

2004

# Verification and measurement of software component testability

Ming-Chih Shih  
*San Jose State University*

Follow this and additional works at: [https://scholarworks.sjsu.edu/etd\\_theses](https://scholarworks.sjsu.edu/etd_theses)

---

## Recommended Citation

Shih, Ming-Chih, "Verification and measurement of software component testability" (2004). *Master's Theses*. 2626.  
DOI: <https://doi.org/10.31979/etd.useu-4uzu>  
[https://scholarworks.sjsu.edu/etd\\_theses/2626](https://scholarworks.sjsu.edu/etd_theses/2626)

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](mailto:scholarworks@sjsu.edu).

VERIFICATION AND MEASUREMENT OF  
SOFTWARE COMPONENT 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 Engineering

by

Ming-Chih Shih

August 2004

UMI Number: 1424497

Copyright 2004 by  
Shih, Ming-Chih

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**<sup>®</sup>

---

UMI Microform 1424497

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

Copyright © 2004  
Ming-Chih Shih  
ALL RIGHTS RESERVED

**APPROVED FOR THE COLLEGE OF ENGINEERING**



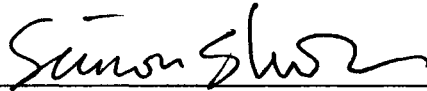
---

Dr. Jerry Gao, Thesis Advisor



---


Dr. Xiao Su, Committee Member



---

Dr. Simon Shim, Committee Member

**APPROVED FOR THE UNIVERSITY**



## ABSTRACT

### VERIFICATION AND MEASUREMENT OF SOFTWARE COMPONENT TESTABILITY

by Ming-Chih Shih

Component-based software engineering is an extremely popular research topic, due to its advantages in cutting product development costs and increasing productivity. Since components are the major building blocks for component-based systems, developing high quality components has become a critical task. Enhancing component testability has been recognized as one of the most important research topics in component-based software engineering. The purpose of this thesis is to present, examine, and discuss an advanced method of reducing software testing costs and enhancing software quality. The focus will be on the process and methods for verification and measurement of component testability in a software component development process.

### **Dedication**

The author would like to dedicate this thesis to his parents, Peter Shih and Mei Cho, and thank them for their support and encouragement.

### **Acknowledgments**

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



## Table of Contents

<b>1. Introduction .....</b>	<b>1</b>
<b>2. Background and Literature Review .....</b>	<b>5</b>
2.1 Definitions of Software Component .....	5
2.2 The Concept of Software Component Testability .....	6
2.3 Verification of Component Testability .....	8
2.4 Measurement of Component Testability .....	10
<b>3. Verification of Component Testability .....</b>	<b>12</b>
3.1 Requirements Verification for Component Testability .....	14
3.1.1 Requirements Verification for Understandability .....	14
3.1.2 Requirements Verification for Observability .....	18
3.1.3 Requirements Verification for Controllability .....	20
3.1.4 Requirements Verification for Traceability .....	24
3.1.5 Requirements Verification for Test Support Capability .....	27
3.2 Design Verification for Component Testability .....	30
3.2.1 Design Verification for Understandability .....	31
3.2.2 Design Verification for Observability .....	32
3.2.3 Design Verification for Controllability .....	39
3.2.4 Design Verification for Traceability .....	40
3.2.5 Design Verification for Test Support Capability .....	41
<b>4. Measurement of Component Testability .....</b>	<b>42</b>
4.1 Requirements Measurement for Component Testability .....	42
4.1.1 Requirements Measurement for Understandability .....	43
4.1.2 Requirements Measurement for Observability .....	46
4.1.3 Requirements Measurement for Controllability .....	47
4.1.4 Requirements Measurement for Traceability .....	48
4.1.5 Requirements Measurement s for Test Support Capability .....	50
4.2 Design Measurement for Component Testability .....	53
4.2.1 Design Measurement for Understandability .....	54
4.2.2 Design Measurement for Controllability .....	55
4.2.3 Design Measurement for Controllability .....	57
4.2.4 Design Measurement for Traceability .....	59
4.2.5 Design Measurement for Test Support Capability .....	60
4.3 Pentagram Model of Software Testability Measurement .....	61
<b>5. A Case Study and Application Examples .....</b>	<b>64</b>
5.1 A Work Flow to Compare the Two Versions .....	64
5.2 Requirements Verification and Measurement for Component Testability .....	65
5.2.1 Requirements Verification and Measurement for Understandability .....	66
5.2.2 Requirements Verification and Measurement for Observability .....	71
5.2.3 Requirements Verification and Measurement for Controllability .....	72

5.2.4 Requirements Verification and Measurement for Traceability .....	75
5.2.5 Requirements Verification and Measurement for Test Support Capability...	78
5.3 Design Verification and Measurement for Component Testability .....	83
5.3.1 Design Verification and Measurement for Understandability .....	83
5.3.2 Design Verification and Measurement for Observability .....	85
5.3.3 Design Verification and Measurement for Controllability .....	90
5.3.4 Design Verification and Measurement for Traceability.....	92
5.3.5 Design Verification and Measurement for Test Support Capability.....	95
5.4 Final Results and Comparison.....	96
<b>6. Conclusions .....</b>	<b>100</b>
6.1 Summary .....	100
6.2 Conclusions and Recommendations for Future Research.....	101
<b>References .....</b>	<b>102</b>
<b>Appendices .....</b>	<b>104</b>
Appendix A. Notation Page .....	104
Appendix B. Binary Tree .....	106

## List of Figures

Figure 1.	Software Verification.....	13
Figure 2.	Requirements Verification for Component Testability.....	14
Figure 3.	Understandability Model.....	15
Figure 4.	Observability Model.....	19
Figure 5.	Controllability Model.....	21
Figure 6.	Traceability Model.....	24
Figure 7.	Test Support Capability Model.....	28
Figure 8.	Design Verification for Component Testability.....	30
Figure 9.	Data Observability.....	33
Figure 10.	Communication Observability.....	34
Figure 11.	GUI Data Observability.....	35
Figure 12.	GUI Event Observability.....	36
Figure 13.	Function Call Data Observability.....	37
Figure 14.	Function Call Message Observability.....	38
Figure 15.	Design for Controllability.....	40
Figure 16.	Requirements Measurement for Testability.....	43
Figure 17.	Design Measurement for Component Testability.....	53
Figure 18.	Pentagram Model.....	62
Figure 19.	Comparing Two Pentagram Areas.....	63
Figure 20.	The Work Flow for Comparison.....	65
Figure 21.	Pentagram of Requirements (Binary Tree - Version 1).....	96
Figure 22.	Pentagram of Requirements (Binary Tree - Version 2).....	97
Figure 23.	Pentagram of Design (Binary Tree - Version 1).....	98
Figure 24.	Pentagram of Design (Binary Tree - Version 2).....	99
Figure 25.	Comparing Testability of The Two Versions.....	99

## List of Tables

Table 1.	Data Observability Evaluation Table .....	33
Table 2.	Communication Observability Evaluation Table .....	34
Table 3.	GUI Data Observability Evaluation Table .....	35
Table 4.	GUI Event Observability Evaluation Table .....	36
Table 5.	Function Call Data Observability Evaluation Table .....	37
Table 6.	Function Call Message Observability Evaluation Table .....	38
Table 7.	Evaluation Table for Functional Requirements .....	44
Table 8.	Evaluation Table for Non-Functional Requirements .....	45
Table 9.	Evaluation Table for Observability in Requirements.....	46
Table 10.	Evaluation Table for Controllability Types in Requirements .....	47
Table 11.	Evaluation Table for Controllability Properties in Requirements.....	48
Table 12.	Evaluation Table for Traceability Types in Requirements .....	49
Table 13.	Evaluation Table for Traceability Properties in Requirements .....	50
Table 14.	Evaluation Table for Component Test Generation in Requirements .....	51
Table 15.	Evaluation Table for Component Test Management in Requirements .....	51
Table 16.	Evaluation Table for Component Test Coverage in Requirements.....	52
Table 17.	Evaluation Table for Component Test Scripting in Requirements .....	52
Table 18.	Evaluation Table for Understandability in Design.....	54
Table 19.	Evaluation Table for Data Observability in Design .....	55
Table 20.	Evaluation Table for Communication Observability in Design.....	55
Table 21.	Evaluation Table for GUI Data Observability in Design .....	56
Table 22.	Evaluation Table for GUI Event Observability in Design .....	56
Table 23.	Evaluation Table for Function Call Data Observability in Design .....	56
Table 24.	Evaluation Table for Function Call Message Observability in Design.....	57
Table 25.	Evaluation Table for Built-in Mechanism Design of Controllability .....	58
Table 26.	Evaluation Table for API Design of Controllability .....	58
Table 27.	Evaluation Table for Built-in Mechanism Design of Controllability .....	59
Table 28.	Evaluation Table for API Design of Controllability .....	60

Table 29.	Evaluation Table for Component Test Support Capability in Design .....	61
Table 30.	Comparison Table for Functional Requirements .....	68
Table 31.	Comparison Table for Non-Functional Requirements .....	70
Table 32.	Comparison Table for Observability Types in Requirements.....	71
Table 33.	Comparison Table for Controllability Types in Requirements.....	73
Table 34.	Comparison Table for Controllability Properties in Requirements .....	74
Table 35.	Comparison Table for Traceability Types in Requirements .....	76
Table 36.	Comparison Table for Traceability Properties in Requirements.....	77
Table 37.	Comparison Table for Test Generation in Requirements.....	80
Table 38.	Comparison Table for Test Management in Requirements .....	81
Table 39.	Comparison Table for Test Coverage in Requirements .....	81
Table 40.	Comparison Table for Test Scripting in Requirements.....	81
Table 41.	Comparison Table for Understandability in Design .....	84
Table 42.	Comparison Table for Data Observability in Design.....	88
Table 43.	Comparison Table for Communication Observability in Design.....	88
Table 44.	Comparison Table for GUI Data Observability in Design.....	88
Table 45.	Comparison Table for GUI Event Observability in Design .....	88
Table 46.	Comparison Table for Function Call Data Observability in Design.....	89
Table 47.	Comparison Table for Function Call Message Observability .....	89
Table 48.	Comparison Table for Built-in Mechanism Controllability in Design .....	91
Table 49.	Comparison Table for API Design Controllability in Design.....	91
Table 50.	Comparison Table for Built-in-Mechanism Traceability in Design .....	93
Table 51.	Comparison Table for API Traceability in Design .....	94
Table 52.	Comparison Table for Component Test Support Capability in Design .....	95
Table 53.	Summary Table of Requirements Metrics .....	96
Table 54.	Summary Table of Design Metrics .....	97

## **1. Introduction**

Component-based software engineering has become a very important research area in past decades. The present chapter gives us a general idea of why this thesis topic is chosen in the component-based field, what the issues and challenges are, and how the thesis addresses solving the problem.

### **Motive**

Software managers or engineers always encounter the same problem— finding an efficient solution for eliminating errors. Defects in software have a high impact on the quality of software components. Software engineers and managers spend time and effort to find out where specific errors are located. These maintenance and testing efforts always cost a lot in the development of software components. In the past, people have considered software testing a cost effective method to detect faults in software (Myers 1979). Some people have tried to use automated testing tools to check the correctness of the program specification and program structure; however, the outcome is neither accurate nor efficient. Thus, checking software reliability is still a puzzle. Since we cannot improve quality by using the old ways, how can people improve software quality? Voas et al. (1995) point out that “software verification is often the last defense against disasters caused by faulty software development” and that “software component testability is an efficient solution to verify the quality of a software component.” Verifying software components by improving the testability has the following two benefits: reducing testing development costs and increasing software quality.

1. **Reducing Software Testing Costs:** Jungmayr (1999) indicates that software testing costs about 40 to 50 percent of the overall development budget. Being aware of testability in the initial stage of, and throughout, software component development will have developers devise trustworthy design patterns at every step of process to reduce software testing costs.
2. **Increasing Software Quality:** Until now, and even presently, engineers have spent a lot of time testing due to insufficient manageability and supportability. When engineers are working in the maintenance phase, it is always past the time, illogical, and, actually, counterproductive for them to try figuring out software defects. Thus, one of the best ways to increase software quality is by focusing, at the initial stage, on the verification of software component testability.

Hence, in this thesis, software component testability will be discussed, and an approach to verifying software component testability will be proposed. After discussing the importance of software component testability, the next issue is that of understanding what software testability measurement is, and how to perform it. Accurate measurement of software components can help us understand their quality easily and efficiently.

### **The Problem**

What problems do the verification of software component testability and the measurement of software component testability pose for us? Here are several concerns regarding building and testing of software components.

1. How to construct software components in a systematical way which allows engineers to perform testing easily?

2. How to construct components with high testability?
3. How to increase the testability of software components?
4. How to verify the testability of software components in a systematic way?
5. How to measure the testability of software components in a systematic way?

Having indicated these problems, the purpose for this thesis is described in the following section.

### **Purpose**

The purpose of this thesis is to develop a systematic approach to verification and measurement of component testability in order to reduce the cost of software testing development and to improve the quality of software components. It is understood that all software engineers and managers hope to improve software quality and to detect defects in software components as early as possible. If we can develop software components by predefined criteria of testability, engineers would be better able to test and maintain software components efficiently.

By evaluating the testability of software components, software engineers can comprehend their quality. Also, this measurement makes software engineers more facile at improving the quality of software components and at comparing testability between different versions of the same component.

### **Scope**

Based on the purpose, the scope has been divided into five categories:

1. Understanding the concept and issue of software testability
2. Developing a systematic approach to verification of software component testability



3. Developing a systematic approach to measurement of software component testability
4. Providing a case study to show the results of two versions of testable component design
5. Comparing testability between two different versions of software components

### **Organization**

The thesis is organized into six chapters and two appendices. In chapter 2, the related works of software component testability are reviewed. In chapter 3, a systematic approach to verifying software component testability is proposed. In chapter 4, this approach will be applied in order to evaluate the testability of software components. Metrics and evaluation tables are tools used here to measure software component testability. In chapter 5, two versions of a software component, a binary tree application, are compared in order to assure and prove the effectiveness of the approach. Finally, in chapter 6, this thesis is summarized, and directions are presented for future research. Appendix A illustrates notations for describing software components. Appendix B provides two versions of binary tree application used for my case study.

## 2. Background and Literature Review

This chapter reviews and analyzes the literature on software testability. This background section describes current issues in the field of software testing and introduces several important terms of testability in order to present the work that will be discussed in this thesis.

### 2.1 Definitions of Software Component

Different perspectives (i.e. different fields of study) use different definitions for the word *component*. In the early stage of software engineering, Szyperski (1996) gives the idea of a *software component* as “a unit of composition with contextually specified interfaces and explicit context dependencies only.” A *software component* can be deployed independently and is subject to third-party composition. The following definition is considered a general notion of a software component: “A *software component* is an independently deployable implementation of some functionality, to be reused in a broad spectrum of applications” (Bass et al. 2001). In other words, a component can be a class, an interface, a module or a subsystem.

### Software Component Testing

The quality of software depends on the quality of components. Veryard(1996) indicates that “component-based development and software reuse places new demands on *software testing* and *quality assurance*.” This is especially true if components are to be traded among organizations. This concept gives us an idea of the relationship between software testing and software component. Thus, *software component testing* refers to

testing activities that check the correctness and reliability of a system from the perspective of its components.

Baldini & Pronetto (2003) give a broad definition for components, which includes both hardware and software. Their design for components is based, and focuses, on hardware testability concepts. What they are trying to do is to apply the concept of testing consumer electronics to the testing of software components. Baldini & Pronetto's paper not only presents a different way to test software components, but also gives us a greatly expanded idea of software testability concepts.

## **2.2 The Concept of Software Component Testability**

The IEEE Standard Glossary of Software Engineering Terminology (1990) defines *testability* as:

1. 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.
2. the degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met.

Voas and Miller (1995) are the first to present the benefits of their new approach, software testability. This new verification approach differs from traditional views of verification and testability. Voas and Miller propose the domain/range ratio (DRR) metric to classify software components into three groups: variable domain/fixed range (VDFR), variable domain/variable range (VDVR) and fixed domain/fixed range (FDFR). Their

solution only focuses on the relationship between the input and output domains of a component.

Next, what is “Software Testability?” According to Voas et al. (1995), “Software testability is a characteristic that either suggests how easy software is to test, how well the tests are able to interact with the code to reveal defects, or some combination of the two.” Freedman (1991) is the first person who defines a new concept, domain testability, by applying the concepts of two factors, observability and controllability, to software. He defines the domain testability for software components as a combination of component observability and controllability. In short, he focuses on the relationship between the input and corresponding output of software components. In his definition, observability is the ease of determining if specific input affects the output, and controllability is the ease of producing specific output from specific input.

### **Five Software Component Testability Factors**

Gao (2002) extends the definition of component testability and divides it into five factors: component understandability, observability, controllability, traceability, and testing support capability. He claims that component testability can be verified and measured by processes based on these five factors. The definitions for each factor are described as below.

1. **Component Understandability** — includes the availability of component information and the understandability of the provided component information.
2. **Component Observability** — indicates the mapping relationships between inputs and corresponding outputs for each test. Also, it tracks and monitors all component

tests and test results.

3. **Component Controllability** — includes the component environment control, component execution control, component state-based behavior control, component test control, and component function feature control.
4. **Component Traceability** — refers to the extent of its built-in capability to track functional operations, component attributes, and behaviors.
5. **Component Testing Support Capability** — includes component test management capability, component test generation capability, test suite support capability, and test coverage analysis capability.

The first part of this thesis focuses on the verification of software component testability. After having understood the meaning of component testability, the importance of verifying component testability and maturity of verifying component testability are introduced in the next section.

### **2.3 Verification of Component Testability**

It is impossible to eliminate all software errors by using any current methods or technique. The only way to reduce software errors is to improve the quality of the software, which also means increasing software reliability. The purpose of software verification is to measure software component quality. The IEEE Standard Glossary of Software Engineering Terminology (1990) defines software verification to be “the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.”

