Software Engineering 2012, 2(3): 43-48
DOI: 10.5923/j.se.20120203.01

Unit Test Case Design Metrics in Test Driven Development

Divya Prakash Shrivastava

Department of Computer Science and Engineering, Al Jabal Al Garbi University, Gharyan, Libya

Abstract Testing is a validation process that determines the conformance of the software's implementation to its specification. It is an important phase in unit test case design and is even more important in object-oriented systems. We want to develop test case designing criteria that give confidence in unit testing of object-oriented system. The main aim testing can be viewed as a means of assessing the existing quality of the software to probe the software for defect and fix them. We also want to develop and executing our test case automatically since this decreases the effort (and cost) of software development cycle (maintenance), and provide re-usability in Test Driven Development framework. We believe such approach is necessary for reaching the levels of confidence required in unit testing. The main goal of this paper is to assist the developers/testers to improve the quality of the ATCUT by accurately design the test case for unit testing of object-oriented software based on the test results. A blend of unit testing assisted by the domain knowledge of the test case designer is used in this paper to improve the design of test case. This paper outlines a solution strategy for deriving Automated Test Case for Unit Testing (ATCUT) metrics from object-oriented metrics via TDD concept.

Keywords ATCUT Metrics,TDD,Test Case Design, OOM, UTF

1. Introduction

The object-oriented paradigm plays a prominent role in the development of many modern software systems. The different structure and behavior of object oriented software helps in solving or mitigating several problems of procedural software; but raise new problems that often cannot be addressed with traditional techniques. In particular; object oriented software presents new classes of faults that require new testing techniques. System testing requires more than simply creating and executing tests. Before beginning to test, the overall strategy must devise, including how to record problems found during testing.

Here we focus on the issue of class testing. Major problems in class testing derive from the presence of instance variables and their effects on the behavior of methods defined in the class. A given method can produce an erroneous result or a correct one; depending on the value of the receiver's variables when the method is invoked. It is therefore crucial that techniques for the testing of classes exercise each method when the method's receiver is in different states[1].

Change made to production classes must be reflected in the code used to test the classes. This requires a high degree of traceability from the production code into the test code.

Reliability is defined as the probability that a test case function according to its specification in a specified environment. It gives a measure of confidence in the unit testing of object-oriented system. A failure occurs when the unit testing behavior does not match the specification. A fault is a static component of the test case that causes the failure. A fault causes failure only when that part of the code is executed, and hence, not all faults result in failures. An error is a programmer action or omission and results in a fault. Testability is a statistical study of the failures.

In parallel with the rise to prominence of the object-oriented paradigm has come the acceptance that conventional software testing metrics are not adequate to measure object-oriented unit testing. This has inspired to develop new metrics that are suited to the Automated Test Case Design paradigm for testing of unit of object-oriented system. The design of Automated Test Case for Unit Testing metrics have been much empirical evaluation, with claim made regarding the usefulness of metrics to assess external attributes such as quality.

To achieve high test case reliability, extensive test case designing is required, and hence, some terms are defined next. A Test Case is defined as a single value mapping for each input of the unit of the system that enables a single execution of the software system.

The Unit Test is the lowest level of testing performed during software development, where individual units of software are tested in isolation from other parts of

* Corresponding author:
dp_shrivastava@yahoo.com(Divya Prakash Shrivastava)
Published online at http://journal.sapub.org/se
Copyright © 2012 Scientific & Academic Publishing. All Rights Reserved

program/software system.

What can be tested?

One instruction

One feature

One class

One cluster

One program

One library

Good test cases increase the rate of early fault detection and correction by finding bugs early so they can be corrected early[2].

2. Test Styles and Paradigms

Of all testing levels, the unit level of testing has undergone the most recent and most dramatic change. With the introduction of new agile (aka, "lightweight") development methods, such as XP (eXtreme Programming) came the idea of Test-Driven Development (TDD). The TDD is a software development technique that melds program design, implementation and testing in a series micro-iterations that focus on simplicity and feedback. Programmer tests are created using a unit testing framework and are 100% automated.

