Journal of

Software Engineering

ISSN 1819-4311

Academic
Journals Inc.

www.academicjournals.com

Journal of Software Engineering 4 (2): 122-136, 2010
ISSN 1819-4311

Predicting Testability of Eclipse: A Case Study

Y. Singh and A. Saha
GGSIP University, India

Abstract: This study attempts to predict the testability of Eclipse (one of the biggest open source project). Testability is an important attribute of software quality. Assessment of software testability can help in predicting the testing effort required for a given product. In this study the testability is predicted at package level. Eclipse is a project whose functional testing is done at the package level. We have developed an Eclipse plugin to extract the value of source code metrics and test metrics from the source code of Eclipse project. The source code metrics and test metrics of Eclipse project are calculated at the package level. Test metrics are obtained from the J unit test classes of test packages. The package level metrics are obtained from the class level metrics. A correlation is found between source code metrics and test metrics. The results show that there is a significant correlation between source code metrics and test metrics at the package level. Since, the test metrics assess the testing effort and through testing effort an assessment of testability is made. The results show that testability can be assessed from the source code metrics. This study can help software practitioners to have an understanding of package level testability and to assess the testing effort for a given package.

Key words: Metrics, object-oriented, empirical study, JUnit, package

INTRODUCTION

Software testability is an important attribute of software quality. By predicting software testability, improvements can be made in the quality of the software. Software quality can be assured through software testing or through testability improvement. There are software testing techniques which aim at improving the testability of the software (Ying *et al*., 2008; Liu *et al*., 2001; Ananya and Bhattacharya, 2004; Beizer, 1990). This study focuses on the testability measurement only. A number of definitions of testability have been given in the literature. Most of them relate testability to the testing effort. Software testability has been defined by a number of researchers with different viewpoints. The IEEE standard glossary (IEEE Press, 1990) has defined testability as: the degree to which a system or component facilitates the establishment of test criteria and performance of tests to determine whether those criteria have been met. Voas and Miller (1994) claims that testability analysis is a kind of validation as it quantifies the semantic content of the program. Software testability has two key concepts: controllability and observability. To test a given component, one must be able to control its input and observe its output (Binder, 1994). If we cannot control the input, we cannot say what produced the given output. If we cannot observe the output of a component under test, we cannot be say how a given input is processed.

Software testability is an important characteristic of the software and is also defined in terms of the software development process and as a characteristic of the architecture, design,

Corresponding Author: Yogesh Singh, GGSIP University, India

and implementation of the software (Kolb and Muthig, 2006). There are a number of different approaches for the assessment of software testability which have been proposed in the literature. These approaches work at various stages of the software development life cycle. i.e., at the design level, implementation level etc. The area of improving the software testability had also been the focus of many researchers. Although, a lot of research has been done in the area of software testability and there are a number of approaches for the assessment of software testability, design for testability and to improve the testability. Although a number of empirical studies exist to assess the testability in procedural and object oriented software. These studies focus on the class level testability only. There are no empirical studies at the package level to the best knowledge of the authors. This study focuses on the assessment of package testability of object oriented software.

Testability is difficult to measure directly since it is an external attribute. Hence the measurement of the testing effort is done to help in assessing the testability. This paper quantifies testability through the testing effort. In this study, we evaluate the effect of the source code metrics on testing effort for object oriented development at the package level. This study is performed using 50 packages (25 packages of source code and 25 test packages) of the Eclipse project. The Eclipse project is tested through functional testing method. Corresponding to java packages there are test packages which contain JUnit test classes for the java classes in the source packages. A number of metrics related to size, inheritance, coupling, cohesion and polymorphism are calculated at the package level and their correlation is found with the test metrics. The correlation is found to be quite significant, which helps in predicting the effect of source metrics on the test metrics and hence on testability. The results show that testability can be assessed through different kinds of metrics at the package level.

This study is motivated by a number of factors: (1) Software testability is one of most important attributes of the software quality. Hence assessing software testability will provide indications to improve the software quality. (2) Testability measurement helps in measuring software reliability, one of the critical aspects of the software (Yang *et al*., 1998). (3) Assessment of software testability helps in planning of testing activities as it provides information about the testing effort required to test the system (Mouchawrab *et al*., 2005). (4) Class testability has been the focus of many studies but empirical studies on software testability at the package level are rare.

Testability has been defined by a number of researchers with different viewpoints. Bache and Mullerburg (1990) define testability in terms of the effort required for testing. They measure it by the number of test cases required to satisfy a given coverage criterion. They focus on only the control flow based testing strategy and have used Fenton-Whitty Theory for measuring testability. Flow graphs are used to measure the testability.

Binder defines software testability as the relative ease and expense of revealing software faults (Binder, 1994). Reliability of a system is dependent on how testable a system is. Testability helps in reducing the cost of testing. Testability has two key factors: controllability and observability and the main concern of testability is to have more controllable input and observable output. Observability can be increased by making use of assertions. He states that the testability can be accessed through the use of software metrics. Binder claims that, the software testability is dependent upon six factors: (1) characteristics of requirements and, specification, (2) the process of software development, (3) the test suite, (4) built-in test capabilities, (5) the environment which supports testing, and (6) characteristics of the source code.