“The importance of software verification is increasing as more and more safety and security critical applications emerge, and there is greater demand for high-quality software” (Beckert 2004). He also points out that “current tools and methods for software verification do not adequately support the design formalisms (such as UML) and programming languages (such as Java) that are used in practice.”

Voas et al. (1995) introduces a new concept of software verification by using software testability. Based on the concept of improving software testability, software engineers can increase software reliability and reduce testing costs. He points out two advantages of software verification by using software testability.

1. It can enhance testing.
2. It can reduce correctness in testing.

Gao (2002) categorizes the process of improving software testability into four levels, based on the maturity that each one of the process' levels represents.

**Level 1 - Initial:** At this level, component developers and testers use an ad hoc approach to enhance component testability in a component development process.

**Level 2 - Standardized:** At this level, component testability requirements, design methods, implementation mechanisms, and verification criteria are defined as a part of software component standard.

**Level 3 - Systematic:** At this level, a well-defined component development and test process is used to facilitate component testability at all phases.

**Level 4 - Measurable:** At this level, component testability can be evaluated and measured using systematic solutions and tools in all phases of a software component development process, including component testing.

Most of the current solutions for improving software testability are still in the first two levels. Thus, in this thesis, a systematic and measurable approach is proposed to verify software testability.

#### **2.4 Measurement of Component Testability**

Measurement in science has a long history. In 1889, Lord Kelvin explains the nature and value of measurement as "When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind." Basically, software measurement is something that can determine the quality of software by quantifying it. Kaner (2001) points out that "measurement is the assignment of numbers to objects or events according to a rule derived from a model or theory."

#### **The Advantages of Software Measurement**

Measurement of software components has the following three advantages:

1. Determining the quality and reliability of software components.
2. Evaluating the cost and effort of developing software components.
3. Estimating and predicting the cost and effort of maintaining software components.

Software measurement is defined differently according to different fields' (i.e. scientific disciplines) perspectives. Fenton, Krause and Neil (2002) show how software

can be measured by using Bayesian networks. They describe how a Bayesian network for software quality risk management provides accurate predictions of software defects in a range of real projects. In recent years, the concept of measurement of software component testability has been introduced. Robach and Traon(1995) used the data-flow specification to measure program testability. Data flow modeling performs testability analysis based on information transfers and program specification. Finally, they also provide a real case study, “Aerospatiale,” that graphically illustrates all the assumptions.

Jungmayr (2002) uses metrics to measure the testability that depends on each component’s dependency. The author also gives four case studies to confirm his theory. Readers can easily understand his measurement solution and how it works by its well organized structure.

Gao(2002) explains that “the software testability measurement refers to the activities and methods that study, analyze, and measure software testability during a software product life cycle.” According to his definition, in chapter 4, an approach of measurement is proposed to measure and evaluate software testability.



### 3. Verification of Component Testability

Even though it is difficult to remove all software errors, software engineers still hope to produce high quality software. By increasing software testability, software quality and reliability are also improved. In this chapter, a systematic approach is proposed to verify component testability.

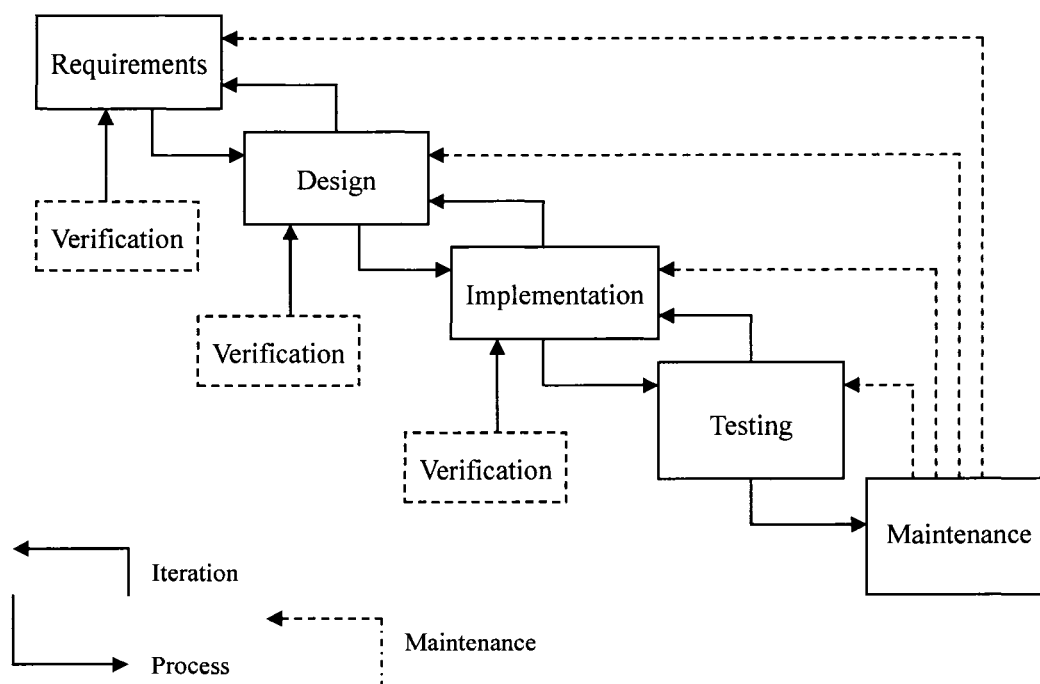
#### Waterfall Model

The waterfall model is a software development model, first proposed by Royce (1970), in which development proceeds linearly through the following phases: requirements, design, implementation, testing, and maintenance. In the past, software products were developed without beginning the process by first considering testing issues. Consequently, software or testing engineers spent a lot of effort on maintenance. The major problem in doing maintenance is that we cannot do it on software products that are not designed for maintenance. Thus, if we can develop software products and consider maintenance in the beginning phase of development, we can save a lot of cost and effort in coding, testing, and maintenance, which are all expensive phases in the software product life cycle.

In this thesis, *testability review* refers to activities of verifying software components, and the general idea of this thesis is to perform a testability review at the first two phases of the software development cycle in order to verify component testability. The purpose of the review is to enhance component testability in order to make sure that the requirements and design phases are correct and complete. By doing testability reviews of

the two initial phases, engineers can save on the effort and cost of developing software components through the remaining phases of the software life cycle.

During the software component development process, software verification refers to the activity of checking component testability using well-defined verification, as shown in Figure 1.

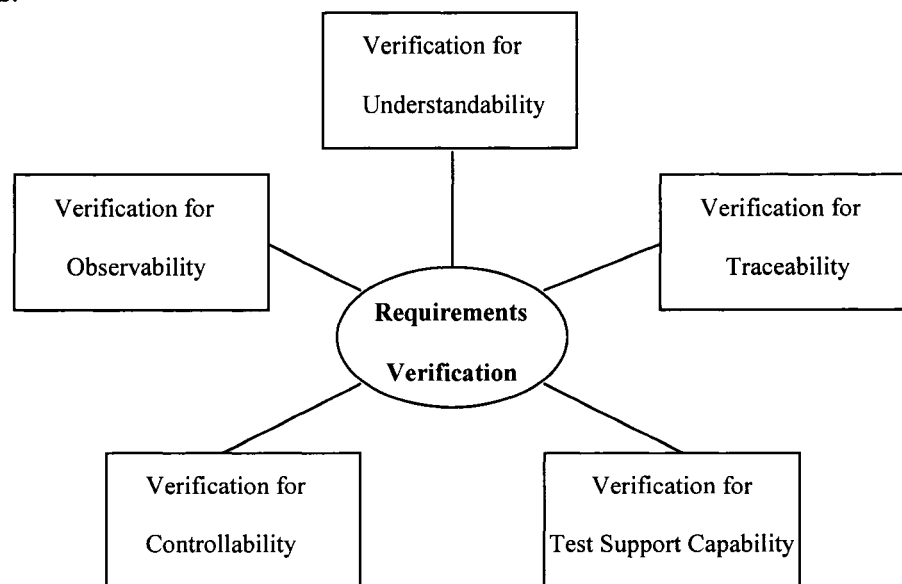


**Figure 1. Software Verification**

The focus in this thesis is only on the first two phases, requirements and design, since, after these, implementation is relatively easier in programming. This systematic approach is a diagnostic scheme for verifying software component development from the requirements phase to the design phase using the five factors of testability. The verification of software components at the requirements phase is described in chapter 3.1, and the design phase, in chapter 3.2.

### 3.1 Requirements Verification for Component Testability

This verification checks whether component requirements are clearly specified so that they can be tested and measured for a given test criteria— testability. Gao (2002) has defined testability by five factors. The process model is based on five factors as shown in Figure 2.

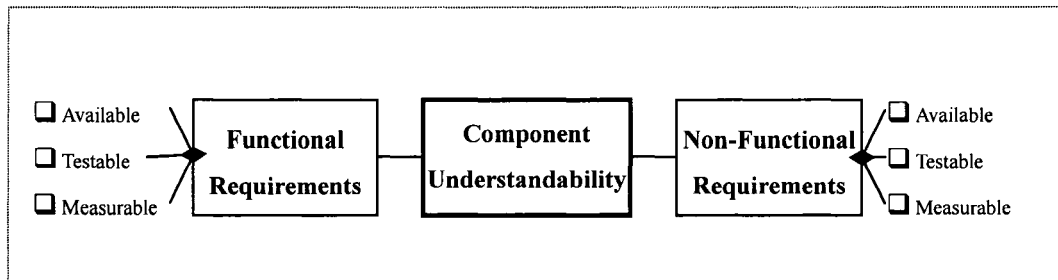


**Figure 2. Requirements Verification for Component Testability**

#### 3.1.1 Requirements Verification for Understandability

Component understandability refers to the activity of separately checking the understandability of, first, functional requirements, and then, afterwards, non-functional requirements. The understandability of functional and non-functional requirements refers to checking requirements for whether they are: available, testable, and measurable. These

two activities and three essential qualities can be adequately described in the understandability model, which is shown as Figure 3.



**Figure 3. Understandability Model**

### **Available**

Checking the availability of component information is the first part of verifying the understandability of a component. Collecting all the component information before checking the testability of a component is also an essential procedure.

### **Testable**

Blackburn (2001) defines testable requirements as “complete, consistent and unambiguous requirements.” He also indicates, “Any potential misinterpretation of the requirement is a defect.” In order to decrease the quantity of software defects, it is necessary to have testable (i.e., precise and unambiguous) requirements.

In this thesis, the case study will be a binary tree application. Examples are taken from this case study in order to explain several essential concepts of testability.

An example of a *non-testable requirement* could be, “If users want to insert a node, the binary tree application can get their input from the text field.” Inherent in the term itself, a *non-testable requirement* is so because it is not specific enough to be testable. A more

specific, accurate, and representative statement of a *testable requirement*, and the working example used in this thesis, will be: “If users want to insert a node, the binary tree application can get their input from the text field. This input will be verified to see if it is an integer. If the value is not an integer, a message will be displayed stating that there is an input error.” At this point, programmers are able to test this requirement, but still unable to measure it.

### **Measurable**

Robertson (1997) explains measurable requirements as “requirements with quantifiable characteristics”. He also points out the importance of measurable requirements by stating that, if programmers incorporate quantifiable characteristics in their requirements, then, “Testers can accurately determine whether the eventual solution satisfies a requirement so that the client can determine if the requirement's value is worth the cost of construction.” In order to enhance the previous testable requirement so that it can become a *measurable requirement*, another specification needs to be added and this specification is that, “The validation of an input will be less than 0.1 second.”

### **Activity 1. Verification of Functional Requirements**

It is understood that functional requirements describe what a software system must accomplish; however, what is a functional requirement with good understandability? It has to be a testable and measurable requirement which allows users to understand its meaning, observe its behavior, control its behavior, and trace its operation.

## **Activity 2. Verification of Non-functional Requirements**

There are seven types of non-functional requirements: environment, availability, performance, scalability, throughput, reliability, and utilization requirements. Every requirement needs to be testable and measurable for better understandability.

**1. Environment Requirements:** This refers to the minimum needs of hardware and software resources that are required by the application. For example, the minimum PC specification that the system must be run on is a 200 MHz Pentium PC with 100 MB of spare disk space and 60 MB of memory.

**2. Availability Requirements:** This refers to the availability of either software or hardware parts, such as computer hardware, computer software, components, servers, functions and processes. It is very crucial for some online systems to be available all the time. For example, in 99% of cases, users can get a response from the binary tree within 3 days.

**3. Performance Requirements:** This refers to the requirements of system process speed and system latency, such as system processes, tasks, transactions, responses, data retrieval, data loading, and message/event latency. The major purpose of performance requirements is to check the current product capacity in order to answer questions from customers and marketing people and to identify performance issues and performance degradation in a given system. For example, over a three-day period of running the binary tree application, the system response to each user action must be less than 1 second in 99% of the cases.

**4. Scalability Requirements:** This refers to the boundaries, limits, or improvement of a software component by a given platform and system loads, such as data volume, database connectivity, numbers of current users, and numbers of concurrent transactions. For example, the binary tree will hold no more than 64 nodes, which means users cannot insert more than the maximum nodes, 64 nodes.

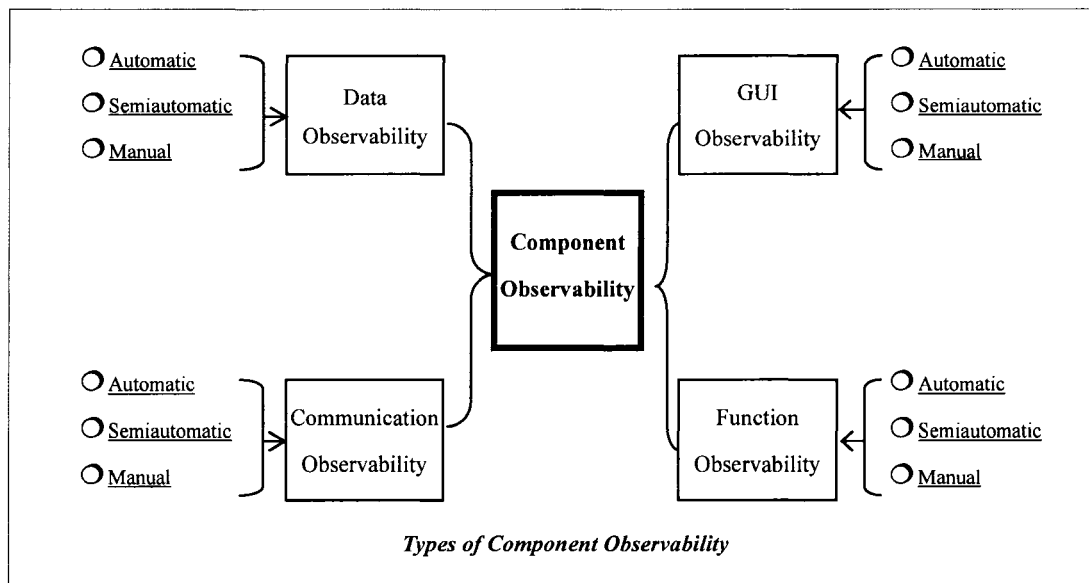
**5. Throughput Requirements:** This refers to product capacity, such as event, task, transaction, and message. For instance, we require the minimum volume of processing events, tasks, transactions and messages in a given period of time.

**6. Reliability Requirements:** This refers to a minimum defect of a software component, computer hardware, network and application in a given time period. For example, users can get 99% correct action when inserting, deleting, searching a node from the binary tree within 3 days.

**7. Utilization Requirements:** This refers to system resource, such as CPU time, memory, cache, disk, and network bandwidth.

### **3.1.2 Requirements Verification for Observability**

Component observability refers to the ability of checking the mapping relationship between the input and output of data, communication, GUI events, and function's behavior. These activities are described in the observability model, which is shown in Figure 4. The observability model is mainly based on the types of component observability. The symbol '○', before each essential quality, represents the option of choosing that one, and only one can be chosen.



**Figure 4. Observability Model**

### Checking for the Four Types of Component Observability

There are four types of component observability. For each type, users can increase the testability by the easiness of observation. Automatic observation is the best, semiautomatic observation is the next best, and manual observation is the least preferred.

**1. Data Observability:** It checks whether requirements include the ability of observing input data and its corresponding output data of a component. In addition, it also checks whether requirements describes the ease of observation that can be categorized by automatic, semiautomatic and manual. One example of a requirement with automatic data observability follows. Users will observe the input and output data of a component automatically by means of a specifically designed monitor.



**2. Communication Observability:** It checks whether requirements include the ability of observing incoming data and its corresponding outgoing message of a component. In addition, it also checks whether requirements describes the ease of observation that can be categorized by automatic, semiautomatic and manual.

**3. GUI Observability:** It checks whether requirements include the ability of observing GUI input data/event and its corresponding output data/event of a component. In addition, it also checks whether requirements describes the ease of observation that can be categorized by automatic, semiautomatic and manual.

