Survey on Software Test Design Metrics

Raj Kumari and Jaspreet Kaur

Abstract*— Measurement has always been an integral part of software engineering. Software projects can be assessed and risks reduced by using software metrics. In this paper we discuss software test metrics and their impact on software testing. This paper focuses on software test design metrics its key role in software testing process along with classification and analysis of various test metrics.

***Index Terms*—Software testing,Sofware metrics,software test design metrics.**

I. Introduction

Computer software is an important element in growth of social-economic development which requires new techniques and strategies.Software application‘s demand for quality has grown.Hence testing becomes one of the indispensable components of software development which is the indicator of quality [1].

—Testing is questioning a product in order to evaluate it".
------James Bach [2]

Software testing is a crucial element in the SDLC and can deliver excellent results if executed accurately and efficiently [3] and software measurement can play a key role in increasing the effectiveness of testing process. Software metrics are used to evaluate the software development process and the quality of the resultant product [4].

This paper investigates metrics from the view point of unit test case designing. In this our units consist of the classes, the smallest testable unit. The approach used is to evaluate a set of metrics and predict the testing effort of those metrics.The basic reason for choosing this approach was that software metrics can efficiently investigate various aspects of software.For better results an operation has to be tested as part of class not in isolation. Thus we consider the unit testability with respect to various test design metrics [5].

II. Software test design metrics

Software Metric is generally used to describe a dimension of a particular attribute of a software project. The Software Metrics that the QA team produces are concerned with the test activities that are part of the Test Phase and so are formally known as software test metrics. Figure1 shows various test measures and resulting test metrics.

Raj Kumari is with the University Institute of Engineering and Technology,PanjabUniversity,Chandigarh,India (email:rajkumari_bhatia5@yahoo.com).
Jaspreet kaur is student at University Institute of Engineering and Technology,PanjabUniversity,Chandigarh,India (email:jsprtkaur15@gmail.com)

Software metrics can be broadly classified into two types software product metrics and software process metrics. Figure 2[17] shows the categories of software testing metrics hierarchy.

TEST MEASURES & METRICS

Measure	Metrics
Test Progress	* Tracking testing progress * Tracking testing defect * Backlog * Staff productivity
Test Quality	* Test process efficiency * Test productivity
Cost of testing	* Direct cost * Indirect cost
Test effectiveness	* Residual Defect Density * Defect Distribution (severity) * Defect Rejection

Figure 1

Test process metrics provide information about preparation for testing, test execution and test progress. They are mainly used in measuring progress of the Test Phase but don‘t provide any information regarding the test state. Process metrics describe the effectiveness, quality and efficiency of the processes that produce the software product. For example, effort required in the process, total time taken to produce the product, defect removal efficiency throughout development, no. of defects found at some stage in testing, no. of defects removed maturity of the process [6]. Some Test process metrics are:

(i.) Number of test cases designed.
(ii.) Number of test cases executed.
(iii.) % of test cases executed.
(iv.) % of test cases passed.
(v.) % of test cases failed.
(vi.) Average execution time of a test case.

Test product metrics present information about the test state as well as testing status of a software product and are generated by test execution and code fixes or deferment. Using these metrics we can measure the products test state and indicate the quality level, valuable for product release decisions. Product metrics help in describing the characteristics of the product such as size, complexity, design features, scalability, efficiency, reliability, portability and most importantly testability [7].

Some Test product metrics are:

(i.) Estimated time for testing.
(ii.) Actual testing time.
(iii.) Average time interval between failures.
(iv.) Average number of failures experienced in time intervals.
(v.) Time remaining to complete the testing.