Voas and Miller (1995) define software testability as the probability that during testing the software will fail on its next execution if it contains faults. They make use of sensitivity analysis to predict the testability. In that, the software and its mutant versions are repeatedly executed and estimation is made about the likelihood of the mutants be detected. Depending upon this, more testing effort is applied on the parts of the software where there is low likelihood. Their technique depends upon the testing method used and also on the mutation testing method used for seeding the faults.

Bruntink and Deursen predict the testability of a class using two test metrics: (1) the number of test cases needed to test a class and (2) the effort required to develop each test case (Bruntink and Deursen, 2006). They have used five java systems for the experiments. Their technique finds a correlation between class level metrics (like LOC, NOA etc.) and the test metrics. The testability is assessed at the class testing level. The results show significant correlations with size related metrics but inheritance related metrics did not show any good correlation with test metrics. Such correlations are highly dependent upon the testing technique used and the testing criterion used in the systems used for the study.

Briand and Labiche describe a method in which software contracts (pre and post conditions of methods and class invariants) are used to improve the value of software testability (Briand *et al*., 2003). They claim that the software contracts increase the observability i.e., the probability that a fault will be detected upon the execution of test cases. A case study shows that using software contracts, a large number of failures can be detected as compared to the software using no contracts.

Mouchawrab et al. provides a generic framework for the testability of the object oriented software (Mouchawrab *et al*., 2005). A number of hypotheses are stated which explain, how an attribute can relate to testability and under what conditions. The software testability is described at the design level, before the start of coding. They claim that evaluation of testability at the analysis and design stage can reduce the testing cost. Design attributes, that affect the testability for each testing activity, are stated. For each attribute a set of measures is provided and model elements from UML models are identified that are required to evaluate these measures. The guidelines are provided on how the testability measurement helps in providing suggestions to change the design so that the testability can be improved.

Baudry *et al*. (2002) emphasized upon making use of the design artifacts for analyzing the testability. Mainly, they have used UML Class diagrams and state diagrams to analyze software testability. Their work aims at finding the parts of the software where complex interactions occur among objects and make the testing process difficult. A testing criterion is introduced which exercises the object interactions. Testing effort is estimated as number of object interactions in the UML class diagram. The number of object interactions also provides an estimate of the number of test cases required to test a system. The testability measurement proposed is derived from the design artifacts hence it would be different for the implementation of the system.

Jungmayr (2002) measuresd testability based on the dependencies between components. The number of dependencies increase the testing effort required to test the system. The metrics related to dependency are defined. The dependency information is used for possible refactoring that improves the testability. The level of testability investigated is the integration level.

ISO has defined software testability as: attributes of software that bear on the effort needed to validate the software product (ISO, 1991).This study follows this definition. The attributes of software that we consider are the source code metrics (For example LOC, NOA, NOM etc.) of object oriented software at the package level. The focus of this study is on the

source code factors only and the effect of source code factors on the testing effort is investigated. The testability is assessed through the testing effort.

A number of empirical studies (Yogesh and Goel, 2008; Chae *et al*., 2007) have been done to predict and assess various attributes of software quality like maintainability, fault proneness etc. These studies make use of differ rent statistical methods or neural networks to predict the software quality attributes. Software testability is a software quality attribute which is a part of software maintainability attribute. The similarity between these studies and our study is that we also use various metrics and statistical method to assess software testability.

JUNIT TESTING FRAMEWORK

JUnit is an open source framework which has been designed for the purpose of writing and running tests in the Java programming language. It was originally written by Beck and Gamma (1998). JUnit has gained a lot of popularity as a part of test driven development (Beck, 2002), agile software development (Cockburn, 2002) and extreme programming methodology (Beck, 1999). JUnit defines how to write the test cases and provides the tool to run them. JUnit helps in testing a java class by writing the corresponding JUnit class. A number of other frameworks exist for class testing in other languages like CPPUnit (for C++) etc. JUnit is described here as it is being used in this study. An example java class (calculator. java) and its JUnit test class (calculator Test.java) is shown in Fig. 1 and 2, respectively.

