TESTABILITY OF SOFTWARE SYSTEMS

Sanjeev Patwa [1] & Anil Kumar Malviya [2]

[1] Mody Institute Of Technology & Science , Lakshmangarh,Sikar(Raj.), India
[2] KNIT, Sultanpur,U.P, India

ABSTRACT

Software testing is one of the most expensive phase of the software development life cycle. Testing object oriented software is more expensive due to various features like abstraction, inheritance etc. The cost of testing can be reduced by improving the software testability. Software testability of a class is generally measured in terms of the testing effort which is equal to the number of test cases required to test a class. Hence testability can be improved if the test cases can be reduced. Software contracts (method preconditions, method postcondtions, and class invariant) can be used in improving the testability of the software. The paper provides some suggestions that will inspire teams to make their software products more testable. It cites examples of testability features that have been used in testing various software products, it also provides a detailed exploration of how testability issues affect GUI test automation and what kinds of provisions testers must make when testability is lacking.

Keywords: *Software, Testing, Object oriented, Testability.*

1. INTRODUCTION

Testability is a software quality characteristic that is of major relevance for test costs and software dependability. Still, testability is not an explicit focus in today's industrial software development projects. Testing consumes a significant amount of time and effort within an average software development project. There are different approaches to keep test costs under control and to increase the quality of the product under test:

- improve the software specification and documentation,
- reduce or change functional requirements to ease testing,
- use better test techniques,
- use better test tools,
- improve the test process,
- train people, and
- improve the software design and implementation.

The degree to which a software system or component facilitates testing is called *testability*. Designing a software system with testing in mind is called *design for testability*. There are different ways to improve the testability of a software design, for example to limit the use of class inheritance, to limit the state behavior of objects, to avoid overly complex classes, to avoid no determinism, and to prepare graphical user interfaces for the use of test automation tools. we concentrate on system dependencies and observability , describe related testing problems, and give guidelines on how to improve testability. There are many definitions of testability. Binder defines two facets of testability succinctly:

"To test a component, you must be able to control its input (and internal state) and observe its output. If you cannot control the input, you cannot be sure what has caused a given output. If you cannot observe the output of a component under test, you cannot be sure how a given input has been processed."

Based upon these definitions, it is intuitive how controllability and observability impact the ease of testing.
Without controllability, seemingly redundant tests will produce different results. Without observability,
incorrect results may appear correct as the error is contained in an output that we are unable to see. Other facets of testability that impact the ease of testing include:

- the amount of difficulty in setting up drivers to execute a component and creating stubs for functionality that does not yet exist,
- the complexity and amount of inherited, parameterized, and polymorphic types in your software, and
- the thoroughness of specification and design information available for test design.

Another definition of testability focuses on the *value* of testing. Voas argues that controllability and observability do not adequately represent all the costs associated with testing, though they are certainly part
of the testability equation [4-6]. A key component is the ability of a test to reveal faults. Testing is of less value if a particular testing activity fails to locate existing problems. This value definition of testability attempts to measure the amount of effort necessary to adequately test a system such that all faults are found. Even more broadly

testability may be anything that makes software easier to test, improves its testability, whether by making it easier to design tests and test more efficiently Bach describes testability as composed of the following.

- *Control*. The better we can control it, the more the testing can be automated and optimized.
- *Visibility*. What we see is what we test.
- *Operability*. The better it works, the more efficiently it can be tested.
- *Simplicity*. The less there is to test, the more quickly we can test it.
- *Understandability*. The more information we have, the smarter we test.
- *Suitability*. The more we know about the intended use of the software, the better we can organize our testing to find important bugs.
- *Stability*. The fewer the changes, the fewer the disruptions to testing.

This broader perspective is useful when you need to estimate the effort required for testing or justify your estimates to others.

2. IMPORTANCE OF TESTABILITY

Several software development and testing experts pointed out the importance of testability and design for testability, especially in the context of large systems:

"During the design of new systems we do not have only to answer the question 'can we built it?' but also the question 'can we test it?'. Good testability of systems is becoming more and more important."

"Design for testability, although rarely the first concern of smaller projects is of paramount importance when successfully constructing large and very large systems." The importance of software testing for a particular software system increases with

- The size and complexity of the system,
- The risk for life and business if errors remain undetected,
- The frequency of the test activities, and
- The life-time of the system (assuming that maintenance and regression testing are permanent tasks).

Testability is important for software tester and programmers because it helps them to keep the test effort under control; additionally it is relevant to consumers as well.

