

2004

Design of software components with increased testability

Yi-Tien Lin
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Lin, Yi-Tien, "Design of software components with increased testability" (2004). *Master's Theses*. 2669.
DOI: <https://doi.org/10.31979/etd.7zz4-mn63>
https://scholarworks.sjsu.edu/etd_theses/2669

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

DESIGN OF SOFTWARE COMPONENTS WITH INCREASED TESTABILITY

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Software Engineering

by

Yi-Tien Lin

December 2004

UMI Number: 1425471

Copyright 2004 by
Lin, Yi-Tien

All rights reserved.

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 1425471

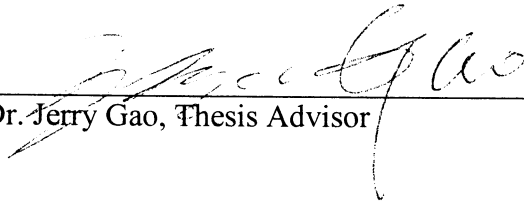
Copyright 2005 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

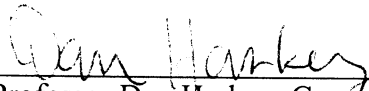
ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

© 2004
Yi-Tien Lin
ALL RIGHTS RESERVED

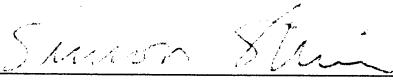
APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING


11/3/04

Dr. Jerry Gao, Thesis Advisor

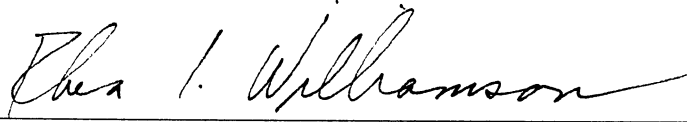

11-3-04

Professor Dan Harkey, Committee Member


11/3/04

Dr. Simon Shim, Committee Member

APPROVED FOR THE UNIVERSITY



ABSTRACT

DESIGN OF SOFTWARE COMPONENTS WITH INCREASED TESTABILITY

by Yi-Tien Lin

Testability is an important quality indicator of component-based software and its components since its measurement helps to understand and improve software quality. This thesis introduces an innovative concept of building testable components, and discusses its perspectives and a systematic approach to increase component testability. The major contribution of this thesis is its systematic method to construct testable components based on the given COTS or in-house components. With this method, testable software components could be validated in a cost-effective manner using a reusable component test bed and common component test framework. A case study and examples show a strong potential of using this solution to achieve component test automation. In addition, the development of a general component test bed is also reported.

Dedication

The author would like to dedicate this thesis to his father Hsien Chen Lin and his mother Li-Yuan. Thank you for your support and encouragement.

Acknowledgments

The author is deeply indebted to Dr. Gao for his encouragement and guidance in the preparation of this research. In addition, the author sincerely thanks the committee members, Professor Harkey and Dr. Shim, for their invaluable feedback and assistance with this thesis.

Table of Contents

CHAPTER

1. Introduction.....	1
The Problems.....	1
Purpose	2
Limitations.....	3
Organization	3
2. Review of the Literature.....	5
Background of the Study	5
Related Work.....	6
3. Perspectives of Testable Components	9
What Is a Testable Component?.....	9
Why Are Testable Components Needed?.....	10
How Can Component Testability Be Increased?.....	10
The Proposed Architecture Model for Testable Components	14
4. A Systematic Solution for Building Testable Components	16
Two Assumptions of Testable Components.....	17
Formats of the Component API and Test Case	18
Steps for Constructing a Testable Component	22
5. Component Test Framework and Supporting Environment.....	33
Component Test Framework	34
Supporting Environment	37
6. Application Examples of COMP_TEST Platform.....	40
Two Examples of Testable Components.....	40
The COMP_TEST Platform.....	43

7. Case Study	55
8. Conclusions and Future Work.....	65
Conclusions	65
Future Work.....	67
REFERENCES.....	68

List of Figures

Figure 3.1	Building Different Types of Testable Components	13
Figure 3.2	The Proposed Architecture Model for Testable Components.....	15
Figure 4.1	The Component API Format (Class Level).....	19
Figure 4.2	The Component API Format (Function Level)	20
Figure 4.3	The Test Case, Test Data and Test Result Data	22
Figure 4.4	The Test External Interface.....	23
Figure 4.5	The Test Mode Code	23
Figure 4.6	The Test Driver Implementation	25
Figure 4.7	The Test Entity Constructor	26
Figure 4.8	The Test Entity Setup Test Case Function.....	27
Figure 4.9	The Test Runner Run Test Case Function.....	28
Figure 4.10	The Test Internal Interface	28
Figure 4.11	The Component Test Wrapper	30
Figure 4.12	The Test Adapter Set Test Mode Function.....	30
Figure 4.13	The Test Adapter Setup Test Function	31
Figure 4.14	Test Adapter Run Test Function	31
Figure 5.1	The Component Test Framework.....	34
Figure 5.2	The Test Project Controller Program	36
Figure 5.3	The COMP_TEST High Level Architecture Diagram	38
Figure 6.1	A Simple Calculator Testable Component.....	41
Figure 6.2	A Binary Search Tree Testable Component	42
Figure 6.3	A Profiler Implementation Example	44
Figure 6.4	A List of all component APIs.....	45
Figure 6.5	A Test Case Work Space Implementation	47
Figure 6.6	An Example to Create Test Data	48
Figure 6.7	An Example to Create Test Data for Binary Search Tree Node.....	49
Figure 6.8	An Example to Create Test Case Using Pre-Compiled Test Data	50

Figure 6.9 An Example to Edit Test Script in the Comp_Test Platform..... 51
Figure 6.10 Commands to Perform Testing..... 53
Figure 6.11 A Result Collector Implementation Example..... 54
Figure 7.1 Distribution of Test Cases in Test Suites..... 60
Figure 7.2 Distribution of Test Drivers in Test Suites..... 61
Figure 7.3 The Code Size of Test Drivers 62
Figure 7.4 Summary of Test Efforts and Test Results 63

List of Tables

Table 3.1	Comparisons of Three Different Approaches.....	12
Table 4.1	A Template for the Component Profile.....	17
Table 7.1	The Amount and Distribution of Test Cases.	56
Table 7.2	The Amount and Distribution of Test Drivers in Manual Approach.	57
Table 7.3	The Code Size of Test Drivers in Manual Approach.	58
Table 7.4	Summary of Test Efforts and Bugs Report.	64

CHAPTER 1

Introduction

In the field of software engineering, the usage of third-party software components is gaining substantial interest due to the cost reduction that it brings. Many workshops have begun to use the component engineering approach to develop component-based software. Although there are many published papers addressing the issues of building component-based software, very few of them focus on the problems and challenges in testing and maintaining component-based software.

The Problems

Weyuker (1998) pointed out that the industry needs new methods to test and maintain software components to make them highly reliable and reusable, if they are deployed in diverse software projects, products, and environments. In the field of software engineering, engineers have encountered some new problems and challenges in the testing of software components and component-based software (Gao 1998, 2000, 2003). One such challenge is the question of how to build software components with sufficient testability. Voas & Miller (1995) considered testability an important quality indicator for component quality and reliability. In component-based software engineering (CBSE), component-based software is made of components; hence software testability is highly dependent on component testability. Consequently, increasing component testability is a primary key to improving and enhancing the testability of component-based software.

In the real world, engineers are seeking answers to the following questions:

1. What is software component testability?
2. How can we increase the testability of software components during component design?
3. How can we construct testable software components with a systematic approach?
4. How can we achieve test automation of software components by providing a reusable component test-bed?

Purpose

In this thesis, component testability is considered as a very important factor to increase component quality and reduce testing cost. Therefore, it is essential for component developers to construct deployable, executable, testable and manageable components. This thesis focuses on how to build software components with testability. First of all, we discuss the concept of component testability in terms of its basic requirements, properties, and benefits to help readers to understand its contributing factors. The literature contains different approaches to develop testable components, including components with built-in tests, testable beans, and components with test wrappers. Second, we explain a systematic approach to build testable components, including architecture, interfaces, and facilities. In addition, a well-defined test bed has been designed and developed to demonstrate the proposed test automation solution. The major contribution of this thesis is to provide a systematic approach to generate testable components that can be executed and validated in a reusable test-bed. The case study and application results indicate the proposed solution has a strong potential to reduce

component validation cost for component testers because it provides them a practice-oriented systematic solution to achieve black-box component test automation in a plug-in-and-test approach.

Limitations

Although the research work for the proposed approach has proven its potential to reduce component testing cost and increase software quality, there is still a long way to go. In this thesis, for the sake of narrowing down the scope, the current implementation has several limitations:

1. **Deployable and executable components**: we assume that all given COTS components are deployable and executable components. In addition, we use only small-sized components in the proposed approach.
2. **Single-thread components**: the implementation is not concerned with handling multi-threaded components in the reusable test bed.
3. **Single-API components**: we assume that each component has a single API. In addition, the API is the framework / library type. The implementation can not use other types of APIs, such as Graph User Interface (GUI) API, database access API, or communication / protocol API.

Furthermore, the current implementation is done by Java SDK 1.4, though the approach can be also easily implemented in other languages.

Organization

The thesis is structured as follows. Chapter 2 reviews the related work on designing software components with increased testability. It also explains the basic concept of software components. In Chapter 3, we compare different ways to design testable components to facilitate component test automation. Chapter 4 presents a systematic solution to construct testable components and/or convert COTS components to testable components. Chapter 5 reports the research efforts in building a reusable component test environment to support test automation of testable components. Chapter 6 describes two application examples. Chapter 7 provides the case study results based on the application examples. The conclusions and future work are given in Chapter 8.

CHAPTER 2

Review of the Literature

Background of the Study

What is testability? How can we measure and evaluate testability of software?

According to IEEE Standard (1990), the “testability” is: “the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met; the degree to which a requirement is stated in terms that permit the establishment of test criteria and performance of tests to determine whether those criteria have been met.”

Recently, there have been many papers addressing the concept of component testability and methods for building testability into software components. For example, Binder (1994) defined the concept of testability with two key facets: controllability and observability. In other words, “Removing obstacles to controllable input and observable output is the primary concern of design for testability.” From that perspective, Binder enumerated six primary testability factors: representation, implementation, built-in test, test suite, test tools and test process. The details can be found in Binder (1994). Voas & Miller (1995) considered software reliability as a three-piece puzzle: software testability, software testing and formal verification. They pointed out that software must have high testability to be highly reliable. But why is testability useful? First, testability can enhance testing by suggesting the testing intensity or estimating how difficult it will be to detect a software fault. Second, testability can give us confidence of correctness in fewer tests if we know the software will not hide faults. To extend the concept, Gao et

al. (2003) analyzed software testability based on five factors: understandability, observability, traceability, controllability, and test support capability. Their view contributed a lot to our proposed component automated testing solution. The details can be found in *Testing and Quality Assurance for Component-Based Software* (Gao et al., 2003).

Related Work

To build testability into software components, new systematic methods are needed. So far, there are a number of papers discussing how to increase testability of software components, which can be classified into three approaches. The first one is known as Built-In Tests (BITs) approach. The concept of embedding BITs in components has been using in hardware components for quite a long time. In a paper of Martins, Toyota and Yanagawa (2001), they used this method to construct self-testable software components. They presented an approach to improve component testability by integrating testing resources into it. Besides, Wang et al. (2000) implemented a systematic reuse method of BITs in object-oriented framework development. Similar to hardware components, their BITs components can be setup and executed under two modes: the normal and test modes. In the normal mode, a BIT object is nothing but a conventional object. However, in the test mode, the built-in test cases can be activated as member functions of software components. Each implemented test case, as part of test drivers, can automatically report test results.

The second approach is known as the BIT wrapper. Kropp, Koopman & Siewiorek (1998) suggested to create protective hardening “software wrappers” against

software component failures caused by exceptional inputs. However, a clearly BIT wrapper architecture was proposed by Edwards (2000). His paper focuses on a general strategy for automated black-box testing of software components. His approach can automatically generate component test drivers, black-box test data, and automatically or semi-automatically generate component BIT wrappers. By adding “Hooks” for BIT test components, a number of self-testing features can then be provided. The BIT wrapper contains the inner layer and outer layer. The inner layer, also called representation layer, is responsible for accessing component’s internals while the outer layer, called abstract layer, is responsible for checking clients’ obligations using a model from the inner layer. Edwards’s BIT wrapper approach facilitates automated black-box testing capability to software components.

The third methodology is known as testable beans introduced by Gao et al. (2002) to facilitate automated component testing. Different from the previous approaches, this approach constructs testable components based on a well-defined test interface to the test supporting framework. Gao et al. considered each testable bean has the following parts:

1. A component test interface supporting test operations
2. Built-in test code supporting the interactions between component APIs and the test interface
3. Component tracking interface for monitoring component operations and behaviors
4. Built-in tracking code for component tracking.

The research work reported in this thesis is an extension of previous work in Gao et al. (2000, 2002). It is also influenced by Edward’s BIT wrappers (2001). The basic

idea and contribution is to present a systematic approach to construct testable components based on a well-defined component architecture model, test interface, and supporting test framework. Using this methodology, we can convert a given COTS component to a testable component by automatic generated component wrappers. Component wrappers play as pluggable test harnesses to support test automation. With the proposed supporting test environment, there is strong potential to achieve the goal to test components in the plug-in-and-test manner.

CHAPTER 3

Perspectives of Testable Components

To generate high quality component-based software, developers must increase component testability. In this chapter, we propose a systematic method to build testable software components with testability so that they can be easily tested, controlled, observed, traced, and understood. The implementation work refers to the concept of testable beans from Gao et al. (2000, 2002). In addition, it focuses the discussion on testable beans by answering the following questions:

1. What is a testable component?
2. Why are testable components needed?
3. How can component testability be increased?

What Is a Testable Component?

A testable component is not only deployable and executable, but also testable. It should be constructed in a way to facilitate component testing and automation to reduce the validation cost of component testers and users. Unlike normal components, testable components must be constructed using a well-defined component architecture model and consistent test interface. In addition, they have the following basic requirements and features (Gao et al., 2003).

- Requirement #1: A testable component should be deployable and executable.
- Requirement #2: A testable component must be traceable by supporting basic component tracking capability. As defined in Gao et al. (2000), traceable

components are ones constructed with a built-in tracking mechanism for monitoring various component behaviors.