To test our calculator.java class we define calculator Test.java as a subclass of TestCase. The JUnit test class contains various test methods to test the methods of the source class. A JUnit test method does not contain any parameters. In writing the test cases in a JUnit class a number of asserts are used. For example assertEquals (expected output, actual output), assert Equals (message, expected output, actual output), etc. The calculatorTest class in Fig. 2 uses asser Equals() to compare the expected and actual output.

```
public class calculator {
public calculator() { }
public int add(int a, int b)
{
return a+b;
}
public int subtract(int a, int b)
{
return a-b;
} }
```

Fig. 1: Calculator. java class

```
import junit.framework.TestCase;
public class calculatorTest extends TestCase
{
private calculator cal = new calculator();
private int a = 4;
private int b = 3;
public void testAdd() {
assertEquals(7, cal.add(a,b)); }
public void testSubtract() {
assertEquals(1, cal.subtract(a,b));}}
```

Fig. 2: CalculatorTest.java class

If they are equal the test case passes otherwise test case fails. In Java, classes are contained in packages. The source class and test class can be kept in the same or different packages. In this example calculator is the source class and calculatorTest is the test class. There can be a number of source classes and a number of test classes corresponding to these source classes in a given system. In the source system used for this study, the source classes are kept in the source package and test classes are kept in the test packages. In Eclipse project there are different source packages and they have corresponding test packages.

EXPERIMENTAL DESIGN

The aim of this study is to evaluate whether source code metrics can be used to assess the testability of a package or not. This study was conducted in December 2009 at University school of Information Technology. In this section, we identify the object oriented metrics under consideration and formulate a hypothesis to be analyzed. Subsequently, we identify test metrics based on JUnit test classes.

Object-Oriented Metrics

Object oriented metrics chosen in this work are given in Table 1; these metrics are calculated at the class level and then calculated at the package level for analysis. These metrics can be divided into 5 categories viz. size, coupling, cohesion, inheritance and polymorphism. The source metrics are defined in the appendix.

Test Metrics

The four test metrics used for this study are: TLOC (Lines of Code for Test Class), TM (Number of Test Methods), TA (Number of asserts) and NTClass (number of test classes per test package). These metrics are calculated from the JUnit test classes of the test packages of Eclipse source code. The first three metrics are class level metrics. They are

Table 1: Metrics calculated in the Eclipse Data Set

Level	Metric name	Source	Package level
Class	Line of code per class (LOC)	Henderson-Sellers (1996)	Max of class level metric. Total of class level metric.
	No. of Attributes per Class (NOA)	Henderson-Sellers (1996)	
	No. of Methods per Class (NOM)	Henderson-Sellers (1996)	
	Weighted Methods per Class (WMC)	Chidamber and Kemerer (1994)	
	Response for Class (RFC)	Chidamber and Kemerer (1994)	
	Coupling between Objects (CBO)	Chidamber and Kemerer (1994)	
	Data Abstraction Coupling (DAC)	Henderson-Sellers (1996)	
	Message Passing Coupling (MPC)	Henderson-Sellers, 1996)	
	Tight Class Cohesion (TCC)	Briand *et al.* (1991)	
	Information based Cohesion (ICH)	Lee *et al.* (1995)	
	Lack of Cohesion (LCOM)	Chidamber and Kemerer (1994)	
	Depth of Inheritance (DIT)	Chidamber and Kemerer (1994)	
	No. of Children (NOC)	Chidamber and Kemerer (1994)	
	No. of Methods Overridden by a subclass (NMO)	Henderson-Sellers (1996)	
Class	Test Lines of code per Class (TLOC)	(Bruntink and Deursen, 2006)	Total of class level metric.
	Test methods per class (TM)		
	Test Cases per class (TA)	(Bruntink and Deursen, 2006)	
Package	NClass (No. of Classes)		Value
	NTClass (No. of Test Classes)		

calculated at the package level for this study. TLOC and TA metrics are proposed by Brutink and Deursen (2006). We proposed a new test metric TM. All the test metrics are defined below:

TLOC (Test Lines of Code) Metric

It is defined as the number of lines of code (non comment and non blank) in a JUnit test class. In Fig. 2, calculatorTest class has TLOC = 10. Since the total lines of code for calculatorTest class is 12.

TM (Number of Test Methods) Metric

The TM metric is defined as the number of test methods in the Junit test class. In Fig. 2 there are two test methods in calculatorTest class: testAdd and testSubtract. Hence for calculatorTest class TM = 2.

TA (Number of Asserts) Metric

It is defined as the number of asserts in the test class. In Fig. 2 there are two asserts in the calculatorTest class. Hence for this class the value of TA = 2.

NTClass (Number of Test Classes)

It is defined as the number of test classes in a given test package.

The above three test metrics (TLOC, TM, and TA) are first calculated at the class level and then calculated at the package level. Eclipse source code contains test packages corresponding to source java packages.

Goal and Hypotheses

This study evaluates the object oriented metrics using the framework proposed by Basili (Briand *et al*., 2002). The goal, perspective and environment for this study are described below:

Goal

To judge the capability of the object oriented metrics to assess the testability of a package.

Perspective

The testing level considered is package level. Thus we judge if the object oriented metrics can assess the testability at the package level.

Environment

This study uses eclipse packages as the source system, which is written in java and tested at the package level using JUnit testing framework. JUnit framework allows users to create a test class for every java class. The source code of Eclipse is executed using Eclipse IDE (http://www.eclipse.org).