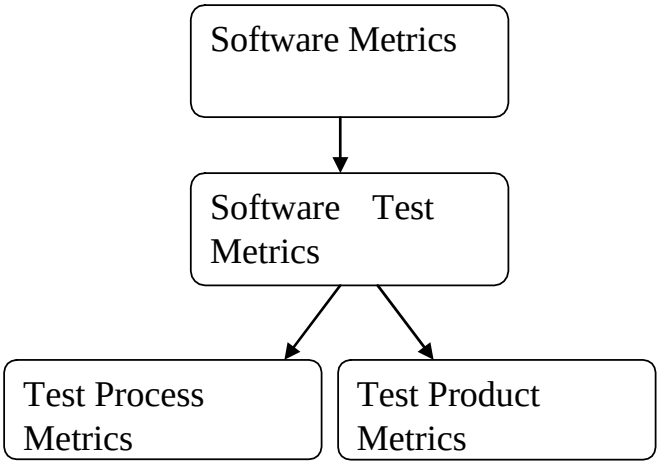

Figure 2

III. Characteristics of usable metrics

Metrics should not be collected because they are prescribed in the literature or because they are recognized as popular in some companies, but because they are helpful in taking decisions on a project in particular or within a given organization [8]. A useful metric is precisely defined (i.e., measurable or quantifiable), It also helps indicate whether an organization is achieving software goals [9]. There are several fundamental characteristics associated with useful software metrics. The useful software metrics should be:

- Simple and easy to understand.
- Measurable
- Economical.
- Metrics must be timely.
- Robust.
- Reliable
- Valid.
- Consistent and used over time.
- Unobtrusively collected.
- Independent.
- Accountable.
- Precise.

Useful metrics must be accompanied by data that is correct (correct according to the rules of the definition of the metric), accurate, exact and consistent (no large difference in the value occur, even if the person or measuring device changes). The measurement process should be clearly described clearly enough for someone else to be able to repeat the measurement Units[17].

IV. Evaluation of test design metrics

This evaluation is aimed at determining the role of test design metrics in predicting the effort required to implement the design and the quality of the code produced that is to determine total testing effort required to test based upon various metrics.
A typical empirical validation of test design object-oriented metrics is done by investigating the relationship between each metric and the outcome of interest. The results for different metrics are studied. The metrics RFC, CBO, and LCOM were defined in [10] and the NMA metric was defined in [11]. They have revealed that size can have an important perplexing result on the validity of object-oriented metrics [12].
The evaluation of test metrics is the core topic of this research. In this research we have defined our set of metrics, and set up the experiments to evaluate them. Software engineering rarely involves empirical analysis.
First, we state the objective of our experiments:
Objective: To assess the capability of the proposed source-based metrics (CBO,RFC,NOC,NOA and DITC) in predicting the testing effort.
Point of view: We are assessing that whether or not the degree of testing effort required for the class values can be predicted on the basis of the above mentioned source based metrics [5].

V. Testability

IEEE defines 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―[13]. In the unit testing of object oriented system, the testing for classes brings in some issues one of which is that a class cannot be tested directly, only an instance of it can be tested and the second one is that when an object is considered in an object oriented system, the state related with that object also influences the path of execution and methods of a class can communicate among themselves through this state. Thus we took into consideration the unit testability of the object oriented system with respect to the test case design for unit testing [5].
Encapsulation of attributes and operations make testing difficult as for testing the solid and abstract state of object is -required. No-doubt code reuse has been achieved through the use of inheritance but it poses further usage requirement on retesting[14], so does multiple inheritance in which further the testing is further complicated by increasing the number of contexts for which testing is required. How the test cases are applied within super class and subclass also needs to be considered with care [5],[14],[15].
The realm of object orientation lays emphasis on the encapsulation of information and implementation of operations performed on the information. Coupling provides us an evaluation of strength of association established by a connection between object classes to which a class is coupled. It is measured by calculating the number of distinct non

inheritance related class hierarchies on which a class depends[5],[10],[14],[15].The larger the number of couples the higher will be the sensitivity to change and errors in other parts of design and make testing difficult. This would increase the testing effort (TE) and decrease the testability. Therefore, we say that testability is inversely proportional to CBO.

$TE \propto CBO$

$ITb \propto 1/CBO$ (1)

Estimating the total CBO (TCBO) over all classes (i=1 to n), the sum is divided by two because the same relationship will be counted twice, when the two coupled classes are considered independently. Therefore we have:

$$TCBO = \frac{1}{2} \times \sum_{i=1}^{n} CBO \quad (2)$$

Now, if we consider the combination of the complexity of a class through the number of methods and the amount of communication with other classes. It was found that the complexity of the class increases with number of methods that can be invoked from a class through messages. Larger number of methods that can be invoked in response to a message, the more complicated the testing is, which in turn decreases the testability.
Using the metric, response for a class (RFC) which is defined as the number of methods in response set [5],[10],[14],[15], we say that the testing effort (TE) is directly proportional to RFC and hence testability is inversely proportional to it.