- Requirement #3: A testable component must provide a consistent, well-defined and built-in interface, called test interface, to support external interactions for software testing.
- Requirement #4: A testable component must include inner program code that facilitates component testing by interacting with external testing facilities or tools to support test set-up, test execution and test validation.

Why Are Testable Components Needed?

It is desired to find an approach to build software components with sufficient testability so that they are easily executed, traced, observed, controlled, and tested. Using testable components enhances component testability and achieves component test automation. As pointed out by Gao et al. (2003), the proposed systematic approach should be able to help in the following aspects:

1. Converting components to testable components easily.
2. Standardizing the test interface for components so that various test tools and facilities can be deployed and used more easily.
3. Generating component test drivers and stubs automatically.
4. Reducing the effort and cost of setting up component test beds and enabling component test automation in a plug-in-and-test manner.

How Can Component Testability Be Increased?

According to Gao et al. (2003), there are three different systematic mechanisms to construct testable beans by adding program codes and interfaces to support and facilitate component testing and program testing. Table 3.1 lists the three basic approaches to add a consistent testing capability into software components to increase component testability.

Framework-based testing facility Approach

Approach 1: Framework-based testing facility – In this approach, component engineers facilitate software components with program testing code according to the application interfaces of a given framework. This approach is simple and flexible to use, however, there are several drawbacks (Gao et al., 2003). Firstly, it requires a high programming overhead. Secondly, it relies on engineers' willingness to add the testing code due to its high overhead in programming. Moreover, this approach assumes that component source code is available. Therefore, it is difficult to deal with COTS components because usually they do not provide any source code to clients.

Built-in test Approach

Approach 2: Build-in tests – This approach requires component developers to add test cases and test drivers inside a software component as its parts. This causes a high programming overhead during component development. The major advantage of this approach is that tests are built-in inside components. Therefore, engineers can perform component tests without any external support from testing environment. The drawback of this approach is that components become more complex because of the additional

Different Perspectives	Framework-Based Testing Facility	Built-in Tests	Systematic Component Wrapping for Testing
Source Code	Needed	Needed	Not needed
Programming Overhead	Low	High	Very Low
Testing Code Separated from Source Code	No	No	Yes
Component Tests inside Components	No	Yes	No
Test Change Impact on Components	No	Yes	No
Standard Component Test Interface	Yes	No	Yes
Component Complexity	Low	Very High	High
Usage Flexibility	High	Low	Low
Component Change Impact on Component Testing Interfaces	No	Yes	No – If a dynamic wrapping method is used. Yes – If a static wrapping method is used.
Applicable Components	In-house components and newly developed components	In-house-built components and newly-developed component	In-house components and COTS as well as newly constructed components

Table 3.1

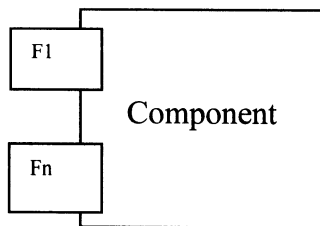
Comparisons of Three Different Approaches (Gao et al., 2003)

non-conventional functional features. In addition, not many types of tests can be built into components.

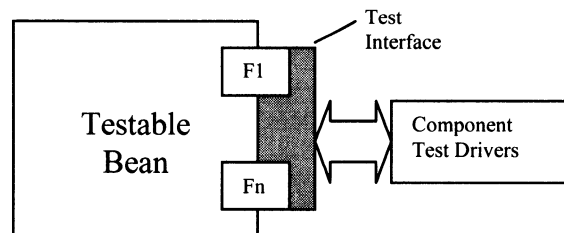
Automatic component wrapping for testing

Approach 3: Automatic component wrapping for testing - This approach converts a software component into a testable component by wrapping it to facilitate software testing. Compared with the first two methods, this approach has three advantages (Gao et al., 2003). First, its programming overhead is low because the wrappers are automatically created by the test framework. Next, it separates the testing code from the original source code of a component. Third, this method can be used for COTS components. The most important research issue is how to achieve this using a systematic way.

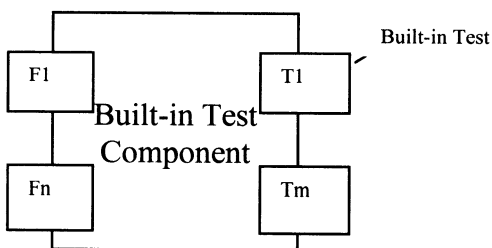
A SOFTWARE COMPONENT



A TESTABLE BEAN WITH INTERFACE



A BUILT-IN TEST COMPONENT



A COMPONENT WITH TEST WRAPPER

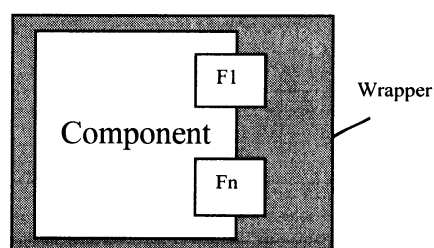


Figure 3.1

Building Different Types of Testable Components (Gao et al., 2003)

As shown in Table 3.1, each approach has its own advantages and limitations. In real practice, engineers need to use them together to support different types of testing in an effective and efficient manner.

As given by Gao et al. (2002), intuitively, a testable bean is a software component, which is designed to facilitate component testing. Figure 3.1 shows three types of testable components. The first one is a built-in test component consists of a set of component tests cases and test drivers. The testable bean is the second one which provides a consistent test interface to support testing. The third one shows a testable component which is created using the third approach. Edward (2001) described one way to implement this approach based the formal component specification language RESOLVE.

The Proposed Architecture Model for Testable Components

Figure 3.2 shows the architecture model for testable components proposed by this thesis. The idea is an extension of the previous work of Gao et al. (2000, 2002 & 2003). In this architecture model, a testable component consists of the following parts:

1. A component test adapter. It is a generated adapter to facilitate the interactions between the component internal test interface and diverse component API interfaces of each under test component.
2. Two well-defined test interfaces:
 - a. External component test interface. It is used as a standard test interface to interact with component test and management tools, such as test execution tool, and test management tool.

- b. Internal component test interface. It is used to interact with the original component APIs by a generated component test adaptor.
3. A generic built-in component test driver. It is pre-defined and implemented as a part of component test bed to support component testing based on the consistent external component test interface.

Clearly, the proposed testable component model here not only provides well-defined external test interface between components and testing tools, but also a clear picture about the functional elements of a component test wrapper. With this component architecture model we can easily construct testable components and convert COTS components into testable components in a plug-in-and-test approach. The detailed procedure and methods to construct testable components are given in the following chapter.

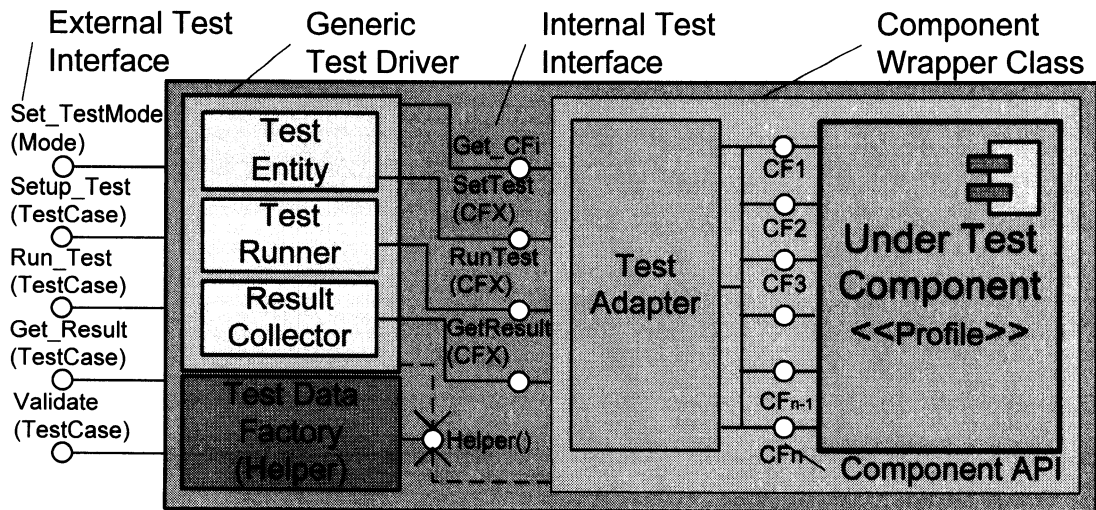


Figure 3.2

The Proposed Architecture Model for Testable Components

CHAPTER 4

A Systematic Solution for Building Testable Components

To increase component testability and support component test automation, we propose a new way to construct testable components with a well-defined architecture model and the standard component interfaces. The proposed approach also could be used to convert a given component into a testable component. Its motivation is to reduce the cost of component test harness to support diverse component APIs by regulating the testing interfaces to test tools.

Since COTS components could be developed based on different application logics using diverse component APIs and various technologies, they are classified into five basic groups:

1. Framework-based components. Components consist of a set of reusable classes.
2. Application components. Components implement as domain-specific reusable application modules.
3. Data access components. Components are written to access a data repository.
4. User interface components. Components are written as interface components to interact with users.
5. Communication components. Components are written to support a communication protocol over a network.

Two Assumptions of Testable Components

Currently, the proposed approach provides a framework to test COTS components through their component APIs. It has made two simple assumptions about the given COTS components.

1. Each component must come with a well-formatted component profile (Table 4.1).
2. Each component must be provided with a well-formatted component API.

Table 4.1

A Template for the Component Profile

General Information		Run-time Settings	
Cmp_ID	Cmp_Name	Work_Space	BIN_path
Cmp_Version	Last_Updated	EXE_Path	LIB_Path
Cmp_Vendor	Vendor_URL	LIB_Version	Export_Env
Bug_Report_URL	Contact_Info	File_Encoding	File_Separator
Tech_Support	Wrapper (Object)	Dependent Tech	
Programming Language		Operating Platform	
Name	Vendor	OS Arch	Name
Version	LIB_Path	Version	
Specification & API Documents		User-Defined Information	
Spec_Name	API_Name	User_Name	Country
Spec_Version	API_Version	User_Home	Language
Spec_Location	API_Location	TimeZone	

Formats of the Component API and Test Case

To support a component's test interface to interact with component test tools, we have developed a well-defined component test framework, which consists of a set of classes representing component APIs, test scripts, test cases, and test stubs as well as test data. The basic idea is to provide a mechanism for the component test framework to recognize and access the under test component API. On the other hand, the under test component must support that mechanism and provide a function to map test functions (from the component test framework) into actual component functions. To achieve the objectives of the mechanism, this thesis proposes a solution of formatting APIs and test data. Suppose the under test component knows how test functions are formatted, it would be easier to map test functions into actual functions. The same rule applies to test data. Consequently, in the second assumption, the provided well-formatted component API should be consistent with the component API format of the component test framework.

To illustrate the proposed solution of formatting API and test data, the three basic requirements are listed below:

1. The solution should use a universal data type to define the function signature of component APIs.
2. The solution should be able to handle recursive structures corresponding to fundamental programming languages in the API level.
3. The solution should provide a way to define different data types including primitive data types, user-defined class types and interface types.

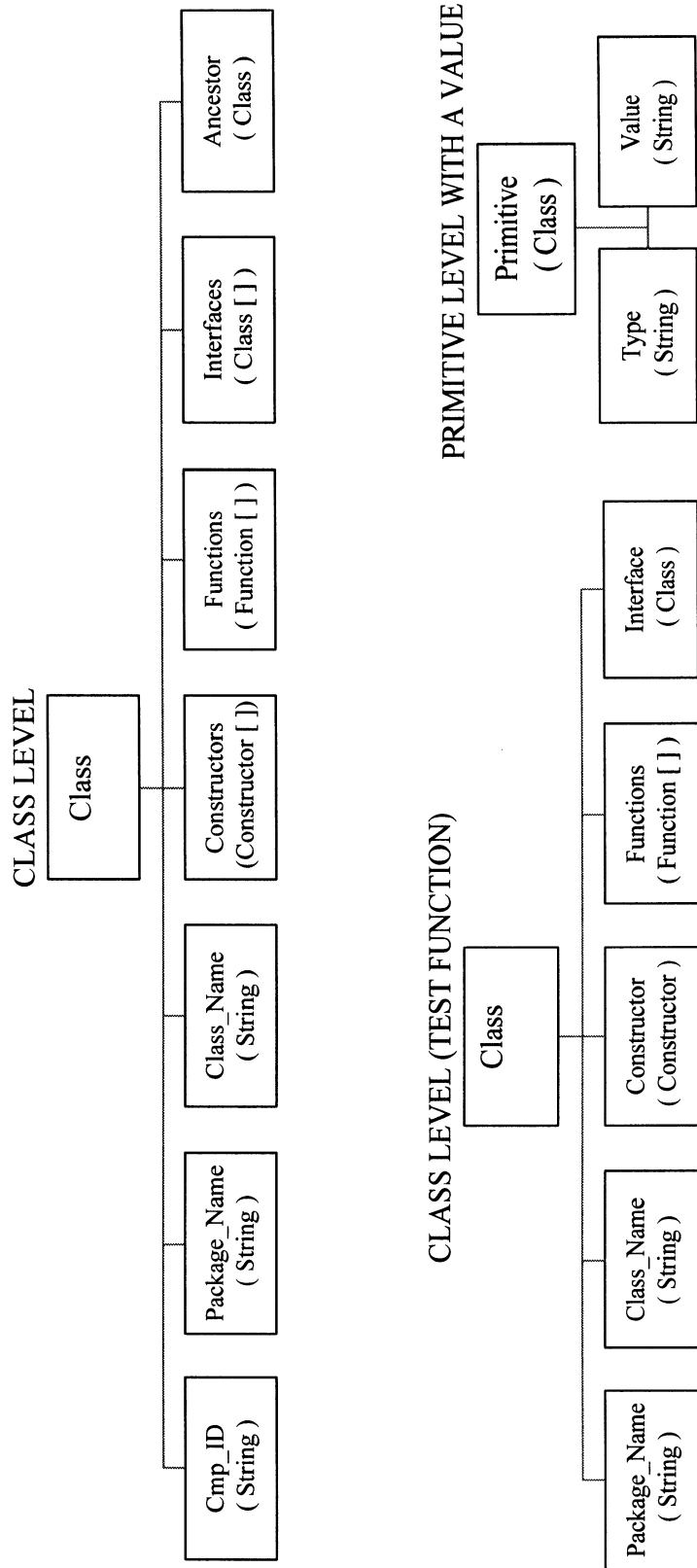


Figure 4.1

The Component API Format (Class Level)

