Estimation of a Set of Package Metrics to Determine the Package Testability Efforts on Open Source Software

Aleem Khalid Alvi

Computer Engineering Department,
College of Computer Sciences and Engineering,
King Fahd University of Petroleum & Minerals,
Dhahran 31261, SAUDI ARABIA

Abstract

Open source software projects are the resource of the joint effort of the individuals and industry. eXtreme programming is an attractive paradigm for Open source software development. For supporting the testing activities in eXtreme programming JUnit framework is implemented for writing repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks. It is used to test the software on both the functional and package level by implementing the corresponding test cases. In this paper we are investigating the package level functional testing efforts using a case study. We have selected the Eclipse platform SDK 3.2 as case study and its implementation. JUnit test cases were used for package level functional testing in Eclipse SDK 3.2. The objective of this paper is to identify a set of package metrics that can be used to determine the package testability efforts using Java based software system and correlations between test metrics.

1. Introduction

Success of any software product depends on its quality. The quality can be measure in attribute as reliability of the system. For ensuring the reliability; testing is the main activity in software development cycle. However after launching its final release; defects are reported which shows the loop hole in the testing activities. More requirement changes require more testing efforts; which is the case of extreme programming. In extreme programming the changes are implemented on ad hoc bases. JUnit [1] is a simple framework to write repeatable tests for component level. Our goal is to identify and evaluate a set of package level metrics that can be used to determine the package testability efforts using Java based software system and correlations between test metrics.

JUnit testing framework [2] is much famous now in agile development processes like eXtreme programming [3]. The resources for implementing this strategy in Java are open source and support complete development cycle. JUnit is used both for class and package level unit testing. It provides the test class C_T for each source class C for class level testing. Similarly, for package level functional testing it provides the test package P_T for source package P for package level testing. Our work is the extension of the work given in M. Bruntink et al. [4], where the authors did the empirical study into class testability using the pair <C, C_T> and by using these pairs to find class level metrics on C that are good predictors of test-related metrics on C_T. We are investigating using <P, P_T> pair to find package level metrics on P that are good predictors of the test-related metrics on P_T.

We have taken Eclipse platform SDK (version 3.2), which is the java system as a subject for our empirical research. Eclipse platform SDK has JUnit test suites, which were available from the Eclipse website [5]. This system is developed based on eXtreme programming paradigm.

Our motivation of this work is to understand relationship between the package level metrics and software testability. Discovery of significant effects between the metrics and testability will lead the Test Manager to write test cases that are more efficient or in other way, he can plan and monitor his testing activities productively. Software developers can use this data to review their code and improve them using refactoring in more practical purposes.

On software testability, we have done literature survey and following two sections are based on this. In section 2 we discuss the related research work. Section 3 covers the background material for this paper. We discuss about evolution towards measuring testability. In section 4 we describe the list of package level metrics which we selected for our experiment. We discuss complete setup of our experiments using subsections package metrics, hypothesis, and test suite metrics and statistical analysis. Further, we give details about our selected subject. In section 5 and 6 we present and analyze the results. Section 7 is dedicated for threat of validity. Finally, we conclude our experimental results and discuss possible future extensions of our work.
2. Related work

We found lot of testability paradigm and procedures; however, we have selected only those that are closer to our work. Freedman [6] defines the domain testability by applying the terms controllability and observability to software. These terms are already in use for hardware components. Domain testable program in his approach is observable and controllable. It means this program has consistency among test input and output. He showed the testability approach is much successful in domain testable specifications of program and take less time for test case building and execution. Y. Le Traon used the improved version proposed in [7] and uses it to find testability of data flow software designs. In addition, he defines the testability of data flow software design and the axiomatization of its expected behavior. Both in [6-7] the authors are interested in domain level testability where as our work is to evaluate the package level testability. Which gives more information then domain level because domain consideration gives more abstract information then package.

In 1995, M. V. Jeffrey [8] described new verification way for finding the hidden faults in software system. They used the fault injection concepts to introduce the faults and capture system behavior to verify and predict the more likeliness of the hidden faults. We used in our work JUnit testing framework, this elaborate hidden faults more. So by our approach the hidden fault occurrence will be minimized because module or package level testing gives stronger information regarding the system fault.

Lamoreaux et al. [9] propose a process for improving software testability where the authors developed a process for creating test harnesses in an automated testing environment. The process allows building maintainable products that contain hooks for testability at the time of products release.