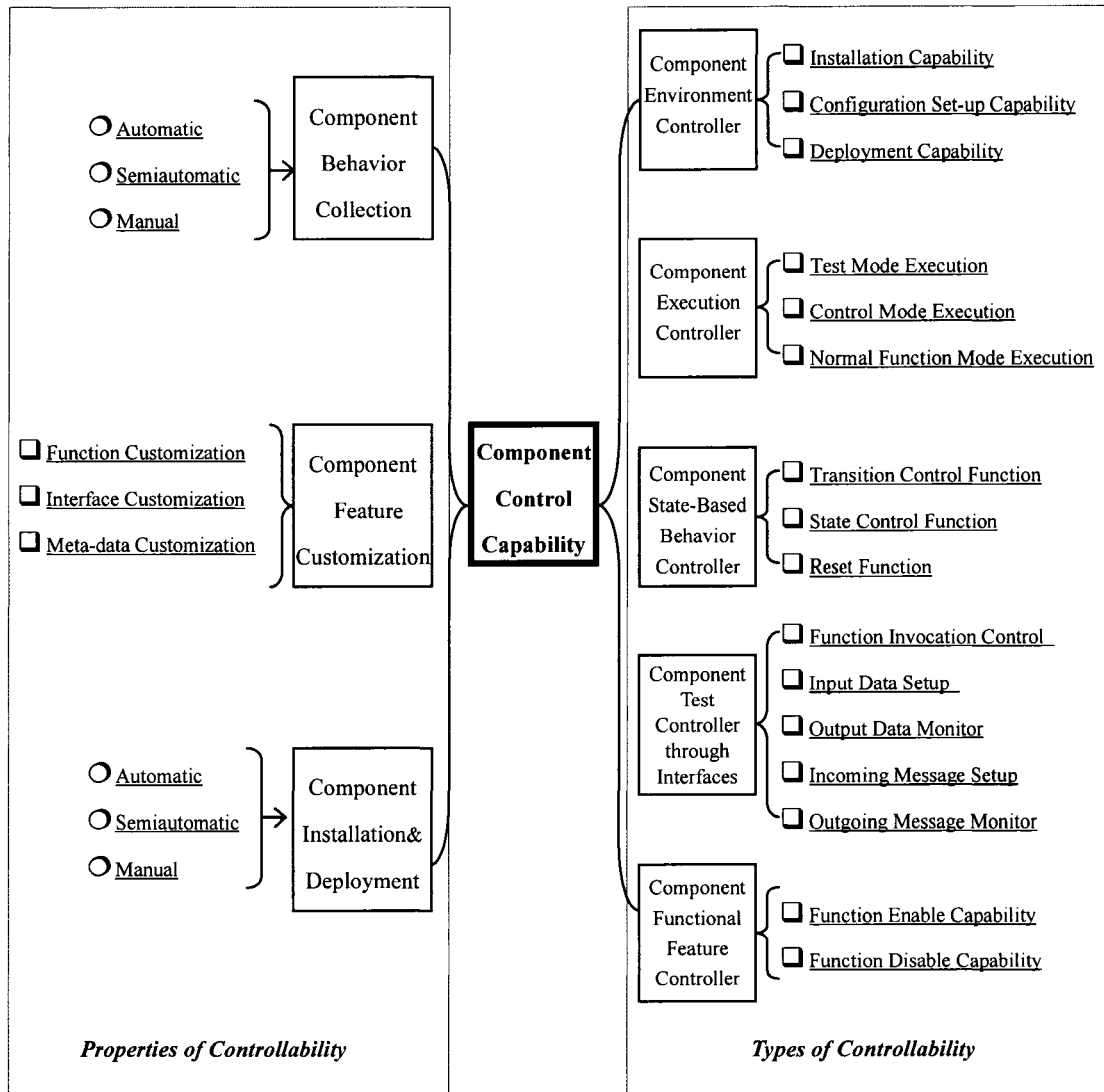
**4. Function Observability:** It checks whether requirements include the ability of observing input function call data/messages and its corresponding output data/message of a component. In addition, it also checks whether requirements describes the ease of observation that can be categorized by automatic, semiautomatic and manual.

### 3.1.3 Requirements Verification for Controllability

Component controllability refers to the activities of checking five types, and three properties, of component controllability. Each of the five types of controllability has its own three properties. For example, a component environment controller should be able to collect component behavior, customize component features, and install/deploy a component.

These activities are described in the controllability model, which is shown as Figure 5. The symbol ‘○’, before each essential quality, represents the option of choosing that one,

and only one can be chosen. The symbol ‘’, before each essential quality, represents the option of making more than one choice, which means, as many as desired can be chosen.



**Figure 5. Controllability Model**

### **Activity 1. Checking Requirements for Five Types of Component Controllability**

There are five types of component controllability.

**1. Component Environment Control:** It refers to the built-in component capability that supports component environment installation, configuration set-up, and deployment.

**2. Component Execution Control:** This refers to the built-in component execution control capability that enables executions of a component in different modes, such as test mode, normal function mode, control mode, and so on. With this capability, users and testers can start, restart, stop, pause, and abort a program as they wish.

**3. Component State-based Behavior Control:** This refers to the built-in capability that facilitates the control of component state-based behavior, such as re-setting component states, state control function and transition control function.

**4. Component Test Control:** This refers to the built-in capability that facilitates the control of component tests through component interfaces, such as function invocation control, input data setup, output data monitor, incoming message setup and outgoing message monitor.

**5. Component Function Feature Control:** This refers to the built-in capability that facilitates the selection and configuration of component functional features. It is only applicable to software components with customizable functional features that allow component users to select and configure its features based on their needs.

## **Activity 2. Checking Requirements for Three Properties of Component**

### **Controllability**

There are three properties of component controllability: component behavior collection, component feature customization, and component feature customization.

**1. Component Behavior Collection:** It refers to a mechanism, which collects the controllability of its behaviors and output data responding to its functional operations and input data. By the easiness of the collection, it can be categorized into automatic, semiautomatic, and manual collection. One example of a requirement with automatic component behavior collection would be: users can collect all component behaviors by saving it automatically into files or a database.

**2. Component Feature Customization:** It refers to a built-in capability of supporting customization and configuration of its internal function features, such as function, meta-data and interface.

**3. Component Installation & Deployment:** It refers to the control capability of component installation and deployment. By the easiness of the collection, it can be categorized into automatic, semiautomatic, and manual installation & deployment.

### 3.1.4 Requirements Verification for Traceability

Verification of Component traceability refers to the activities of checking five types, and five properties, of component traceability. Each of the five types of traceability has its own five properties. These activities are described in the traceability model, which is shown in Figure 6.

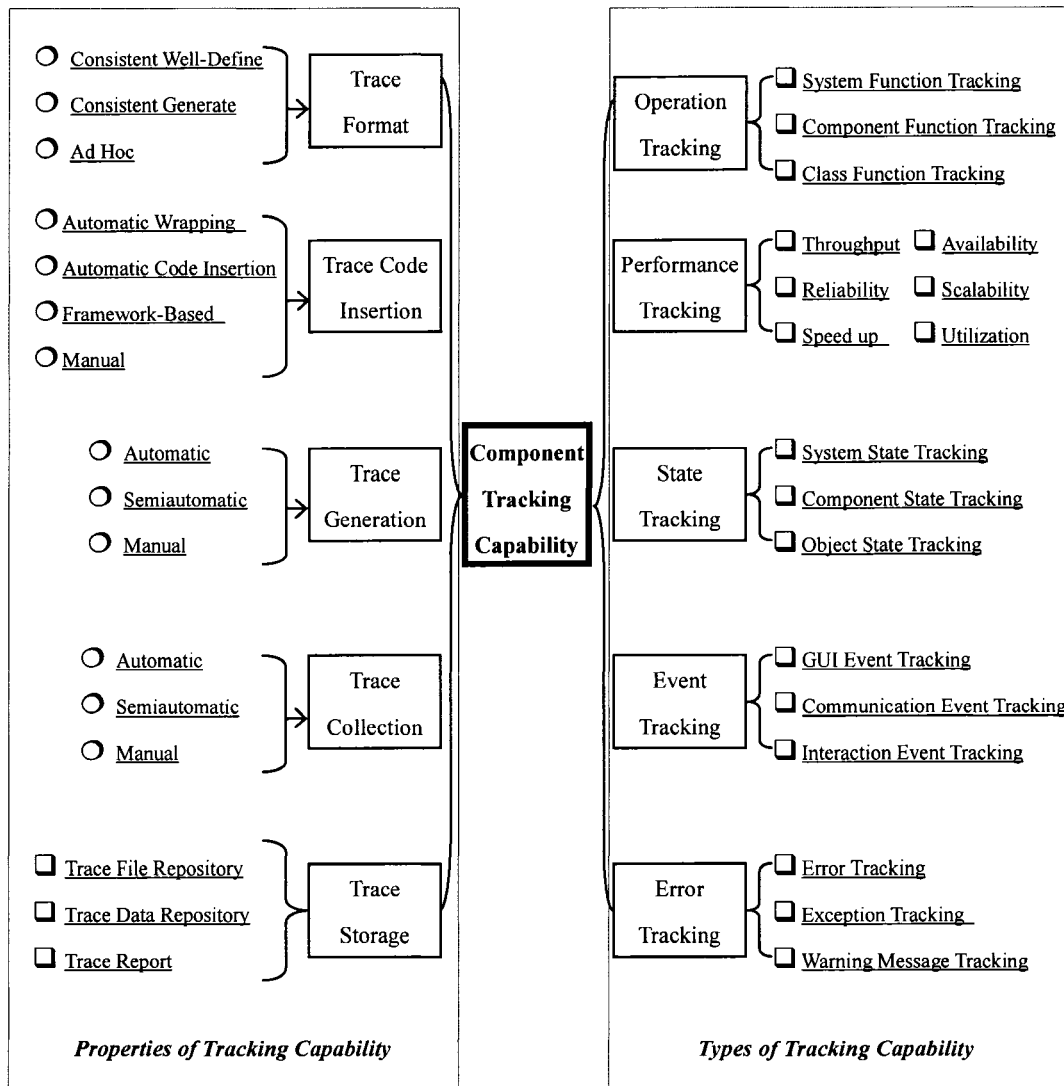


Figure 6. Traceability Model

For example, if an operation tracking mechanism has its own five properties, examiners need to check: whether the trace format of the operation tracking is well-defined, what kind of trace code insertion this operation tracking performs, whether this operation tracking has trace generation or trace collection, what kind of storage this operation tracking has. The following section describes the detail of the two activities.

### **Activity 1. Checking Requirements for Five Types of Component Tracking**

#### **Capability**

There are five types of component tracking capability: operational tracking, performance tracking, state tracking, event tracking and error tracking.

**1. Operational Tracking:** It refers to the ability of recording the interactions of component operations. Also it can be further categorized into tree groups: system function tracking, component function tracking and class function tracking.

**2. Performance Tracking:** It refers to the ability of recording the performance data and benchmarks for each function of a component in a given platform and environment. Performance trace is very useful for developers and testers to identify the performance bottlenecks and issues in performance tuning and testing.

Performance tracking usually can be discussed in six fields: throughput, reliability, speed up, availability, scalability and utilization.

**3. State Tracking:** It refers to the ability of tracking the object states or data states in a component. It can be further categorized into three groups: system state tracking, component state tracking and object state tracking.

**4. Event Tracking:** It refers to the ability of recording the events and sequences that have occurred in a component. It can be further categorized into three groups: GUI event tracking, communication event tracking, and interaction event tracking.

**5. Error Tracking:** It refers to the ability of recording the error messages generated by a component. It can be further categorized into three groups: error tracking, exception tracking and warning message tracking.

## **Activity 2. Checking Requirements for Five Properties of Component Tracking**

### **Capability**

There are five properties of component tracking capability: trace format, trace code insertion, trace generation, trace collection, and trace storage.

**1. Trace Format:** It refers to standardized trace format and understandable trace messages. The trace format should be consistent well-defined to include all necessary information for engineers. For example, the trace format should include class function name, component identifier and thread identifier.

**2. Trace Code Insertion:** It refers to the ability of inserting or deleting trace code into software components. This could reduce the workload for engineers to increase the traceability and the cost for the maintenance of component-based software. The solution for trace code insertion can be categorized into four levels: automatic wrapping, automatic code insertion, framework-based insertion and manual insertion.

**3. Trace Generation:** It refers to the flexibility on selecting different trace types. Since each trace type has its special usage on testing and maintenance of a component-based program, it is important to provide diverse trace types and facilities

for engineers to use according to their needs. The solution for trace generation can be categorized into three levels: automatic, semiautomatic and manual levels.

**4. Trace Collection:** It refers to the ability of collecting trace messages. The solution for trace generation can be categorized into three levels: automatic, semiautomatic and manual collection.

**5. Trace Storage:** It refers to the ability to manage and store the selective trace messages. Engineers need this function in debugging, system testing and maintenance to help them manage and monitor the trace messages. The solution for trace generation can be categorized into three ways: trace file repository, trace data repository and trace report.

### **3.1.5 Requirements Verification for Test Support Capability**

Component test support capability refers to the ability of checking four types of component test support capability. These activities are described in the test support capability model, which is shown in Figure 7.





Figure 7. Test Support Capability Model

## Checking For the Four Types of Test Support Capability

There are four types of test support capability: test generation, test management capability, test coverage and component test scripting capability.

**1. Test Generation:** It refers to the extent to which component tests and test scripts can be generated by using systematic test generation methods and tools. There are two major component test generations, which are white-box and black-box generations. White-box test generation can be categorized into five aspects, which are basic path testing, data flow testing, branch-based testing, syntax-based testing and state-based testing. Black-box test generation can be also categorized into five aspects, which are boundary value testing, equivalence partitioning, graphed-based testing, random testing and requirements-based testing.

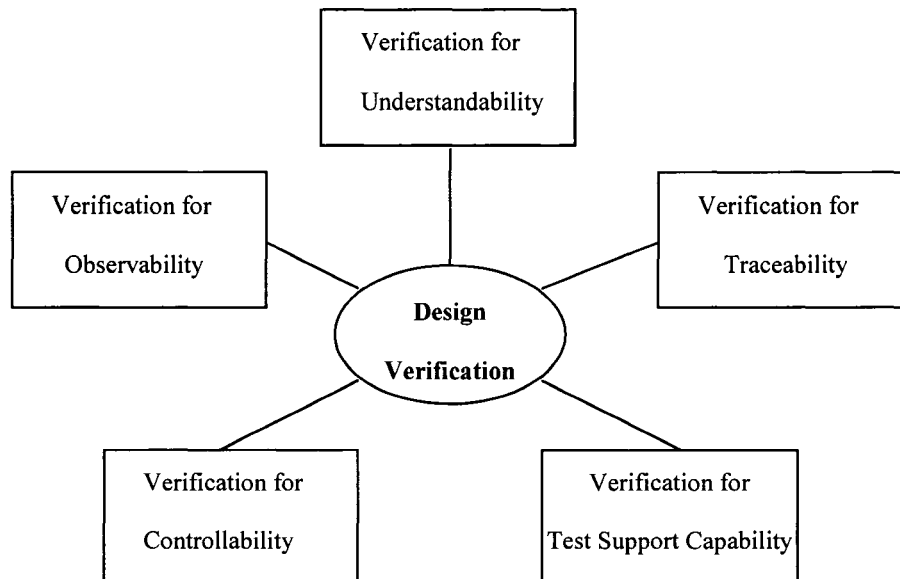
**2. Test Management Capability:** It refers to how well a systematic solution is provided to support the management of various types of test information. There are three common test managements, which are problem, test and suit management. Each test management capability can be systematic with tools, systematic without tools, or ad-hoc.

**3. Test Coverage:** It refers to the extent to which the component test coverage could be easily measured, monitored, and reported. In order to increase this capability, engineers need two things; the first is a set of well-defined component coverage criteria and standards, and the other is the test coverage analysis function with testing tools that provides measure, monitor and report.

**4. Component Test Scripting Capability:** It refers to how easy it is for test engineers and test developers to create, maintain, and execute test scripts, test drivers and stubs. This capability is essential to component testing and evaluation for users.

### 3.2 Design Verification for Component Testability

This verification focuses on how the current component designs to meet the given testability requirements, including component model, architecture, interfaces for testing, and test facility design. The process model, which is shown in Figure 8, describes a series of procedure for verify the design of software components by five factors, which are understandability, observability, controllability, traceability and test support capability.



**Figure 8. Design Verification for Component Testability**

### **3.2.1 Design Verification for Understandability**

Verification of Component understandability in the design phase refers to the verification activities of functional and non-functional design.

#### **Activity 1. Verification of Non-functional Design**

There will be a series of testing reviews to ensure that non-functional design documents provide a consistent, complete, and correct solution for the corresponding requirements of availability, performance, scalability, throughput, reliability, and utilization.

**Consistency:** To ensure that the requirements and design are consistent, we will need to conduct a series of document reviews, especially during transition periods between phases. Each requirement will be compared its with design to maximize consistency.

**Completeness:** A complete set of design document is the one that correctly and completely states the desires and needs from the requirement. Completeness ensures that no necessary elements are missing from the requirement phase to design phase.

**Correctness:** The design solution must be technically correct and also can be judged based on standards. The content of each section must fulfill the intended purpose.

#### **Activity 2. Verification of Functional Design**

There will be a series of testing reviews to ensure that non-functional design documents provide a consistent, complete, and correct solution for users to observe its behavior, control its behavior, and trace its operation.

### 3.2.2 Design Verification for Observability

Component observability in the design phase refers to the activity of checking whether the input and corresponding output can be observed or not. The first thing is to make sure that the design for observability is consistent with requirements for observability. Thus, we need to check whether the corresponding design provides consistent, complete, and correct solutions that allow users to observe input/output data, communication data, GUI events and related input/output data, and function operation/behavior.

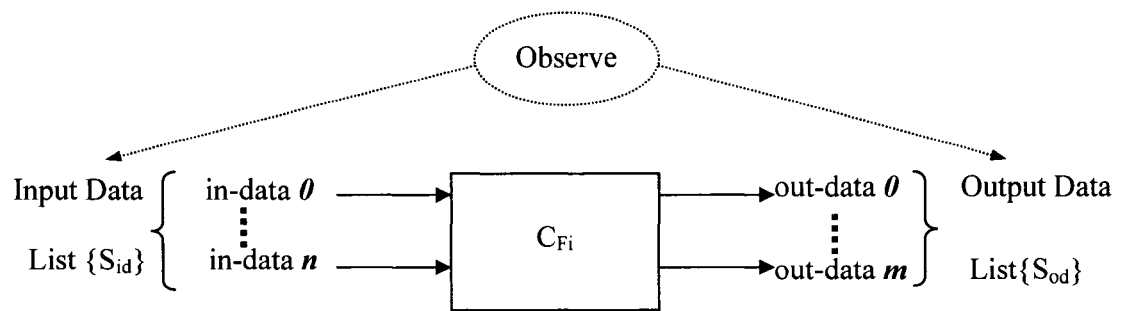
Next, the software component can be regarded as a functional black box with a set of inputs and corresponding outputs. The task is to observe and check the mapping relationship between the inputs and corresponding outputs. The hidden inputs and missing outputs are the defects causing bad testability. In addition, a mechanism that allows users to monitor and observe the mapping relationship will also automatically increase the observability. There are four types of software component observability: component data, component communication, component GUI, and component function observability.