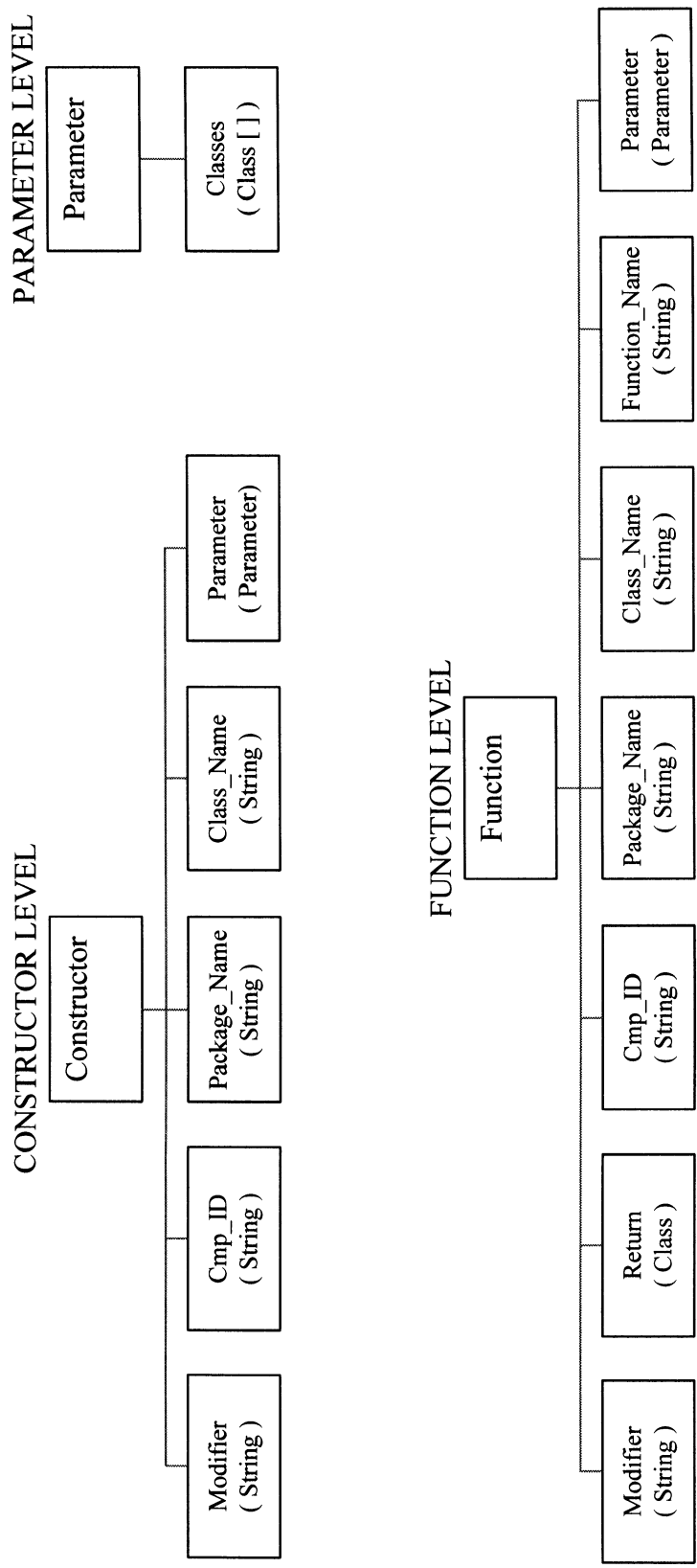


Figure 4.2

The Component API Format (Function Level)

The implementation of this thesis meets all three requirements above. It supports primitive data types, user-defined class types including programming language library and component classes, and interface types. The formatting of component function signatures is in two categories: one is for the original component API while the other is for test functions, which usually contain test data (Figure 4.1 and Figure 4.2). The test data values or object values could be set-up and accessed in three ways: a) interacting with a tester directly through a user-friendly interface, b) accessing a test repository, and c) accessing XML-based data files/scripts.

Figure 4.3 shows the design of test cases and its related programs. The format of test cases has been defined in the component test framework to contain three types of information:

1. Test management information: This refers to information that is used by test management systems. Typically, a test case ID and a version number are examples.
2. Test function and test data information: It specifies the information to set-up test functions and test data. In Figure 4.3, the “constructor” variable is an example. This variable is instantiated from the component API format to create a under test component class object. In our implementation, we assume test cases can be created even without the under test component class object.
3. Test running object information: After set-up a test case, the under test component will return outputs for the test function and test data. These outputs will be stored in the test case for execution or validation purposes.

The implementation of test cases has another issue: cloning. Since a test case can be executed for many times, we have the need to make each run a separate instance. Our approach is to make a deep copy of each test case. “Deep copy”, instead of shallow copy, clones an object including its associated mutable objects.

<pre>public class TestCase implements Cloneable { private String tcaseID; private String cmpID; private CnstrSign constructor; private MethodSign function; private ClassSign funcRtnType; private String tcaseType; private String testerID; private String tcaseVersion; private String tcaseDate; private String tcaseDesc; private TestData[] cnstrTData; private TestData funcRtnData; private TestData[] funcTData; // The following objects will be instantiated // after setup Test Case. private Object underTestObj; private Object funcObj; private Object funcRtnObj; private Object[] tdataObj; private TestResultData trData; public Object clone(){ ---- code to make a deep copy of this object ---- } }</pre>	<pre>public class TestData implements Cloneable{ private ClassSign classType; private Object valObj; private Class valClass; public Object clone(){ ---- code to make a deep copy of this object ---- } } public class TestResultData implements Cloneable{ private long startT=0; private long endT=0; private long durationT; private short result; private Object resultObj; private String reason; private String cmpID; private String tcaseID; public Object clone(){ ---- code to make a deep copy of this object ---- } }</pre>
---	--

Figure 4.3

The Test Case, Test Data and Test Result Data

Steps for Constructing a Testable Component

The common steps for creating a testable component based on a given component (including a COTS component) are given below.

1. Define the external component test interface.
2. Instantiate the external test interface.

3. Define the internal component test interface.
4. Instantiate the internal test interface.
5. Set up a reusable test bed.

<pre>public interface TExtInterface { public boolean setTestMode(boolean testModeValue); public int setupTest(TestCase testCase); public int runTest(TestCase testCase); public int validateTestResult(TestCase testCase); public TestResultData getTestResult(TestCase); }</pre>	<pre>public class TestDriver implements TExtInterface{ String cmpId; String[] cmpClassName; TIntInterface tii; // Test Internal Interface from components TestEntity testEntity; TestRunner testRunner; ResultCollector rtColl; --- A constructor to instantiate above variables --- public boolean setTestMode(boolean testModeValue){ if (testModeValue) // Go into test mode return tii.getCFi(cmpClassName); return false; // NOT in test mode } }</pre>
---	--

Figure 4.4

The Test External Interface

Figure 4.5

The Test Mode Code

Step 1: Define the component test external interface

Testable components must include consistent and standard test interfaces that regulate the testable component's interaction with testing tools to facilitate component testing. Each testable component has to (a) implementing the test internal interface that directly interacts with the original component API, and (b) supporting the test external interface that interacts with external test tools. Most of the test framework can wrap components with the test external interface automatically. In case it is not supported, components have to implement the test external interface as well.

Figure 4.4 shows the test external interface of a testable component. In it, test operations are activated in the test case level. These interface operations include:

Test mode: Each testable component has two operational modes: the test and normal modes. Figure 4.5 is an example. In the test mode, the component's functions are

executed based on its test interfaces. In the normal mode, the component is executed as its normal functions through its API.

Setup a test case: It helps in setting-up a test case including its test classes, test functions and test data. It processes the given test case from the external test interface, dynamically identifies the involved component functions, and sets up these functions for the test case.

Run a test case: First, it gets the set-up test functions with test data from the test case. Then, test environment information such as a timer is recorded. Next, it exercises test functions through the component test internal interface. Finally, test results are saved back to the test case.

Validate a test case: After execution, the actual test results will be saved in the test case. This operational function checks the actual results against the expected results for the test case. The possible validation results are: pass, fail, exception, and unknown.

Get test result from a test case: The result of each test case validation is saved as the Result Test Data format (Figure 4.3) in the test case. It can be used to generate test reports since it holds testing information from test case information, test environment information and test execution information to test validation information.

Step 2: Instantiate the test external interface

In Figure 3.2, we identify elements according to their functionality to instantiate the test external interface. These elements are:

1. **Test Driver:** It instantiates the test external interface. In our scenario, testing tools prepare test cases and run them through the test external interface. Actually, the

procedure is done by calling this generic test driver. Each component has a single test driver in the reusable test framework. In addition, each test driver has a component ID.

```
/*
   This function has the output values:
   return 1 if set up is ready.
   return 0 if failed.
   return -1 if insufficient test case information (an invalid test case)
*/
public int setupTest(TestCase testCase){
    return testEntity.setupTestCase(testCase);
}
public int runTest(TestCase testCase){
    return testRunner.runTestCase(testCase);
}
public int validateTestResult(TestCase testCase){
    return rtColl.validateTestResult(testCase);
}

public TestResultData getTestResult(TestCase testCase){
    return rtColl.getTestResult(testCase);
}
}
```

Figure 4.6

The Test Driver Implementation

2. **Test Entity**: The major task of the Test Entity is to set up test cases. It reads the settings of test functions and test data from a test case, prepares them into objects through the test internal interface before testing, and returns these objects to the test case. In short, the Test Entity must guarantee test cases can be executed under test. (Figure 4.7 & Figure 4.8)

```

public class TestEntity {
    TIntInterface tii;    // Test Internal Interface
    TDataClassHelper classHelper;    // A Helper class to get test data in class type.
    TDataObjectHelper objectHelper;    // A Helper class to get test data in object type.
    public TestEntity(TIntInterface tii){
        this.tii = tii;
        classHelper = new TDataClassHelper();
        objectHelper = new TDataObjectHelper();
    }
}

```

Figure 4.7

The Test Entity Constructor

3. **Test Runner**: This class is responsible for executing test cases. In order to do so, it gets the test class objects, test function objects and test data objects from a test case. These objects can be invoked for testing through the component test internal interface. It is also essential for this class to record the test environment information while testing. The actual results are saved in a standard format. (Figure 4.9)
4. **Result Collector**: After a test case is executed, it is also important to validate the actual test results against the expected results. The Result Collector can validate test results for a test case and generate test reports.

Step 3: Define the component test internal interface

As mentioned before, the component internal test interface is defined to adopt the original component API. Figure 4.10 shows an example of the test internal interface. The implementation is mainly dependent on the programming language.

```

public int setupTestCase(TestCase testCase){
    if (testCase == null) return -1;           // Invalid test case
    Object underTestObj = testCase.getUnderTestObj();
    MethodSign funcSign = testCase.getFunction();
    if (funcSign == null) return -1;         // No test function.
    String testClassName = funcSign.getPackageName() + "." + funcSign.getClassName();
    String testFuncName = funcSign.getMethodName();
    // Set up the under test object for running test functions
    if (underTestObj == null){
        CnstrSign underTestConstr = testCase.getConstructor();
        if (underTestConstr != null)        // Test case has specified the constructor
            underTestObj = objectHelper.getDataObject(underTestConstr);
        else                                // Default constructor
            underTestObj = objectHelper.getDataObject(testClassName);
        testCase.setUnderTestObj(underTestObj); // Set up the under test object to the test case
    }
    // Set up the expected output value object
    TestData funcRtnData = testCase.getFuncRtnData();
    ClassSign rtnData = funcRtnData.getClassType();
    if (rtnData == null) return -1;
    funcRtnData.setValClass(classHelper.getDataClass(rtnData)); // The signature of return data
    funcRtnData.setValObj(objectHelper.getDataObject(rtnData)); // The return data object
    testCase.setFuncRtnObj(funcRtnData.getValObj()); // Set up return data object to the test case
    // Set up the test data value objects
    TestData[] funcTData = testCase.getFuncTData();
    if (funcTData == null) return -1; // Invalid test cases
    Object[] tdataObjs = new Object[funcTData.length];
    for (int i=0; i<funcTData.length; i++){
        ClassSign tData = funcTData[i].getClassType();
        if (tData == null) return -1;
        funcTData[i].setValClass(classHelper.getDataClass(tData)); // The signature of test data
        Object tObj = objectHelper.getDataObject(tData); // The test data object
        funcTData[i].setValObj(tObj);
        tdataObjs[i] = tObj;
    }
    testCase.setTdataObj(tdataObjs); // Set up test data object to the test case
    Object funcObj = null;
    try { // Pass the test function and data to Internal Interface and get the actual component function.
        funcObj = tii.setTestCFX(testClassName, testFuncName, funcTData);
    }
    catch (Exception e){ return 0; } // Failed to get the actual component function.
    testCase.setFuncObj(funcObj); // Set up test function object to the test case
    return 1;
}
}

```

Figure 4.8

The Test Entity Setup Test Case Function


```

public int runTestCase(TestCase testCase){
    TestResultData trData = new TestResultData ();
    testCase.setTrData(trData);           // Set test environment information to Test Result Data
    trData.setCmpID(testCase.getCmpID());
    trData.setTcaseID(testCase.getTcaseID());

    Object underTestObj = testCase.getUnderTestObj(); // Get the test class object from the test case
    if (underTestObj == null) return -1;
    Object funcObj = testCase.getFuncObj();           // Get the test function object from the test case
    if (funcObj == null) return -1;
    Object funcRtnObj = testCase.getFuncRtnObj();    // Get the expected result object from the test case
    Object[] tdataObj = testCase.getTdataObj();      // Get test data from the test case
    Object resultObj = null;
    trData.setStartT(System.currentTimeMillis());    // Set the start testing time
    try {
        // Call Test Internal Interface to run the test function.
        resultObj = tii.runTestCFX(underTestObj, funcObj, tdataObj);
    }
    catch (Exception e){
        trData.setEndT(System.currentTimeMillis());
        trData.setReason(e.toString());              // If an exception happens, put it in the result
        return 0;
    } // Failed to run test function.
    trData.setEndT(System.currentTimeMillis());    // Set the end testing time
    trData.setResultObj(resultObj);                  // Return the actual result to the test case
    return 1;
}

```

Figure 4.9

The Test Runner Run Test Case Function

```

public interface TintInterface {
    public boolean getCFi(String[] className);
    public Object setTestCFX(String className, String funcName, TestData[] testData)
        throws TestException;
    public Object runTestCFX(Object underTestObj, Object funcObj, Object[] testData)
        throws TestException;
}

```

Figure 4.10

The Test Internal Interface

Get Component Functions: By specifying which component classes will be using for testing, this operation instantiates every constructor and function of the selected component classes and maintains links or pointers with them. The component will then be in the test mode. Only when the test external interface calls to cancel the test mode will the component constructor and function instances be in the pool. Later, by supplying a component API signature, the pool would return the specified component function for it.

