International Journal of Computer Theory and Engineering, Vol.2, No.6, December, 2010
1793-8201

Metrics for Test Case Design in Test Driven Development

Divya Prakash Shrivastava and R.C. Jain

Abstract—**The demand for quality in software applications has grown, and awareness of software testing-related issues plays an important role. In research, we formulate the Automated Test Case of Unit Testing design metrics from a collection of internal testing design metrics that are correlated. In this paper the test case design metrics estimation is based on object oriented software design metrics determine the testability of test case in Test Driven Development approach. The estimation of the Automated Test Case for Unit Testing (ATCUT) design metrics, measures that quantify attributes of a test case design, that can be measured by the size, complexity and reusability of test case and testability of its product.**

Index Terms—**ATCUT, Metrics, TDD, Test Case.**

I. Introduction

Computer software is an engine of growth of social-economy development which requires new techniques and strategies. The demand for quality in software applications has grown. Hence testing becomes one of the essential components of software development which is the indicator of quality [1].

"Testing proves the presence, not the absence of bugs"
-------- E.W.Dijkstra

This paper investigates metrics from the perspective of unit test case designing., where our units consist of the classes. Our approach is to evaluate a set of metrics with respect to test case designing to predict the testing effort of its product. We choose this approach because metrics are a good driver for the investigation of aspects of software. The evaluation of metrics that are thought to have a bearing on the test case design and effort allows us, on the one hand, to gain insight into the factors of testability, and to obtain refined metrics on the other.

Here, the smallest testable unit is the encapsulated class or object. A single operation cannot be tested in isolation; it has to be tested as part of class. Thus we consider the unit testability under TDD with respect to ATCUT (Automated Test Case for Unit Testing) metrics.

Divya Prakash Shrivastav, Asst. Professor,Department of Computer Science, El Gabal El Garbi University, Nalut, LIBYA(Mob-00218-922913965, email: dp_shrivastava@yahoo.com).

Prof. R.C. JAIN, Head of Computer Science and Application, DirectorSamrat Ashoka Technological Institute of Technology, Vidisha (M.P.), INDIA(e-mai: jain_rc@ymail.com).

II. Test-Driven Development

Test Driven Development (TDD) is the core part of the Agile code development approach derived from eXtreme Programming (XP) and the principles of the Agile manifesto. It provides to guarantee testability to reach an extremely high test coverage, to enhance developer confidence, for highly cohesive and loosely coupled systems, to allow larger teams of programmers to work on the same code base, as the code can be checked more often. It also encourages the explicitness about the scope of implementation. Equally it helps separating the logical and physical design, and thus to simplify the design, when only the code needed.

The TDD is not a testing technique, rather a development and design technique in which tests are written prior to the production code. The tests are added its gradually during its implementation and when the test is passed, the code is refactored accordingly to improve the efficacy of internal structure of the code. The incremental cycle is repeated until all functionality is implemented to final.

The TDD cycle consists of six fundamental steps:

1) Write a test for a piece of functionality,
2) Run all tests to see the new test to fail,
3) Write corresponding code that passes these tests,
4) Run the test to see all pass,
5) Refactor the code and
6) Run all tests to see the refactoring did not change the external behavior.

The first step involves simply writing a piece of code to ensure the tests of desired functionality. The second is required to validate the correctness of test, i.e. the test must not pass at this point, because the behavior under implementation must not exist as yet.

Nonetheless, if the test passes over, means the test is either not testing correct behavior or the TDD principles have not been strictly followed. The third step is the writing of the code.

However, it should be kept in mind to only write as little code as possible to enable to pass the test. Next, step is to see that the change has not introduced any of the problems somewhere else in the system. Once all these tests are passed, then the internal structure of the code should be improved by refactoring. The afore mentioned cycle is presented in Fig 1.

Design of Model

Implementation of the design is a straightforward matter of translating the design into code, since most difficult decisions are made during design. The code should be a simple

translation of the design decision into the peculiarities of a peculiar language. Decision do have to be made while writing code, but each one should affect only a small part of the program so they can be changed easily.