We formulate the following hypothesis that would be tested by this study:

- **H_0(c,t):** There is no correlation between object oriented metric c and test metric t for a package
- **H_1(c,t):** There is a correlation between object oriented metric c and test metric t for a package

Where c belongs to a set of object oriented metrics and t belongs to the set of test metrics for a package.

Statistical Analysis

In order to evaluate the above hypotheses, we calculate Spearman's rank-order correlation coefficient (rs), for each object oriented metric of eclipse packages and four test metrics of the corresponding packages. Spearman's rank-order correlation between object-oriented metric c and test metric t is denoted by rs(c, t). Spearman's rank-order correlation coefficient measures the association between two variables, measured in an ordinal scale (Siegel and Castellan, 1988). This kind of correlation is used because it does not depend upon the underlying data distribution. The value of rs ranges from -1 to 1. A value of 1 indicates a perfect positive correlation and -1 indicates a perfect negative correlation. A value of 0 indicates that there is no correlation. To calculate rs, there should be corresponding test metrics calculated for each source package. Each eclipse source package contains a number of java classes and the corresponding test package contains a number of test classes. Spearman's rank-order correlation coefficient, rs is calculated for each object oriented metric c and all the test metrics, t. The values for each object oriented metric for all the eclipse packages is calculated using the Eclipse plugin which is developed and then each value is paired with the value of test metrics of the corresponding test package. The resulting pairs are used to calculate the value of the correlation.

EMPIRICAL DATA COLLECTION

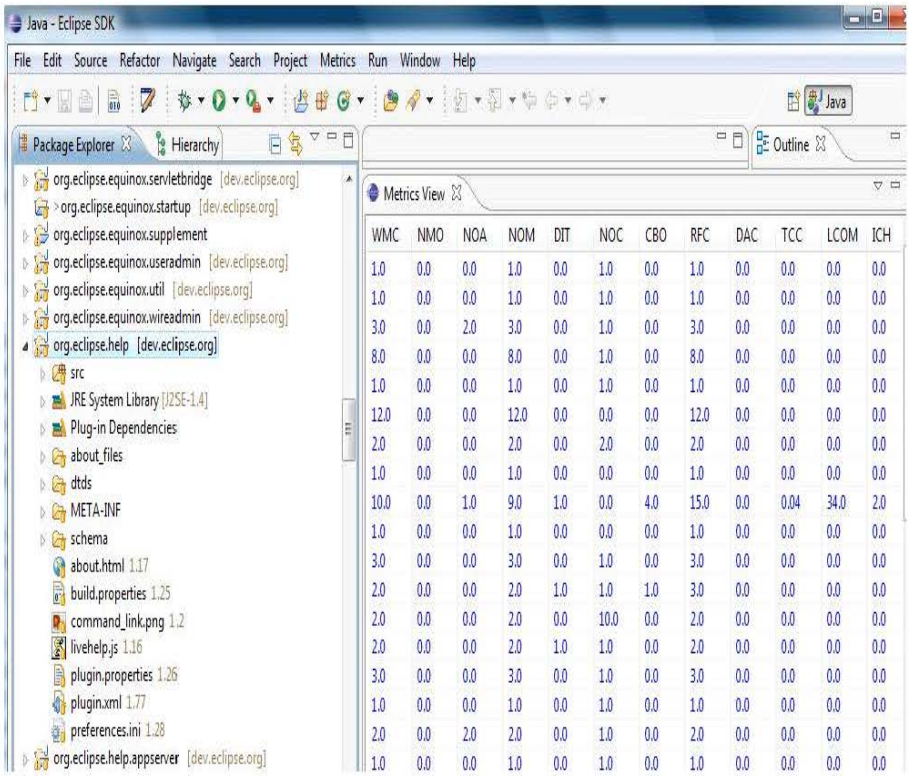
This study has used the source code of Eclipse (version 3.0), which is available as an open source system. The Eclipse (http://www.eclipse.org) Project is an open source project which provides a robust and freely available industry platform for developing the highly integrated tools. The Eclipse Project is made from the following Projects:

- Platform: The platform upon which all other Eclipse based tools are built
- JDT (The Java development tooling) also called Java IDE
- PDE (Plug-in development environment)

The Platform Project is further divided into various components, which are Ant, Compare, Core, Debug, Doc, Help, Releng, Scripting, Search, SWT, Text, UI, Update, VCM, and WebDAV. The JDT Project is divided into the following components: JDT Core, JDT Doc, JDT UI and JDT Debug. The PDE Project has following three components: PDE Build, PDE Doc, and PDE UI. These Projects are managed by a group called the Project Management Committee (the PMC). Eclipse is tested using functional testing at the package level by means of JUnit. The size measurements, of the source system are given in the Table 2. In order to collect data from the Eclipse system, we have used the Eclipse platform (http://www.eclipse.org). We have developed an Eclipse plugin to calculate the set of metrics used for this study. Figure 3 shows the calculated metrics using this plugin. In the Fig. 3, the metric values are shown for the package org.eclipse.help. The shown metric values are class level values. These are calculated for all the 25 packages. Corresponding to all the packages there are test packages. Foe example org.eclipse.help has org.eclipse.help.tests as its test package. To calculate the package level metrics we take the sum of the metrics at the class level.