Setup Test Function: This operation seeks particular component function in the pool and returns the test function as an object. The candidate component function should have exactly the same component API signature with that of the test function.

Run Test Function: This operation directs the test class object to run test functions with test data. The detailed implementation of this component internal test interface will be done in the next step.

Step 4: Instantiate the Test Internal Interface

The instantiation of the component test internal interface is known as a component test wrapper (Figure 4.11). The actual implementation, as in Figure 3.2, is a Test Adapter. The Test Adapter usually is dependent on the used technology of the component, for example, the underlying programming language. However, it is totally possible to reuse a Test Adapter for the same programming language. Through the Test Adapter, test functions can be set up dynamically. Therefore, the impact of changing code inside the component API is very small to the test framework. In Figure 4.12, it shows an implementation of the Test Adapter.

```

public class BSTWrapper implements TIntInterface, Serializable {
    BSTAdapter testAdapter;
    public boolean getCFi(String[] compClassName) {    // Go into test mode
        try {
            testAdapter = new BSTAdapter(compClassName);
        }
        catch (TestException e){
            return false;
        }
        return true;
    }
    // Set up a component function
    public Object setTestCFX(String className, String funcName, TestData[] testData)
        throws TestException {
        try {
            return testAdapter.setupCmpMethod(className, funcName, testData);
        }
        catch (Exception e){
            throw new TestException(e.toString());
        }
    }
    // Run a component function
    public Object runTestCFX(Object underTestObj, Object funcObj, Object[] testData)
        throws TestException {
        try {
            return testAdapter.runCmpMethod(underTestObj, funcObj, testData);
        }
        catch (Exception e){
            throw new TestException(e.toString());
        }
    }
}

```

Figure 4.11

The Component Test Wrapper

```

public BSTAdapter (String[] cmpClassNames) throws TestException {
    cmpCnstrs = new ArrayList(); // All the component constructors
    cmpMethods = new ArrayList(); // All the component functions
    if (cmpClassNames == null)
        throw new TestException("Invalid component class names");
    for (int i=0; i < cmpClassNames.length; i++){
        if (!setCmpClass(cmpClassNames[i])) // instantiate constructors and functions into the pool
            throw new TestException ("Failed to get component classes");
    }
}

```

Figure 4.12

The Test Adapter Set Test Mode Function

```

public Object setupCmpMethod (String className, String methodName,
                             TestData[] testData) throws TestException {
    String cmpMethodName = className + "." + methodName; // Test Function name
    Class[] paramClasses;
    if (testData == null) paramClasses = new Class[0];
    else {
        paramClasses = new Class[testData.length]; // Get test data in class types for checking signature
        for (int i = 0; i < testData.length; i++) {
            paramClasses[i] = testData[i].getValClass();
        }
    }
    // Search for the mapped API signature.
    if (!(cmpMethods == null)) {
        for (int i = 0; i < cmpMethods.size(); i++) {
            Method m = (Method) cmpMethods.get(i); // A component function name in the pool
            String mName = m.getDeclaringClass().getName() + "." + m.getName();
            if (cmpMethodName.equals(mName)) { // Function names are the same
                Class[] paramTypes = m.getParameterTypes();
                if (paramTypes.length == paramClasses.length) {
                    if (paramClasses.length == 0) return m; // Check for length of parameters
                    boolean isPassed = true;
                    for (int j=0; j < paramTypes.length; j++) {
                        if (!(paramTypes[j].equals(paramClasses[j]))) // Check for parameter class types
                            isPassed = false;
                    }
                    if (isPassed) return m; // Return the component function for the set up test function
                }
            }
        }
    }
    return null; // If not found.
}

```

Figure 4.13

The Test Adapter Setup Test Function

```

public Object runCmpMethod (Object underTestObj, Object funcObj,
                           Object[] testData) throws TestException {
    Method testFunc = (Method) funcObj;
    try {
        return testFunc.invoke(underTestObj, testData);
    }
    catch (Exception e) {
        throw new TestException(e.toString());
    }
}

```

Figure 4.14

The Test Adapter Run Test Function

Step 5: Setup for the reusable test bed

To achieve the objective of plug-in-and-test, we propose a two-stage scenario for development and packaging testable components:

1. In the development stage: The test definition library including the component profile and API must be imported. A graphic user interface helper program can regulate the test definition library classes as shown in our application examples. The outputs of the component profile and API are saved as files.
2. In the packaging stage: A testable component has two additional classes for the test internal interface: the component test wrapper and test adapter classes. In addition, it writes a property file to indicate the file names and locations where the component profile and API have been saved.

Once the testable component has been plugged-in, the test framework first reads the property file from it and restores the component profile and API to their corresponding class types. Next, the wrapper class is cast by the test internal interface. Finally, the test framework wraps the test external interface and starts testing.

CHAPTER 5

Component Test Framework and Supporting Environment

To achieve test automation for components, this thesis has developed a test automation solution for software components. The objective is to provide a cost-effective black-box test environment to support test automation for components in a plug-in-and-test manner. The focus is to provide users and developers a simple and easy way to validate components. The component test automation solution consists of three parts:

1. The proposed systematic solution to construct testable components or convert components to testable components.
2. A well-defined component test framework.
3. A test automation platform, known as **COMP_TEST**.

Among them, we have discussed Part 1 in Chapter 3 and Chapter 4. The whole picture of testable component has been explained in details by the given program source code. This chapter is for Part 2. Our focus is on the design for the component test framework and the component supporting test environment. Later, in Chapter 6, we demonstrate the advantages of using the proposed solution for Part 3. The implemented test automation platform, **COMP_TEST** reduces much coding effort for test engineers by the well-designed graphic user interface and reusable test data mechanism. The proposed solution can also be adapted to web-based architecture easily.

Component Test Framework

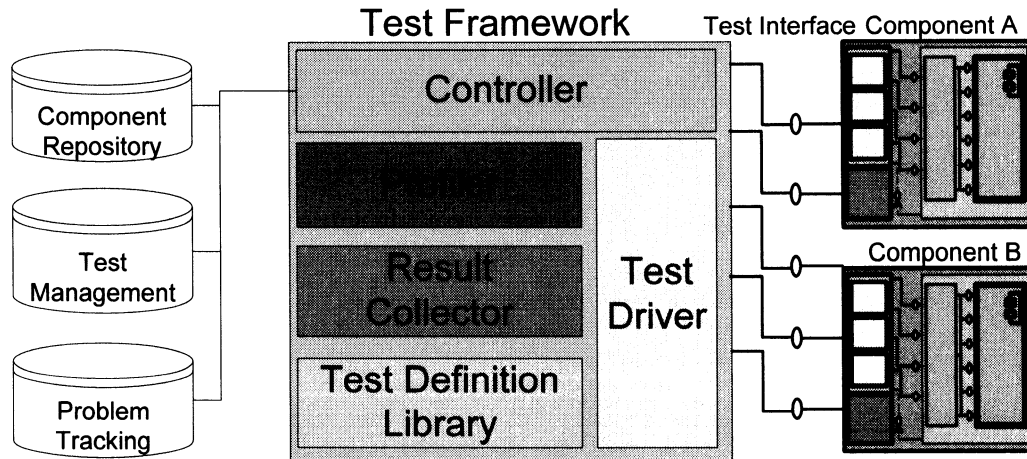


Figure 5.1

The Component Test Framework

As shown in Figure 5.1, the component test framework consists of the following parts:

Controller – The test project controller (Figure 5.2) controls the work flow and sets the entry point into a testable component. Operations to the test external interface are dominated by this controller program. It also accesses test cases, test scripts or test suites from external test tools or repository. The detailed procedures to execute test cases are described later in this chapter.

Profiler – The profiler handles and manages all component profiles and APIs. It is one of our assumptions that testable components must support the component test framework. In addition, the profiler uses certain logic to manage component profiles. For example, the profiler can show a group of component APIs for a test model.

Test driver – As mentioned in the previous sections, the test driver implements the test external interface for use by the control program. In the component test framework, a test driver automatically wraps a testable component and interacts with its component test wrappers. Therefore, the control program is able to manage and schedule tests by accessing each test driver.

Result collector – A reusable program to validate test results and generate test reports. An informative test report will help test engineers to analyze test results and identify possible software problems.

Test definition library – The component test framework defines a set of classes to represent essential test elements. These test definitions should be transferred to test requirements for building testable components. All the component profile, API, test case, test script, test suite, and test internal/external interfaces are our test definitions.

Procedures to run testing

Although different implementations can produce various procedures to the test framework, they can be summarized in the following steps to perform testing:

1. **Importing testable components.** In this step, the test framework gets component profiles, APIs and component test wrapper objects from testable components. The sources of testable components are from external COTS component packages, in-house built components or a component repository. When components have been plugged-in, the profiler program manages all the profiles and APIs for the test framework.

2. Selecting test cases. The next step is to select test cases. In general, testers design test cases and analyze test coverage before actual testing. The test framework has a work space that can be used to write test cases. If it is supported by external test tools, the test framework can produce test cases for components automatically.

```
public class TPController {
    private String projID = ""; // Test project ID
    private DBAccess dbAccess; // Database connection
    private Profiler profiler; // Profiler used in this test project
    private Hashtable testDrivers; // All generic component drivers (each component has a driver)
    private Hashtable allTestCases; // All test cases associated with this test project
    private ArrayList testCases; // Actual test cases to be setup/ run / validated
    private Hashtable allTestScripts; // All test scripts associated with this test project
    private ArrayList testScripts; // Actual test scripts to be executed.
    private ResultCollector rtColl; // Can be used to validate test cases and generate test reports
    private Hashtable testData; // All reusable test data
}
```

Figure 5.2

The Test Project Controller Program

3. Setting components to the test mode. Once the test cases have been selected, the test framework will set the under test components to the test mode. It enables component testing using the selected test cases.
4. Setting up the test cases. The selected test cases are then be set up with the test data and the expected results through the test external interface.
5. Running the test cases. The test framework can execute a test case once or multiple times by calling the test external interface. Each run of a test case is a separate instance object in the test framework. Therefore, it might obtain different test results.

6. Validating actual test results against expected results. After executing a test case, the actual test results are saved but not validated. For the validating operation, the test framework needs to call the test external interface to validate each test case.
7. Getting test results and generating test reports. The final step is to generate clear and subject-oriented test reports for testers. Testers analyze test results from the test reports.

Supporting Environment

To achieve component test automation, component testers need a component test supporting environment. In this thesis, the proposed high level system architecture is in Figure 5.3.

As discussed in Chapter 4 (Figure 4.1 & Figure 4.2), the proposed solution has designed a system to define component APIs in a universal data type – the string data. The system is now extended to have the capabilities of handling test data. It supports all the three data types – primitive types, class types and interface types. In this section, the COMP_TEST platform employs this test data formatting system as the logic of the client side presentation level.

The implemented functions of COMP_TEST include:

1. **API List**: A User Interface (UI) that lists all public component functions and specifications for testing. It also serves as an API helper to generate the component API file.

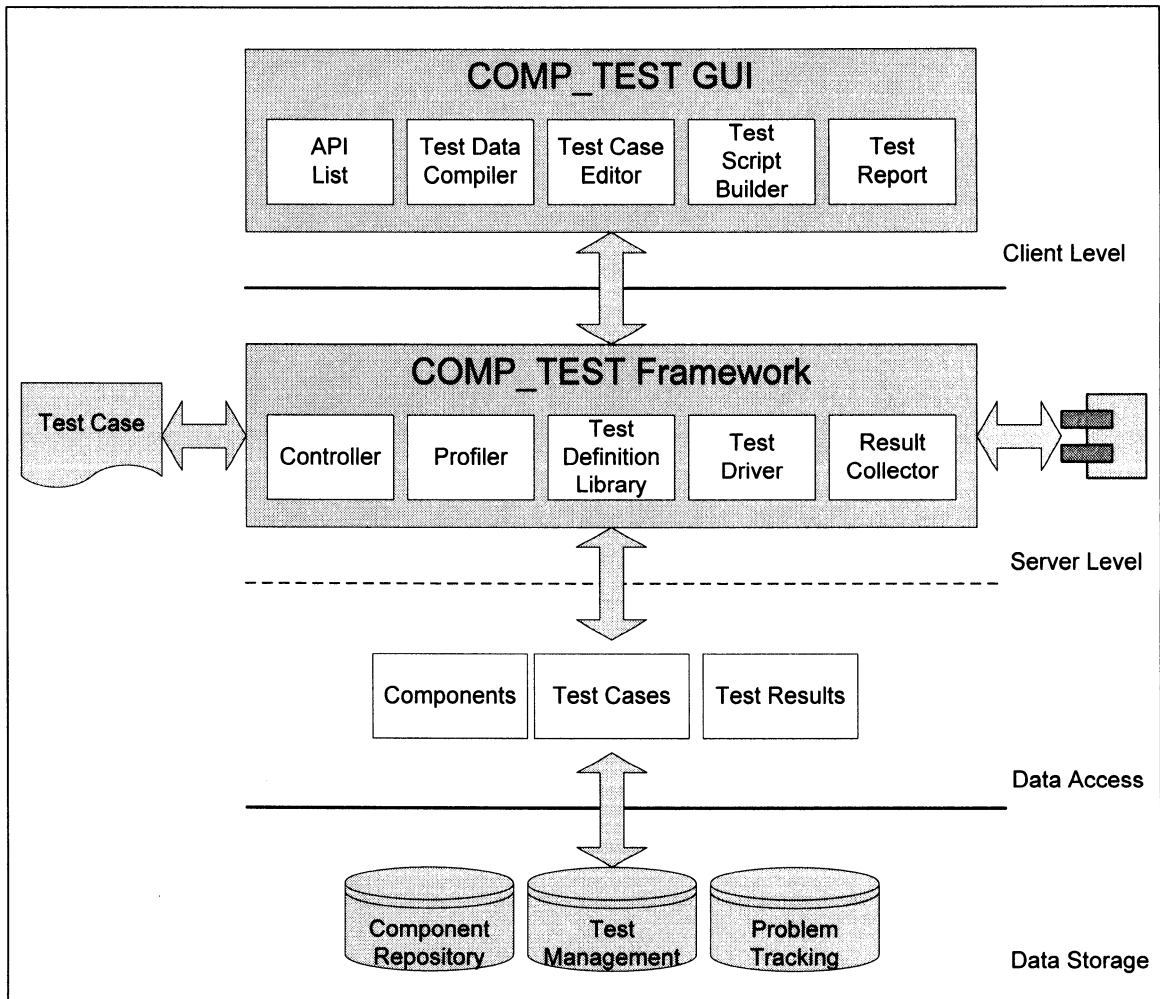


Figure 5.3

The COMP_TEST High Level Architecture Diagram

2. **Test Data Compiler**: To prevent addition testing cost, COMP_TEST is designed to reuse test data. In this UI, users can create new test data or extend them from the existing test data. By carefully planning, testing cost will be reduced greatly if the platform can reuse test data effectively.
3. **Test Case Editor**: This is the UI for testers to write new test cases or load test cases from a test repository or other sources, such as from XML files.