Under TDD, all test cases are automated with the help of a Unit Testing Framework (UTF). The UTF assists developers in creating, executing and managing test cases (UTFs exist for almost every development environment, an indication of the popularity of this technique.). All test cases must be automated because they must be executed frequently in the TDD process with every small change to the code.

TDD uses a "test first" approach in which test cases are written before code is written. These test cases are written one-at-a-time and followed immediately by the generation of code required to get the test case to pass. Software development becomes a series of very short iterations in which test cases drive the creation of software and ultimately the design of the programme.

3. Test Case for Unit Testing

Before Just as the procedure and function are the basic units of abstraction in procedure-oriented languages, the class is the basic unit of abstraction in object-oriented languages (and object based). Naturally, it makes sense that testing applied to these types of languages should focus on their primary abstraction mechanisms. This view is reflected by the proportion of literature on testing object-oriented software that is devoted to the testing of classes[3].

The design of test cases has to be driven by the specification of software. For Unit testing, test cases are designed to verify that an individual unit implements all design decisions mode in the unit's design specification. A thorough unit test specification should include positive testing that the unit does what it is supposed to do, and also negative testing, that the unit does not do anything that it is not supposed to do.

A unit test specification comprises a sequence of unit test cases. Each unit test case should include four essential elements.

- A statement to the unit, the starting point of test case.
- The inputs to the unit, including the value of any external data read by the unit.
- What the test case actually tests, in terms of functionality of unit and the analysis used in the design of the test case.
- The expected outcome of the test case.

A software metrics defines a standard method of measuring certain attributes of process or product or services. Our approach is to evaluate a set of object-oriented metrics with respect to their capabilities to design the unit in V model. We choose this approach because metrics are a good driver for the investigation and design aspects of software. The analysis of metrics that are thought to have a bearing on the design allows us, and to obtain refined ATCUT metrics.

4. Principles of OOD

Object oriented principles advise the designers what to support and what to avoid. We categorize all design principles into three groups in the context of design metrics. These are, general principles, cohesion principles, and coupling principles. These principles are collected by Martin[4]. The following discussion is a summary of his principles according to our categories.

4.1. General Principles

The Open/Closed Principle (OCP): Open close principle states a module should be open for extension but closed for modification i.e. Classes should be written so that they can extend without requiring the classes to be modified.

The Liskov Substitution Principle (LSP): Liskov Substitution Principle mention subclasses should be substitutable for their base classes i.e. a user of a base class instance should still function if given an instance of a derived class instead.

The Dependency Inversion Principle (DIP): Dependency Inversion Principle states high level classes should not depend on low level classes i.e. abstractions should not depend on the details. If the high level abstractions depend on the low level implementation, dependency is inverted from what it should be[5]

The Interface Segregation Principle (ISP): Interface Segregation Principle state Clients should not be forced to depend upon interfaces that they do not use. Many client-specific interface are better than one general purpose interface.

4.2. Cohesion Principles

Reuse/Release Equivalency Principle (REP): The granule of reuse is the granule of release. Only components that are released through a tracking system can be efficiently reused.

A reusable software element cannot really be reused in practice unless it is managed by a release system of some kind of release numbers. All related classes must be released together.

Common Reuse Principle (CRP): All classes in a package should be reused together. If one of the classes in the package reused , reuse them all. Classes are usually reused in groups based on collaborations between library classes.

Common Closure Principle (CCP): The classes in a package should be closed against the same kind of changes. A change that affects a package, affects all the classes in that package. The main Goal of this principle is to limit the dispersion of changes among released packages i.e. changes must affect the smallest number of released packages.

Classes within a package must be cohesive. Given a particular kind of change, either all classes or no class in a component needs to be modified.

4.3. Coupling Principles

Acyclic Dependencies Principle (ADP): The dependency structure for a released component must be a Directed Acyclic Graph (DAG) and there can be no cycles.