3. APPLICATION PROBLEM AND TESTABILITY

Approaches to programming have been changed dramatically since the invention of computers. The primary reasons for the change are: to accommodate the increasing complexity; to ensure correctness; and to prove the productivity of software. Object-oriented programming (OOP) has been broadly accepted since the 1980s when C++ emerged as a powerful OOP language .OOP has taken the best ideas of structured programming and combines them with several new concepts such as abstraction, encapsulation, inheritance and reusability. An object is a specific instance of a class. Conventionally, the design and testing of OO software *(OOS) are* relatively separate phases and independent activities.. Communication softwares play an important role in the successful operation of large and distributed computer systems applications. Within the communications software engineering process, testing is a critical activity in which the quality and functions of the software implementation are checked against the software specification. Testing communications software and distributed systems in general is becoming an increasingly complex activity requiring lots of effort and time.

Despite enormous efforts devoted to developing testing techniques, serious problems with respect to the generation and application of tests remain to be solved. Although improved testing methods had undoubtedly helped alleviating some of these problems, it is unlikely that these will lead to effective and economical testing of arbitrary software. Issues related to testing can no longer be considered in isolation from the specification and implementation of software products. Effective and economical testing requires that a software specification to contain mechanisms to make the implementation easier to test, thereby reducing development costs and increasing software reliability and maintainability.

Design for testability (DFT) is the process of introducing some features into a protocol specification to facilitate and enhance the testing process. For a better understanding of communication software testability issues, we adopt the following general definition: "software possesses the testability property if it includes facilities allowing the easy application of testing methods, and the detection and isolation of existing faults". Unfortunately, most existing protocols have been designed and documented without consideration for the testing requirements. We argue that design for testability should start at the earliest possible phase of the protocol life cycle, i.e. at the specification level. DFT may only be able to improve the effectiveness of the testing process under certain constraints.

The set of properties that characterize testable software is not yet well-defined, but software designers have an intuitive idea as to what constitutes testable software. Observability and controllability of software are widely

acknowledged as two important attributes that influence software testability. DFT aims at reducing the difficulties encountered in the testing process. Specifically, we list the following difficulties:
1. Selection of the test suite with minimal length and maximal coverage of faults: a test set may be infinite; in which case, total test coverage cannot be achieved unless a fault domain is restricted in advance.
2. Application of a test suite to the implementation under test: most test suites are derived from protocol specifications, and therefore are called abstract test suites. The efforts spent in adapting these test suites for specific implementations can diminish the advantage of using them.
3. Identification of the design schemes that lead to parts which are not-testable or difficult to test.
4. Analysis and interpretation of test results (traces): the problem of finding a matching trace in the specification requires the identification of each input and output in both the expected and the observed traces.
5. Fault diagnostics: when a test case fails, it may be difficult to determine the cause of the failure. The presence of multiple faults in the implementation can further complicate the fault isolation problem.

3.1 Problem and Solution in Dependencies to Classes

Problem: If a client class depends on a server class we can implement a stub as a subclass of the server class. Implementing the stub as a subclass of the server class is not possible in case that

- The stub should inherit from some other (test framework) class (but multiple inheritance is not
available in Java) or
- The class implements the Singleton pattern. In this case it can not be subclassed because of its private constructor (a super-call to the constructor is not possible) and because of its static
getInstance()method which can not be overridden (e.g. to return the stub instead of the singleton instance).

Solution: Let every test relevant class implement a corresponding interface. If the class is part of an inheritance tree the root class shall implement an interface or it shall be an abstract class each client has only dependencies to the interface that the server implements or to the abstract superclass of the server..

3.2 Problem and Solution in High Coupling of Classes

Problem: The class under test (CUT) depends on a large number of other classes. This makes test tasks more difficult, for example the definition and selection of test cases, the definition of the test order, the creation of stubs, test-setup and integration testing.
Solution: Adhere to the general software engineering principles on how to reduce coupling. Limit the visibility of classes within the system by making them private or protected whenever appropriate. Prefer primitive and language defined parameter types instead of developer defined classes and interfaces.

3.3 Measurement Model of OOS Testability

The *TOOP* method is developed by Y. Wang et al. to support the implementation of design for testability in *OOS*, and of building testable mechanisms into objects during coding or compiling. A systematic approach to *TOOP* has been provided, and the testability of *OOS* at *BCS* level *(CBcs)*, object level *(OTA) and* system level *(STA) are* quantitatively modeled.

4. A TESTABILITY FRAMEWORK

Testability-oriented communications software development cycle based on an iterative process consisting of specification transformations and testability measurements. Present a testability-oriented specification classification based on the determinacy, specifiedness and minimality of the specifications.

5. LIFE CYCLE APPROACH

We consider design for testability as an iterative process of modifying specifications and measuring the testability of the newly obtained specification. These transformations can be partially automated and integrated in the traditional communications software development cycle as shown in Fig.1.