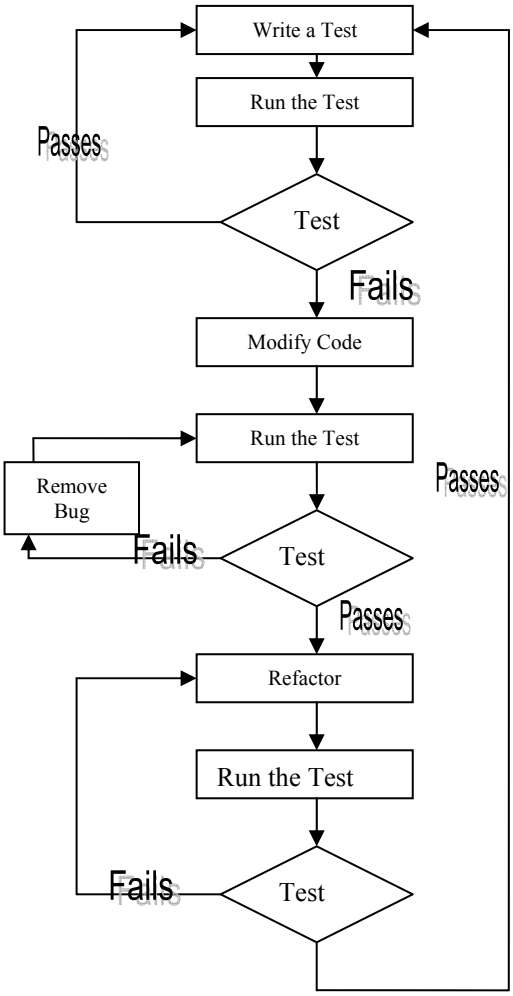

Fig 1. TDD Cycle.

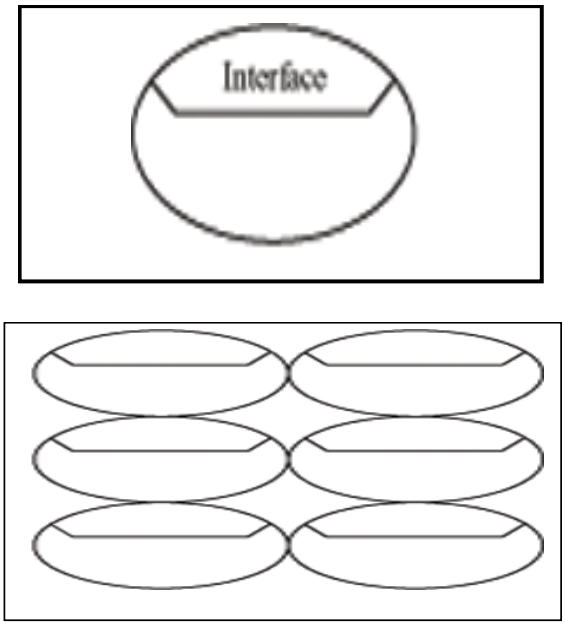

Fig 2. A unit and Aggregation of Units

Test Driven Development involved analysis, design, implementation and testing are very interleaved activities performed in an incremental fashion.

Here, I show figures of a unit and of an aggregation of units, which I will call a subsystem[4].

As we seen the V model says That someone should first test each unit. When all the subsystem's units are tested, they should be aggregated the subsystem tested to see if it works as a whole.

So how do we test the unit? We look at its interface as specified in the detailed design, or at the code, or at both, pick inputs that satisfy some test design criteria, feed those inputs to the interface, then check the results for correctness. Because the unit usually can not be executed in isolation, we have to surround it with stubs and drivers, as in the fig. The arrow represents the execution trace of a test.

Fig 3. Testing of a particular unit

A test case designed to find bugs in a particular unit might be best run with the unit in isolation, surrounded by unit-specific stubs and drivers.

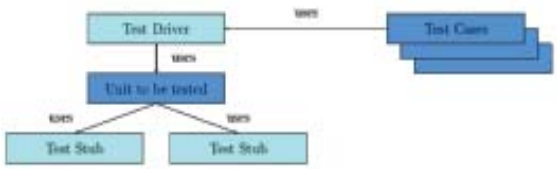

Fig 4. Test case

III. Proposed ATCUT Metrics

In this section we try to find software metrics can indicate the quality of a Test Case and determine whether they match the selection criteria. Many quality measures can be collected from literature, the main goal of metrics is to measure errors and defects. The following quality factor should have every metrics [2] –[4]:

• Efficiency - Are the constructs efficiently designed? The amount of computing resource and code required by a program to perform its function.

• Complexity - Could the constructs be used more effectively to decrease the architectural complexity?

• Understandability - Does the design increase the psychological complexity?

• Reusability - Does the design quality support possible

reuse? Extent to which a program or part of a program can be reused in other application , related to the packaging and scope of the functions that the program performs.

• Testability/Maintainability - Does the structure support ease of testing and changes? Effort required locating and fixing an error in a program, as well as effort required to test a program to ensure that it performs its intended function.