Stable Dependencies Principle (SDP): The dependencies between components in a design should be in the direction of stability. A component should only depend upon components that are more stable than it is.

Stable Abstractions Principle (SAP): The abstraction of a package should be proportional to its stability. Packages that are maximally stable should be maximally abstract. Instable packages should be concrete.

5. Test Case for Unit Testing

This section describes the metrics collected and their source. The research focuses on metrics that pertain to internal design and code quality. In order to measure internal quality, test case from the industry experiments was collected and metrics were generated and analyzed statistically.

Desirable Attributes of Quality Test Case

Desirable attributes of high quality software were identified

- Understandability

– low complexity, high cohesion, simplicity

- Maintainability

– low complexity, high cohesion, low coupling

- Reusability

– low complexity, high cohesion, low coupling, inheritance

- Testability

– high cohesion, low coupling, high test coverage component needs to be modified.

5.1. Review of Literature on Design based Metrics

In order to improve the object oriented design, software measures or metrics are needed. The main objective of object oriented design metrics is to understand the quality of product to assess the effectiveness of the process and to improve the quality of work in the project. Application of design metrics in software development saves the time and money on redesign. The following figure shows metrics hierarchy according to our categorization.

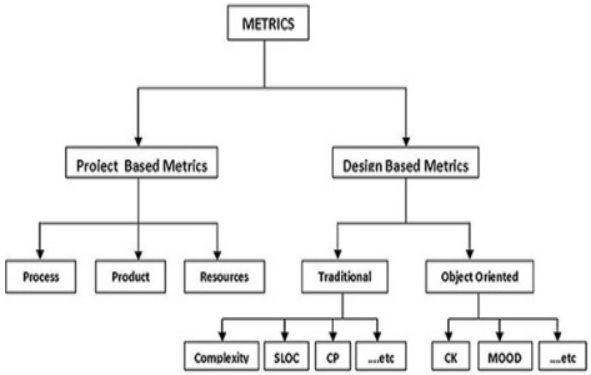

Figure 1. Metrics Hierarchy

A significant number of object oriented metrics have been developed in literature. In object oriented design there are major sets of design metrics proposed by Abreu[6], C.K metrics[7], Li and Henry[8] metrics, MOOD metrics[9], Lorenz and Kidd[10] metrics etc. C.K metrics are the most popular (used) among them. Another comprehensive set of metrics is MOOD metrics. This subsection will focus on traditional metrics and above mention metrics (mainly C.K and MOOD metrics).

5.2. C.K. Metrics

Chidamber and Kemerer define the so called CK metric suite[11]. This metric suite offers informative insight into whether developers are following object oriented principles in their design[8]. They claim that using several of their metrics collectively helps managers and designers to make better design decision. CK metrics have generated a significant amount of interest and are currently the most well known suite of measurements for OO software[12]. Chidamber and Kemerer proposed six metrics; the following discussion shows their metrics.

WMC Weighted Methods per Class (Complexity)
DIT Depth of Inheritance Tree
NOC Number of Children
CBO Coupling between Object Classes
RFC Response for a Class
LCOM Lack of Cohesion in Methods

5.3. MOOD Metrics

Abreu[4] defined MOOD (Metrics for Object Oriented Design) metrics. MOOD refers to a basic structural mechanism of the object-oriented paradigm as encapsulation (MHF, AHF), inheritance (MIF, AIF), polymorphism (POF), and message passing (COF). Each metrics is expressed as a measure where the numerator represents the actual use of one of those feature for a given design[13].

Britto e Abreu have proposed the following set of metrics

for object oriented design called MOOD set.