The following section presents four different kinds of verification of component observability in the design phase. For example, in the first activity, checking the design for component data observability, input data numbers and its corresponding output data numbers need to be examined. In other words, if every input datum can be mapped to its corresponding output datum, then the component has good data observability. Conversely, if an input datum cannot be mapped to a corresponding output datum, this leads to an

unequal outcome, which shows itself as poor data observability. If both input and output data are hidden or unreachable, it would be the worst condition of data observability.

### Activity 1. Checking Design for Component Data Observability

The diagram, as shown in Figure 9, demonstrates the relationship between input and output data. By giving a list of input data list  $S_{id}$  {in-data<sub>0</sub>...in-data<sub>n</sub>}, there should be a list of corresponding output data list  $S_{od}$  {out-data<sub>0</sub>...out-data<sub>m</sub>}.



**Figure 9. Data Observability**

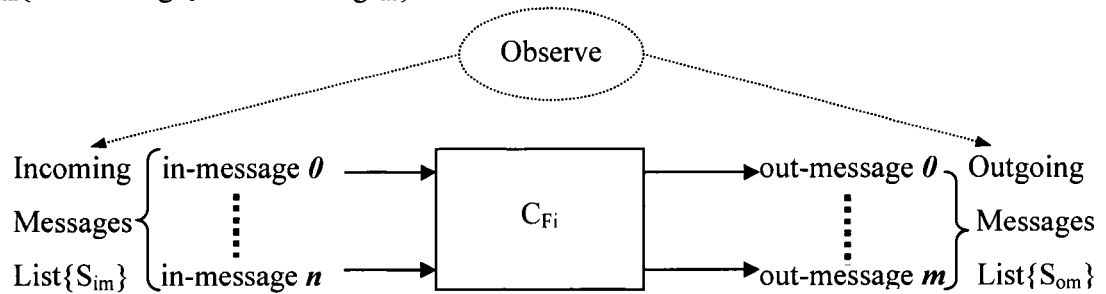
Having hidden and unreachable data is the major reason to lower data observability. A component with good data observability should have the same amount of input data and output data. The data observability evaluation table is shown in Table 1.

Mapping Relationship	Observability
$ S_{id}  =  S_{od}  \neq 0$	Good
$ S_{id}  \neq  S_{od} $	Poor
$ S_{id}  =  S_{od}  = 0$	Worst

**Table 1. Data Observability Evaluation Table**

### Activity 2. Checking Design for Component Communication Observability

The diagram, as shown in Figure 10, demonstrates the relationship between incoming and outgoing messages. By giving a list of incoming messages list  $S_{im}$   $\{in-message_0 \dots in-message_n\}$ , there should be a list of corresponding output message list  $S_{om}\{out-message_0 \dots out-message_m\}$ .



**Figure 10. Communication Observability**

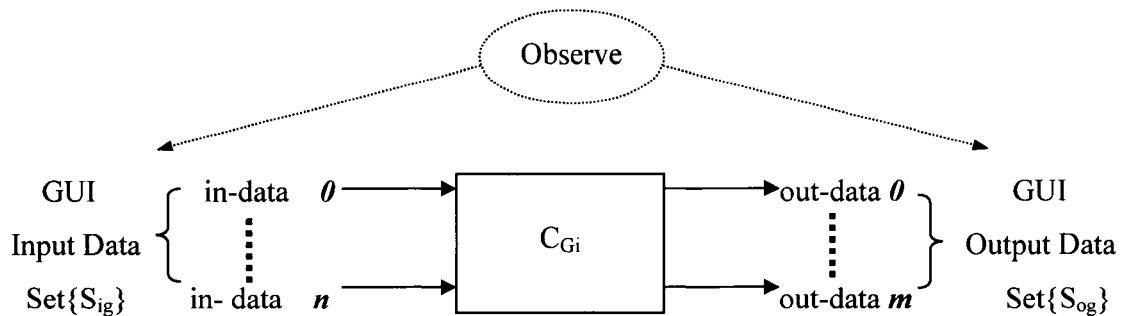
Having hidden and unreachable message is the major reason to lower communication observability. A component with good communication observability should have the same amount of incoming and outgoing messages. The communication observability evaluation table is shown in Table 2.

Mapping Relationship	Observability
$ S_{im}  =  S_{om}  \neq 0$	Good
$ S_{im}  \neq  S_{om} $	Poor
$ S_{im}  =  S_{om}  = 0$	Worst

**Table 2. Communication Observability Evaluation Table**

### Activity 3. Checking Design for Component GUI Observability

GUI observability covers GUI data observability and GUI event observability. The diagram, as shown in Figure 11, demonstrates the relationship between GUI input and output data. By giving a list of GUI input data list  $S_{ig}\{\text{in-data}_0 \dots \text{in-data}_n\}$ , there should be a list of corresponding output data list  $S_{og}\{\text{out-data}_0 \dots \text{out-data}_m\}$ .



**Figure 11. GUI Data Observability**

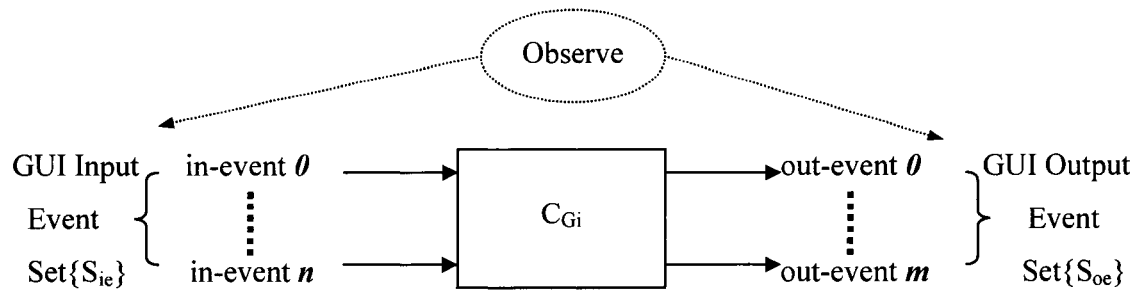
Having hidden and unreachable GUI data is the major reason to lower GUI data observability. A component with good GUI data observability should have the same amount of input and output GUI data. The GUI data observability evaluation table is shown in Table 3.

Mapping Relationship	Observability
$ S_{ig}  =  S_{og}  \neq 0$	Good
$ S_{ig}  \neq  S_{og} $	Poor
$ S_{ig}  =  S_{og}  = 0$	Worst

**Table 3. GUI Data Observability Evaluation Table**



The diagram, as shown in Figure 12, demonstrates the relationship between GUI input and output event. By giving a list of GUI input event list  $S_{ie}\{\text{in-event}_0 \dots \text{in-event}_n\}$ , there should be a list of corresponding output data list  $S_{og}\{\text{out-data}_0 \dots \text{out-data}_m\}$ .



**Figure 12. GUI Event Observability**

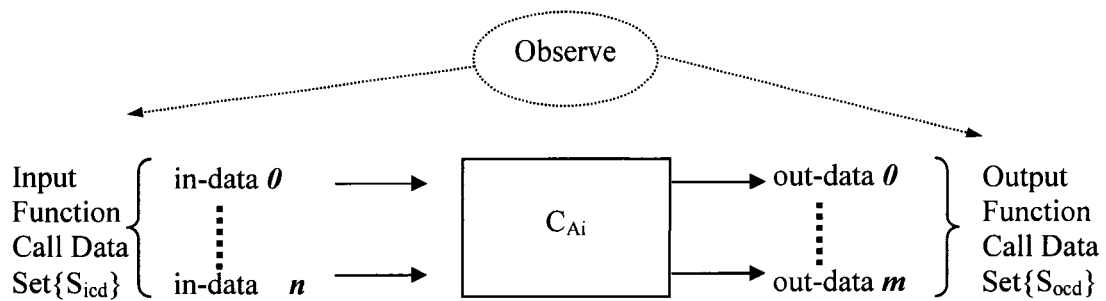
Having hidden and unreachable GUI event is the major reason to lower GUI event observability. A component with good GUI event observability should have the same amount of input and output GUI event. The GUI event observability evaluation table is shown in Table 4.

Mapping Relationship	Observability
$ S_{ie}  =  S_{oe}  \neq 0$	Good
$ S_{ie}  \neq  S_{oe} $	Poor
$ S_{ie}  =  S_{oe}  = 0$	Worst

**Table 4. GUI Event Observability Evaluation Table**

#### Activity 4. Checking Design for Component Function Observability

Component function call observability covers observability of function call data and function call messages. The diagram, as shown in Figure 13, demonstrates the relationship between input and output function call data. By giving a list of input function call data list  $S_{icd}\{in-data_0...in-data_n\}$ , there should be a list of corresponding output function call data list  $S_{ocd}\{out-data_0...out-data_m\}$ .



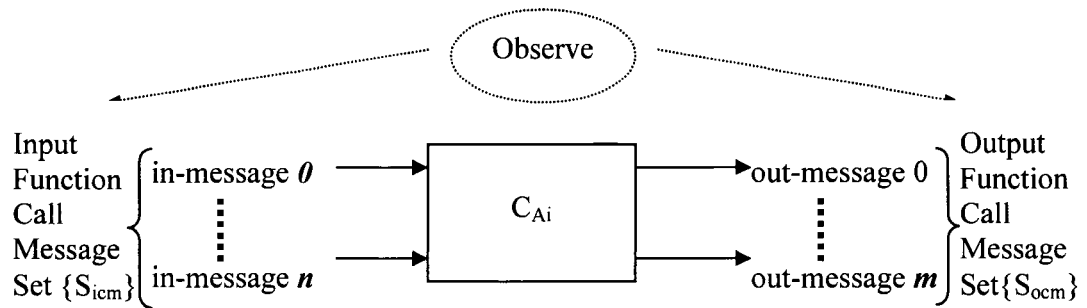
**Figure 13. Function Call Data Observability**

Having hidden and unreachable function call data is the major reason to lower function call data observability. A component with good function call data observability should have the same amount of input and output function call data. The function call data observability evaluation table is shown in Table 5.

Mapping Relationship	Observability
$ S_{icd}  =  S_{ocd}  \neq 0$	Good
$ S_{icd}  \neq  S_{ocd} $	Poor
$ S_{icd}  =  S_{ocd}  = 0$	Worst

**Table 5. Function Call Data Observability Evaluation Table**

The diagram, as shown in Figure 14, demonstrates the relationship between GUI input and output data. By giving a list of GUI input function call message list  $S_{icm}\{in-message_0 \dots in-message_n\}$ , there should be a list of corresponding output function call message list  $S_{og}\{out-message_0 \dots out-message_m\}$ .



**Figure 14. Function Call Message Observability**

Having hidden and unreachable function call message is the major reason to lower function call message observability. A component with good function call message observability should have the same amount of input and output function call message. The function call message observability evaluation table is shown in Table 6.

Mapping Relationship	Observability
$ S_{icm}  =  S_{ocm}  \neq 0$	Good
$ S_{icm}  \neq  S_{ocm} $	Poor
$ S_{icm}  =  S_{ocm}  = 0$	Worst

**Table 6. Function Call Message Observability Evaluation Table**

### 3.2.3 Design Verification for Controllability

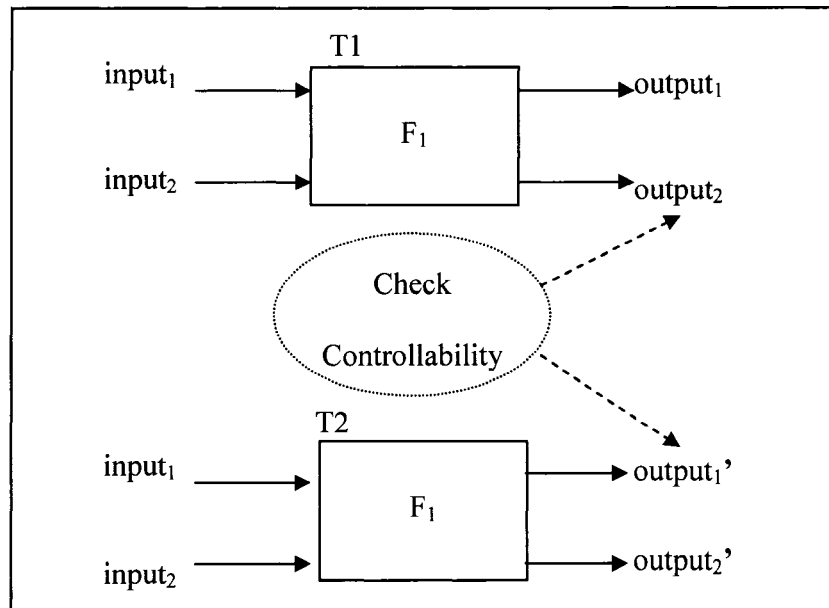
There are two activities that verify the design for controllability. One is by checking the built-in mechanism design, and the other is by checking the API design.

#### Activity 1. Checking the Built-in Mechanism Design for Component Controllability

The major purpose of checking the built-in mechanism design is to verify whether the following eight essential characteristics are consistent with their requirements. The eight essential characteristics, once again, are: component behavior collection, component feature customization, component installation & deployment, component environment controller, component execution controller, component state-based behavior controller, component test controller through interfaces, and component functional feature controller. In addition, it is necessary to ensure that the design solutions are complete and correct.

#### Activity 2. Check API Design for Component Controllability

By giving T1, T2, the same component  $F_1$ , the mapping relationship can be observed between input and output, as shown in Figure15. In component  $F_1$ , the input is controllable form output if output is uniquely determined by input, i.e., if  $[(input1, input2), (output1,output2)], [(input1, input2), (output1',output2')]\in F_1 \Rightarrow output1 = output1'$  and  $output2 = output2'$ . In other words, for the same input (data, message, GUI data, function call data), it should always receive the same output for a better component controllability.



**Figure 15. Design for Controllability**

### 3.2.4 Design Verification for Traceability

There are two activities of verifying design for traceability. One is to check the built-in mechanism design and the other is to check the API design.

#### Activity 1. Checking the Built-in-Mechanism Design for Component Traceability

The major purpose of checking the built-in mechanism design is to verify whether the following ten essential characteristics are consistent with their requirements. The ten essential characteristics, once again, are: trace format, trace code insertion, trace generation, trace collection, trace storage, operational tracking, performance tracking, state tracking, event tracking and error tracking are consistent with its requirement or not. In addition, it is necessary to assure that the design solutions are complete and correct.

**Activity 2. Checking the API Design for Component Traceability**

A design with good traceability has seven features: including source code, allowing code separation, considering overhead, lowering complexity, heighten flexibility, providing applicability, and including applicable component.

**3.2.5 Design Verification for Test Support Capability**

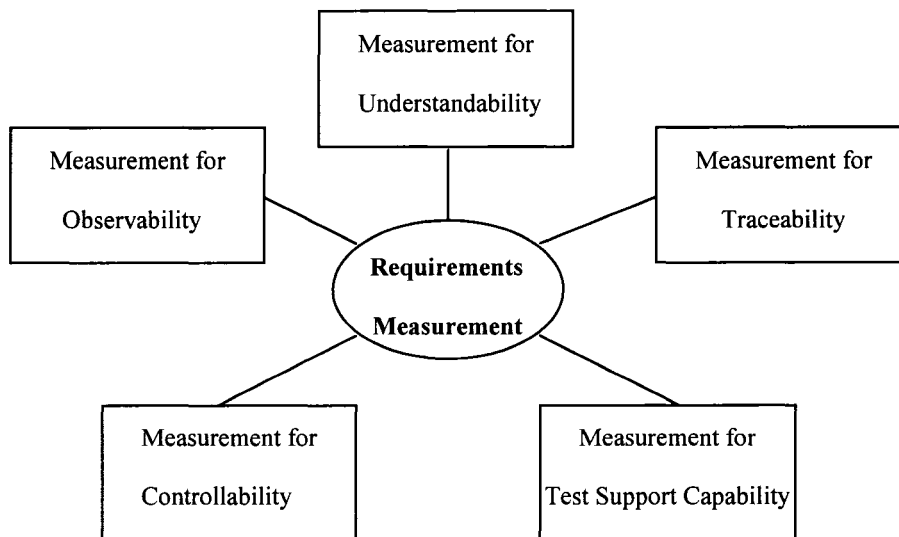
The major purpose of checking the design is to verify whether the four following test support capability are consistent with their requirements. The four essential characteristics, once again, are: test generation mechanism, test management capability, test coverage, and test scripting capability. In addition, it is necessary to ensure the design solutions are complete and correct.

## 4. Measurement of Component Testability

During a software component development cycle, software testability measurement refers to the activities and methods that study, analyze, and measure software testability. The main purpose of software testing review is to evaluate whether the requirement has been satisfied; however, the major objective of software component testability measurement is to evaluate the quality of software components by the given testability factors. The measurement consists of two phases: requirements and design.

### 4.1 Requirements Measurement for Component Testability

Evaluation tables will be created based on the requirement of testability model described in the previous chapter. The values in the evaluation table have been purposefully assigned in order to calculate testing points. These values can be changed differently according to different examiner views or purpose. In other words, examiners can increase certain values when addressing certain features. The idea of testing points are similar to the Albrecht's (1979) idea of the *function point*, which has been used to measure software size and productivity. Thus, testing points are defined in order to measure and represent the different levels of component testability. For each testability model, a designed metric measures the value of the component testability. The process model, which is shown in Figure 16, describes a series of procedures to measure the requirements of software components by five metrics: understandability, observability, controllability, traceability, and test support capability metrics.



**Figure 16. Requirements Measurement for Testability**

#### **4.1.1 Requirements Measurement for Understandability**

Measurement of understandability is based on two activities. The first activity is to design evaluation tables, and the second activity is to design the average understandability metrics for requirements.

##### **Activity 1. Evaluation Tables for Functional and Non-functional Requirements**

As described in chapter 3.1.1, verification of requirements is divided into two parts. Consequently, there are two corresponding evaluation tables to assess functional and non-functional requirements.