Further M. Brunstink et al. [4] use the class testability concepts and facilitate their experiments by using the JUnit test to investigate factors of the testability of object-oriented software systems by using initial model of testability and existing object-oriented metrics related to testability. They identify and evaluate a set of metrics that can be used to assess the testability of the classes in object-oriented system. In this paper, the authors reported the significant relationship between the class-level metrics and the test code metrics. For this purpose, they are using the class and test class pair as input from five objects oriented systems to their experiment. Where as we intend to consider the relationship between the package level metrics and test metrics. Our work gives another picture for the relationship between these. We have set the pilot experiment on Eclipse C/C++ Development Tooling (CDT). Based on the results we have applied some changes i.e. in metric selection and strategy for selecting the pair of < P, P_i > packages. This refines our results and we have come with the more promising conclusion.

We have mentioned the eclipse-testing framework [10] for more understanding about how the test packages were formed.

3. Software testability

Software testability is one of the important aspects for controlling the software development activities in the fruitful direction. For getting quality results in the software product it is require to view the testability activities from start and measure it. Monitoring gives the control on the testing activities. By understanding, the relationship of the testability and package metrics software development activities will be more strengthen. The IEEE Standard Glossary of Software Engineering Terminology [11] defines testability as: (1) the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met, and (2) the degree to which a requirement as stated an terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met. Where The ISO [12] defines testability as attributes of software that bear on the effort needed to validate the software product (ISO, 1991).

Many testability definitions are available from researchers one of the dominant is from M. V. Jeffrey [13] that defines it as Software testability is a software characteristic that refers to the ease with which some formal or informal testing criteria can be satisfied. The simplest definition is the ease of performing testing is in [14]. This definition has its roots in hardware testing and is usually defined in terms of observability and controllability [6]. B. Pettichord [15] has given broader view of testability to explain in term of operability, controllability, observability, simplicity, suitability, understandability, stability.

4. Experimental design

The objective of experiment is to identify and estimate the package metrics that have significantly effect in predicting the package testability. In subsection 4.1, we discuss package level metrics used here for our experiment. We use these metrics to elaborate our hypothesis in section 4.2. Further, in last subsection we select the test metrics for the package.

4.1. Package level metrics

The selection of metrics is based on our interest to find the complexity and scope of the software testability. For this purpose we select the metrics suite proposed by Robert Cecil Martin [16] and addition to that, we have added Single lines of code (SLOC) by Henderson-Sellers [17]. We added one metric in this list is the Number of Packages (NOP), for finding the relationships with test metrics as mentioned in section
4.3. The metrics used in this experiment are listed below. These all metrics are the independent variables in our experiments.  

**Metric 1: Concrete class count (CC)**
Concrete class count is the simple metric for finding the size of the package and it shows the enlargement of package.

**Metric 2: Abstract class count (AC)**
Abstract class count is a metric for finding the number of abstract classes in a package.

**Metric 3: Number of classes and interfaces (NOCI)**
The number of concrete and abstract classes (and interfaces) in the package is an indicator of the extensibility of the package.

**Metric 4: Efferent Couplings (Ce)**
It is an indicator of the package's independence and it defines as the classes in the package depend on the number of other packages.

**Metric 5: Afferent Couplings (Ca)**
It is the inverse in definition from Efferent couplings i.e. the number of other packages that depend upon classes within the package is an indicator of the package's responsibility.

**Metric 6: Abstractness (A)**
The ratio of the number of abstract classes (and interfaces) in the analyzed package to the total number of classes in the analyzed package. The range for this metric is 0 to 1, with A=0 indicating a completely concrete package and A=1 indicating a completely abstract package.

**Metric 7: Instability (I)**
The ratio of efferent coupling (Ce) to total coupling (Ce + Ca) such that I = Ce / (Ce + Ca). This metric is an indicator of the package's resilience to change. The range for this metric is 0 to 1, with I=0 indicating a completely stable package and I=1 indicating a completely unstable package.

**Metric 8: Distance from the Main Sequence (D)**
The perpendicular distance of a package from the idealized line A + I = 1. This metric is an indicator of the package's balance between abstractness and stability. A package squarely on the main sequence is optimally balanced with respect to its abstractness and stability. Ideal packages are either completely abstract or stable (x=0, y=1) or completely concrete and instable (x=1, y=0). The range for this metric is 0 to 1, with D=0 indicating a package that is coincident with the main sequence and D=1 indicating a package that is as far from the main sequence as possible.