4. **Test Script Builder**: Test Script, in the implementation, indicates a set of test cases in a particular order. The COMP_TEST controller directly runs test scripts instead of test cases for the intentions to run more test cases at the same time.
5. **Test Result Collector**: This UI allows Testers to review the test results of each executed test case.

CHAPTER 6

Application Examples of COMP_TEST Platform

To demonstrate the proposed test automation solution, this chapter presents the applications of testable components in our component test bed. The intentions of this chapter are to:

1. Apply the systematic approach to construct testable components.
2. Develop a reusable test bed (platform) for testable components to achieve plug-in-and-test scenario.
3. Build the fundamental parts for the case study in the next chapter.

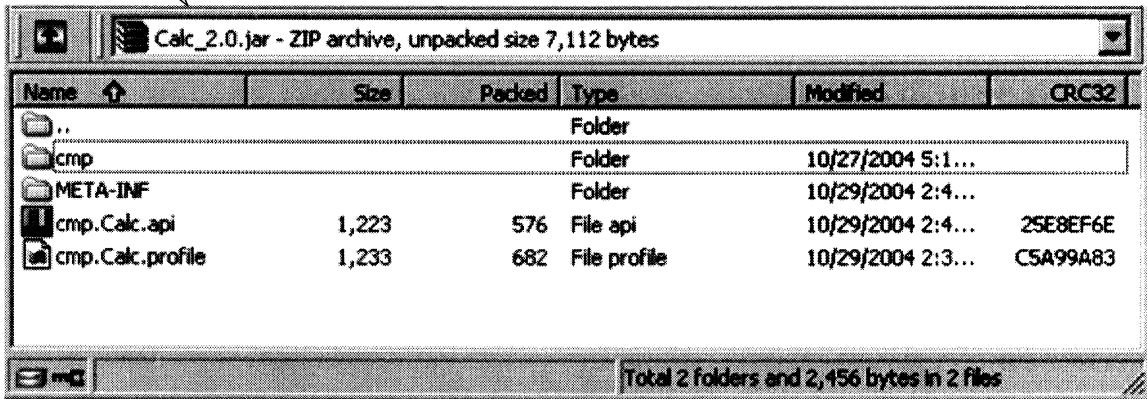
We shall examine the application examples by two steps. In the first step, we illustrate two testable components: one is a simple calculator component and the other is a binary search tree component. In the second step, we will see how to test components in the COMP_TEST platform.

Two Examples of Testable Components

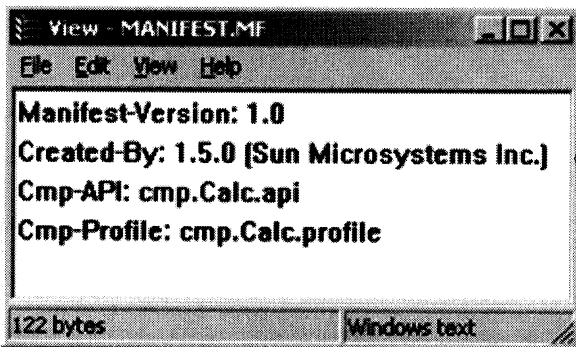
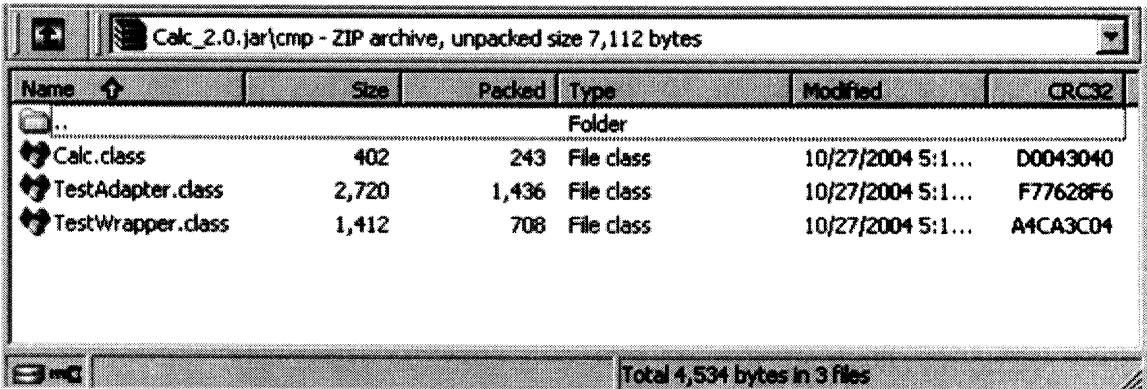
In this part, we selected two Java testable components as the examples.

1. Figure 6.1 shows a simple calculator (Calc) testable component. Originally, this component contains only one class – the “Calc.class.” However, to be testable, the vendor has to implement the test internal interface for this component. Hence, the “TestAdapter.class” and “TestWrapper.class” are added. Besides, it also required to supply the component profile and component API files. Finally, the “MANIFEST.MF” text file records the setting information.

The Simple Calculator testable component comes with a component profile and component API.



Inside the "cmp" package, the "Calc.class" is the main component class. The "TestAdapter.class" and "TestWrapper.class" are implemented by the proposed approach.

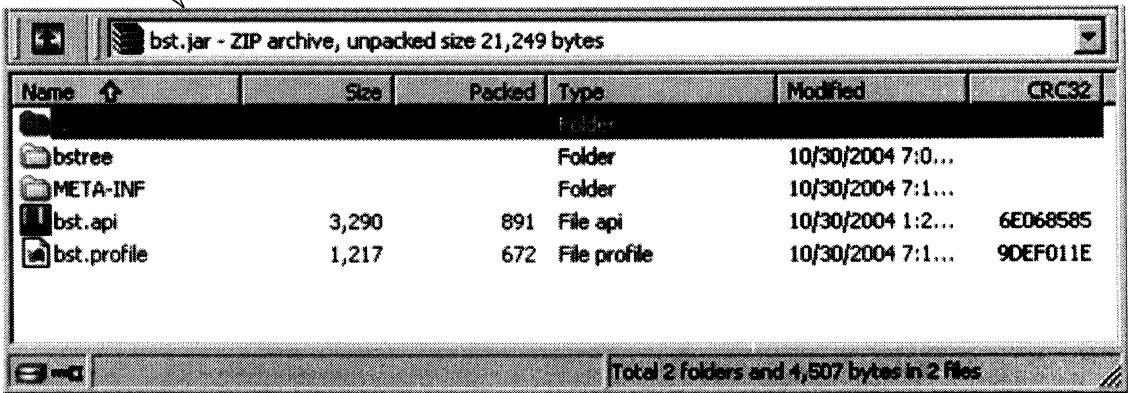


The Manifest file tells where the component profile and API files are.

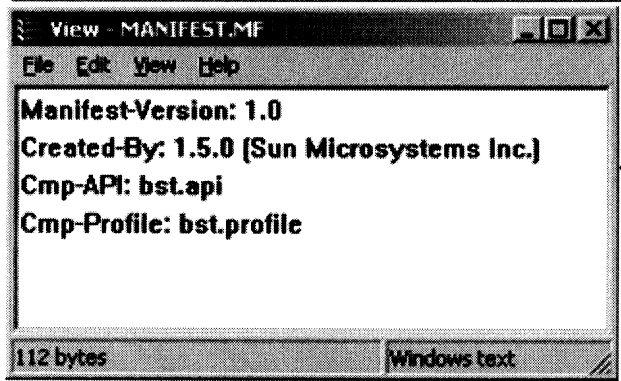
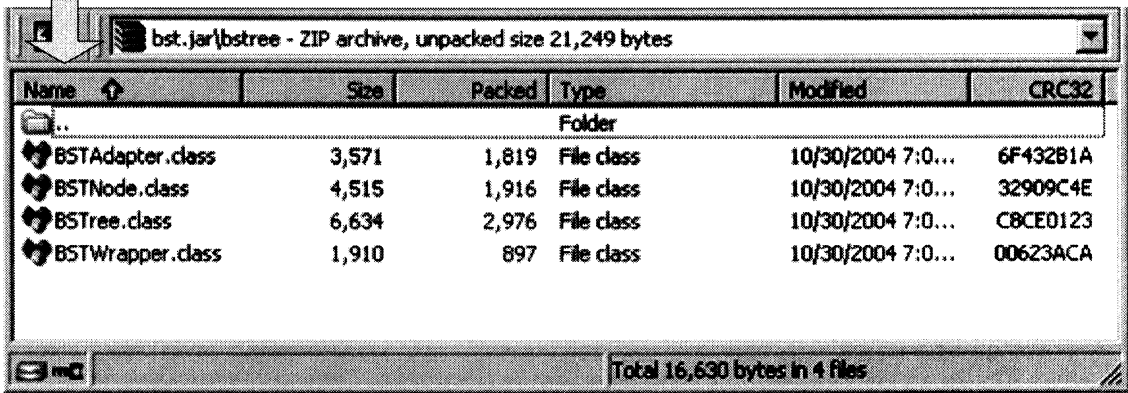
Figure 6.1

A Simple Calculator Testable Component

The Binary Search Tree testable component comes with a component profile and component API.



Inside the "bstree" package, the "BSTree.class" and "BSTNode.class" are the original component classes. The "BSTAdapter.class" and "BSTWrapper.class" are implemented by the proposed approach.



The Manifest file tells where the component profile and API files are.

Figure 6.2

A Binary Search Tree Testable Component

2. Figure 6.2 shows a binary search tree (BST) component.

This component has two classes: the “BSTree.class” and “BSTNode.class.” Similar to the Calc component, the “BSTAdpater.class” and “BSTWrapper.class” are added to implement the test internal interface. It also provides the “bst.profile” as the component profile and the “bst.api” as the component API.

In these two examples, the Test Adapter and Test Wrapper classes are reusable among components based on the same programming language. It takes no additional effort to implement the test internal interface in our proposed approach.

The COMP TEST Platform

This part demonstrates how to test components in the Comp_Test Platform. In general, the procedures to perform testing in the COMP_TEST platform are three:

1. Importing testable components into the COMP_TEST platform.
2. Editing test cases and test scripts for testing components.
3. Running commands to set under-test components to the test mode, setup test cases, run test cases, validate test cases and get results from all test cases.

Procedure 1: importing testable components.

Both the Calc and BST components are packaged as JAR files with no source code as in the examples. From the menu, COMP_TEST imports these two components by accessing the JAR files. Immediately, the profiler of the test framework manages the testable component profiles and APIs in a tree structure, as shown in Figure 6.3. Users can add, edit or modify component functional signatures in the work space of APIList as shown in Figure 6.4.

The Profile and API information of the 1st Binary Search Tree Component. There are two folders under BST component:

- 1) Info: the component profile information. (the profile template is in Table 2)
- 2) API: the component API information.

The API folder contains:

- 1) "Test Wrapper" indicates the component test wrapper object. Each testable component has to provide one test wrapper object.
- 2) bstree.BSTree: the main tree class API of the component.
- 3) bstree.BSTNode: the tree node class API of the component.

The API information of the Binary Search Tree component lists all the public functions and constructors.

The other testable component, Simple Calculator Component, in the Comp_Test Platform. The API information shows that this component has:

- 1) One component test wrapper. The wrapper object is in cmp.TestWrapper
- 2) Only class. The class is "cmp.Calc" which contains 1 constructor (Calc ()) and 4 functions inside.

There are 2 Under Test Components

```

Testable Bean ID: BST
  Total Constructors:3
    Functions:17
Testable Bean ID: Calc_2.0
  Total Constructors:1
    Functions:4
  
```

The status of all testable component profile information in the Comp_Test platform.

Figure 6.3

A Profiler Implementation Example

API List | Test Data Compiler | Test Case Editor | Test Script Builder | Result Collector
Component API Browser

Comp #	Package & Class	Modifier	Return	Constructor / Method
BST	bstree.BSTree	public	bstree.BSTNode	getMaximum ()
BST	bstree.BSTree	public	int	getMinimum ()
BST	bstree.BSTree	public	String	inorder ()
BST	bstree.BSTree	public	boolean	isEmpty ()
BST	bstree.BSTree	public	void	makeEmpty ()
BST	bstree.BSTree	public	String	postorder ()
BST	bstree.BSTree	public	String	preorder ()
BST	bstree.BSTree	public	bstree.BSTNode	removeMaximum ()
BST	bstree.BSTree	public	bstree.BSTNode	removeMinimum ()
BST	bstree.BSTree	public	bstree.BSTNode	removeNode (bstree.BSTNode)
BST	bstree.BSTree	public	bstree.BSTNode	removeNode (java.lang.Comparable)
BST	bstree.BSTree	public	void	insertNode (BSTNode)
Calc_2.0	cmp.Calc	public	Calc ()	
Calc_2.0	cmp.Calc	public	int	sum (int, int)
Calc_2.0	cmp.Calc	public	int	difference (int, int)
Calc_2.0	cmp.Calc	public	int	product (int, int)
Calc_2.0	cmp.Calc	public	int	division (int, int)

The table lists public constructors and functions of all testable components in the Comp_Test Platform.

Users can add / edit / modify component API in this workspace.

API Editor & Details

This is?

