. The mutation score is 86.67% for PureMVC, 100% for Cinema, and 81.19% for ApacheCLI. Therefore, our approach is effective in detecting at least 81.19% of mutants.

Table.2. PureMVC results.

Class	Killed	Lived	Total
Façade	16	0	16
Mediator	0	2	2
Notification	4	0	4
Observer	2	0	2
Proxy	4	2	6
Total	26	4	30

Table.3. Cinema results.

Class	Killed	Lived	Total
Asiento	51	0	51
Cine	1491	0	1491
Sesion	3	0	3
Total	1545	0	1545

Table.4. ApachiCLI results.

Class	Killed	Lived	Total
CommandLine	50	0	50
HelpFormatter	724	22	746
GnuParser	42	0	42
Option	261	0	261
OptionBuilder	39	8	47
OptionGroup	7	2	9
Options	5	0	5
OptionValidator	127	90	217
PatternOptionBuilder	157	240	397
PosixParser	118	0	118
TypeHandler	26	0	26
Util	6	0	6
Total	1562	362	1924

A comparison with a base line approach is also conducted. For the base line approach, all test cases available with the selected software systems are executed. We perform two comparisons with the baseline approach. The first one is performed by measuring the savings that can be achieved by the proposed approach. The savings is measured using the following equation:

$$Savings = \frac{|T_{base}| - |T|}{|T_{base}|} * 100$$

Table.5. The comparison with the baseline approach.

Application	The Proposed Approach			The Baseline Approach		
	# of test cases	Mutation Score	Score	# of test cases	Mutation Score	Score
PureMVC	70	86.67%	1.24	139	93.30%	0.67
Cinema	53	100%	1.89	106	100%	0.94
ApacheCli	104	91.19%	0.88	207	94.33%	0.46

Where T_{base} represents the number of test cases of the base line approach while T represents the total number of test cases that are chosen by the proposed approach. The proposed approach achieves 50% savings for the selected applications. The second comparison is conducted by calculating a score that considers both mutation score and number of developed test cases. The score is computed as follows:

$$Score = \frac{Mutation\ Score}{|T|}$$

Table 5 shows the results of comparing the proposed approach with the base line approach. The results indicate that the proposed approach achieves a higher score than the base line approach for all of the selected applications. Based on the two comparisons, we conclude that the proposed approach is feasible and outperforms the baseline approach.

5. Related Work

A number of approaches have been proposed in literature to reduce the cost of software testing which include test prioritization[13, 28], test selection [7, 19], and test minimization [25, 16]. The purpose of test prioritization is to rank test cases so that test cases that are more effective according to a given criteria will be executed first to maximize the fault detection rate. The purpose of test selection is to identify test cases that are not needed to run on the new version of the software. The purpose of test minimization is to remove test cases that are redundant based on a given criteria.

Another direction to reduce the cost of software testing is to predict fault-proneness modules and then concentrate the testing effort on the modules that are recognized to be more error prone. Different approaches are available in literature about building fault-proneness prediction models using of object-oriented metrics[6, 9, 8, 14]. Benlarbi and Melo[6] identified and used polymorphism measures to predict fault-proneness on class level. Their results reveals that system's quality can be predicted by their measures. Briand, Wüst, Daly, and Porter[9] studied the relationships between object-oriented design measures and fault proneness at the class level. In their work, they outlined the fault-proneness as the probability of detecting a fault in a class. Their results showed that coupling induced by method invocations, the rate of change in a class due to specialization, and the depth of a class in its inheritance hierarchy are strongly associated to the fault-proneness in a class. Denaro and Pezze[10] used logistic regression to predict fault-prone modules. They reported that their best model required about 50% of the modules to be investigated in order to find 80% of the software faults. The primary goal of this work is to find most of the errors (80%) by using a small number of test cases (half number of methods).

Predicting which parts of the system are more fault-proneness can help the testers during unit and integration testing to concentrate on the faulty classes. However, most of the work presented so far do not provide experimental evidence of the effectiveness of predicting fault-prone classes on tuning the testing process. In addition, most of previous work do not integrate prediction models into the development process (i.e., how to use the prediction models in testing). Most approaches for predicting faulty classes use binary classifiers to build predictive models. Therefore, it cannot tell which methods are faultier than other methods. Therefore, testing team cannot know which methods in each class they can spend more resources on. Therefore, the proposed approach works on a finer-grain level (method level) since it is the smallest unit of object-oriented systems.

Some approaches were proposed to reduce the cost of testing by reducing the number of executed test cases. Bouchaib[15] used a set of complexity metrics to select test cases for regression testing. The experimental results showed that the executed test cases based on the proposed approach detected 100% of seeded errors and at least 60% of mutants. Banitaan, Alenezi, Nygard, and Magel[4] proposed an approach to select the test cases in integration testing. Their approach used a combination of object oriented metrics to give a weight for each method-pair connection. After that, the approach gives a recommendation about the number of test cases needed to test each method-pair connection. Their experimental results on Java applications showed that the small number of developed test cases detected most of the integration errors. Banitaan, Daimi, Wang, and Akour[5] proposed an approach for test case selection using software metrics. They examined the ability of two complexity metrics and three size metrics to find the most error prone classes. Their experimental results showed that their proposed approach significantly reduce the number of test cases needed for execution while detecting most of errors.