**Metric 9: Single Line of Code (SLOC)**
This metric is count the number of code lines in the method where comments and empty lines are not counted. This is the good indicator of the real size of the class and in our case, it is collectively count the methods in all classes of the package, however large methods are usually a sign of poor design.

**Metric 10: Number of Package (NOP)**
Number of package (NOP) metric used here to further find the relation with the test metrics number of packages in matched testing code package (NOPt). This is the good indicator for the size.

## 4.2. Goal and Hypotheses

First, we evaluate this set of metrics by applying the GQM technique [18]. Therefore, we first describe the goal, perspective and environment of our experiment.

**Goal:** To assess the capability of the proposed package metrics to predict the testing effort.

**Perspective:** We evaluate the metrics at the package level. Thus, we are assessing whether or not the values of the package metrics can predict the required amount of effort needed for unit testing a package.

**Environment:** The experiments are targeted at Java systems, which are unit tested at the package level using the JUnit testing framework. In section 4.4, we discuss about the case study more elaborately. The JUnit framework allows creating the package level functional test cases in the form of group of test classes dedicated to the targeted source component as test suits.

We set the questions for evaluating goal for measuring package testability.

**Question 1:** What is the relationship of the package level metrics to measure the testing effort for the package?

The actual development effort (time) are proportional to size of the project. In COCOMO, size of the project is used for measuring the time spends on the project [19, 20]. Therefore, the size of the corresponding test suite is a good indicator of the testing effort. As we have discussed that, in JUnit framework, the test package used to test a package and similarly for the class, however all source packages may not be tested in this way. A special case is the test first approach [21], where we write test first then code therefore every code component has a corresponding test component. In section 4.3, we precisely discuss the test suite metrics for measuring the testing effort.

**Question 2:** Is there any association that exists between the package metrics for a package and the size of the corresponding test suite?

**Question 3:** Are the values of the package level metrics proportional to the corresponding test suite size for that package?

Based on these questions we set the following hypothesis for the experiments.

**Hypotheses:** These hypotheses translate our goal into measurement statements as follows:

The null hypothesis is

\[ H_0(m, n) : \text{There is no association between package metric } m \text{ and test suite metric } n. \]

The alternative hypothesis is

\[ H_1(m, n) : \text{There is association between package metric } m \text{ and test suite metric } n, \text{ where } m \text{ ranges over our set of package metrics, and } n \text{ over our set of test suite based metrics.} \]

## 4.3. Test metrics

Proposed test metrics for our experiments are,

Single Lines Of Code for Package (SLOCP), Number Of
test cases/Classes in a Package (NOCP) and number of sub packages in a test package (NOP) as the indicator of the size of the test suites. These all are our dependent variables. SLOCP metrics is calculated by using SLOCC (Single Lines Of Code for Class) metric, this can be shown as below

\[ SLOCP = \sum_{i=1}^{n} SLOCC_i \text{, where } n = \text{no. of classes} \]

Therefore sum of all SLOCC’s give the SLOCP measure.

NOCP is the measure of the number of test case/classes. One class is called one test case and one method is called test. where group of test cases are called test suite [22]. These classes are responsible to test the source package by running the test cases.

\[ NOCP = \sum_{i=1}^{n} TC_i \text{, where } TC = \text{Test classes} \]

**Procedure:**
We use following procedure to find the metrics values.
1. Use the package metrics tool, which is mentioned, in next section to find the package metrics measurements.
2. Use the metrics tool to find the package level measurement.
3. Use the statistical analysis (using Spearman’s rank order correlation coefficient) for analyzing the data collected from above steps.
4. Come to the result and analysis of results for the result extracted.
5. Conclusion based on step 4.

**Supporting tools:** We are using Diff Commander 1.1.1 [23] (for directories and file comparison), JDepend4Eclipse [24] as plug in which uses JDepend version 2.9.1 [25] for test package metrics. For SLOC and NOP metrics calculation, we use DependencyFinder 1.1.1 [26]. In addition, we use Eclipse metric 1.3.6 tool [27] as plug-in for cross verification.