The major reason for our selection of these metrics is the close match between our and different metric suite. Chidamber and Kemerer also suggest that some of their metrics have a bearing on the testing effort; in particular, their Coupling Between Objects (CBO) and Response For Class (RFC) metrics. Other metrics from Chidamber and Kemerer's suite that are included in our set are Depth Of Inheritance Tree (DIT), Number Of Children (NOC), Weighted Methods Per Class (WMC) and Lack Of Cohesion Of Methods (LCOM).

IV. Evaluation of ATCUT Metrics

The aims of this evaluation is to determine whether the test case metrics could be used to predict the effort required to implement the design and the quality of the code produced and whether or not the number of test cases is significantly different for functional and units/classes in this domain.

A typical empirical validation of ATCUT object-oriented metrics proceeds by investigating the relationship between each metric and the outcome of interest. Here we can see the results for different metrics studied. The metrics WMC, RFC, CBO, and LCOM were defined in [5] and the NMA metric was defined in [6]. They have shown that size can have an important confounding effect on the validity of object-oriented metrics[7].

The evaluation of 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. A description of the methods used for the experiments concludes the research.

The test case design for unit testing is directly related to observe the output on pre decided ATCUT metrics and measure the testability of the units/components.

Now that we have defined a set of metrics, it is time to set up experiments to evaluate them. Empirical study within the field of software engineering is relatively rare. First, we state the goal of our experiments:

Goal: To assess the capability of the proposed source-based metrics to predict the testing effort.

Perspective: We evaluate the source-based metrics at the class level, and limit the testing effort to the unit testing of classes. Thus, we are assessing whether or not the values of the source-based metrics can predict the required amount of effort needed for unit testing a class.

V. Testability

The ISO defines testability as “attributes of software that bear on the effort needed to validate the software product” [8]. In the unit testing of object oriented system, the testing for classes brings in some issues that are not present in testing of functions as in the case of structured systems. These are: for object-oriented systems, a class cannot be tested directly, only an instance of the class can be tested. The control flow is characterized by message passing among objects unlike the structured systems where there is sequential flow within a function and when an object is considered in an object oriented system, the state associated with that object also influences the path of execution and methods of a class can communicate among themselves through this state, because this state is persistent across innovations of methods. Thus we considered the unit testability of the object oriented system with respect to the test case design for unit testing in test driven development.

The testing requires reporting on the concrete and abstract state of an object, encapsulation of attributes and operations makes it difficult. Inheritance poses further usage requirement on retesting even though reuse has been achieved[9], multiple inheritance further complicates the testing by increasing the number of contexts for which testing is required. The applicability of test cases within super class and subclass also need to be considered with care [9],[10].

The key concept of object orientation lies in the encapsulation of information along with the implementation of operations performed on the information.

Coupling provides us a measure of strength of association established by a connection between entities. The measure, coupling between object classes to which a class is coupled. It is measured by counting the number of distinct non inheritance related class hierarchies on which a class depends [5],[9],[10]. The larger number of couples will be the higher sensitivity to change and errors in other parts of design and making testing difficult. This would increase the testing effort (TE) and decrease the testability. Therefore, we say that testability is inversely proportional to CBO.

$$TE \propto CBO$$

$$ITb \propto 1/CBO \quad (1)$$

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 individually. Thus we have :

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

Now, considering the combination of the complexity of a class through the number of methods and the amount of communication with other classes. It is 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, that complicated the testing is, which in result decreases the testability.

Using the metric, response for a class (RFC) which is defined as the number of methods in response set [5],[9],[10], we say that the testing effort (TE) is directly proportional to RFC and hence testability is inversely proportional to it.

$$TE \propto RFC$$

$$ITb \quad \infty \quad 1/RFC \tag{3}$$

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

$$TRFC = \sum_{i=1}^{n} RFC \tag{4}$$

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. This hierarchy is formed due to the inheritance relation between classes, where inheritance leads to super class accumulating all the common features of the subclass.