6. USER INTERFACE TESTABILITY

GUI test automators know that testability issues often bedevil automation. A common problem that GUI test automators face is custom controls. *Custom controls* are controls that are not recognized by a given GUI test tool. The process of assessing and ensuring testability of our product with a given GUI test tool is thus:
1. Test compatibility of your test tool and development tools early.
2. Define standards for naming user interface elements.
3. Look for custom controls. If there are any, plan to provide testability support.
a. Ensure that the tool can recognize the control as an object with a name, class and location.

b. Ensure that the name used by the tool to identify the control is unique. It won't do if multiple windows or controls are given the same identifiers
c. Ensure that the tool can verify the contents of the control. Can it access the text of a text field? Can it determine whether a check box is checked? (This step doesn't apply to some controls, like push buttons, that don't have contents.)
d. Ensure that the tool can operate the control. Can it click a push button? Can it select the desired item from a menu? (This step doesn't apply to some controls, like static text fields, that can't be operated by the user.)

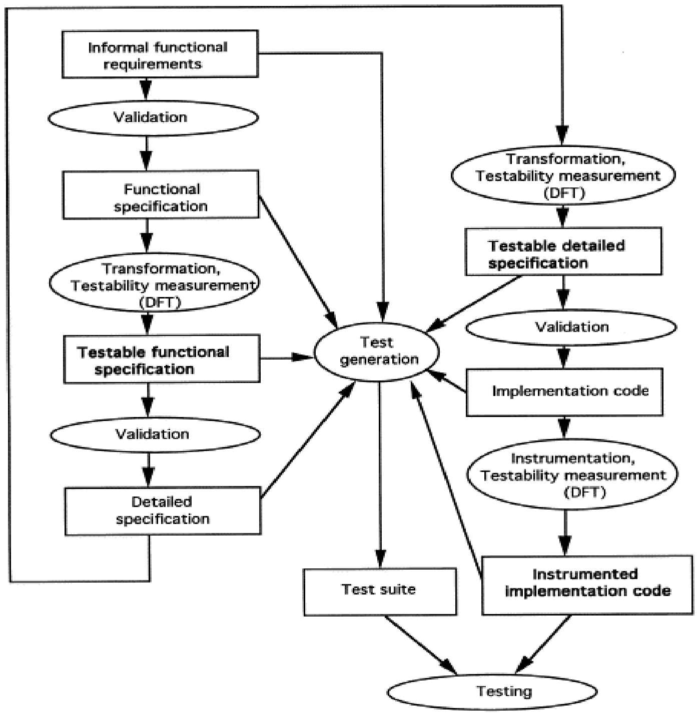
Informal functional requirements
Validation
Functional specification
Transformation, Testability measurement (DFT)
Testable functional specification
Validation
Detailed specification
Test generation
Transformation, Testability measurement (DFT)
Testable detailed specification
Validation
Implementation code
Instrumentation, Testability measurement (DFT)
Instrumented implementation code
Test suite
Testing

Fig 1 Testability-oriented communication software Development Cycle

Successful techniques often reduce veracity or require calls to testability interfaces.

7. OVERALL CONCLUSIONS

The benefits of object-oriented development are threatened by the testing burden that inheritance and information hiding place upon objects. In terms of both the ease of testing and the value of testing, object oriented software has been demonstrated to have lower testability than procedural implementations. To address these concerns, software design-for-testability is becoming important. There are numerous practical approaches to increasing the testability of systems during design. Of course, testability has to be considered throughout the entire software development project. This means to start with testable requirements as well as testability requirements, i.e. requirements related to the testability of the software product. Stakeholders for testability requirements include the customers and software users since testability is important to shorten maintenance cycles and to locate residual errors. Future empirical studies about the effects and economics of testability are necessary in order to establish testability engineering as a new discipline of software engineering.

8. BIBLIOGRAPHY

[1]. Y. Wang, G. King, I. Court, M. Ross and G. Staples, "On Testable Object-Oriented Programming" ACM SIGSOFT**,**Software Engineering Notes vol 22 no 4, pp. 84-90,July 1997.
[2]. Pressman,R.[1992] Software Engineering: A Practitioner's Approach (3rd ed.),McGraw-Hill international editions, pp 595-630.
[3]. Andy Carmichael, "Applying analysis patterns in a component architecture," TogetherSoft UK Ltd,1998, URL http://www.togethersoft.co.uk/.
[4]. Arthur J. Riel, "*Object-Oriented Design Heuristics*," Addison-Wesley, 1996
[5]. Binder, R.V.," Design for Testability with Object-Oriented Systems" Communications of the ACM, 1994. 37(9): p. 87-101.
[6]. Stefan Jungmay ,"Design for Testability", CONQUEST 2002 – 57
[7]. http://www.sciencedirect.com/science