Component ID:

Modifier:

Return Type:

Package Name:

Class Name:

Method Name:

Parameter #1:

Parameter #

Data(Class) Type

New Save Cancel Delete

Figure 6.4
A List of all component APIs

Procedure 2: editing test cases and test scripts.

According to different types of test data, the test case examples have two:

1. Test cases with only primitive type test data.

Figure 6.5 demonstrates how to edit such test cases.

2. Test cases with user-defined class and interface type test data:

The proposed examples are supposed to answer the following two questions:

- a. Where and how can users create user-defined class and interface type test data?
- b. How can users use these test data in test cases?

For the first question, the COMP_TEST provides a Test Data Compiler work space in helping users to create class type or interface test data (Figure 6.6 & 6.7). Currently, users can create test data from: (a) the standard Java SDK library, (b) the under-test testable component, and (c) other testable components

For the second question, the COMP_TEST uses the same concept from the Test Data Compiler. All the available user-defined test data will show up in the pop-up list in the test data table (Figure 6.8).

Figure 6.9 shows how to edit a test script.

Procedure 3: executing commands to perform testing.

The COMP_TEST platform calls the test external interface by commands (Figure 6.10).

These commands are embedded in the menu.

1. “Test Mode”: this command sets all under-test testable components to the test mode.
2. “Set Up”: the COMP_TEST will setup test cases in the selected test scripts.
3. “Run!”: it runs all test cases in the selected test scripts.

API List | Test Data Compiler | Test Case Editor | Test Script Builder | Result Collector

Component Test Case Browser

TC Case #	Comp #	Package & Class	Method	Expected
TC_Calc_9	Calc_2.0	cmp.Calc	product ((int)999, (int)0)	(int)0
TC_Calc_8	Calc_2.0	cmp.Calc	product ((int)1, (int)2)	(int)2
TC_Calc_7	Calc_2.0	cmp.Calc	difference ((int)0, (int)-999)	(int)999
TC_Calc_6	Calc_2.0	cmp.Calc	difference ((int)999, (int)0)	(int)999
TC_Calc_5	Calc_2.0	cmp.Calc	difference ((int)1, (int)2)	(int)-1
TC_Calc_4	Calc_2.0	cmp.Calc	sum ((int)999, (int)-999)	(int)0
TC_Calc_3	Calc_2.0	cmp.Calc	sum ((int)0, (int)-999)	(int)-999
TC_Calc_10	Calc_2.0	cmp.Calc	product ((int)0, (int)-999)	(int)0
TC_Calc_2	Calc_2.0	cmp.Calc	sum ((int)999, (int)0)	(int)999
TC_Calc_1	Calc_2.0	cmp.Calc	sum ((int)1, (int)2)	(int)3

Test Case Editor & Details

Test Case: TC_Calc_11

Component: Calc_2.0

Package: cmp

Class: Calc_0

Method: product (int, int)

Return type: int

Constructor Data: Class Type Value

Expected Input: Class Type Value

Test Script: Black-box

WriteBy: Yi-Tien Lin

Version: 2.0

Doc Date: 11/11/04

Description: Test Product function using two boundary values.

Expected Input: Class Type Value

Test Script: int 999

Input Script: int -999

The table lists all the test cases for testing. Each test case has a unique ID.

Users can add / edit / modify test case in this workspace. In this example, the test case is going to be used to test the "product" function of the Simple Calculator component. This "product" function accepts 2 int values as the parameters and returns another int value for the result. Users edit the test data and the expected result in the lower right table.

Figure 6.5

A Test Case Work Space Implementation

This example shows how to create test data. The Test Data ID is assigned by the user to identify test data. Once the user has finished editing test data, the COMP_TEST platform will compile test data automatically. Later, the user can use these compiled test data in their test cases.

Test Data Editor & Details

New Save Cancel Delete

Source: JAVA SDK

Test Data#: Integer (5)

Component#: Java SDK 1.4

Package: java.lang

Class: Integer

Interface? Cast by Interface

Component#: Java SDK 1.4

Package: java.lang

Class: Comparable

Constructor Data	ClassType	Value
Parameter #1	int	5

This Test Data is an Integer object which is constructed by a primitive int value of 5. The Integer object implements the Comparable Interface.

Figure 6.6

An Example to Create Test Data

This example shows how to create test data for the Binary Search Tree Node. In it, the constructor of the BSTNode accepts one Comparable Interface object. Since we have created such an object in the last procedure, the COMP_TEST platform shows the test data in the pop-up list for selection. By selecting the "Integer(5)" test data in the constructor table, the user specifies that the BSTNode should be constructed by the Integer class object.

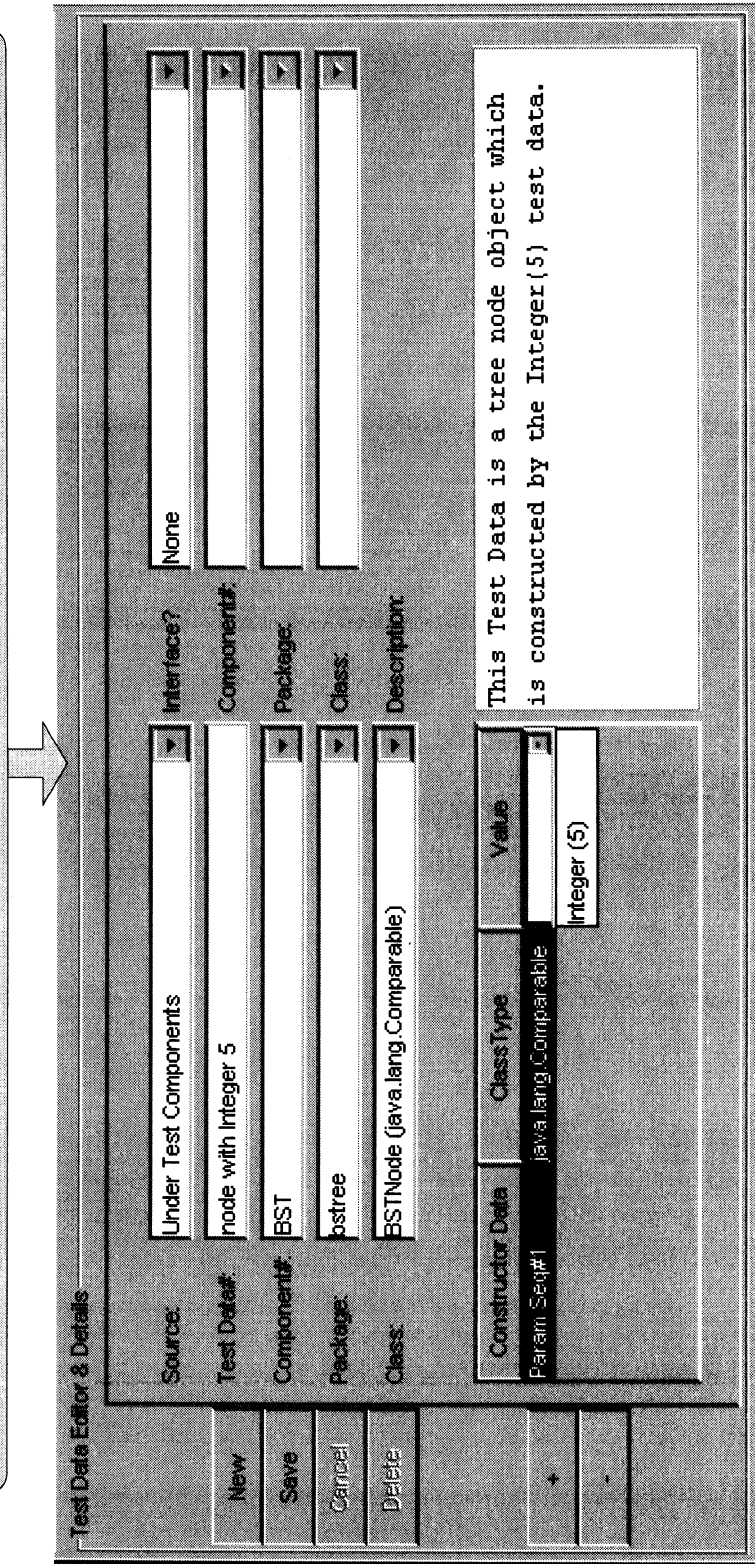


Figure 6.7

An Example to Create Test Data for Binary Search Tree Node

The same concept applies to editing test data in the test case. In this example, we construct a binary search tree object using a previous created tree node as the root. Then, an Integer object is passed in to the “find” function to search such a value in the binary search tree. The expected output is supposed to be the node with Integer value of 5.

The screenshot shows the 'Test Case Editor & Details' window with the following fields and values:

- TestCase#:** TC_BST_1
- Component#:** BST
- Package:** bstree
- Class:** BSTree (bstree.BSTNode)
- Method:** find (java.lang.Comparable)
- Return type:** bstree.BSTNode
- Case Type:** Black-Box
- WroteBy:** Yi-Tien Lin
- Version:** 1.0
- DocDate:** 11/11/04
- Description:** This Test Case tests "find" a value in a BSTree.

At the bottom, there are buttons for 'New', 'Save', 'Cancel', and 'Delete'.

The 'Constructor Data' table is as follows:

Param Seq#1	ClassType	Value
bstree.BSTNode	bstree.BSTNode	node with Integer 5

The 'Expected Input' table is as follows:

Input Seq#1	ClassType	Value
Expected	java.lang.Comparable	Integer (5)
	bstree.BSTNode	node with Integer Integer (5)

Figure 6.8

An Example to Create Test Case Using Pre-Compiled Test Data

To edit a test script, the user selects test cases from the left table and set to the right table. The selected test cases will be executed in the order as specified in the right table.

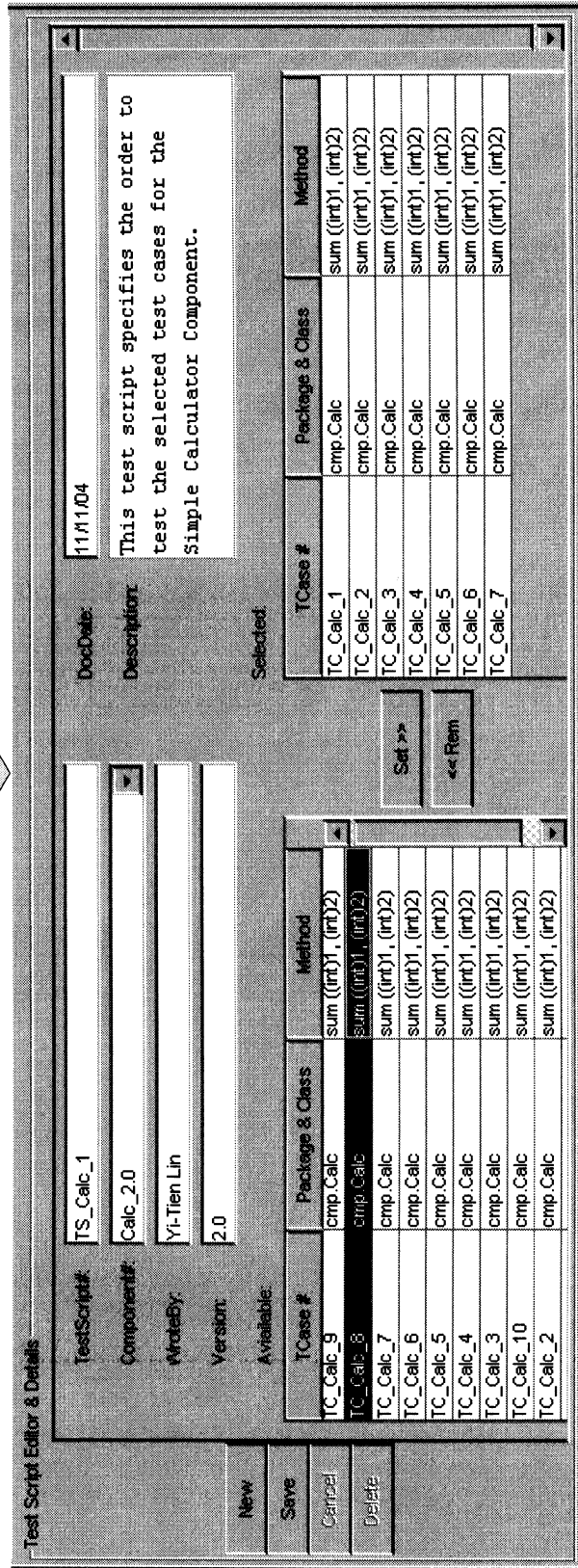


Figure 6.9

An Example to Edit Test Script in the Comp_Test Platform

4. “Validate”: it validates test results against expected result for test cases.
5. “Get Results”: it gets all test results from the executed test cases. It will also trigger the Result Collector window (Figure 6.11).

From the application examples of **Part I** and **Part II**, the COMP_TEST platform shows the capability to run test cases in the simple test script level. Although the COMP_TEST starts to support test models and coverage analysis based on the test suite level, it's beyond the scope of this thesis.