MHF Method Hiding Factor
AHF Attribute Hiding factor
MIF Method Inheritance Factor
AIF Attribute Inheritance Factor
CF Coupling Factor
PF Polymorphism Factor

6. Similarity of OO Metrics

Object oriented metrics can be collected in different ways. Although different writers have described different metrics, according to object oriented design, there are some similarities found in different metrics. The following table shows similar OO metrics. We have categorized metrics in class, attribute, method, cohesion, coupling, and inheritance category because most of the object oriented metrics are defined in above mention categories. In this table we will discuss only CK metrics suite, MOOD metrics, and metrics defined by Chen & Lu, Li & Henry. Since other metrics are defined from different context and different point of views, we have not considered those metrics in our table.

Table 1. Similar Object Oriented Metrics

Category	Class	Attribute	Method	Cohesion/ Coupling	Inheritance
MOOD	MHF, AHF, POF, COF	AHF, AIF	MHF, MIF, POF		MIF, AIF
Chidamber & Kemerer	WMC, RFC, LCOM	LCOM	WMC, RFC, LCOM	CBO	DIT, NOC
Chen & Lu	OXM, RM, OACM			CCM, OCM	CHM
Li & Henry	DAC	Size2	MPC, NOM, Size1, Size2	MPC	

We categorized these metrics (shown above table) to find out the right metrics to measure class, methods, attributes, etc. Since we did not collect all proposed metrics and we did not categorize all of them, this table is not complete to give clear recommendation as to which metrics should be used. But these categorizations focus on common metrics which will be helpful for novice designers to support their design measurements. The following is a brief discussion of that categorization. These metrics will be categorized as method level, class level, and interface level.

For each experiment and the case study, a large number of metrics will be reported and analyzed. This section describes the metrics collected. It also describes the statistical analysis conducted. The research focuses on metrics that pertain to internal design and test case quality.

7. Proposed ATCUT Metrics Set

The major reason for our selection of these metrics is the close match between our model and different metric suite. Compared to several other metrics presented in the above description, our metrics have the additional advantage of being easier to implement and understand.

Chidamber and Kemerer also suggest that some of their metrics have a bearing on the design and 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).

7.1. The ATCUT Metrics Discussion

The design quality was assessed through six ATCUT metrics. The data were gathered from case projects, in order to make comparison between Test At First (Test Driven Development) and Test at Last traditional approach.

In software development, test case design is the earliest stage on which the structure of the system is defined, and therefore, term "design" is generally used when referring to architectural design. The traditional product metrics such as size, complexity and performance are not sufficient for characterizing and assessing the quality of test case for OO software systems, and they should be supplemented with notions such as encapsulation and inheritance, which are inherent in object orientation. The OO design metrics for measuring the test case design quality effects of TDD within this study were selected from the ones proposed ATCUT metrics: Line of codes (LOC),Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number Of Children (NOC), and Response For a Class (RFC). Dependency Inversion Principle (DIP) metric measure the ratio of dependencies that have abstract classes.

Table II summarizes the used ATCUT metrics. The WMC value presents class's complexity, and it can be used for predicting how much effort certain class requires when it is maintained. A high value of WMC implies that the class is probably less reusable and requires more maintaining effort, because it is likely to have a greater impact on its derived classes and it may be more application specific. The proposed threshold for WMC is 20 when the value presents the number of methods.

The DIT value measures the depth of each class within its hierarchy, and it can be utilized for predicting the complexity and the potential reuse of a class. A high value of DIT indicates increased complexity, because more methods are involved, but also increased potential for reuse of inherited methods. A proposed threshold for DIT is around 6.

NOC presents the number of class's immediate subclasses, and it may be used for assessing class's potential reuse and the effort the class requires when it is tested. A high value of NOC may be a sign of an improper use of sub classing and it implies that the class may require more testing, because it has many children using its methods. On the other hand, a high NOC value indicates greater potential for reuse,

because inheritance is one form of reuse. There is no specific threshold proposed for NOC, at least to the author's knowledge.

The value of RFC presents the number of methods that can be invoked in response to a message to an object of the class or by some method in the class. The testing and debugging of a class with a high value of RFC is difficult, because the increased amount of communication between classes requires more understanding on the part of the tester. It also indicates increased complexity. A threshold of 50 is proposed for RFC, although the value is acceptable up to 100.