The evaluation table, shown in Table 7, checks the availability of each feature for each functional requirement. A testable and measurable description increases the testing points, or, in other words, increases the testability. Conversely, less testable or less



measurable requirements will lead to lower testability. For example, if the traceable operation is available in the functional requirement, a value of '0.5' is given in the evaluation table; if not, a value of '0' is given. If this traceable operation requirement is testable and measurable, as defined in the previous chapter, the sum of the testing points is assigned the value of '1.0'.

For each Functional Requirement	Available?	Testable?	Measurable?	Testing Points (SUM)
Understandable Requirement	0.5	0.25	0.25	
Observable Behavior	0.5	0.25	0.25	
Controllable Behavior	0.5	0.25	0.25	
Traceable Operation	0.5	0.25	0.25	
Total Testing Points of Each Functional Requirement (TPEFR)				
Average of Total Testing Points of Each Functional Requirement (ATPEFR)				

**Table 7. Evaluation Table for Functional Requirements**

After evaluating each functional requirement, we obtain the TPEFR (Testing Points of Each Functional Requirement). Testing points are numbers that represent the level of each testing character, and would be measured and given by testing engineers. Testing points of functional requirements (FReq) are the average testing points of TPEFR.

$$ATPEFR = \frac{1}{4} (TPEFR)$$

$$FReq = \frac{1}{n} \sum_{i=1}^n ATPEFR_i \quad \text{Where, } n = \text{Numbers of Functional Requirement,}$$

$$FReq_{Max} = 1, FReq_{Min} = 0$$

The evaluation table, as shown in Table 8, checks the availability of each feature for each non-functional requirement. In the same manner that evaluation tables verify

functional requirements, a testable and measurable description increases the testing points or, in other words, increases the testability.

Non-Functional Requirements	Available?	Testable?	Measurable?	Testing Points(SUM)
Environment Requirements	0.5	0.25	0.25	
Availability Requirements	0.5	0.25	0.25	
Performance Requirements	0.5	0.25	0.25	
Scalability Requirements	0.5	0.25	0.25	
Throughput Requirements	0.5	0.25	0.25	
Reliability Requirements	0.5	0.25	0.25	
Utilization Requirements	0.5	0.25	0.25	

**Table 8. Evaluation Table for Non-Functional Requirements**

After evaluating each non-functional requirement, we obtain the NFReq (testing points of non-functional requirements).

$$\text{NFReq} = \frac{1}{7} \sum_{i=1}^7 \text{NFReq}_i, \text{ Where } \text{NFReq}_{\text{Max}} = 1, \text{NFReq}_{\text{Min}} = 0$$

### **Activity 2. Average Understandability Metrics for Requirements (AUMR)**

After summing the Freq and NFReq, and dividing the total by 2, we obtain the average understandability metrics.

$$\text{AUMR} = \frac{1}{2} (\text{FReq} + \text{NFReq})$$

AUMR represents the understandability for component requirements. A value of '1.0' means that the requirements have perfect understandability.

### 4.1.2 Requirements Measurement for Observability

Measurement for observability is based on two activities. The first activity is to design an evaluation table, and the second is to design the average observability metrics for requirements.

#### Activity 1. An Evaluation Table for Observability in Requirements

As described in chapter 3.1.2, there are four types of component observability. Consequently, the evaluation table, as shown in Table 9, checks the types of observability and the ease of the capability to observe data. There are three levels, based on ease of observation, to observe data. These levels are: automatic, semiautomatic, and manual.

Types of Observability(TO)	Count	TP	Types of Observability(TO)	Count	TP
Data Observability	Select		GUI Observability	Select	
Automatic	1.0		Automatic	1.0	
Semiautomatic	0.8		Semiautomatic	0.8	
Manual	0.6		Manual	0.6	
Communication Observability	Select		Function Observability	Select	
Automatic	1.0		Automatic	1.0	
Semiautomatic	0.8		Semiautomatic	0.8	
Manual	0.6		Manual	0.6	

**Table 9. Evaluation Table for Observability in Requirements**

#### Activity 2. Average Observability Metrics for Requirements (AOMR)

Average Observability Metrics for Requirements are calculated by dividing the total testing points of each separate type of observability (TO) by 4.

$$AOMR = \frac{1}{4} \sum_{i=1}^4 TO_i$$

### 4.1.3 Requirements Measurement for Controllability

Measurement for controllability is based on two activities. The first activity is to design evaluation tables, and the second is to design the average controllability metrics for requirements.

#### Activity 1. Evaluation Tables of Observability for Requirements

As described in chapter 3.1.3, verification for requirements refers to the activity of verifying the five types and three properties of component controllability. Consequently, there are two corresponding evaluation tables, shown in Table 10 and Table 11, checking requirements for the five types and the three properties of component controllability.

#### Evaluation Table 1-Five Types of Component Controllability

Testing Points for Types of Controllability in Requirements (TPTCR) =  $\frac{1}{5} \sum_{i=1}^5 TC_i$

Types of Controllability (TC)	Count	TP	Type of Controllability (TC)	Count	TP
Component Environment Controller	SUM		Component State-Based Behavior Controller	SUM	
Installation Capability	0.33		Transition Control Function	0.33	
Configuration Set-up Capability	0.33		State Control Function	0.33	
Deployment Capability	0.33		Reset Function	0.33	
Component Execution Controller	SUM		Component Test Controller through Interfaces	SUM	
Test Mode Execution	0.33		Function Invocation Control	0.2	
Control Mode Execution	0.33		Input Data Setup	0.2	
Normal Function Mode Execution	0.33		Output Data Monitor	0.2	
Component Functional Feature	SUM		Incoming Message Setup	0.2	
Function Enable Capability	0.5		Outgoing Message Monitor	0.2	
Function Disable Capability	0.5				

**Table 10. Evaluation Table for Controllability Types in Requirements**

### Evaluation Table 2- Three Properties of Component Controllability

$$\text{Testing Points for Properties of Controllability in Requirement (TPPCR)} = \frac{1}{3} \sum_{i=1}^3 PC_i$$

Properties of Controllability (PC)	Count	TP	Properties of Controllability (PC)	Count	TP
Component Behavior Collection	Select		Component Installation & Deployment	Select	
Automatic	1.0		Automatic	1.0	
Semiautomatic	0.8		Semiautomatic	0.8	
Manual	0.6		Manual	0.6	
Component Feature Customization	SUM				
Function Customization	0.33				
Interface Customization	0.33				
Meta-data Customization	0.33				

**Table 11. Evaluation Table for Controllability Properties in Requirements**

#### Activity 2. Average Controllability Metrics of Requirements (ACMR)

Average controllability metrics of requirements are the product of TPTCR and TPPCR.

$$\text{ACMR} = \text{TPTCR} * \text{TPPCR}$$

#### 4.1.4 Requirements Measurement for Traceability

Measurement of traceability is based on two activities. The first activity is to design evaluation tables and the second is to design the average traceability metrics for requirements.

##### Activity 1. Evaluation Tables for Traceability in Requirements

As described in chapter 3.1.4, verification of requirements refers to the activity of verifying the five types of component controllability and the five properties of component traceability. Consequently, there are two corresponding evaluation tables, which are shown as Table 12 and Table 13, checking the requirements for the five types and properties of component controllability.

**Evaluation Table 1 — Five types of component traceability**

$$\text{Testing Points for Types of Traceability in Requirements (TPTTR)} = \frac{1}{5} \sum_{i=1}^5 TT_i$$

Types of Tracking Capability(TT)	Count	TP	Types of Tracking Capability(TT)	Count	TP
Operation Tracking	SUM		State Tracking	SUM	
System Function Tracking	0.33		System State Tracking	0.33	
Component Function Tracking	0.33		Component State Tracking	0.33	
Class Function Tracking	0.33		Object State Tracking	0.33	
Performance Tracking	SUM		Event Tracking	SUM	
Throughput	0.16		GUI Event Tracking	0.33	
Reliability	0.16		Communication Event Tracking	0.33	
Speed up	0.16		Interaction Event Tracking	0.33	
Availability	0.16		Error Tracking	SUM	
Scalability	0.16		Error Tracking	0.33	
Utilization	0.16		Exception Tracking	0.33	
			Warning Message Tracking	0.33	

**Table 12. Evaluation Table for Traceability Types in Requirements****Evaluation Table 2 — Five properties of component traceability**

$$\text{Testing Points for Properties of Traceability in Requirement (TPPTR)} = \frac{1}{5} \sum_{i=1}^5 PT_i$$

Properties of Tracking Capability(PT)	Count	TP	Properties of Tracking Capability(PT)	Count	TP
Trace Format	<b>Select</b>		Trace Generation	<b>Select</b>	
Consistent Well-Define	<b>1.0</b>		Automatic	<b>1.0</b>	
Consistent Generate	<b>0.8</b>		Semiautomatic	<b>0.8</b>	
Ad Hoc	<b>0.6</b>		Manual	<b>0.6</b>	
Trace Code Insertion	<b>Select</b>		Trace Collection	<b>Select</b>	
Automatic Wrapping	<b>1.0</b>		Automatic	<b>1.0</b>	
Automatic Code Insertion	<b>0.8</b>		Semiautomatic	<b>0.8</b>	
Framework-Based	<b>0.6</b>		Manual	<b>0.6</b>	
Manual	<b>0.4</b>		Trace Storage	<b>SUM</b>	
			Trace File Repository	<b>0.33</b>	
			Trace Data Repository	<b>0.33</b>	
			Trace Report	<b>0.33</b>	

**Table 13. Evaluation Table for Traceability Properties in Requirements**

### **Activity 2. Average Traceability Metrics for Requirements (ATMR)**

Average traceability metrics for requirements are the product of TPTTR and TPPTR.

$$\text{ATMR} = \text{TPTTR} * \text{TPPTR}$$

### **4.1.5 Requirements Measurements for Test Support Capability**

Measurement of test support capability is based on two activities. The first activity is to design evaluation tables and the second activity is to design the average test support capability metrics for requirements.

#### **Activity 1. Evaluation Tables for Test Support Capability in Requirements**

As described in chapter 3.1.5, verification of requirements refers to the four aspects of component test support capability. Consequently, there are four corresponding evaluation

tables, which are as shown from Table 14 to Table 17, checking requirements for test support capability.

**Evaluation Table 1 — Component Test Generation in Requirements**

Testing Points for Test Generation (TPTG) =  $\frac{1}{2}$  (White-Box Test Generation TP + Black-Box Test Generation TP)

Component Test Support Capability	Count	TP	Component Test Support Capability	Count	TP
Component White-Box Test Generation	SUM		Component Black-Box Test Generation	SUM	
Basic Path Testing	0.2		Boundary Value Testing	0.2	
Data Flow Testing	0.2		Equivalence Partitioning	0.2	
Branch-Based Testing	0.2		Graphed-based Testing	0.2	
Syntax-Based Testing	0.2		Random Testing	0.2	
State-Based Testing	0.2		Requirements-Based Testing	0.2	

**Table 14. Evaluation Table for Component Test Generation in Requirements**

**Evaluation Table 2 — Component Test Management in Requirements**

Testing Points for Test Management (TPTM) =  $\frac{1}{3}$  (Problem Management TP + Test Management TP+ Suit Management TP)

Component Test Support Capability	Count	TP	Component Test Support Capability	Count	TP
Problem Management	Select		Suit Management	Select	
Systematic with tools	1.0		Systematic with tools	1.0	
Systematic without tools	0.8		Systematic without tools	0.8	
Ad-hoc	0.6		Ad-hoc	0.6	
Test Management	Select				
Systematic with tools	1.0				
Systematic without tools	0.8				
Ad-hoc	0.6				

**Table 15. Evaluation Table for Component Test Management in Requirements**



**Evaluation Table 3 — Component Test Coverage in Requirements**

Testing Points for Test Coverage (TPTC) =  $\frac{1}{3}$  ( Test Coverage Criteria TP + Test

Coverage Standards TP+ Test Coverage Analysis Tools TP)

Component Test Support Capability	Count	TP	Component Test Support Capability	Count	TP
Test Coverage Criteria	Select		Test Coverage Analysis Tools	SUM	
Well-defined Criteria	1.0		Provide Measure	0.33	
Defined Criteria	0.8		Provide Monitor	0.33	
Non-defined Criteria	0.6		Provide Report	0.33	
Test Coverage Standards	Select				
Well-defined Standards	1.0				
Defined Standards	0.8				
Ad-hoc	0.6				

**Table 16. Evaluation Table for Component Test Coverage in Requirements**

**Evaluation Table 4 — Component Test Scripting in Requirements**

Testing Points for Test Scripting Capability (TPTSC) =  $\frac{1}{3}$  ( Test Scripts TP + Test

Drivers TP + Test Stubs TP)

Component Test Support Capability	Count	TP	Component Test Support Capability	Count	TP
Test Scripts	Select		Test stubs	SUM	
Create/Editing	1.0		Create	0.33	
Book-keeping/ Management	0.8		Maintain	0.33	
Execute(EXE)	0.6		Execute	0.33	
Test Drivers	Select				
Automatic Creation	1.0				
Systematic Creation	0.8				
Ad-hoc Creation	0.6				

**Table 17. Evaluation Table for Component Test Scripting in Requirements**

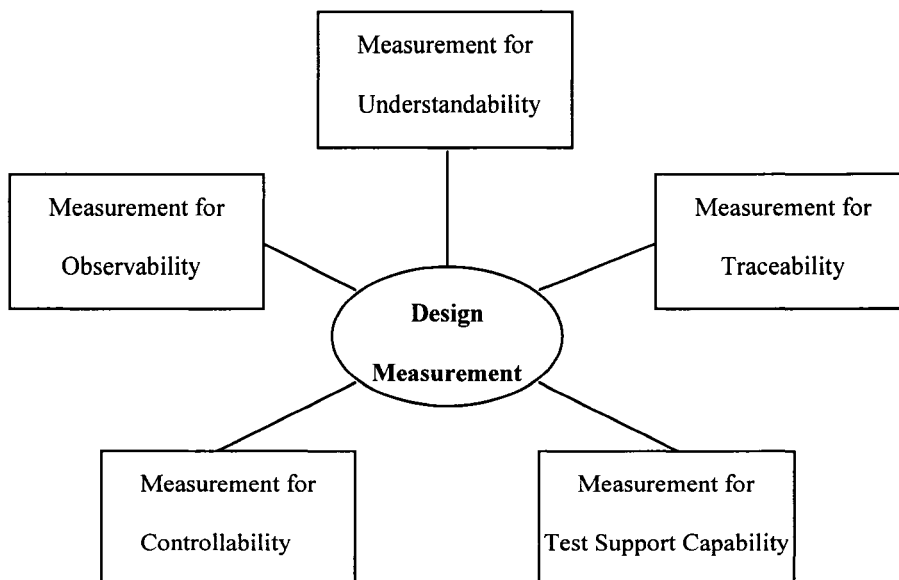
### Activity 2. Average Test Support Capability Metrics for Requirements (ATSMR)

Average test support capability metrics of requirement are the total of the four testing points and divide by 4.

$$\text{ATSMR} = \frac{1}{4} (\text{TPTG} + \text{TPTM} + \text{TPTC} + \text{TPTSC} )$$

### 4.2 Design Measurement for Component Testability

Evaluation tables will be created based on the design of testability model described in the previous chapter. For each testability model, a designed metric can measure the value of the component testability. The process model, which is shown in the Figure 17, describes a series of procedures to measure the design of software component by five metrics. These five metrics are: understandability, observability, controllability, traceability, and test support capability metrics.



**Figure 17. Design Measurement for Component Testability**

#### 4.2.1 Design Measurement for Understandability

Measurement of understandability is based on two activities. The first activity is to design evaluation tables and the second is to design the average understandability metrics for design.

##### Activity 1. Evaluation Tables for Functional and Non-functional Requirements

As described in chapter 3.2.2, verification of requirements is divided into two parts. Consequently, there is a corresponding evaluation table, as shown in Table 18, checking whether the design consist with its requirements and the design is a complete and correct solution. Testing Points of functional and non-functional design are the total counts of their features.

Understandability	Count	TP	Understandability	Count	TP
Functional Design			Non-Functional Design		
Design for Observable Behavior	0.33		Design for Availability	0.16	
Design for Controllable Behavior	0.33		Design for Performance	0.16	
Design for Traceable Operation	0.33		Design for Scalability	0.16	
			Design for Throughput	0.16	
			Design for Reliability	0.16	
			Design for Utilization	0.16	

**Table 18. Evaluation Table for Understandability in Design**

##### Activity 2. Average Understandability Metrics for Design (AUMD)

After summing the Freq and NFRreq, and dividing the total by 2, we obtain the average understandability metrics for design.

$$\text{AUMD} = \frac{1}{2} (\text{FReq} + \text{NFRreq})$$

#### 4.2.2 Design Measurement for Controllability

Measurement of observability is based on two activities. The first activity is to design an evaluation table and the second is to design the average observability metrics for design.

##### Activity 1. Evaluation Tables for Observability in Requirements

As described in chapter 3.2.3, there are six different evaluation tables of component observability. Consequently, here are six corresponding evaluation tables, which are shown from Table 19 to Table 24, checking the relationship between the input and corresponding output. Different weight of percentage will be given based on the ease of observing data.