The metric depth of the inheritance that measure the depth of the class within the inheritance hierarchy is defined as "the maximum length from the node to the root of the tree" [5],[9]. It shows that the deeper a class is within the hierarchy, the greater the number of methods it is likely to inherit. This makes 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 DIT.

$$TE \quad \infty \quad DIT$$

$$ITb \quad \infty \quad 1/DIT \tag{5}$$

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

$$TDIT = \sum_{i=1}^{n} DIT \tag{6}$$

From the inheritance hierarchy, another metric, number of children (NOC) is given by [5],[9],[10]. It is given as the number of descendents of the class. As NOC increases, the effort of testing (TE) of methods of that class increases. This decreases the testability providing an inverse relationship:

$$TE \quad \infty \quad NOC$$

$$ITb \quad \infty \quad 1/NOC \tag{7}$$

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

$$TNOC = \sum_{i=1}^{n} NOC \tag{8}$$

Effective object oriented design helps to maximize cohesion as they promote the property of encapsulation. Cohesion here is defined as the degree to which methods within a class are related to one another and work together to provide well-bounded behavior.

The metric lack of cohesion of method (LCOM) measures the degree of similarity of methods that access one or more of the same attributes [11]. Higher value of LCOM indicates that the methods may be coupled to one another via attributes, increasing the complexity of class design and the likelihood of errors during the development process. This leads to increase in effort of testing (TE), decreasing the interface testability. Thus, we say that LCOM is indirectly proportional to testability:

$$TE \quad \infty \quad LCOM$$

$$ITb \quad \infty \quad 1/LCOM \tag{9}$$

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

$$TLCOM = \sum_{i=1}^{n} LCOM \tag{10}$$

The number of methods and the complexity of the methods involved is a predictor of how much time and effort required to develop and maintain the class. The number of methods and their complexity is a reasonable indicator of the amount of effort required to implement and test a class.

The metric, weighted method per class (WMC), is a count methods implemented within a class, or the sum of the complexities of the method [11]. As the value of WMC increases, testing become difficult and testability would decrease. Thus we say that the testing effort (TE) is directly proportional to WMC and interface testability is inversely proportional to WMC.

$$TE \quad \infty \quad WMC$$

$$ITb \quad \infty \quad 1/WMC \tag{11}$$

Estimating total WMC (TWMC) over all classes (i=1 to n), we get the testability with respect to a class :

$$TWMC = \sum_{i=1}^{n} WMC \tag{12}$$

Eequations (1,3,5,7,9,11) we get the testability with respect to a class :

$$ITb \quad \infty \quad (1/CBO) \times (1/RFC) \times (1/DIT) \times (1/NOC) \times (1/LCOM) \times (1/WMC) \tag{13}$$

$$ITb = \frac{k}{(CBO \times RFC \times DIT \times NOC \times LCOM \times WMC)} \tag{14}$$

Where 'k' is the constant of proportionality and other values are defined above.

By equations (2,4,6,8,10,12) the total interface testability (TITb) of the object oriented software over all classes (i=1 to n) can be given as

$$TITb = \frac{k}{(TCBO \times TRFC \times TDIT \times TNOC \times TLCOM \times TWMC)} \tag{15}$$

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. The value will have to be worked out for specific software teams of concerned organization.

VI. Conclusion

These ATCUT design metrics cover the key concepts for test case design and its testability. In this work , I have considered the possible testability for unit testing in test driven development.

In summary, we have seen that the results have shown us that the ATCUT metrics seem to measure test case construction factors/testability. Most notably, the results allow for explanations of the LOC, WMC, RFC, DIT, NOC and DIP metrics in terms of test case construction factors. We conclude that the results will allow us to improve the set of metrics and the development approach.

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] Chen, J-Y., Lum, J-F.: A New Metrics for Object-Oriented Design, Information of Software Technology 35,4(April 1993):232-240.
[3] Jon Avotins: Defining and Designing a Quality OO Metrics Suite, Depertment of Software Development, Monash University, Australia 3145.
[4] Rosenberg, H Linda: Applying and Interpreting Object Oriented Metrics, Software Assurance Technology Office (SATO).
[5] S. Chidamber and C. Kemerer: A Metrics Suite for Object-Oriented Design, In IEEE Transactions on Software Engineering, 20(6):476-493, 1994.
[6] M. Lorenz and J. Kidd: Object-Oriented Software Metrics. Prentice-Hall, 1994.
[7] Saida Benlarbi, Khaled El Emam, Nishith Goel. Issues in Validating Object-Oriented Metrics for Early Risk Prediction, Accessed on April 2008.
[8] ISO. International standard ISO/IEC 9126. information technology: Software product evaluation: Quality characteristics and guidelines for their use, 1991.
[9] R.S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill, 1997.
[10] P. Jalote, An Integral Approach to Software Engineering", Spring Verlog, 1997.
[11] B.-K. Kang and J. M. Bieman. A Quantitive Framework for Software Restructuring. Journal of Software Maintenance, 11:245-254, 1999.