Table 2: Size measurements of the packages of Eclipse

Source system	No. of packages used	Total java classes	Total test classes
Eclipse	25	4420	1564



WMC	NMO	NOA	NOM	DIT	NOC	CBO	RFC	DAC	TCC	LCOM	ICH
1.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0
3.0	0.0	2.0	3.0	0.0	1.0	0.0	3.0	0.0	0.0	0.0	0.0
8.0	0.0	0.0	8.0	0.0	1.0	0.0	8.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0
12.0	0.0	0.0	12.0	0.0	0.0	0.0	12.0	0.0	0.0	0.0	0.0
2.0	0.0	0.0	2.0	0.0	2.0	0.0	2.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
10.0	0.0	1.0	9.0	1.0	0.0	4.0	15.0	0.0	0.04	34.0	2.0
1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
3.0	0.0	0.0	3.0	0.0	1.0	0.0	3.0	0.0	0.0	0.0	0.0
2.0	0.0	0.0	2.0	1.0	1.0	1.0	3.0	0.0	0.0	0.0	0.0
2.0	0.0	0.0	2.0	0.0	10.0	0.0	2.0	0.0	0.0	0.0	0.0
2.0	0.0	0.0	2.0	1.0	1.0	0.0	2.0	0.0	0.0	0.0	0.0
3.0	0.0	0.0	3.0	0.0	1.0	0.0	3.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0
2.0	0.0	2.0	2.0	0.0	1.0	0.0	2.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0

Fig. 3: Metric calculation through metrics plugin

ANALYSIS

The results of this study are given in Table 3-5, the Tables contain the values of Spearman's rank order correlation coefficient (rs) between object oriented metrics and test metrics. This correlation allows us to accept hypotheses H1(c, t) and reject hypotheses H0(c, t). The software used in this study, is SPSS 16. (www.spss.com).

The impact of source code metrics on testability of object oriented systems is explained through the correlation values. The very first observation is that, all the source code metrics are highly correlated amongst themselves as shown in Table 4. Secondly, there is a high correlation amongst test metrics as shown in Table 5. A detailed discussion of the correlation between the source code metrics and test metrics is given below.

Size Related Metrics

The size related metrics for a package are: LOC, NOA, NOM, WMC and NSClass. Intuitively, a package with more number of classes and more lines of code should require more testing effort, the values in Table 3 show that all the five size metrics are correlated with all the four test metrics.

LOC

The size of a package is a significant factor in the determination of the testing effort of a package. A large package will require large number of test classes (i.e., high NTClass). More the number of test classes, more is the number of test methods and test cases. Hence high LOC leads to high testing effort and same is reflected by the correlation values in Table 3. LOC is significantly correlated to all the test metrics. Hence high LOC increases the testing effort and decreases the testability.

NOA

There is a significant correlation between NOA and all the test metrics. Number of attributes increases the number of test cases as more effort is required to initialize all the attributes for testing. High NOA leads to lower testability.

NOM

NOM is positively correlated to all the test metrics. High NOM leads to high test methods(TM), hence high TA and higher TLOC. Hence high NOM leads to lower testability.

Table 3: Correlation of source metrics (sum) with test metrics for Eclipse

	LOC	NOA	NOM	WMC	LCOM	ICH	TCC	CBO	RFC	DAC	MPC	DIT	NOC	NMO	NSClass
LOC	1														
NOA	0.965	1													
NOM	0.978	0.942	1												
WMC	0.993	0.951	0.990	1											
LCOM	0.801	0.814	0.842	0.837	1										
ICH	0.858	0.815	0.865	0.883	0.888	1									
TCC	0.892	0.902	0.910	0.896	0.772	0.837	1								
CBO	0.866	0.882	0.853	0.862	0.819	0.896	0.919	1							
RFC	0.952	0.942	0.978	0.969	0.875	0.901	0.946	0.913	1						
DAC	0.917	0.920	0.900	0.901	0.734	0.778	0.922	0.869	0.901	1					
MPC	0.847	0.863	0.853	0.858	0.871	0.912	0.900	0.972	0.919	0.844	1				
DIT	0.933	0.919	0.938	0.925	0.677	0.751	0.909	0.811	0.904	0.900	0.770	1			
NOC	0.901	0.887	0.923	0.911	0.793	0.835	0.954	0.909	0.957	0.911	0.907	0.878	1		
NMO	0.854	0.784	0.860	0.869	0.753	0.834	0.797	0.736	0.820	0.739	0.712	0.824	0.716	1	
NSClass	0.925	0.928	0.945	0.919	0.779	0.806	0.925	0.858	0.934	0.946	0.805	0.930	0.914	0.786	1