##### Evaluation Table 1 — Component Data Observability

Testing Points for Component Data Observability (TPDO) = Count \* Weight

Numbers of Data	Count	Observation Level	Weigh
Input Data   =   Output Data   ≠ 0	1.0	Automatic	100%
Input Data   ≠   Output Data	0.8	Semiautomatic	80%
Input Data   =   Output Data   = 0	0.6	Manual	60%

**Table 19. Evaluation Table for Data Observability in Design**

##### Evaluation Table 2 — Component Communication Observability

Testing Points for Component Communication Observability (TPCO) = Count \* Weight

Numbers of Message	Count	Observation Level	Weigh
Incoming Message   =   Outgoing Messages   ≠ 0	1.0	Automatic	100%
Incoming Message   ≠   Outgoing Messages	0.8	Semiautomatic	80%
Incoming Message   =   Outgoing Messages   = 0	0.6	Manual	60%

**Table 20. Evaluation Table for Communication Observability in Design**

### Evaluation Table 3 — Component GUI Data Observability

Testing Points for Component GUI Data Observability (TPGDO) = Count \* Weight

Numbers of GUI Data	Count	Observation Level	Weight
GUI Input Data   =   GUI Output Data   ≠ 0	1.0	Automatic	100%
GUI Input Data   ≠   GUI Output Data	0.8	Semiautomatic	80%
GUI Input Data   =   GUI Output Data   = 0	0.6	Manual	60%

**Table 21. Evaluation Table for GUI Data Observability in Design**

### Evaluation Table 4 — Component GUI Event Observability

Testing Points for Component GUI Event Observability (TPGEO) = Count \* Weight

Numbers of GUI Event	Count	Observation Level	Weight
Input Event   =   Output Event   ≠ 0	1.0	Automatic	100%
Input Event   ≠   Output Event	0.8	Semiautomatic	80%
Input Event   =   Output Event   = 0	0.6	Manual	60%

**Table 22. Evaluation Table for GUI Event Observability in Design**

### Evaluation Table 5 — Component Function Call Data Observability

Testing Points for Component Function Call Data Observability (TPFDO)

= Count \* Weight

Numbers of Function Call Data	Count	Observation Level	Weight
Input Function Call Data   =   Output Function Call Data   ≠ 0	1.0	Automatic	100%
Input Function Call Data   ≠   Output Function Call Data	0.8	Semiautomatic	80%
Input Function Call Data   =   Output Function Call Data   = 0	0.6	Manual	60%

**Table 23. Evaluation Table for Function Call Data Observability in Design**

### Evaluation Table 6 — Component Function Call Message Observability

Testing Points for Component Function Call Message Observability (TPFMO)

= Count \* Weight

Numbers of Function Call Message	Count	Observation Level	Weight
Incoming Function Call Message   =   Outgoing Function Call Message   ≠ 0	1.0	Automatic	100%
Incoming Function Call Message   ≠   Outgoing Function Call Message	0.8	Semiautomatic	80%
Incoming Function Call Message   =   Outgoing Function Call Message   = 0	0.6	Manual	60%

**Table 24. Evaluation Table for Function Call Message Observability in Design**

#### Activity 2. Average Observability Metrics for Design (AOMD)

Average observability metrics for design are calculated by dividing the total of the six testing points by 6.

$$\text{AOMD} = \frac{1}{6} (\text{TPDO} + \text{TPCO} + \text{TPGDO} + \text{TPGEO} + \text{TPFDO} + \text{TPFMO})$$

#### 4.2.3 Design Measurement for Controllability

Measurement of controllability is based on two activities. The first activity is to design evaluation tables and the second is to design the average controllability metrics for design.

#### Activity 1. Evaluation Tables for Controllability in Design

As described in chapter 3.2.3, verification of design refers to the built-in-mechanism and the API design. Consequently, there are two corresponding evaluation tables, which are shown as Table 25 and Table 26, checking the built-in mechanism and the API design for component controllability.

### Evaluation Table 1 — Built-in Mechanism Design

Testing Points for the Built-in Mechanism Design of Component Controllability (TPBDC)

= Total Counts

Built-in Mechanism for controllability	Count	Built-in Mechanism for controllability	Count
Design for collect component behavior	0.125	Design for component execution controller	0.125
Design for customize component feature	0.125	Design for component state-based behavior controller	0.125
Design for installation and deployment mechanism	0.125	Design for component test controller through interfaces	0.125
Design for component environment controller	0.125	Design for component functional feature controller	0.125

**Table 25. Evaluation Table for Built-in Mechanism Design of Controllability**

### Evaluation Table 2 — API Design

Testing Points for the API Design for Component Controllability's (TPADC) =

Total Counts

API design for Controllability		Count	
Same input data	→	Same output data	0.2
Same incoming message	→	Same outgoing message	0.2
Same GUI input data	→	Same GUI output data	0.2
Same input function call data	→	Same output function call data	0.2
Same input function call message	→	Same output function call message	0.2

**Table 26. Evaluation Table for API Design of Controllability**

### Activity 2. Average Controllability Metrics for Design (ACMD)

Average controllability metrics of design are calculated by dividing the total of the

TPBDC and TPADC by 2.

$$ACMD = \frac{1}{2} (TPBDC + TPADC)$$

#### 4.2.4 Design Measurement for Traceability

Measurement of traceability is based on two activities. The first activity is to design evaluation tables and the second is to design the average traceability metrics for design.

##### Activity 1. Evaluation Tables for Traceability in Design

As described in chapter 3.2.4, verification of design refers to the built-in-mechanism and API design. Consequently, there are two corresponding evaluation tables, which are shown as Table 27 and Table 28, checking the built-in mechanism and the API design for component traceability.

##### Evaluation Table 1 — Built-in Mechanism Design

Testing Points for the Built-in Mechanism Design of Component Traceability (TPBDT) =  
Total Counts

Built-in Mechanism for Traceability	Counts	Built-in Mechanism for Traceability	Counts
Design for trace format	0.11	Design for performance tracking	0.11
Design for trace code insertion	0.11	Design for state tracking	0.11
Design for trace collection	0.11	Design for event tracking	0.11
Design for trace storage	0.11	Design for error tracking	0.11
Design for operational tracking	0.11		

**Table 27. Evaluation Table for Built-in Mechanism Design of Controllability**

##### Evaluation Table 2 — API Design

Testing Points for the API Design of Component Traceability (TPADT) =

$$\frac{1}{7} (\text{Total Testing Points})$$



API design for Controllability	Count	TP	API design for Controllability	Count	TP
Tracking	Select		Complexity	Select	
Need Source Code	1.0		High complexity	1.0	
Without Source code	0.5		Low complexity	0.5	
Code Separation	Select		Flexibility	Select	
Allow	1.0		High flexibility	1.0	
Not Allow	0.5		Low flexibility	0.5	
Overhead	Select		Applicability	Select	
Consider Overhead	1.0		OP Trace	1.0	
Without Overhead	0.5		Performance Trace	0.8	
			Applicable Components	Select	
			In-house Components	1.0	
			COTS	0.8	

**Table 28. Evaluation Table for API Design of Controllability**

#### **Activity 2. Average Traceability Metrics for Design (ATMD)**

Average traceability metrics for design are calculated by dividing the total of the TPBDT and TPADT by 2.

$$ATMD = \frac{1}{2} (TPBDT + TPADT)$$

#### **4.2.5 Design Measurement for Test Support Capability**

Measurement of test support capability is based on two activities. The first activity is to design evaluation tables and the second is to design the average test support capability metrics of design.

#### **Activity 1. An Evaluation Table for Test Support Capability in Design**

As described in chapter 3.2.5, verification of design refers to the four aspects of component test support capability. Consequently, there is a corresponding evaluation table, which is as shown in Table 29, checking the design for test support capability.

### Evaluation Table — Component Test Support Capability in Design

Design for Test Support Capability	TP	Design for Test Support Capability	TP
Design for test generation mechanism	0.25	Design for supporting test coverage	0.25
Design for test management capability	0.25	Design for component test scripting capability	0.25

**Table 29. Evaluation Table for Component Test Support Capability in Design**

#### Activity 2. Average Test Support Capability Metrics for Design (ATSMD)

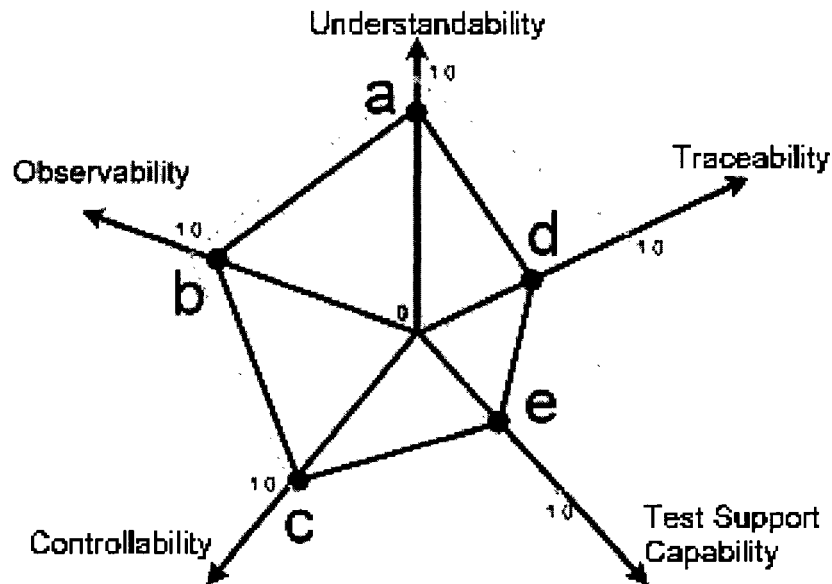
Average test support capability metrics for design is the summation of four testing points.

**ATSMD = Total Testing Points**

#### 4.3 Pentagram Model of Software Testability Measurement

McCall's triangle model of quality was proposed in the early 1970s (McCall et al. 1977). They state that software quality is affected by three factors: product operation, product revision, and product transition, and can be drawn as a triangular model. In this thesis, the same diagram concept is applied in the section presenting the five factors of software testability, but, because there are five factors in the model used here, the diagram is of a pentagram, rather than a triangle.

Using the testability review, a value is given for each of the five factors by its own metric, and a value of 1.0 is the maximum length of each vector. The value can be represented as a point in an X-Y Coordinate system. After connecting each dot, a pentagram is represented, which is shown in Figure18.



**Figure 18. Pentagram Model**

The area of the pentagram is the measurement of component testability. A higher value of the area means better testability. As a result, we can easily measure component testability by the value of the sum. The maximum value of the pentagram's area is approximately 2.4. The following is how the final value is calculated for each pentagram, and how the final metrics are represented for calculating component testability. As we can see, the pentagram can be divided into five triangles. After summing up the areas of these five triangles, we can obtain the area of the pentagram.

For each triangle, the area would be:

$$\frac{1}{2} \lambda_1 \times \lambda_2 \times \sin \alpha \quad \text{where } \lambda_1, \lambda_2 \text{ represent the sides of the triangle, and } \alpha \text{ represents}$$

the angles between the two sides. Angle  $\alpha$  measures 72 degrees, obtained by dividing the turn angle (360 degrees) by 5.

From center of the pentagram, we have five values for each factor: understandability, Observability, Controllability, Traceability, and Test Support capability. In the pentagram mode, these factors, respectively, are designated by the following letters: a, b, c, d, and e. The maximum value of any single factor is 1.0. At this point, the area of the pentagram can be calculated by using the following equations.

### Testability of Pentagram Area

$$= \frac{1}{2} ab \sin 72^\circ + \frac{1}{2} bc \sin 72^\circ + \frac{1}{2} ce \sin 72^\circ + \frac{1}{2} de \sin 72^\circ + \frac{1}{2} ad \sin 72^\circ$$

$$= \frac{1}{2} \sin 72^\circ (ab + bc + ce + de + ad) \cong \frac{1}{2} \times 0.9511 \times (ab + bc + ce + de + ad)$$

$$\cong 0.48 \times (ab + bc + ce + de + ad)$$

In brief, by using the above equation, we define the testability of software components.

$$\text{Testability} = 0.48 \times (ab + bc + ce + de + ad)$$

After calculating the area, we can easily distinguish the testability between two components in each phase. As can be seen in Figure 19, the area of the left pentagram is bigger than the right one, which means that, in that phase, the testability of the left component is better than the right one.

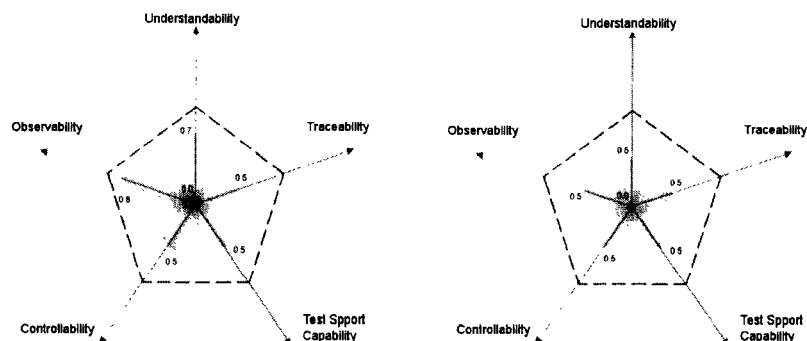


Figure 19. Comparing Two Pentagram Areas

## **5. A Case Study and Application Examples**

This case study will provide a comparison of the requirements and design documents used in the development of the Binary Search Tree application. This application is intended to be small in scope, with a limited capacity, in order to clearly classify two different versions of requirements and design. The purpose is to see whether the testability for an up-to-date version can get a better score than is calculated by the designed metrics. See Appendix B for the requirements and design documents for the two different versions of binary tree application.

### **5.1 A Work Flow to Compare the Two Versions**

There are four steps, shown in Figure 20, which illustrate a series of procedures to compare the two versions of binary tree application. Both requirements and design documents will go through these steps in order to compare their testability using the five factors.

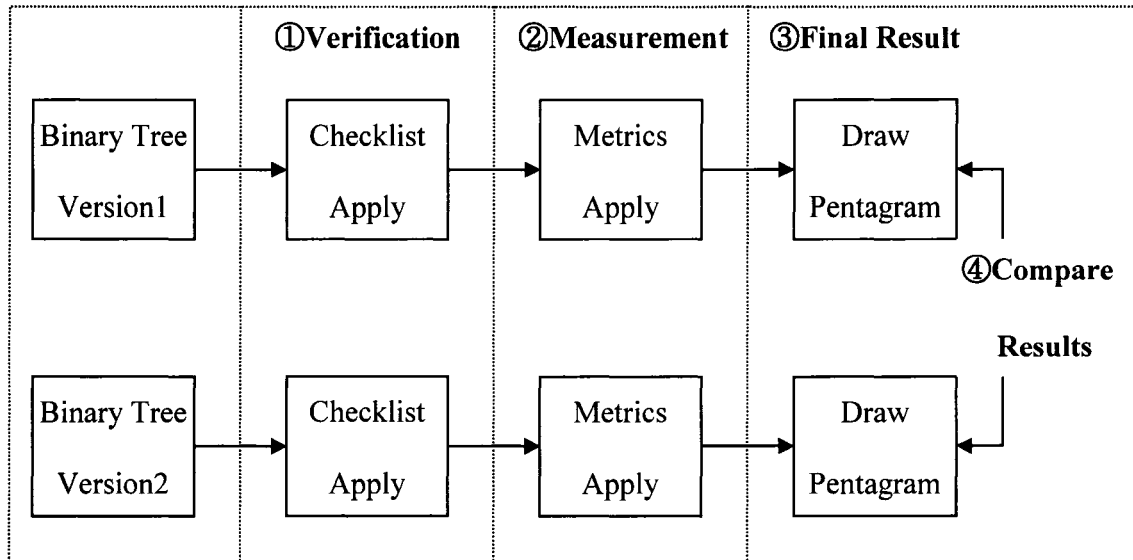
**Step 1: Verification** — Verify two versions of binary tree application using the given checklist. This refers to two activities: applying the checklist to the requirements and applying it, also, to the design.

**Step 2: Measurement** — Fill in the evaluation table based on the checklist. Then apply the metrics, according to the data in the evaluation table.

**Step 3: Final Results** — Draw the pentagram based on the values calculated from the metrics.

**Step 4: Comparing Results** — Calculate the area of the two pentagrams and compare the results.

In short, four elements: *check lists, evaluation tables, metrics, and pentagrams*, are used to verify and measure software component testability.



**Figure 20. The Work Flow for Comparison**

## 5.2 Requirements Verification and Measurement for Component Testability

Requirements of two different versions of binary tree application can be found in Appendix B. There are four steps for conducting requirements verification and measurement of component testability. First, the checklist is applied in order to verify component testability. Second, the checklist results are entered into the comparison table. So as to label the two different versions of binary tree application, BT1 will represent version 1, and BT2 will represent version 2.

### 5.2.1 Requirements Verification and Measurement for Understandability

There are two activities to check understandability: one is that of checking functional requirements, and the other is that of checking non-functional requirements.