$TE \propto RFC$

$ITb \propto 1/RFC$ (3)

Estimating total RFC (TRFC) over all classes (i=1 to n) we get:

$$TRFC = \sum_{i=1}^{n} RFC \quad (4)$$

Since a class is a set of objects that have common properties (i.e methods and instance variables), an abstraction of the application domain is prepared/developed by arranging classes in a hierarchy which is formed due to inheritance between classes.This leads to super class accumulating all or desired common features of the subclass.

A class is composed of attributes and methods. In this proposal the Depth of Inheritance Tree of a Class (DITC) metric for class inheritance hierarchy is measured in terms of sum of the attributes (Private, Protected, public and inherited) and Methods (Private, Protected, public and inherited) at each level. The DITC metric of a class is calculated as:

$$DITC(C) = \sum_{i=1}^{L} LEV(i) * i$$

Where,

LEV_i = Attribute (C_i) + Method (C_i)

C_i = A class in the i_{th} level of class inheritance hierarchy.
Attribute (C_i) = Count the total number of protected, private, public and inherited attributes within a class in the class inheritance hierarchy at each level.
Method (C_i) = Count the total number of protected, private, public and inherited methods within a class in the class inheritance hierarchy at each level.
L = Total height in the class inheritance hierarchy i.e. the maximum distance from the last node (last level in the class inheritance hierarchy) to the root node (first level in the class inheritance hierarchy), ignoring any shorter paths in case of multiple inheritance is used. The metric depth of the inheritance that measure the depth of the class within the inheritance hierarchy is defined as ―the maximum distance from the node to the root of the tree‖ [10],[14].

It shows that the deeper a class is within the hierarchy, the more the number of methods it is will inherit. Thus making it more complex to predict its behavior. Deeper trees involved more methods and classes increasing the design complexity. This increases the testing effort and decrease the testability. This leads to testability being inversely proportional to DITC [16].

$TE \propto DITC$

$ITb \propto 1/DITC$ (5)

Estimating total DITC (TDITC) over all classes (i=1 to n), we get:

$$TDITC = \sum_{i=1}^{n} DITC \quad (6)$$

Chidamber and Kemerer proposed the Number Of Children of a class as the NOC metric for the class, which is the number of immediate subclasses subordinate to a class in the class hierarchy [10]. NOC is a measure that the methods of parent class are to be inherited by how many sublasses, the greater the value the greater will be the potential for reus. The greater the number of children of a class, the greater is the probability of improper abstraction of that class. NOC gives an approximate idea of the potential influence a class has on the overall design. It is given as the Number of descendents of the class. As number of descendents increase, the Effort of testing (TE) of methods of that class increases. This Decreases the testability providing an inverse relationship:

$$TE \propto NOC$$

$$ITb \propto 1/NOC \quad (7)$$

Estimating total NOC (TNOC) over all classes (i=1 to n), We get :

$$TNOC = \sum_{i=1}^{n} NOC \quad (8)$$

The *Number Of Attributes* [18] metric is used to calculate the average number of attributes a class contains in the model. This is useful in identifying the following probable problems:

(a) A class with numerous attributes may signify the existence of coincidental cohesion and necessitate additional decomposition, to handle the complexity of the model.

(b) In case of no attributes serious consideration should be given to the semantics of the class. This may possibly be a class utility rather than a class.