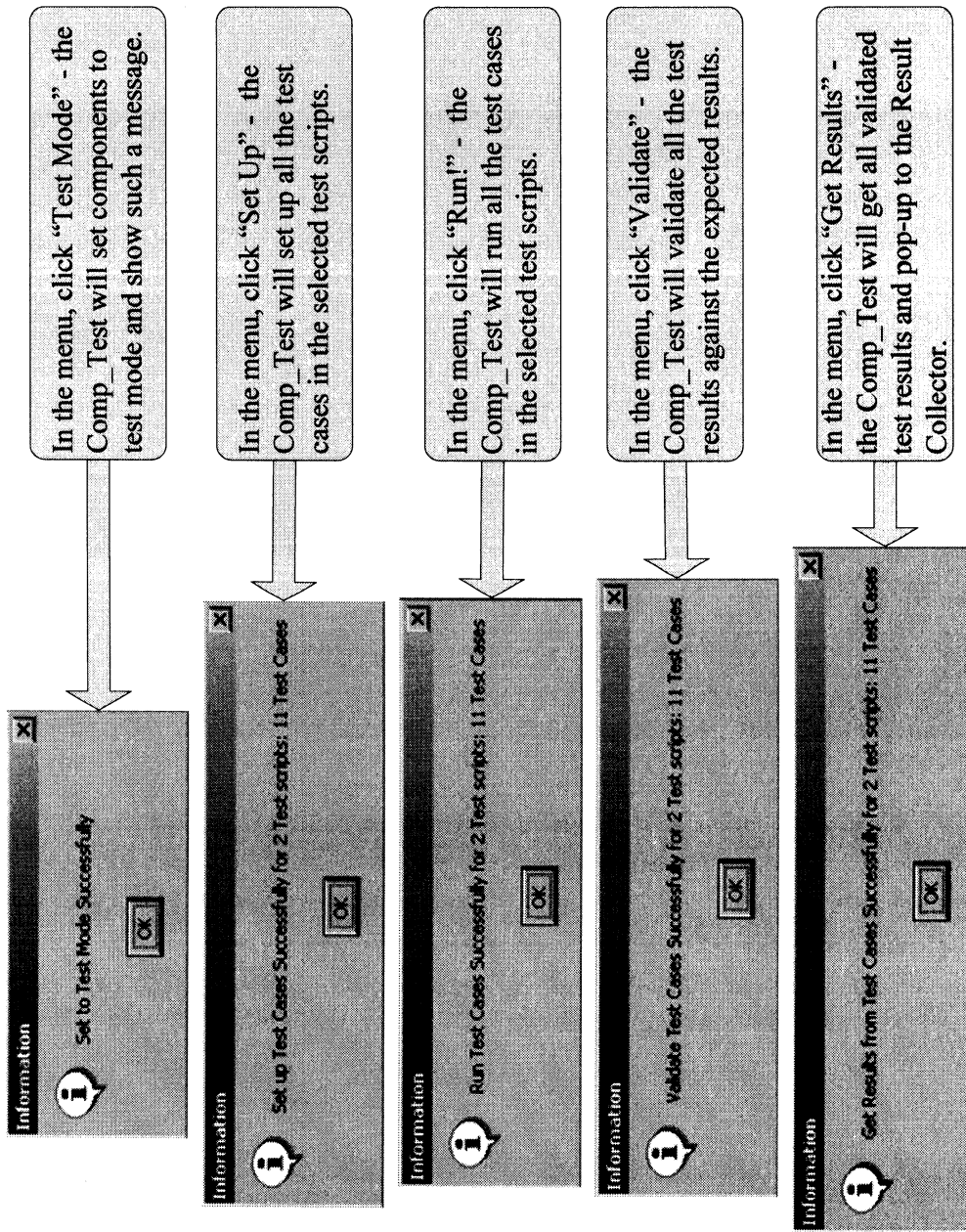


Figure 6.10

Commands to Perform Testing

After the script has been executed, the Result Collector will pop-up to show the test results.

Test Result Browser					
API List	Test Data Compiler	Test Case Editor	Test Script Builder	Result Collector	
Top	Up	Down	Bottom		
Comp #	TestCase #	Result	Used Time	Explanation	
EST	TC_EST_1	Passed	12		
Calc_2.0	TC_Calc_1	Passed	2		
Calc_2.0	TC_Calc_2	Passed	4		
Calc_2.0	TC_Calc_3	Passed	2		
Calc_2.0	TC_Calc_4	Passed	5		
Calc_2.0	TC_Calc_5	Passed	2		
Calc_2.0	TC_Calc_6	Passed	2		
Calc_2.0	TC_Calc_7	Passed	3		
Calc_2.0	TC_Calc_8	Passed	1		
Calc_2.0	TC_Calc_9	Passed	0		
Calc_2.0	TC_Calc_10	Passed	5		

Figure 6.11

A Result Collector Implementation Example

CHAPTER 7

Case Study

This thesis has applied the proposed approach to conduct a number of case studies for several COTS components and in-house built components using the developed reusable test framework and test bed. The major purposes of the case studies are: (a) validating the feasibility of constructing testable components in a systematic way, (b) comparing the cost differences between the conventional validation approach and the proposed approach. Here we report the case study for one selected COTS component.

This case study for a selected component contains three parts.

Part I: Conduct a conventional black-box validation process for a COTS component as component users. This includes the following tasks:

1. Understand the given component based on the provided component specifications.
2. Conduct black-box test design and document test cases based on pre-selected test coverage criteria.
3. Develop required test drivers and/or stubs for supporting component black-box tests.
4. Run black-box test cases and report bugs for the component.
5. Report test cost and development efforts for constructing test drivers and/or stubs.

Part II: Apply the proposed approach to validate the given component. It consists of the following tasks:

1. Convert the COTS component into a testable component using the systematic solution.

2. Apply and run the same set of component black-box test cases using the COMP_TEST Framework with the support of our component test framework.
3. Report test cost and other efforts in Step #1 and Step #2.

Part III: Compare and present the test cost and efforts using two different approaches.

Here, we report the case study based on the COTS component Binary Search Tree. It has contains two Java classes:

1. BSTree.class — the component main program, the actual Binary Search Tree implementation.
2. BSTNode.class — the component library program, the implementation of Binary Search Tree Node.

	Test Suite 1	Test Suite 2	Test Suite 3	Total Test Cases
Boundary Value	39	0	0	39
Equiv. Partition	0	19	146	165
Num. of Test Cases	39	19	146	204

Table 7.1

Distribution of Test Cases in Test Suites

In this case study, we apply two black-box testing methods (boundary value analysis and equivalent partition methods) to this COTS component. Table 7.1 lists the details of three test suites. Test Suite 1 is a collection of boundary value test cases. Test Suite 2 and Test Suite 3 include equivalent partition test cases. The difference

between Test Suite 2 and Test Suite 3 is in the pre-conditions. Test Suite 2 takes an empty binary search tree.

# of Test Drivers	Test Suite 1	Test Suite 2	Test Suite 3	Required Test Drivers
Precondition Test Drivers	6	1	10	14
Functional Test Drivers	13	18	27	27
Case-Oriented Test Drivers	12	19	19	50
Generic Test Drivers	0	0	0	1
Total	31	38	56	92

Table 7.2

Distribution of Test Drivers in Test Suites

This case study has applied the same suite of tests to validate the Binary Search Tree component using two approaches. First, we use a traditional way to validate the component using the given test suites by creating component test drivers using a traditional way. Table 7.2 shows the developed four types of test drivers to support component black-box unit tests. They are:

1. Functional test drivers. They are created for accessing component functions through component APIs. In general, each component function needs one functional test driver.
2. Condition-setup test drivers. They are created as control programs that set-up a pre-defined binary search tree before running the tests.
3. Case-oriented test drivers. They are created for setting up test cases and test data with expected outputs.
4. Generic test drivers. They are set up to control and execute test suites.

Complexity of Test Drivers (LOC)	Test Suite 1	Test Suite 2	Test Suite 3	Total Test Driver Size
Precondition Test Driver Size (LOC)	114	75	198	241
Functional Test Driver Size (LOC)	98	115	236	279
By Case Test Driver Size (LOC)	158	104	311	573
Generic Test Driver Size (LOC)	0	0	0	8
Total Size (LOC)	370	294	745	1101

Table 7.3

The Code Size of Test Drivers in Manual Approach

In Tables 7.2 and 7.3, we report the detailed cost of involving the development of various component test drivers and their complexity in term of Line-Of-Code (LOC). After converting Binary Search Tree component into a testable component using the systematic way defined in the previous section, we perform the same suite of tests to validate the testable component in the supporting test environment. The test framework only needs to generate *one* generic driver with **41 (LOC)** to support the execution of all test cases in the test suites when they are stored in a test repository of the component test environment.

Table 7.4 shows varying test costs and test harness development times. It is clear that both approaches applied to the same set of test cases yield the same set of program bugs. However, testing of the testable component reduces a great deal of cost in test harness development and set-up. Figures 7.1, 7.2, 7.3, and 7.4 show the comparison of the case study results using the two different approaches.