#### ● A Checklist for Verifying Understandability of Functional Requirements

For each functional requirement, check the appropriate answer.		
● Is this component functional requirement understandable?	<input type="radio"/> Yes	<input type="radio"/> No
If yes, is it: <i>testable</i> ?	<input type="radio"/> Yes	<input type="radio"/> No
<i>measurable</i> ?	<input type="radio"/> Yes	<input type="radio"/> No
● Does this functional requirement allow users to observe its behavior?	<input type="radio"/> Yes	<input type="radio"/> No
If yes, is it: <i>testable</i> ?	<input type="radio"/> Yes	<input type="radio"/> No
<i>measurable</i> ?	<input type="radio"/> Yes	<input type="radio"/> No
● Does this functional requirement allow users to control its behavior?	<input type="radio"/> Yes	<input type="radio"/> No
If yes, is it: <i>testable</i> ?	<input type="radio"/> Yes	<input type="radio"/> No
<i>measurable</i> ?	<input type="radio"/> Yes	<input type="radio"/> No
● Does this functional requirement allow users to trace its operation?	<input type="radio"/> Yes	<input type="radio"/> No
If yes, is it: <i>testable</i> ?	<input type="radio"/> Yes	<input type="radio"/> No
<i>measurable</i> ?	<input type="radio"/> Yes	<input type="radio"/> No

● **A Comparison Table for Measuring Understandability of Functional Requirements**

<b>Functional Requirement 1</b> <b>Load a Tree</b>	Available?		Testable?		Measurable?		Testing Points(SUM)	
	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>
Understandable Requirement	0.5	0.5	0	0.25	0	0.25	0.5	1.0
Observable Behavior	0.5	0.5	0	0	0	0	0.5	0.5
Controllable Behavior	0.5	0.5	0	0	0	0	0.5	0.5
Traceable Operation	0.5	0.5	0	0	0	0	0.5	0.5
Total Testing Points of Each Functional Requirement (TPEFR)							2.0	2.5
Average of Total Testing Points of Each Functional Requirement (ATPEFR)							0.5	0.63
<b>Functional Requirement 2</b> <b>Save a Tree</b>	Available?		Testable?		Measurable?		Testing Points(SUM)	
	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>
Understandable Requirement	0.5	0.5	0	0.25	0	0.25	0.5	1.0
Observable Behavior	0.5	0.5	0	0	0	0	0.5	0.5
Controllable Behavior	0.5	0.5	0	0	0	0	0.5	0.5
Traceable Operation	0.5	0.5	0	0	0	0	0.5	0.5
Total Testing Points of Each Functional Requirement (TPEFR)							2.0	2.5
Average of Total Testing Points of Each Functional Requirement (ATPEFR)							0.5	0.63
<b>Functional Requirement 3</b> <b>Insert a Node</b>	Available?		Testable?		Measurable?		Testing Points(SUM)	
	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>
Understandable Requirement	0.5	0.5	0	0.25	0	0.25	0.5	1.0
Observable Behavior	0.5	0.5	0	0	0	0	0.5	0.5
Controllable Behavior	0.5	0.5	0	0	0	0	0.5	0.5
Traceable Operation	0.5	0.5	0	0	0	0	0.5	0.5
Total Testing Points of Each Functional Requirement (TPEFR)							2.0	2.5
Average of Total Testing Points of Each Functional Requirement (ATPEFR)							0.5	0.63
<b>Functional Requirement 4</b> <b>Delete a Node</b>	Available?		Testable?		Measurable?		Testing Points(SUM)	
	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>
Understandable Requirement	0.5	0.5	0	0.25	0	0.25	0.5	1.0
Observable Behavior	0.5	0.5	0	0	0	0	0.5	0.5



Controllable Behavior	0.5	0.5	0	0	0	0	0.5	0.5
Traceable Operation	0.5	0.5	0	0	0	0	0.5	0.5
Total Testing Points of Each Functional Requirement (TPEFR)							2.0	2.5
Average of Total Testing Points of Each Functional Requirement (ATPEFR)							0.5	0.63
<b>Functional Requirement 5</b> <b>Find a Node</b>	Available?		Testable?		Measurable?		Testing Points(SUM)	
	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>
Understandable Requirement	0.5	0.5	0	0.25	0	0.25	0.5	1.0
Observable Behavior	0.5	0.5	0	0	0	0	0.5	0.5
Controllable Behavior	0.5	0.5	0	0	0	0	0.5	0.5
Traceable Operation	0.5	0.5	0	0	0	0	0.5	0.5
Total Testing Points of Each Functional Requirement (TPEFR)							2.0	2.5
Average of Total Testing Points of Each Functional Requirement (ATPEFR)							0.5	0.63
<b>Functional Requirement 6</b> <b>Ascending Order</b>	Available?		Testable?		Measurable?		Testing Points(SUM)	
	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>
Understandable Requirement	0.5	0.5	0	0.25	0	0.25	0.5	1.0
Observable Behavior	0.5	0.5	0	0	0	0	0.5	0.5
Controllable Behavior	0.5	0.5	0	0	0	0	0.5	0.5
Traceable Operation	0.5	0.5	0	0	0	0	0.5	0.5
Average Testing Points of Each Functional Requirement (TPEFR)							2.0	2.5
Average of Total Testing Points of Each Functional Requirement (ATPEFR)							0.5	0.63
<b>Functional Requirement 7</b> <b>Paint a Tree</b>	Available?		Testable?		Measurable?		Testing Points(SUM)	
	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>	<b>BT1</b>	<b>BT2</b>
Understandable Requirement	0.5	0.5	0	0.25	0	0.25	0.5	1.0
Observable Behavior	0.5	0.5	0	0	0	0	0.5	0.5
Controllable Behavior	0.5	0.5	0	0	0	0	0.5	0.5
Traceable Operation	0.5	0.5	0	0	0	0	0.5	0.5
Average Testing Points of Each Functional Requirement (TPEFR)							2.0	2.5
Average of Total Testing Points of Each Functional Requirement (ATPEFR)							0.5	0.63

**Table 30. Comparison Table for Functional Requirements**

### ● A Checklist for Verifying Understandability of Non-Functional Requirements

For non-functional requirements, check the appropriate answer.

- Does it include component environment requirements?       Yes     No  
 If yes, is it: *testable?*       Yes     No  
                                  *measurable?*       Yes     No
- Does it include component availability requirements?       Yes     No  
 If yes, is it: *testable?*       Yes     No  
                                  *measurable?*       Yes     No
- Does it include component performance requirements?       Yes     No  
 If yes, is it: *testable?*       Yes     No  
                                  *measurable?*       Yes     No
- Does it include component scalability requirements?       Yes     No  
 If yes, is it: *testable?*       Yes     No  
                                  *measurable?*       Yes     No
- Does it include component throughput requirements?       Yes     No  
 If yes, is it: *testable?*       Yes     No  
                                  *measurable?*       Yes     No
- Does it include component reliability requirements?       Yes     No  
 If yes, is it: *testable?*       Yes     No  
                                  *measurable?*       Yes     No
- Does it include component utilization requirements?       Yes     No  
 If yes, is it: *testable?*       Yes     No  
                                  *measurable?*       Yes     No

● **A Comparison Table for Measuring Understandability of Non-Functional Requirements**

Non-Functional Requirement	Available?		Testable?		Measurable?		Testing Points(SUM)	
	BT1	BT2	BT1	BT2	BT1	BT2	BT1	BT2
Environment Requirements	0.5	0.5	0	0.25	0	0.25	0.5	1.0
Availability Requirements	0.5	0.5	0.25	0.25	0	0.25	0.75	1.0
Performance Requirements	0.5	0.5	0.25	0.25	0	0.25	0.75	1.0
Scalability Requirements	0.5	0.5	0	0.25	0	0.25	0.5	1.0
Throughput Requirements	0	0	0	0	0	0	0	0
Reliability Requirements	0.5	0.5	0.25	0.25	0	0.25	0.75	1.0
Utilization Requirements	0.5	0.5	0	0.25	0	0.25	0.5	1.0

**Table 31. Comparison Table for Non-Functional Requirements**

● **Average Understandability Metrics for Requirements (AUMR)**

$$\text{Binary Tree Version1 FReg} = \frac{1}{7} \sum_{i=1}^7 \text{ATPEFR}_i = \frac{1}{7} \times (0.5 \times 6) = 0.5$$

$$\text{Binary Tree Version2 FReg} = \frac{1}{7} \sum_{i=1}^7 \text{ATPEFR}_i = \frac{1}{7} \times (0.63 \times 6) = 0.63$$

$$\text{Binary Tree Version1 NFReg} = \frac{1}{7} \sum_{i=1}^7 \text{NFReq}_i = \frac{1}{7} \times (3.75) = 0.54$$

$$\text{Binary Tree Version2 NFReg} = \frac{1}{7} \sum_{i=1}^7 \text{NFReq}_i = \frac{1}{7} \times (6) = 0.86$$

$$\text{Binary Tree Version1 AUMR} = \frac{1}{2} (\text{FReg} + \text{NFReg}) = \frac{1}{2} (0.5 + 0.54) = 0.52$$

$$\text{Binary Tree Version2 AUMR} = \frac{1}{2} (\text{FReg} + \text{NFReg}) = \frac{1}{2} (0.63 + 0.86) = 0.75$$

## 5.2.2 Requirements Verification and Measurement for Observability

There are four types of component observability.

### ● A Checklist for Verifying Observability of Component Requirements

There are four types of component observability. Please check if present and qualify the ease of use.

**Type 1:** The capability to observe input/output data  
 How easy is it?    Automatic    Semiautomatic    Manual

**Type 2:** The capability to observe communication data  
 How easy is it?    Automatic    Semiautomatic    Manual

**Type 3:** The capability to observe GUI events and related input/output data  
 How easy is it?    Automatic    Semiautomatic    Manual

**Type 4:** The capability to observe function operation/behavior  
 How easy is it?    Automatic    Semiautomatic    Manual

It doesn't have any of the above-mentioned types.

### ● A Comparison Table for Measuring Observability of Component Requirements

Types of Observability(TO)	TP		Types of Observability(TO)	TP	
	BT1	BT2		BT1	BT2
Data Observability	BT1	BT2	GUI Observability	BT1	BT2
Automatic(1.0)	0.6	1.0	Automatic(1.0)	0.6	1.0
Semiautomatic(0.8)			Semiautomatic(0.8)		
Manual(0.6)			Manual(0.6)		
Communication Observability	BT1	BT2	Function Observability	BT1	BT2
Automatic(1.0)	0.6	1.0	Automatic(1.0)	0.6	1.0
Semiautomatic(0.8)			Semiautomatic(0.8)		
Manual(0.6)			Manual(0.6)		

**Table 32. Comparison Table for Observability Types in Requirements**

● **Average Observability Metrics for Requirements (AOMR)**

$$\text{Binary Tree Version1 AOMR} = \frac{1}{4} * (\text{TO}) = \frac{1}{4} * (2.4) = 0.6$$

$$\text{Binary Tree Version2 AOMR} = \frac{1}{4} * (\text{TO}) = \frac{1}{4} * (4) = 1$$

**5.2.3 Requirements Verification and Measurement for Controllability**

There are five types of controllability and three properties of controllability.

● **A Checklist to Verify Five Controllability Types of Component Requirements**

Which types does it have? Please also indicate its specific feature.

**Type 1: Component Environment Controller**

What kind of capability does it include?

Installation Capability    Configuration Set-up Capability    Deployment Capability

**Type 2: Component Execution Controller**

What kind of capability does it include?

Test Mode Execution    Control Mode Execution    Normal Function Mode Execution

**Type 3: Component Functional Feature Controller**

What kind of capability does it include?

Function Enable Capability    Function Disable Capability

**Type 4: Component State-Based Behavior Controller**

What kind of capability does it include?

Transition Control Function    State Control Function    Reset Function

**Type 5: Component Test Controller through Interfaces**

What kind of capability does it include?

Function Invocation Control    Input Data Setup    Output Data Monitor

Incoming Message Setup    Outgoing Message Monitor

It doesn't have any of the above-mentioned types.

● A Comparison Table to Measure Five Controllability Types of Requirements

Types of Controllability(TC)	TP		Types of Controllability(TC)	TP	
	BT1	BT2		BT1	BT2
Component Environment Controller	BT1	BT2	Component State-Based Behavior Controller	BT1	BT2
Installation Capability(0.33)	0.66	1.0	Transition Control Function(0.33)	0	1.0
Configuration Set-up Capability(0.33)			State Control Function(0.33)		
Deployment Capability(0.33)			Reset Function(0.33)		
Component Execution Controller	BT1	BT2	Component Test Controller through Interfaces	BT1	BT2
Test Mode Execution(0.33)	0	1.0	Function Invocation Control(0.2)	0	1.0
Control Mode Execution(0.33)			Input Data Setup(0.2)		
Normal Function Mode Execution(0.33)			Output Data Monitor(0.2)		
Component Functional Feature	BT1	BT2	Incoming Message Setup(0.2)		
Function Enable Capability(0.5)	0	1.0	Outgoing Message Monitor(0.2)		
Function Disable Capability(0.5)					

**Table 33. Comparison Table for Controllability Types in Requirements**

● A Checklist to Verify Three Properties Controllability of Component Requirements

Which properties does it have? Please also indicate its specific feature.

**Property 1:** Component Behavior Collection

What kind of collection?     Automatic     Semiautomatic     Manual

**Property 2:** Component Feature Customization

What kind of customization?

Function Customization     Interface Customization     Meta-data Customization

**Property 3:** Component Installation & Deployment

What kind of Installation & Deployment?

Automatic     Semiautomatic     Manual

It doesn't have any of the above-mentioned properties.

● **A Comparison Table to Measure Three Controllability Properties of Requirements**

Properties of Controllability(PC)	TP		Properties of Controllability(PC)	TP	
Component Behavior Collection	BT1	BT2	Component Installation & Deployment	BT1	BT2
Automatic(1.0)	0.6	1.0	Automatic(1.0)	0	1.0
Semiautomatic(0.8)			Semiautomatic(0.8)		
Manual(0.6)			Manual(0.6)		
Component Feature Customization	BT1	BT2			
Function Customization(0.33)	0	0.33			
Interface Customization(0.33)					
Meta-data Customization(0.33)					

**Table 34. Comparison Table for Controllability Properties in Requirements**

● **Average Controllability Metrics for Requirement (ACMR)**

$$\text{Binary Tree Version1 TPTCR} = \frac{1}{5} \sum_{i=1}^5 TC_i = \frac{1}{5} (0.66 + 0 + 0 + 0 + 0) = 0.132$$

$$\text{Binary Tree Version2 TPTCR} = \frac{1}{5} \sum_{i=1}^5 TC_i = \frac{1}{5} (1 + 1 + 1 + 1 + 1) = 1$$

$$\text{Binary Tree Version1 TPPCR} = \frac{1}{3} \sum_{i=1}^3 PC_i = \frac{1}{3} * (0.6 + 0 + 0) = 0.2$$

$$\text{Binary Tree Version2 TPPCR} = \frac{1}{3} \sum_{i=1}^3 PC_i = \frac{1}{3} * (1 + 1 + 0.33) = 0.78$$

$$\text{Binary Tree Version1 ACMR} = \text{TPPCR} * \text{TPTCR} = (0.2) * (0.132) = 0.03$$

$$\text{Binary Tree Version2 ACMR} = \text{TPPCR} * \text{TPTCR} = (0.78) * 1 = 0.78$$

### 5.2.4 Requirements Verification and Measurement for Traceability

There are five types of traceability and five properties of traceability.

#### ● A Checklist to Verify Five Traceability Types of Component Requirements

Which types does it have? Please also indicate its specific feature.

**Property 1: Operational Tracking?**

What kind of tracking does it include?

System Function Tracking    Component Function Tracking

Class Function Tracking

**Property 2: Performance Tracking**

What kind of capability does it include?

Throughput    Reliability    Speed up    Availability

Scalability    Utilization

**Property 3: State Tracking**

What kind of tracking does it include?

System State Tracking    Component State Tracking    Object State Tracking

**Property 4: Event Tracking**

What kind of tracking does it include?

GUI Event Tracking    Communication    Event Tracking

Interaction Event Tracking

**Property 5: Error Tracking**

What kind of tracking does it include?

Error Tracking    Exception Tracking    Warning Message Tracking

It doesn't have any of the above-mentioned property.



● **A Comparison Table to Measure Five Traceability Types of Requirements**

Types of Tracking Capability(TT)	TP		Types of Tracking Capability(TT)	TP	
	BT1	BT2		BT1	BT2
Operation Tracking	BT1	BT2	State Tracking	BT1	BT2
System Function Tracking(0.33)	0	1.0	System State Tracking(0.33)	0	1.0
Component Function Tracking(0.33)			Component State Tracking(0.33)		
Class Function Tracking(0.33)			Object State Tracking(0.33)		
Performance Tracking	BT1	BT2	Event Tracking	BT1	BT2
Throughput(0.16)	0	0	GUI Event Tracking(0.33)	0	0.33
Reliability(0.16)			Communication Event Tracking(0.33)		
Speed up(0.16)			Interaction Event Tracking(0.33)		
Availability(0.16)			Error Tracking		
Scalability(0.16)			Error Tracking(0.33)	0.33	0.33
Utilization(0.16)			Exception Tracking(0.33)		
			Warning Message Tracking(0.33)		

**Table 35. Comparison Table for Traceability Types in Requirements**

● **A Checklist to Verify Five Traceability Properties of Component Requirements**

Which properties does it have? Please also indicate its specific feature.

**Type 1: Trace Format**

What kind of format?

Consistent Well-Define    Consistent Generate    Ad Hoc

**Type 2: Trace Code Insertion?**

What kind of insertion?

Automatic Wrapping    Automatic Code Insertion    Framework-Based    Manual

**Type 3: Trace Code Generation**

What kind of Generation?

Automatic    Semiautomatic    Manual

**Type 4: Trace Collection**

What kind of Collection?

Automatic    Semiautomatic    Manual

**Type 5: Trace Storage**

What kind of Storage?

Trace File Repository    Trace Data Repository    Trace Report

It doesn't have any of the above-mentioned properties.

● **A Comparison Table to Measure Five Traceability Properties of Requirements**

Properties of Tracking Capability(PT)	TP		Properties of Tracking Capability(PT)	TP	
	BT1	BT2		BT1	BT2
Trace Format	BT1	BT2	Trace Generation	BT1	BT2
Consistent Well-Define(1.0)	1.0	1.0	Automatic(1.0)	0	1.0
Consistent Generate(0.8)			Semiautomatic(0.8)		
Ad Hoc(0.6)			Manual(0.6)		
Trace Code Insertion	BT1	BT2	Trace Collection	BT1	BT2
Automatic Wrapping(1.0)	0	1.0	Automatic(1.0)	0.6	1.0
Automatic Code Insertion(0.8)			Semiautomatic(0.8)		
Framework-Based(0.6)			Manual(0.6)		
Manual(0.4)			Trace Storage		
			Trace File Repository(0.33)	0.33	0.33
			Trace Data Repository (0.33)		
			Trace Report (0.33)		

**Table 36. Comparison Table for Traceability Properties in Requirements**

● **Average Traceability Metrics for Requirements (ATMR)**

$$\text{Binary Tree Version1 TPTTR} = \frac{1}{5} \sum_{i=1}^5 TT_i = \frac{1}{5} (0 + 0 + 0 + 0 + 0.33) = 0.07$$

$$\text{Binary Tree Version2 TPTTR} = \frac{1}{5} \sum_{i=1}^5 TT_i = \frac{1}{5} (1 + 0 + 1 + 0.33 + 0.33) = 0.53$$

$$\text{Binary Tree Version1 TPPTTR} = \frac{1}{5} \sum_{i=1}^5 PT_i = \frac{1}{5} (1 + 0 + 0 + 0.6 + 0.33) = 0.19$$

$$\text{Binary Tree Version2 TPPTTR} = \frac{1}{5} \sum_{i=1}^5 PT_i = \frac{1}{5} (1 + 1 + 1 + 1 + 0.33) = 0.87$$

$$\text{Binary Tree Version1 ATMR} = \text{TPPTTR} * \text{TPTTR} = (0.19) * (0.07) = 0.01$$

$$\text{Binary Tree Version2 ATMR} = \text{TPPTTR} * \text{TPTTR} = (0.87) * (0.53) = 0.46$$

### 5.2.5 Requirements Verification and Measurement for Test Support Capability

There are four types of test support capability.