Considering a class, this is a simple count of the number of attributes**.**If the number of attributes are high (> 10) it is an indication of poor design, particularly insufficient decomposition, specially if this is coupled with an equally high number of methods. Classes without any attributes are particular cases and are not essentially anomalies. For example these can be interface classes, and must be checked.[17] Therefore as the NOA increases the effort of testing (TE) of methods associated with those attributes increases. Thereby decreasing the testability:

$$TE \propto NOA$$

$$ITb \propto 1/NOA \quad (9)$$

Estimating total NOA (TNOA) over all classes (i=1 to n), we get :

$$TNOA = \sum_{i=1}^{n} NOA \quad (10)$$

Equations (1,3,5,7,9) we get the testability with respect to a class :

$$ITb \propto (1/CBO) \times (1/RFC) \times (1/DITC) \times (1/NOC) \times (1/NOA) \quad (11)$$

$$ITb = \frac{k}{(CBO \times RFC \times DITC \times NOC \times NOA)} \quad (12)$$

Here ‗k' is the proportionality constant and the other values(CBO,RFC,DITC,NOC,NOA) are defined above.

By equations (2,4,6,8,10) the total interface testability (TITb) of the object oriented software over all classes (i=1 to n) can be given as

$$TITb = \frac{k}{(TCBO \times TRFC \times TDITC \times TNOC \times TNOA)} \quad (13)$$

The value of ‗k' will depend on characteristics related to software processes and experience of developer, type of tool available for the development of the unit as we are dealing with unit testing. The value will have to be worked out by specific software teams of concerned organization.

VI. Conclusion

These test design metrics explore the test case design and its testability. The results have shown us that the test design metrics are useful in measuring testability and the effort of testing. Particularly, the results allow for explanations of the CBO, RFC, DITC, NOC and NOA metrics in terms of test case construction factors. To wind up the results will help us to advance the set of metrics and the development approach so that we can increase testability and reduce the testing effort.

References

[1] John A. Fodeh and Niels B. Svendsen, Release Metrics : When to stop Testing with a clear conscience, Journal of Software Testing Professionals, March 2002.

[2] http://www.testrepublic.com/forum/topics/1178155:Topic:33849

[3] Quadri, S. M. K and Farooq, SU, "Software Testing - Goals, Principles, and Limitations", International Journal of Computer Applications (0975 – 8887) Volume 6- No.9, September 2010

[4] Stark, George E; Durst, Robert C; and Pelnik, Tammy M. ―An Evaluation of Software Testing metrics for NASA's Mission Control Center‖ 1992.

[5] Divya Prakash Shrivastava and R.C. Jain, ―Metrics for Test Case Design in Test Drive Development‖, International Journal of Computer Theory and Engineering(1793-8201), Vol.2, No.6, December, 2010

[6] Ogasawara, Hideto, Yamada, Atsushi and Kojo, Michiko, ―Experiences of software Quality Management Using Metrics through Lifecycle‖, Proceedings of ICSE-18, 1996.

[7] Kan, Stephen H, ―Metrics and Models In Software Quality Engineering‖, PEARSON, 2003.

[8] Futrell, Robert T.: Futrell ,Donald F. And Shafer, Linda I., ―Quality Software Project Management‖, PEARSON

[9] Torn, Aimo: Professor, Department of Computer Science Abo, Akademi University; Faculty member Turku Centre for Computer Science (TUCS) Turku, Finland.

[10] S. Chidamber and C. Kemerer: A Metrics Suite for Object-Oriented Design, In IEEE Transactions on Software Engineering, 20(6):476-493, 1994.

[11] M. Lorenz and J. Kidd: Object-Oriented Software Metrics. PrenticeHall, 1994

[12] Saida Benlarbi, Khaled El Emam, Nishith Goel. Issues in Validating Object-Oriented Metrics for Early Risk Prediction, Accessed on April 2008.

[13] IEEE Standard Glossary of Software Engineering TechnologyANSI/IEEE Standard, Washington, DC, USA,2000.

[14] R.S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill, 1997.

[15] P. Jalote, An Integral Approach to Software Engineering‖, Spring Verlog, 1997.

[16] Kumar Rajnish, Vandana Bhattacherjee,‖ Class Inheritance Metrics-An Analytical and Empirical Approach‖ September 13, 2007.

[17] Sheikh Umar Farooq, S. M. K. Quadri, Nesar Ahmad, "Software Measurements and Metrics: Role in Effective Software Testing." International Journal of Engineering Science and Technology (IJEST) 3.1 (2011): 671-680.

[18] Support.objecteering.com/help/us/metrics/metrics_in_detail/ number_of_attributes