4.4. Experimental Subject
Open source software is now days popular and its development is based on highly distributed environment. The best example for this distributed development is SourceForge® Many software starts from open source paradigm and then shifts its mature version to commercial based project; LINUX is an example i.e. Red Hat Linux is the commercial version and its open source version called Fedora Linux, however in the beginning start the LINUX was completely an open software structure, naming conventions and arrangement paradigm and many times their documentations and revision history are poor as compare to commercial based software. Our motivation for open source software is to see and verify the package testability effort corresponding to the package metrics for source code. This gives us some association between them and based on that association we can predict more precisely the testing effort. Especially for those open software whose development teams follow the eXtreme programming paradigm have spent much effort for testing. This is the most important point for selection of JUnit framework for testing. The subject for experiment is a project who has used the JUnit framework for testing because we need to select the pair < P, P > of packages for our experiments. This could only be done in cases where software projects have been developed using JUnit testing framework. Hence based on all our intentions we selected the latest mature version of Eclipse platform SDK (version 3.2) with sub project as plug in i.e Java Development Tooling (JDT), Plug in Development Environment (PDE) and Rich Client Platform (RCP). After analysis of their software structure, naming conventions and arrangement of source and test code, we decided to do pilot experiment first. Therefore, we have selected the Eclipse C/C++ Development Tooling (CDT) for this experiment. After getting the successful results we apply our methodology to large project i.e. Eclipse platform SDK.

4.5. Experiment Implementations
The experimental subject is Eclipse platform SDK (version 3.2). We have used JDepends to measure the Concrete class (CC), Abstract class (AC), Number of Classes (NOC ), Afferent Couplings (Ca), Effferent Couplings (Ce), Abstractness (A), Distance from the main sequence (D), Instability (I) and used dependency finder 1.1.1 to measure the metric Single Line Of Code (SLOC) for source packages. For test packages, we have measured Number of package (NOP), Number of classes in package (NOCP), and Single Lines Of Code in package (SLOC) by using the dependency finder 1.1.1. Based on all the data, we found 51 matched pairs of source and test packages. However, we left more then three time of the test packages because we have not found the exact pair. So the selection is strictly based on the real matched pair of packages. The tools used to create the experimental setup are shown in table 1. Results are shown in the section 5 below.

4.6. Statistical Analysis
We have used the spearman correlation coefficient \( r_s \) (m, n) for the finding the correlation between the source and test packages. Spearman’s rank-

http://sourceforge.net/
http://www.redhat.com/
http://fedoraproject.org/
order correlation coefficient is use to measure the association between two variables where they are measured in an ordinal scale [28]. A nonparametric and distribution-free rank statistic proposed by Spearman in 1904 as a measure of the strength of the associations between two variables [29]. The Spearman rank correlation coefficient can be used to give an R-estimate, and is a measure of monotone association that is used when the distribution of the data make Pearson's correlation coefficient undesirable or misleading [30]. Our whole analysis is based on this statistical technique and we measure the relationship using this and set \( p = 0.05 \) for finding the significant relationship.

5. Results

We use the spearman correlation coefficient \( R \) from statistics, which is the best way to finding the correlation among the source and test package pair. The results are shown in table 2.

6. Analysis of results

The result of Spearman \( R \) and \( p \)-level is shown in bold, where the correlations for source and test metrics are existed. We observe the positive and negative correlation between source and test metrics. We have found the positive correlation between the source metric Efferent coupling (Ce) to test metric Number of test packages (NOPt) (See common cell in row = 4, column = 5). This shows that if the classes in the analyzed package have more dependency on other packages than the corresponding number of test packages will be increased. This proves our hypothesis true for this pair of the source and test metrics.

The other negative correlation is found between source metric, instability (I) and the test metric, Number of test packages (NOPt) (See common cell in row = 7, column = 5). This discovery supports the first correlation. If there is more dependency from classes of analyzed package to other packages then stability of the analyzed package decreases. Therefore, our hypothesis is true for this pair of the source and test metrics.

The last correlation we found is between the source metric, number of source packages (NOP) and corresponding test metrics, number of sub packages in a test package (NOPt). This relationship shows high positive correlation among the pair of variables. This means number of source packages is increased then the corresponding number of test packages will be increased. However, our hypotheses are not accepted for the correlation with every pair shown in table 2.