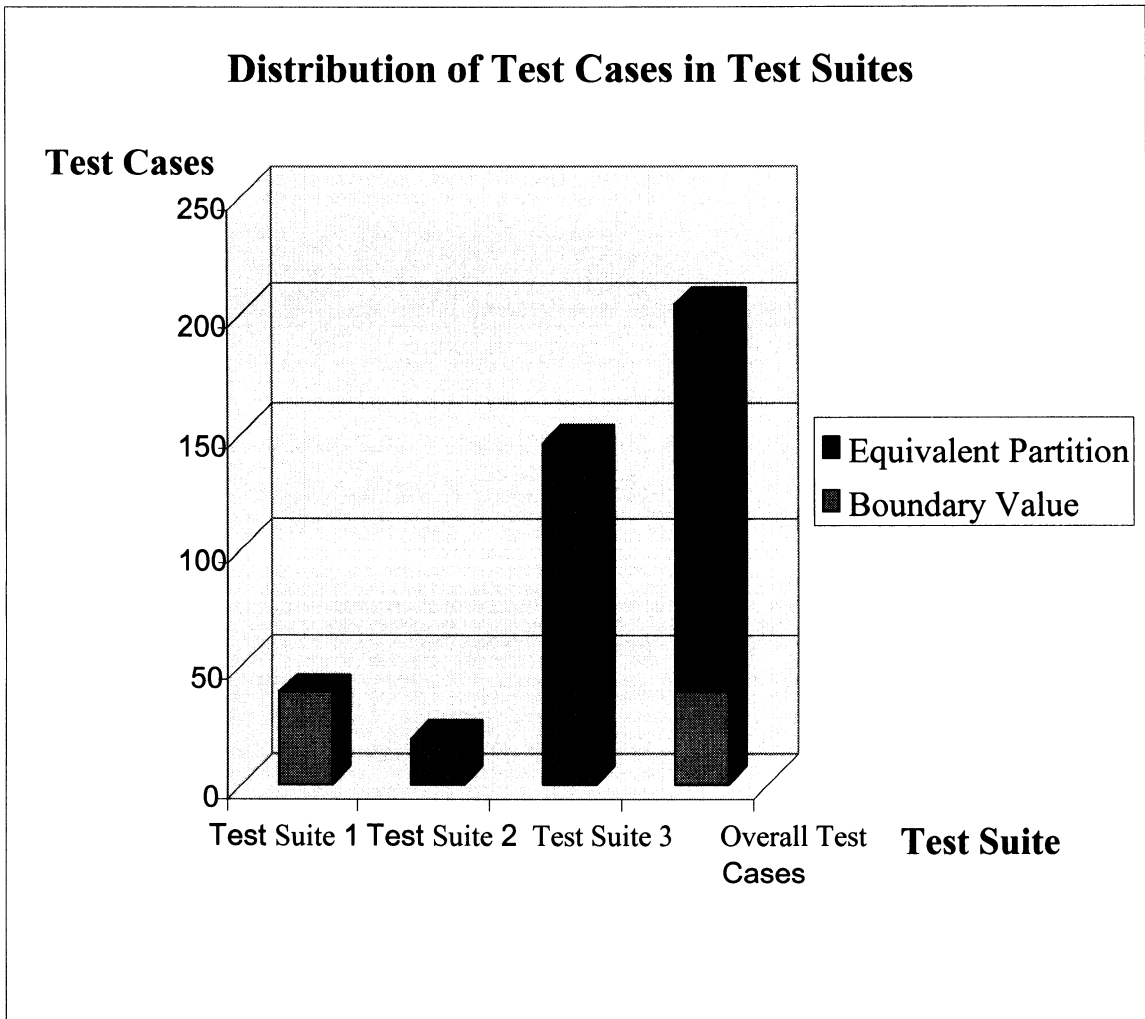


Figure 7.1

Distribution of Test Cases in Test Suites

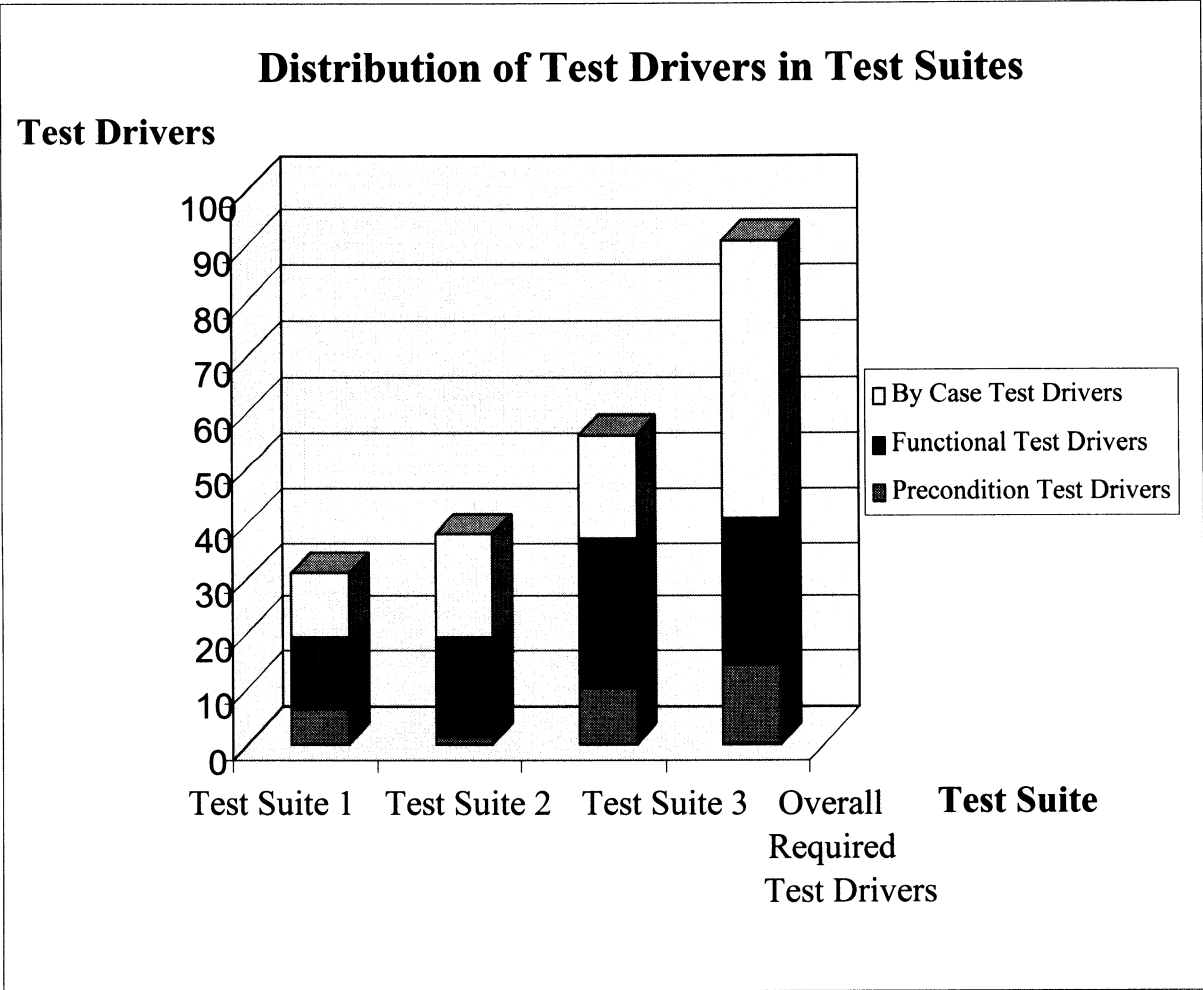


Figure 7.2

Distribution of Test Drivers in Test Suites

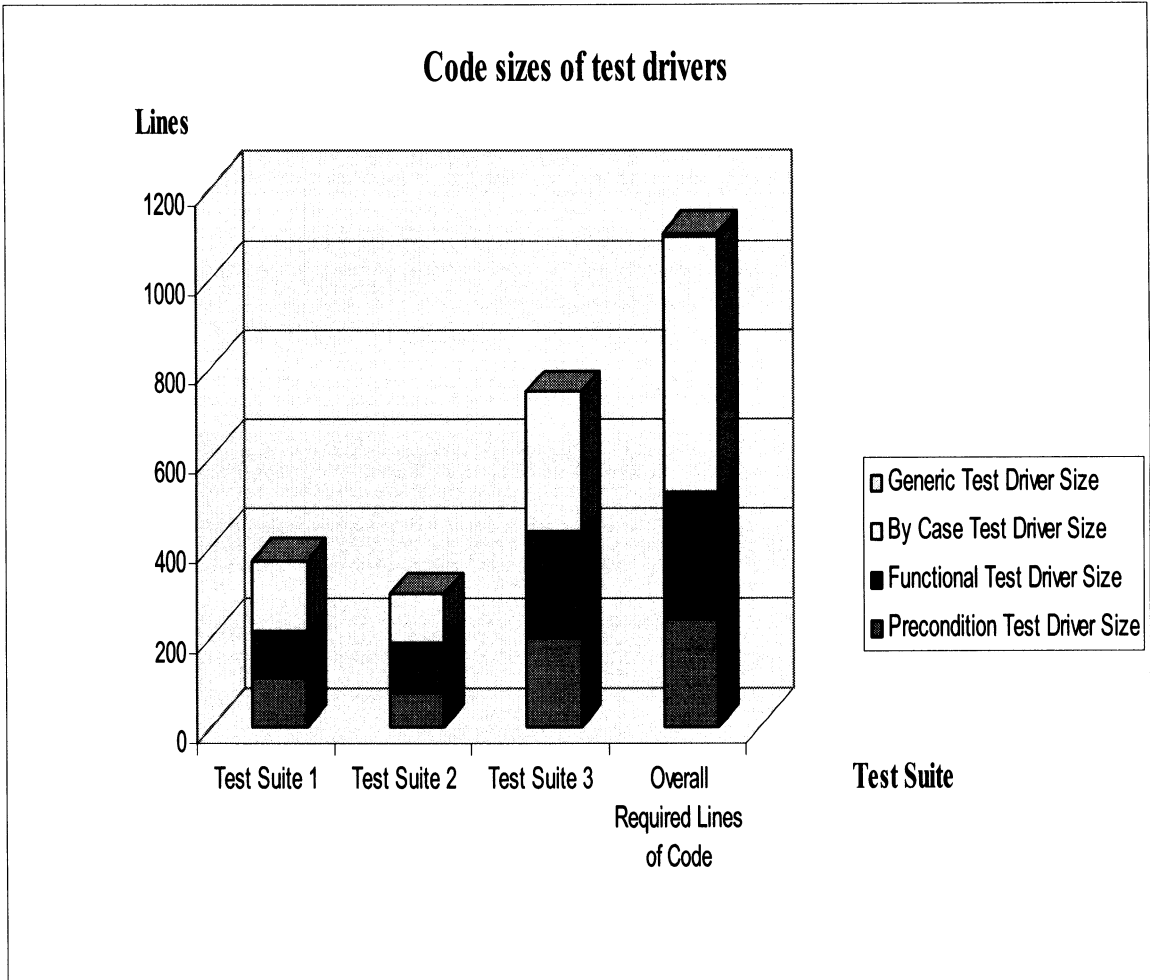


Figure 7.3

The Code Size of Test Drivers

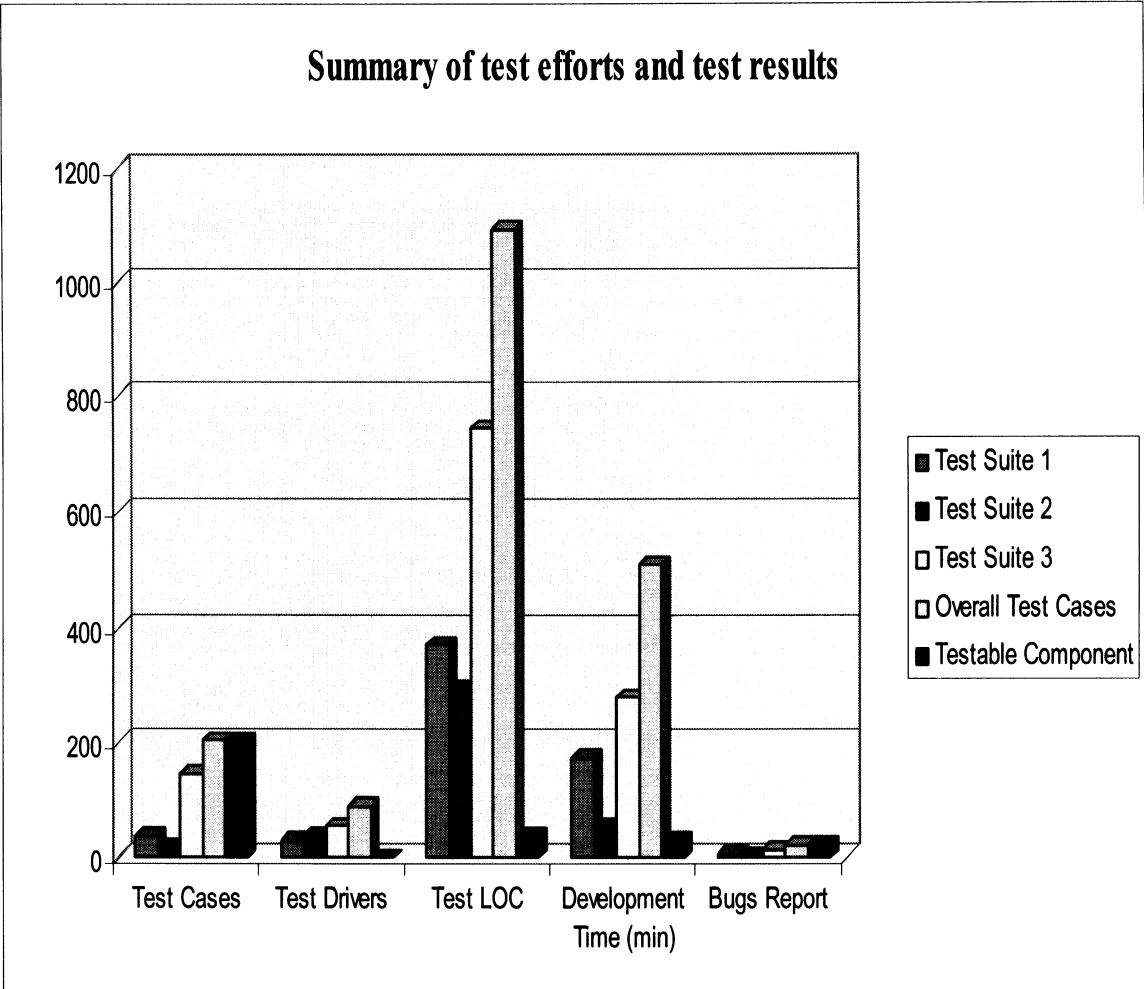


Figure 7.4

Summary of Test Efforts and Test Results

	Test Suite 1	Test Suite 2	Test Suite 3	Overall Required Test Drivers (LOC)	Testable Component
No. of Test Cases	39	19	146	204	204
No. of Test Drivers	31	38	56	91	1
Size of Test Drivers (lines)	370	294	745	1093	41
Development Time (min)	173	57	280	510	33
No. of Bugs Reported	6	2	15	23	23

Table 7.4

Summary of Test Efforts and Bugs Report

CHAPTER 8

Conclusions and Future Work

Conclusions

For component-based software, its testability is dependent on component testability. Component testability is dependent on how well a component is designed and structured to facilitate component testing and software testing. We consider that testable components must be defined with a standard component test interface and a well-defined architecture model for testing. Component test automation needs a systematic solution to generate, test, and support testable components.

This thesis discusses the basic concept of testability of software components in terms of definitions, basic requirements, and contributing factors. It also explains and compares different ways to enhance component testability. The focus of this thesis is to present a systematic way to construct testable software components for COTS components to increase component testability. The distinct contribution of this thesis is its proposed component architecture, well-defined component test interfaces, as well as a systematic wrapping solution to convert COTS (or in-house-built) components into testable components. The proposed method has several distinct features:

1. Convert a given COTS (or in-house-built) component into a testable component using a dynamically generated component wrapper with a well-defined component architecture. This component wrapper provides: (a) a well-defined component test interface to the external world, and (b) a dynamic created internal test adapter to interact with the component API.

2. Use a well-defined common component test framework (which is a set of class library) as a middleware between the testable components and various component test tools and facilities.
3. Provide a common generic component test bed which interacts with testable components based on their common test interfaces.

Its major advantages are summarized below:

1. Giving a practice-oriented solution to enhancing component testability and a systematic approach to support component test automation.
2. Offering a systematic way to construct a common test bed to deal with diverse COTS components.
3. Providing a consistent component test interface between COTS components and test tools and component test repositories.
4. Reducing a great deal of costs of component users in developing component test drivers during a component validation process.

Moreover, the thesis reports the developed component testing environment which supports test automation for testable components in component management, test management, test execution control, and model-based API test coverage monitoring and analysis. Unlike other existing test tools, the system has an intention to offer component users a plug-in-and-test solution to support component functional validation. In addition, this thesis also reports the case study and application example of the proposed solution. The result indicates that this approach has a very sufficient potential to reduce

the component test harness from users and allow them to validate COTS components using the same component test environment in a plug-in-and-test manner.

Future Work

To carry this research into the next step, we are working on applying and extending this solution to other types of components, including graphic user interface components and communication-oriented components. Meanwhile, we are continuously working on applying our approach to middle-sized or large-sized components. In addition, we are adding component API-based test model and coverage analysis features to facilitate component testability measurement based on component tests.

REFERENCES

- Binder, R.V. (1994). Design for Testability in Object-Oriented Systems.
Communications of the ACM, pp.87-101.
- Edwards, S.H. (2000). Black-Box Testing Using Flowgraphs: An Experimental Assessment of Effectiveness and Automation Potential. Software Testing, Verification and Reliability, Vol. 10, No. 4, pp. 249-262.
- Edward, S.H. (2001). A Framework for Practical, Automated Black-box Testing of Component-Based Software. June 2001 Issue of Software Testing, Verification and Reliability, Vol. 11, No.2.
- Freedman, R.S. (1991). Testability of Software Components. IEEE Transactions on Software Engineering, Vol. 17, No. 6.
- Gao, J.Z. (1999). Testing Component-Based Software. STARWEST'99.
- Gao, J.Z. (2000). Challenges and Problems in Testing Software Components. ICSE2000's 3rd International Workshop on Component-based Software Engineering: Reflects and Practice.
- Gao, J.Z., Zhu, E. & Shim, S. (2000). Tracking Software Components. ICSE2000's COTS Workshop: Continuing Collaborations for Successful COTS Development.
- Gao, J.Z. et al. (2000). Monitoring Software Components and Component-Based Software. Proc. of the Twenty-Fourth Annual International Computer Software & Applications Conference (COMPSAC00), Taipei, Taiwan.
- Gao, J.Z., Gupta, K., Gupta, S. & Shim, S. (2002). On Building Testable Software Components. Proc. Of First International Conference on Cost-Based Software

System, pp.108-121.

- Gao, J.Z., Tsao, J. & Wu, Y. (2003). Testing and Quality Assurance for Component-Based Software. Artech House Publishers.
- Harrold, M. J., Liang, D. & Sinha, S. (1999). An Approach to Analyzing and Testing Component-Based Systems. Proc. of FIRST International ICSE Workshop on Testing Distributed Component-Based Systems, Los Angeles.
- IEEE Standard Glossary of Software Engineering Terminology. (1990). ANSI/IEEE Standard 610-12-1990, IEEE Press, New York
- Jhumka, A., Hiller, M., & Suri, N (2002). An Approach to Specify and Test Component-Based Dependable Software. Proc. of the 7th IEEE International Symposium on HASE'02.
- Jungmayr, S. (1999). Reviewing Software Artifacts for Testability. Proc. of the EuroSTAR '99, Barcelona.
- Kropp, N., Koopman, P. & Siewiorek D. (1998). Automated Robustness Testing of Off-the-Shelf Software Components. Proc. of FTCS'98, Munich, Germany.
- Martins, E. & Toyota, C, M. (1998). Reuse in OO Testing: a Methodology for the Construction of Self-Testing Classes. In IX International Conference on Software Technology –Software Quality, Curitiba, Brazil.
- Martins, E., Toyota C.M., & Yanagawa R. (2001). Constructing Self-Testable Software Components. Proc. of the 2001 International Conference on (FTCS) pp. 151-160.
- Voas J. M. & Miller, K.W. (1995). Software Testability: The New Verification. IEEE Software, Vol. 12, No. 3, pp. 17-28.

- Wang, Y., King, G. and Wickburg, H. (1999). A Method for Built-in Tests in Component-based Software Maintenance. Proc. of Third European Conference on Software Maintenance and Re-engineering (CSMR), Holanda, pp. 186-189.
- Wang, Y., King, G., Fayad, M., Patel, D., Court I., Staples G. and Ross M. (2000). On Built-in Test Reuse in Object-Oriented Framework Design. ACM Computing Surveys (CSUR) vol. 32 issue 1es.
- Weyuker, E.J. (1998). Testing Component-Based Software: A Cautionary Tale. IEEE Software.