Table 4: Correlation of source metrics for Eclipse

Sum	TLOC	TM	TA	NTClass
LOC	0.515	0.408	0.439	0.440
NOA	0.502	0.425	0.478	0.432
NOM	0.566	0.465	0.505	0.453
WMC	0.548	0.450	0.482	0.448
LCOM	0.602	0.494	0.586	0.448
ICH	0.601	0.457	0.486	0.540
MPC	0.535	0.455	0.496	0.554
TCC	0.586	0.508	0.495	0.641
CBO	0.500	0.421	0.447	0.586
RFC	0.597	0.504	0.531	0.533
DAC	0.419	0.327	0.352	0.478
DIT	0.485	0.428	0.458	0.450
NOC	0.511	0.448	0.447	0.528
NMO	0.631	0.553	0.557	0.505

Table 5: Correlation of test metrics for Eclipse

	TLOC	TM	TA	NTClass
TLOC	1			
TM	0.874	1		
TA	0.843	0.924	1	
NTClass	0.890	0.771	0.723	1

WMC

WMC is positively correlated to all the test metrics. Hence higher WMC leads to lower testability. The reason is same as for NOM.

Cohesion Metrics

Lack of Cohesion of Methods (LCOM)

LCOM metric is positively correlated with all the test metrics it implies that test effort increases with decrease in cohesion. The results show that LCOM is significantly correlated to all the test metrics. Hence testing effort increases with high LCOM value and leads to lower testability.

Information Based Cohesion (ICH)

By definition, ICH metric suggests that cohesion increases the testing effort. The results show that this metric is significantly correlated to all the test metrics. Hence testing effort increases with increase in cohesion and testability is decreased.

Tight Class Cohesion (TCC)

TCC is significantly correlated to all the test metrics. Hence high TCC increases the testing effort and lowers testability.

Coupling Metrics

Coupling Between Objects (CBO)

Theoretically, increase in CBO should increase the testing effort. The same is shown by the figures in Table 3; there is a significant correlation between CBO and all the test metrics. Hence increase in CBO increases the testing effort and lowers the testability.

Data Abstraction Coupling (DAC)

The amount of testing increases with increase in DAC. DAC is correlated to all the test metrics. Hence increase in DAC increases the testing effort and lowers the testability.

Message Passing Coupling (MPC)

MPC is significantly correlated to all the test metrics, which means that MPC can also predict the testing effort. We can say that as MPC increases, testing effort increases and testability decreases.

Response for Class (RFC)

RFC is significantly correlated to all the test metrics, which means that RFC can predict the testing effort. We can say that as RFC increases, the testing effort increases and the testability decreases.

From the correlation values of all the four coupling metrics (CBO, RFC, DAC and MPC) we can conclude that as coupling increases the testing effort increases and it lowers the testability of the system at the package level.

Inheritance Related Metrics

This Third Category of Metrics Deal with Inheritance

DIT and NOC. DIT measures the parent classes and NOC measures the child classes.

DIT

We find a significant correlation between DIT and test metrics. Hence as the depth of inheritance increases the testing effort increases and testability decreases.

NOC

NOC has a significant correlation with all the test metrics. The reason for correlation among NOC and test metrics is that a thorough testing of parent class is done with the increase in number of children, with an intention that if parent class is having a fault it does not creep into the child class.

The inheritance related metrics are positively correlated to all the test metrics based on the correlation values. Hence the property of inheritance increases the testing effort and decreases the testability.

Polymorphism Related Metrics

NMO

NMO is a metric which deals with the property of polymorphism. Theoretically polymorphism should increase the testing effort and hence decrease the testability. The correlation values show a significant correlation between NMO and all the test metrics. Hence an increase in NMO increases the testing effort and decreases the testability.

IMPLICATIONS OF RESULTS: PRACTICAL LESSONS LEARNT

The results of the study presented have several important implications. Many of these results should provide useful guidance to software practitioners to assess how hard it is to test their code at the package level. An increase in lines of code, number of attributes, number of methods and weighted methods per package and number of classes per package will have a strong increasing impact on testing effort and will thus result in low testability. In other words, the greater the lines of code per package, the number of attributes, the number of methods and weighted methods per package, the lower would be testability. As expected, an increase in coupling in classes is likely to increase the testing effort in the software. Since coupling increases the complexity of the software and reduces encapsulation. Hence for high testability the coupling should be low.

- Lack of cohesion in methods in packages is likely to increase the testing effort. On the other hand, cohesion in methods lowers the testing effort and increases the testability
- An increase in response for class is likely to increase the testing effort and thus decrease its testability. Thus an increase in response for classes is likely to increase the value of testing effort and lower the testability
- Logically it seems that an increase in NOC and DIT should increase the testing effort and thus decrease the testability. The same is shown by the results of this study
- There is statistically a significant positive relationship between test metrics and NMO (an indicator of polymorphism). This is true theoretically because increase in polymorphism increases the testing effort and lowers the testability. Since polymorphism increases the complexity of the software due to run time binding. The same is shown by the system under study

Threats to Validity

There are few limitations of this study which should be taken into account while interpreting the results. The packages considered for this study are the ones which have

corresponding test packages. Since, we considered the testability assessment at the package testing level. The packages which do not have corresponding test package are not considered. Those packages can provide additional information on testability at the package level.