6. Conclusion

In this paper, we proposed an approach to reduce cost and time of the unit testing process. The main goal behind this work is to concentrate testing effort on part of software methods while raising the defect detection and detect at least 80% of errors. The proposed approach is divided into four steps. First, the dependencies are extracted from the Java byte code. Second, metrics are measured by utilizing both the source code and the dependency relationships. Third, a weight is calculated for each method using a set of object oriented metrics. Fourth, the approach gives a recommendation about the number of test cases needed to test each method. The experimental evaluation using mutation testing showed that the test cases executed based on the proposed approach can detect at least 81.19% of mutants. A comparison with the base line approach revealed that the proposed approach is feasible in reducing the number of test cases needed. Future directions include expanding the proposed approach to work on systems developed using other object oriented languages such as C#.

References

[1] M. Akour, A. Jaidev, and T. M. King, Towards change propagating test models in autonomic and adaptive systems, in Engineering of Computer Based Systems (ECBS), 2011 18th IEEE International Conference and Workshops on, IEEE, 2011, pp. 89_96.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche, Is mutation an appropriate tool for testing experiments?, in Proceedings of the 27th international conference on Software engineering, ICSE '05, New York, NY, USA, 2005, ACM, pp. 402411.

[3] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, Using mutation analysis for assessing and comparing testing coverage criteria, IEEE Trans. Softw. Eng., 32 (2006), pp. 608_624.

[4] S. Banitaan, M. Alenezi, K. Nygard, and K. Magel, Towards test focus selection for integration testing using method level software metrics, in Proceedings of the 10th International Conference on Information Technology: New Generations, April 2013.

[5] S. Banitaan, K. Daimi, Y. Wang, M. Akour, Test Case Selection using Software Complexity and Volume Metrics, in the 24th International Conference on Software Engineering and Data Engineering, SEDE'15, San Diego, CA, USA.

[6] S. Benlarbi and W. L. Melo, Polymorphism measures for early risk prediction, in Proceedings of the 21st international conference on Software engineering, ICSE '99, New York, NY, USA, 1999, ACM, pp. 334_344.

[7] L. C. Briand, Y. Labiche, and S. He, Automating regression test selection based on uml designs, Inf. Softw. Technol., 51 (2009), pp. 16_30.

[8] L. C. Briand, J. W€.st, and H. Lounis, Replicated case studies for investigating quality factors in object-oriented designs, Empirical Software Engineering: An International Journal, 6 (2001), pp. 11_58.

[9] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, Exploring the relationship between design measures and software quality in object-oriented systems, J. Syst. Softw., 51 (2000), pp. 245_273.

[10] G. Denaro and M. Pezzè, An empirical evaluation of fault-proneness models, in Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, IEEE, 2002, pp. 241_251.

[11] S. Desikan and G. Ramesh, Software Testing: Principles and Practices, Pearson Education India, 2012.

[12] M. J. T. E. Dietrich, J. and S. M. A. Shah, On the existence of high impact refactoring opportunities in programs, in Australasian Computer Science Conference (ACSC 2012), M. Reynolds and B. Thomas, eds., vol. 122 of CRPIT, Melbourne, Australia, 2012, ACS, pp. 37_48.

[13] S. Elbaum, A. Malishevsky, and G. Rothermel, Test case prioritization: A family of empirical studies, IEEE Trans. Software Eng., 28 (2002), pp. 159182.

[14] K. E. Emam, W. L. Melo, and J. C. Machado, The prediction of faulty classes using object-oriented design metrics, Journal of Systems and Software, 56 (2001), pp. 63_75.

[15] B. Falah, Test Case Selection Based on a Spectrum of Complexity Metrics, PhD Dissertation, North Dakota State University, 2011.

[16] H. Hsu and A. Orso, Mints: A general framework and tool for supporting test-suite minimization, in Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, IEEE, 2009, pp. 419_429.

[17] T. J., Dependency finder, 2008. http://depfind.sourceforge.net/ (Online; accessed 2014).

[18] T. Khoshgoftaar, E. Allen, and J. Deng, Using regression trees to classify fault-prone software modules, Reliability, IEEE Transactions on, 51 (2002), pp. 455_462.

[19] Y. Ledru, G. Vega, T. Triki, and L. Bousquet, Test suite selection based on traceability annotations, in Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2012, pp. 342_345.

[20] M. Lyu, Z. Huang, S. Sze, and X. Cai, An empirical study on testing and fault tolerance for software reliability engineering, in Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on, IEEE, 2003, pp. 119_130.

[21] A. Masood, R. Bhatti, A. Ghafoor, and A. P. Mathur, Scalable and effective test generation for role-based access control systems, IEEE Trans. Software Eng., 35 (2009), pp. 654_668.

[22] G. Myers, C. Sandler, and T. Badgett, The art of software testing, Wiley, 2011.

[23] R Development Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria, 2011.ISBN 3-900051-07-0.

[24] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, An empirical study of the e_ects of minimization on the fault detection capabilities of test suites, in In Proceedings of the International Conference on Software Maintenance, 1998, pp. 34_43.

[25] S. Tallam and N. Gupta, A concept analysis inspired greedy algorithm for test suite minimization, in Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, PASTE '05, New York, NY, USA, 2005, ACM, pp. 35_42.

[26] W. Wang, X. Ding, C. Li, and H. Wang, A Novel Evaluation Method for Defect Prediction in Software Systems, in International Conference on Computational Intelligence and Software Engineering, Wuhan, China, 2010, pp. 1_5.

[27] B. Yu, L. Kong, Y. Zhang, and H. Zhu, Testing java components based on algebraic speci_cations, in Proceedings of the 2008 International Conference on Software Testing, Veri_cation, and Validation, ICST '08, Washington, DC, USA, 2008, IEEE Computer Society, pp. 190_199.

[28] Z. Zhang, Y. Mu, and Y. Tian, Test case prioritization for regression testing based on function call path, in Computational and Information Sciences (ICCIS), 2012 Fourth International Conference on, IEEE, 2012, pp. 1372_1375.