The Dependency Inversion Principle DIP state high level classes should not depend on low level classes i.e. abstractions should not depend upon the details.

Table 2. Quality Metrics for Test Case Design

Metric	Threshold	Description
WMC	20	A count of the methods of a class or the sum of the complexities of the methods. High value of WMC may indicate decreased reusability and maintainability
DIT	6	The depth of a class in an inheritance hierarchy. High value of DIT may indicate increased reusability, but also increased complexity.
NOC	-	The number of immediate subclasses subordinate to a class in the hierarchy. High value of NOC may indicate increased reusability, but also increased testing effort and it can be a sign of a misuse of sub-classing.
RFC	50 (acceptable up to 100)	The combination of the complexity of a class through the number of methods and the amount of communication with other classes. High value of RFC may indicate increased testing effort and complexity.
DIP	-	The DIP metric measure the ratio of dependencies that have abstract classes. If the high level abstractions depend on the low level implementation, the dependency is inverted from what it should be

8. Statistical Analysis of ATCUT Metrics

This section describes the case study conducted in a large, software corporation. Two projects were developed by developers. The first project was completed and the developers employed a test at last approach including automated unit-tests written in JUnit. The project was developed in a traditional mode with a large up-front design and automated and manual testing after the software was implemented. This project will be labeled as test at last (TAL). The second project is the same functionality using a test at first (TDD) approach with automated unit tests. This project will be labeled as test at first (TAF). This section reports, compares, and describes the internal design quality metric results. The metrics are the method, class, and interface level.

In our analysis we use two java projects to measure object oriented metrics. Project 1 which one adopted test at last (TAL) development approach with automated unit testing. That contains 20 classes, 134 methods, and LOC is 1729. Project2 is same what the functionality of project1 is but the project2 is developed in test at first (Test Driven Development) manner. The project will be labeled as test at first (TAF), which contains 25 classes, 103 methods and the total line of code (LOC) is 1023. In this case study, we focused mainly LOC, WMC, RFC, DIT, NOC and DIP metrics, because these are backbone of test case design metrics.

Table 3. Mean and %difference of ATCUT Metrics

S.No	Metrics	TAL Mean	TAF Mean	% Difference
1	LOC	76.25	47.04	38
2	WMC	10.15	8.6	15
3	RFC	18.55	14.32	23
4	DIT	1.65	2.64	60
5	NOC	0.5	0.12	76
6	DIP	0.153	0.059	61

This analysis demonstrates statistically significant differences of the metrics. Fig 2. Illustrates the differences in a radar chart. Each of the metrics has been normalized by multiplying the metric mean for each group (test at first and test at last) by a normalizing factor. The normalizing factor is calculated by dividing the maximum mean of all metrics by the maximum mean of the two (test at first and test at last) groups for the metric being plotted. This chart demonstrates the degree and consistency by which the means of the test at last and test at first metrics varied.

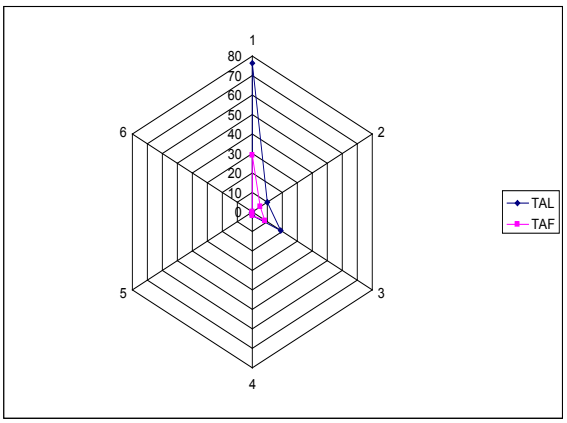

Figure 2. Radar Chart of TAF and TAL Mean

9. Conclusions

The primary contribution of this research is the empirical evidence of the effects that applying TDD has on test case designing for internal software quality.