### Table 2. Measurement for Eclipse platform SDK (version 3.2)

<table>
<thead>
<tr>
<th>Package / Source metrics</th>
<th>Test metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SLOCP</td>
</tr>
<tr>
<td></td>
<td>Spearman R</td>
</tr>
<tr>
<td>CC</td>
<td>-0.056890</td>
</tr>
<tr>
<td>AC</td>
<td>-0.006673</td>
</tr>
<tr>
<td>NOC</td>
<td>0.070491</td>
</tr>
<tr>
<td>Ce</td>
<td>0.047077</td>
</tr>
<tr>
<td>Ca</td>
<td>0.039320</td>
</tr>
<tr>
<td>A</td>
<td>0.100130</td>
</tr>
<tr>
<td>D</td>
<td>-0.040926</td>
</tr>
<tr>
<td>NOP</td>
<td>0.099014</td>
</tr>
<tr>
<td></td>
<td>-0.111602</td>
</tr>
</tbody>
</table>

**Legend:**
- JD Jdepends 1.1.0
- EMp Eclipse Metric plug-in 1.3.6
- DF Dependency Finder 1.1.1
For example, SLOCP and NOCP show no correlation with any of the source metric. This may be because we have left many test packages because they could not be matched 100% with the corresponding source package.

7. Threats to validity

a) Threats to Internal validity:
   i) Package size: It observes that associations between the investigated metrics and fault-proneness may have some effect because of package size. Therefore, it should be taken into consideration. As a result, package size has a strong confounding effect.

b) Threats to External validity:
   i) Testing criteria: A possible confounder for the validity of testability metrics is given by the testing criterion in use at the project under consideration. The validity of the result is threatened by this fact in two ways. First, it is hard to judge applicability of our results to other systems, since the testing criteria may be incompatible. Second, a missing testing criterion hinders evaluation of test suite quality.
   ii) Selection Bias: Another threat may have perplexing effect because we have selected those packages that have associated test suite packages i.e. \(<\text{P}_T, \text{P}> \) pair. A possible selection bias is therefore present.

c) Threats to construct validity:
   i) Metric selection: We have selected the list of well-known metric for measuring the packages. These metrics include the size related, dependency related, and number of classes, stability related and number of packages matrices. However, there may be other metrics, which may shows correlation in test and source packages if we use them in our experiment. e.g., number of public classes, number of synthetic classes, number of inner classes etc., in a package. Therefore, there is the confounding effect for selection of metric with the resulting correlations.

8. Conclusion and future work

The reason for the experiment is to increase our understanding of what can be done during the code development that can decrease the testing effort and to decrease the hardness of test during the source code development. For this purpose, we have selected the JUnit test packages corresponding to source packages and applied the metrics tool to find the source and test metrics. Than we find the relationship between them by analyzing the source and test metrics data. We were able to demonstrate significant correlations between package level metrics (Ce, I, NOP) and test level metric (NOPt).

Our approach is based on strategy, which was further modified and refined after the pilot experiment. After getting the data, we applied the spearman correlation coefficient (\(r_s\)) that gave us the correlation among the package and test metrics. After that, we have discussed the results in detail. Some of the proposed package metrics showed significant relationship with the package testability. Base on this results prediction can be done more efficiently and accurately and will give us understanding on what makes testing the code harder.

We have used strategy-based approach for our experiment. In pilot experiment, we set one strategy and after that, we refine it for the final experiment. That approach has significantly changed the results and we got more appropriate and sensible results. However, we have seen that we were missing many test packages for the selection, as we did not find their exact pairs. Selecting more than five open source projects for this experiment (selected projects must contain good number of pairs \(<\text{P}, \text{P}_T>\) ) will extend this experiment and confirms our results. Further if the projects have different languages, development and testing techniques that might give more generalized results.

Extending number of metrics for the testability measurement may give us more correlation between test and source level metrics. Another study can be done to find and use more powerful statistics than Spearman’s rank-order correlation in order to further assess the predictive capabilities of the metrics.

9. Acknowledgment

The author has done this research work under his PhD studies in King Fahd University of Petroleum & Minerals (KFUPM), Dhahran, Saudi Arabia. He would like to thank KFUPM for supporting his research work. In addition, the author would like to thank Sir Syed University of Engineering & Technology, Karachi, Pakistan, for providing a chance to work in KFUPM as PhD student in computer department.

References