- The source system used (Eclipse) has been tested using the functional testing. The testability assessment would be different if some other kind of testing technique is used
- This study is based on JUnit framework. Results may vary if some other framework is used

CONCLUSIONS AND FUTURE WORK

The main goal of the authors' study was to determine the interrelation between source code metrics and the test metrics, and find the effect of source code metrics on testing effort and on testability at the package level. The relationship between source java packages and their JUnit test packages in a large Java system (Eclipse) was analyzed. The correlation analysis between source code metrics and test metrics shows that there is a significant correlation between all the source code metrics and test metrics at the package level.

The results of correlation show that all the size metrics and coupling metrics are highly correlated to test metrics. It is shown that as the size of the software increases the testability decreases because testing effort increases. As the coupling grows, the testability decreases because of the increased testing effort.

Inheritance and polymorphism increase the testing effort and lower the testability. The same is shown by the correlation between source code metrics and test metrics.

The results show that ICH, TCC and LCOM are correlated to testing metrics. Hence high value of ICH, TCC and LCOM increases the testing effort and decreases the testability.

This work can be extended with the following future work. First, this study should be extended to a large number of systems, using different development methodologies like Test driven development, extreme programming and agile software development. Second, this study has been conducted at the package level. It should be extended to system level testing. Third, this study should be extended to a number of other source code metrics which deal with polymorphism, exception handling etc. Fourth, different statistical methods like multivariate analysis etc. should be used to find the relation between source code metrics and test metrics.

APPENDIX

Size Metrics

In this section four size metrics are discussed. These metrics measure the size of the system in terms of lines of code, attributes and methods included in the class. As these metrics capture the complexity of the class hence they can give an insight into the testability of the class.

Lines of Code per Class (LOC)

It counts the total number of lines of code (non-blank and non-comment lines) in the class.

Number of Attributes per Class (NOA)

It counts the total number of attributes defined in a class.

Number of Methods per Class (NOM)

It counts number of methods defined in a class.

Weighted Methods per Class (WMC)

The WMC is the count of the sum of the McCabe's Cyclomatic Complexity for all the methods in the class. If method complexity is one for all the methods, then WMC = n, the number of methods in the class.

Number of Source Classes per package (NSClass): Number of java classes in a given package.

Cohesion Metrics

Cohesion measures the degree to which the elements of a module are functionally related. A strongly cohesive module does little or no interaction with other modules and implements the functionality which is related to only one feature of the software. This study considers following three cohesion metrics.

Lack of Cohesion in Methods (LCOM)

Lack of Cohesion (LCOM) measures the cohesiveness of the class. It is defined as below:

Let M be the set of methods and A be the set of attributes defined in the class. M_a is the number of methods that access a. Mean be the mean of M_a over A.Then,

$$LCOM = (Mean - |M|)/(1 - |M|)$$

Information Flow Based Cohesion (ICH)

ICH for a class is defined as the number of invocations of other methods of the same class, weighted by the number of parameters of the invoked method.

Tight Class Cohesion (TCC)

The measure TCC is defined as the percentage of pairs of public methods of the class with common attribute usage.

Coupling Metrics

Coupling relations increase complexity and reduce encapsulation.

Coupling Between Objects (CBO)

CBO for a class is the count of the number of other classes to which it is coupled. Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.

Data Abstraction Coupling (DAC)

Data Abstraction is a technique of creating new data types suited for an application to be programmed. Data Abstraction Coupling (DAC) is defined as the number of ADTs defined in a class.

Message Passing Coupling (MPC)

Message Passing Coupling (MPC) is defined as the number of send statements defined in a class. So if two different methods in class C access the same method in class D, then MPC = 2.

Response for a Class (RFC)

The response for a class (RFC) is defined as the set of methods that can be executed in response to a message received by an object of that class.

Inheritance Metrics

This section discusses two different inheritance metrics, which are considered for this study.

Depth of Inheritance Tree (DIT): The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes.

Number of Children (NOC)

The NOC is the number of immediate children of a class in the class hierarchy.

Polymorphism Metrics

Polymorphism is the characteristic of object oriented software through which the implementation of a given operation depends on the object that contains the operation.

Number of Methods Overridden by a Subclass (NMO)

When a method in a child class has the same name and signature as in its parent class, then the method in the parent class is said to be overridden by the method in the child class.

REFERENCES

Ananya, K. and S. Bhattacharya, 2004. Static analysis of object oriented systems using extended control flow graph. Proceedings of the 10th IEEE Region Conference TENCON, Nov. 21-24, Kolkata, India, pp: 310-313.