This analysis demonstrates statistically significant differences of the metrics. Figures illustrate the differences in a radar chart. This chart demonstrates the degree and consistency by which the means of the test at last (TAL) and

test at first(TAF) metrics varied.

This data implies and the radar chart illustrates relatively few significant differences between project developed with a test at last approach and project developed with a test at first approach. Notice that unlike the previous experiments, test at first method tends to have (insignificantly) lower mean metric values than the test at last methods. The data indicates that software developed with a test at last approach is significantly larger in only three of the size, complexity and response for class measures (LOC, WMC and RFC), than software developed with a test at first approach.

9.1. Weighted Method per Class

The graphical representation of WMC indicates that most of the classes have more polymorphism and less complexity. Low WMC indicates greater polymorphism in a class and high WMC indicates more complexity in the class. The WMC figures look quite similar for both Projects but test at first (TDD) development approach is quite better than test at last approach.

9.2. Response For a Class

This result indicates that in test at first approach having less complexity..

9.3. Depth of Inheritance Tree

In our analysis, Project 1, and in Projects 2 result indicates, classes of Project 2 are a higher degree of reuse and fewer complexes.

9.4. Number Of Children

NOC metric measures the number of direct subclass of a class. Since more children in a class have more responsibility, thus it is harder to modify the class and requires more testing. So NOC with less value is better and more NOC may indicate a misuse of sub classing. In our analysis, both Project1 and Project2 need less testing effort.

9.5. Dependency Inversion Principle

The DIP metric measures the ratio of dependencies that have abstract classes. In our analysis, most of the class from project1 indicates more DIP whereas Project2 indicate less DIP. This result shows the classes from Project2 are more depend on abstract classes than Project1's classes.

REFERENCES

[1] Ugo Buy , Alessandro Orso, Mauro Pezz'e, "Automated Testing of classes", Volume 25 Issue 5 pp 39-48 ,copyright 2000.

[2] B. Cohen, P. B. Gibbons, W. B. Mugridge and C. J. Colbourn, " Constructing Test Suites for Interaction Testing", 25th International Conference on Software Engineering (ICSE'30), pp. 38-49, Portland, Oregon, United States, IEEE Computer Society, 2003.

[3] Binder, R.V., Testing Object Oriented Software: A Survey. Journal of Software Testing, Verification & Reliability, 1996. 6(3/4), September/December. P. 125-252.

[4] Robert C. Martin:" Agile Software Development":Principles, Patterns and Practices,2002.

[5] Robert C. Martin , www.objectmentor.com.

[6] Abreu, Fernando B: "Design metrics for OO software system", ECOOP'95,Quantitative Methods Workshop, 1995.

[7] Chidamber, Shyam , Kemerer, Chris F. "A Metrics Suite for Object- Oriented Design." M.I.T. Sloan School of Management E53-315, 1993.

[8] Li,Wei , Henry, Salley.: "Maintenance Metrics for the Object Oriented Paradigm", First International Software Metrics Symposium. Baltimore, Maryland, May 21-22, 1993. Los Alamitos, California: IEEE Computer Society Press, 1993.

[9] Abreu, Fernando B: "The MOOD Metrics Set," Proc. ECOOP'95 Workshop on Metrics, 1995.

[10] Lorenz, Mark & Kidd Jeff: "Object-Oriented Software Metrics", Prentice Hall, 1994.

[11] J. Lakos, "Large-scale C++ software design," Addison-Wesley, 1996.

[12] M. Grand, "Patterns in Java," Volume 2, John Wiley & Sons, 1999.

[13] Abreu, Fernando B, Rita, E., Miguel, G. : "The Design of Eiffel Program: Quantitative Evaluation Using the MOOD metrics", Proceeding of TOOLS'96 USA, Santa Barbara, California, July 1996.