Bache, R. and M. Mullerburg, 1990. Measures of testability as a basis for quality assurance. Software Eng. J., 5: 86-92.

Baudry, B., Y. le Traon and G. Suny, 2002. Testability analysis of a uml class diagram. Proceedings of the 8th International Symposium on Software Metrics, June 4-7, IEEE Computer Society Washington, DC., USA., pp: 54-63.

Beck, K. and E. Gamma, 1998. Test infected: Programmers love writing tests. Java Rep., 3: 51-56.

Beck, K., 1999. Extreme Programming Explained: Embrace Change. Addison-Wesley, Boston.

Beck, K., 2002. Test Driven Development: By Example. The Addison-Wesley, London, ISBN-10: 0321146530.

Beizer, B., 1990. Software Testing Techniques. 2nd Edn., Van Nostrand Reinhold, New York, ISBN: 0-442-20672-0, pp: 550.

Binder, R.V., 1994. Design for testability in object-oriented systems. Commun. ACM, 37: 87-101.

Briand, L., S. Morasca and V. Basili, 2002. An operational process for goal-driven definition of measures. IEEE Trans. Software Eng., 28: 1106-1125.

Briand, L., W. Daly and J. Wust, 1991. A unified framework for coupling measurement in object-oriented systems. IEEE Trans. Software Eng., 25: 91-121.

Briand, L.C., Y. Labiche and H. Sun, 2003. Investigating the use of analysis contracts to improve the testability of object-oriented code. Software Practic Experience, 33: 637-672.

Bruntink, M. and A.V. Deursen, 2006. An empirical study into class testability. J. Syst. Software, 79: 1219-1232.

Chae, H.S., T.Y.K. Woo, S. Jung and J.S. Lee, 2007. Using metrics for estimating maintainability of web applications: An empirical study. Proceedings of the 6th IEEE/ACIS International Conference on Computer and Information Science, July 11-13, Melbourne, Australia, pp: 1053-1059.

Chidamber, S.R. and C.K. Kemerer, 1994. A metrics suite for object oriented design. IEEE Trans. Software Eng., 20: 476-493.

Cockburn, A., 2002. Agile Software Development. Addison-Wesley, Reading, MA USA.

Henderson-Sellers, B., 1996. Object-Oriented Metrics. Prentice Hall, New Jersey, USA.

IEEE Press, 1990. IEEE Standard Glossary of Software Engineering Technology. ANSI/IEEE Standard, Washington, DC, USA.

ISO, International Standard ISO/IEC 9126, 1991. Information technology: Software product evaluation: Quality characteristics and guidelines for their use. http://www.techstreet.com/cgi-bin/detail?doc_no=ISO_IEC%7C9126_1991&product_id=863150.

Jungmayr, S., 2002. Identifying Test-critical dependencies. Proceedings of the 18th IEEE International Conference on Software Maintenance, Oct. 3-6, Montreal, Quebec, Canada, pp: 404-413.

Kolb, R. and D. Muthig, 2006. Making testing product lines more efficient by improving the testability of product line architectures. Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis, (ROSATEA'06), ACM Press, pp: 22-27.

Lee, Y.S., B.S. Liang, S.F. Wu and F.J. Wang, 1995. Measuring the coupling and cohesion of an object-oriented program based on information flow. Proceedings of the International Conference on Software Quality, Nov. 6-9, Maribor, Slovenia, pp: 81-90.

Liu, C.H., D.C. Kung, P. Hsia and C.T. Hsu, 2001. Object-based data flow testing approach for web applications. Int. J. Software Eng. Knowl. Eng., 11: 157-180.

Mouchawrab, S., L.C. Briand and Y. Labiche, 2005. A measurement framework for object-oriented software testability. Inform. Software Technol., 47: 979-997.

Siegel, S. and N.J. Castellan, 1988. Non Parametric Statistics for the Behavioral Sciences. 2nd Edn., McGraw-Hill, New York, pp: 399.

Voas, J.M. and K.W. Miller, 1994. An empirical comparison of a dynamic software testability metric to static cyclomatic complexity. Proceedings of the 2nd International Conference on Software Quality Management. July 1994, IEEE CS Press, Edinburgh, Scotland, pp: 1-11.

Voas, J.M. and K.W. Miller, 1995. Software testability: The new verification. IEEE Software, 12: 17-28.

Yang, M.C.K., W.E. Wong and A. Pasquin, 1998. Applying testability to reliability estimation. Proceedings of the 9th International Symposium on Software Reliability Engineering, Nov. 4-7, Paderborn, Germany, pp: 90-90.

Ying, J., S.S. Hou, J.H. Shan, L. zhang and B. Xie, 2008. An approach to testing black-box components using contract-based mutation. J. Software Eng. Knowl. Eng., 18: 93-117.

Yogesh, S. and B. Goel, 2008. An integrated model to predict fault proneness using neural networks. Software Qual. Profess., 10: 22-32.