● **A Checklist to Verify Four Test Support Capability Types of Requirements**

<p>Does the component provide Test Support Capability? <input type="radio"/> Yes <input type="radio"/> No</p> <p><b>If yes,</b></p> <p>● Does it include Test Generation? <input type="radio"/> Yes <input type="radio"/> No</p> <p>If so, what kind of Test Generation?</p> <p><input type="radio"/> Component White-Box Test Generation <input type="radio"/> Component Black-Box Test Generation</p> <p>→If it uses Component White-Box Test Generation, what specific methods does it use?</p> <p><input type="checkbox"/> Basic Path Testing <input type="checkbox"/> Data Flow Testing <input type="checkbox"/> Branch-Based Testing</p> <p><input type="checkbox"/> Syntax-Based Testing <input type="checkbox"/> State-Based Testing</p>
--

→ If it uses Component Black-Box Test Generation, what specific methods does it use?

- Boundary Value Testing    Equivalence Partitioning    Graphed-based Testing  
 Random Testing

- Does it include Test Management Capability?    Yes    No

If so, what kind of management?

- Problem Management    Test Management    Suit Management

→ If it uses Problem Management, what specific technique does it use?

- Systematic with tools    Systematic without tools    Ad-hoc

→ If it uses Test Management, what specific technique does it use?

- Systematic with tools    Systematic without tools    Ad-hoc

→ If it uses Suit Management, what specific technique does it use?

- Systematic with tools    Systematic without tools    Ad-hoc

- Does it describe Test Coverage?    Yes    No

If so, what is it?

- Test Coverage Criteria    Test Coverage Standards    Test Coverage Analysis Tools

→ If it defines Test Coverage Criteria, how restrict is it?

- Well-defined Criteria    Defined Criteria    Non-defined Criteria

→ If it defines Test Coverage Standards, how restrict is it?

- Well-defined Standards    Defined Standards    Ad-hoc

→ If it includes Test Coverage Analysis Tools, what functionality does it provide?

- Provide Measure    Provide Monitor    Provide Report

● Does it include the Component Test Scripting Capability?  Yes  No

If it does, what is it?

Test Scripts  Test stubs  Test Drivers

→ If it uses Test Scripts, what kind of capability does it include?

Create/Editing  Book-keeping/ Management  Execute

→ If it uses Test stubs, what kind of capability does it include?

Create  Maintain  Execute

→ If it uses Test Drivers, what kind of capability does it include?

Automatic Creation  Systematic Creation  Ad-hoc Creation

● **Four Comparison Tables to Measure Four Test Support Capability Types of Requirements**

Component Test Support Capability	TP		Component Test Support Capability	TP	
	BT1	BT2		BT1	BT2
Component White-Box Test Generation	<b>BT1</b>	<b>BT2</b>	Component Black-Box Test Generation	<b>BT1</b>	<b>BT2</b>
Basic Path Testing(0.2)	<b>0</b>	<b>0.6</b>	Boundary Value Testing(0.2)	<b>0</b>	<b>0.4</b>
Data Flow Testing(0.2)			Equivalence Partitioning(0.2)		
Branch-Based Testing(0.2)					
Syntax-Based Testing(0.2)					
State-Based Testing(0.2)					
			Graphed-based Testing(0.2)		
			Random Testing(0.2)		
			Requirements-Based Testing(0.2)		

**Table 37. Comparison Table for Test Generation in Requirements**

Component Test Support Capability	TP		Component Test Support Capability	TP	
	BT1	BT2		BT1	BT2
Problem Management	<b>BT1</b>	<b>BT2</b>	Suit Management	<b>BT1</b>	<b>BT2</b>
Systematic with tools(1.0)	<b>0</b>	<b>0.6</b>	Systematic with tools(1.0)	<b>0</b>	<b>0.6</b>
Systematic without tools(0.8)			Systematic without tools(0.8)		
Ad-hoc(0.6)					
			Ad-hoc(0.6)		

Test Management	BT1	BT2
Systematic with tools(1.0)	0	0.6
Systematic without tools(0.8)		
Ad-hoc(0.6)		

**Table 38. Comparison Table for Test Management in Requirements**

Component Test Support Capability	TP		Component Test Support Capability	TP	
Test Coverage Criteria	BT1	BT2	Test Coverage Analysis Tools	BT1	BT2
Well-defined Criteria (1.0)	1.0	1.0	Provide Measure(0.33)	0	1.0
Defined Criteria(0.8)			Provide Monitor(0.33)		
Non-defined Criteria(0.6)			Provide Report(0.33)		
Test Coverage Standards	BT1	BT2			
Well-defined Standards(1.0)	0	0			
Defined Standards(0.8)					
Ad-hoc(0.6)					

**Table 39. Comparison Table for Test Coverage in Requirements**

Component Test Support Capability	TP		Component Test Support Capability	TP	
Test Scripts	BT1	BT2	Test Stubs	BT1	BT2
Create/Editing(1.0)	0	1.0	Create(0.33)	0	1.0
Book-keeping/ Management(0.8)			Maintain(0.33)		
Execute(EXE)(0.6)			Execute(0.33)		
Test Drivers	BT1	BT2			
Automatic Creation(1.0)	0	0.6			
Systematic Creation(0.8)					
Ad-hoc Creation(0.6)					

**Table 40. Comparison Table for Test Scripting in Requirements**

● **Average Test Support Capability Metrics for Requirements (ATSMR)**

$$\text{Binary Tree Version1 TPTG} = \frac{1}{2} * (0 + 0) = 0$$

$$\text{Binary Tree Version2 TPTG} = \frac{1}{2} * (0.6 + 0.4) = 0.5$$

$$\text{Binary Tree Version1 TPTM} = \frac{1}{3} * (0 + 0 + 0) = 0$$

$$\text{Binary Tree Version2 TPTM} = \frac{1}{3} * (0.6 + 0.6 + 0.6) = 0.6$$

$$\text{Binary Tree Version1 TPTC} = \frac{1}{3} (1 + 0 + 0) = 0.33$$

$$\text{Binary Tree Version2 TPTC} = \frac{1}{3} (1 + 0 + 1) = 0.66$$

$$\text{Binary Tree Version1 TPTSC} = \frac{1}{3} * (0 + 0 + 0) = 0$$

$$\text{Binary Tree Version2 TPTSC} = \frac{1}{3} * (1 + 0.6 + 1) = 0.87$$

$$\begin{aligned} \bullet \text{ Binary Tree Version1 ATSMR} &= \frac{1}{4} ( \text{TPTG} + \text{TPTM} + \text{TPTC} + \text{TPTSC} ) \\ &= \frac{1}{4} * ( 0 + 0 + 0.33 + 0 ) = 0.08 \end{aligned}$$

$$\begin{aligned} \bullet \text{ Binary Tree Version2 ATSMR} &= \frac{1}{4} ( \text{TPTG} + \text{TPTM} + \text{TPTC} + \text{TPTSC} ) \\ &= \frac{1}{4} * ( 0.5 + 0.6 + 0.66 + 0.87 ) = 0.66 \end{aligned}$$

### 5.3 Design Verification and Measurement for Component Testability

Design of two different versions of binary tree application can be found in Appendix B. There are four steps for conducting design verification and measurement of component testability. First, the checklist is applied in order to verify component testability. Second, the checklist results are entered into the comparison table.

#### 5.3.1 Design Verification and Measurement for Understandability

There are two activities to check understandability: one is to check functional requirements and the other is to check non-functional requirements.

##### ● Two Checklists to Verify Component Design for Understandability

For each non-functional requirement, check the appropriate answers.

- If it states having the availability requirements in the non-functional requirements, does its corresponding design provide a consistent, complete, and correct solution?  Yes  No
- If it states having the performance requirements in the non-functional requirements, does its corresponding design provide a consistent, complete, and correct solution?  Yes  No
- If it states having the scalability requirements in the non-functional requirements, does its corresponding design provide a consistent, complete, and correct solution?  Yes  No
- If it states having the throughput requirements in the non-functional requirements, does its corresponding design provide a consistent, complete, and correct solution?  Yes  No
- If it states having the reliability requirements in the non-functional requirements, does its corresponding design provide a consistent, complete, and correct solution?  Yes  No
- If it states having the utilization requirements in the non-functional requirements, does its corresponding design provide a consistent, complete, and correct solution?  Yes  No



For each functional design, check the appropriate answers.

- If it allows users to observe its behavior in the functional requirements, does its corresponding design provide a consistent, complete, and correct solution?  Yes  No
- If it allows users to control its behavior in the functional requirements, does its corresponding design provide a consistent, complete, and correct solution?  Yes  No
- If it allows users to trace its operation in the functional requirements, does its corresponding design provide a consistent, complete, and correct solution?  Yes  No

● **A Comparison Tables to Measure Component Design for Understandability**

Understandability	TP		Understandability	TP	
	BT1	BT2		BT1	BT2
Functional Design			Non Functional Design		
Design for Observable Behavior(0.33)	1.0	1.0	Design for Availability(0.16)	0.8	0.8
Design for Controllable Behavior(0.33)			Design for Performance(0.16)		
Design for Traceable Operation(0.33)			Design for Scalability(0.16)		
			Design for Throughput(0.16)		
			Design for Reliability(0.16)		
			Design for Utilization(0.16)		

**Table 41. Comparison Table for Understandability in Design**

● **Average Understandability Metrics for Design (AUMD)**

$$\text{Binary Tree Version1 AUMD} = \frac{1}{2} (\text{FReq} + \text{NFReq}) = \frac{1}{2} * (1) = 0.08$$

$$\text{Binary Tree Version2 AUMD} = \frac{1}{2} (\text{FReq} + \text{NFReq}) = \frac{1}{2} * (0.8) = 0.66$$

### 5.3.2 Design Verification and Measurement for Observability

Design verification and measurement of observability refers to the ability of verifying and measuring the input and output of different types of component.

#### ● A Checklist to Verify Design for Component Observability

##### Check Design for Component Data Observability

- If the requirements state that users can observe input/output data, does its corresponding design provide a consistent, complete, and correct solution?  Yes  No

If so, please select the appropriate case for each component access function.

- There are the *same* numbers of Input Data and Output Data
- There is *more* Input Data than Output Data
- There is *less* Input Data than Output Data
- There is *no* Input Data or Output Data

In addition, what kind of design is there for users to observe?

- Automatic observation design  Semiautomatic observation design
- Manual observation design

##### Check Design for Component Communication Observability

- If the requirements states that users can observe communication data in the requirements, does its corresponding design provide a consistent, complete, and correct solution?  Yes  No

If so, please select the appropriate case for each component access function.

- There are *same* numbers of Incoming Message and Outgoing Message
- There are *more* Incoming Messages than Outgoing Messages
- There are *less* Incoming Messages than Outgoing Messages
- There is *no* Incoming Message or Outgoing Message

In addition, what kind of design is there for users to observe?

- Automatic observation design  Semiautomatic observation design
- Manual observation design

**Check Design for Component GUI Observability**

If the requirements states that users can observe GUI events and related input/output data in the requirements, does its corresponding design provide a consistent, complete, and correct solution?

Yes  No

- Does the design allow user to observe GUI input and output data?  Yes  No

If so, please select the appropriate case for each GUI component access function.

- There are *same* numbers of GUI Input Data and of GUI Output Data
- There is *more* GUI Input Data than GUI Output Data
- There is *less* GUI Input Data than GUI Output Data
- There is *no* GUI Input Data or GUI Output Data

In addition, what kind of design is there for users to observe?

- Automatic observation design  Semiautomatic observation design
- Manual observation design

- Does the design allow user to observe input and output event?  Yes  No

If so, please select the appropriate case for each GUI component access function.

- There are same numbers of Input Event and Output Event
- There are *more* Input Events than Output Events
- There are *less* Input Events than Output Events
- There is *no* Input Event or Output Event

In addition, what kind of design is there for users to observe?

- Automatic observation design  Semiautomatic observation design
- Manual observation design

### Check Design for Component Function Observability

If the requirements states that users can observe function operation/behavior in the requirement, does its corresponding design provide a consistent, complete, and correct solution?  Yes  No

- Does the design allow user to observe input and output function call data?  Yes  No

If so, please select the appropriate case for each component access function.

- There are same numbers of Input Function Call Data and Output Function Call Data
- There is *more* Input Function Call Data than Output Function Call Data
- There is *less* Input Function Call Data than Output Function Call Data
- There is *no* Input Function Call Data or Output Function Call Data

In addition, what kind of design is there for users to observe?

- Automatic observation design  Semiautomatic observation design
- Manual observation design

- Does the design allow user to observe incoming and outgoing function call message?

Yes  No

If so, please select the appropriate case for each component access function.

- There are same numbers of Incoming Function Call and Outgoing Function Call Message
- There are *more* Incoming Function Call Messages than Outgoing Function Call Messages
- There are *less* Incoming Function Call Messages than Outgoing Function Call Messages
- There is *no* Incoming Function Call Message or Outgoing Function Call Message

In addition, what kind of design is there for users to observe?

- Automatic observation design  Semiautomatic observation design
- Manual observation design

● **Six Comparison Tables to Measure Design for Component Observability**

Numbers of Data	Count	Observation Level	Weigh
$  \text{Input Data}   =   \text{Output Data}   \neq 0$ (1.0)	<b>BT1</b> <b>BT2</b>	Automatic	100%
$  \text{Input Data}   \neq   \text{Output Data}  $ (0.8)	<b>1.0</b> <b>1.0</b>	Semiautomatic	80%
$  \text{Input Data}   =   \text{Output Data}   = 0$ (0.6)		Manual	60%

**Table 42. Comparison Table for Data Observability in Design**

Numbers of Messages	Count	Observation Level	Weigh
$  \text{Incoming Message}   =   \text{Outgoing Messages}   \neq 0$ (1.0)	<b>BT1</b> <b>BT2</b>	Automatic	100%
$  \text{Incoming Message}   \neq   \text{Outgoing Messages}  $ (0.8)	<b>1.0</b> <b>1.0</b>	Semiautomatic	80%
$  \text{Incoming Message}   =   \text{Outgoing Messages}   = 0$ (0.6)		Manual	60%

**Table 43. Comparison Table for Communication Observability in Design**

Numbers of GUI Data	Count	Observation Level	Weigh
$  \text{GUI Input Data}   =   \text{GUI Output Data}   \neq 0$ (1.0)	<b>BT1</b> <b>BT2</b>	Automatic	100%
$  \text{GUI Input Data}   \neq   \text{GUI Output Data}  $ (0.8)	<b>1.0</b> <b>1.0</b>	Semiautomatic	80%
$  \text{GUI Input Data}   =   \text{GUI Output Data}   = 0$ (0.6)		Manual	60%

**Table 44. Comparison Table for GUI Data Observability in Design**

Numbers of GUI Event	Count	Observation Level	Weigh
$  \text{Input Event}   =   \text{Output Event}   \neq 0$ (1.0)	<b>BT1</b> <b>BT2</b>	Automatic	100%
$  \text{Input Event}   \neq   \text{Output Event}  $ (0.8)	<b>1.0</b> <b>1.0</b>	Semiautomatic	80%
$  \text{Input Event}   =   \text{Output Event}   = 0$ (0.6)		Manual	60%

**Table 45. Comparison Table for GUI Event Observability in Design**

Numbers of Function Call Data	Count		Observation Level	Weigh
$ \text{Input Function Call Data}  =  \text{Output Function Call Data}  \neq 0$ (1.0)	<b>BT1</b>	<b>BT2</b>	Automatic	100%
$ \text{Input Function Call Data}  \neq  \text{Output Function Call Data} $ (0.8)	<b>1.0</b>	<b>1.0</b>	Semiautomatic	80%
$ \text{Input Function Call Data}  =  \text{Output Function Call Data}  = 0$ (0.6)			Manual	60%

**Table 46. Comparison Table for Function Call Data Observability in Design**

Numbers of Function Call Message	Count		Observation Level	Weigh
$ \text{Incoming Function Call Message}  =  \text{Outgoing Function Call Message}  \neq 0$ (1.0)	<b>BT1</b>	<b>BT2</b>	Automatic	100%
$ \text{Incoming Function Call Message}  \neq  \text{Outgoing Function Call Message} $ (0.8)	<b>1.0</b>	<b>1.0</b>	Semiautomatic	80%
$ \text{Incoming Function Call Message}  =  \text{Outgoing Function Call Message}  = 0$ (0.6)			Manual	60%

**Table 47. Comparison Table for Function Call Message Observability**

● **Average Observability Metrics for Design (AOMD)**

$$\text{Binary Tree Version1 TPDO} = \text{Count} * \text{Weight} = 1.0 * 60\% = 0.6$$

$$\text{Binary Tree Version2 TPDO} = \text{Count} * \text{Weight} = 1.0 * 100\% = 1$$

$$\text{Binary Tree Version1 TPCO} = \text{Count} * \text{Weight} = 1.0 * 60\% = 0.6$$

$$\text{Binary Tree Version2 TPCO} = \text{Count} * \text{Weight} = 1.0 * 100\% = 1$$

$$\text{Binary Tree Version1 TPGDO} = \text{Count} * \text{Weight} = 1.0 * 60\% = 0.6$$

$$\text{Binary Tree Version2 TPGDO} = \text{Count} * \text{Weight} = 1.0 * 100\% = 1$$

$$\text{Binary Tree Version1 TPGEO} = \text{Count} * \text{Weight} = 1.0 * 60\% = 0.6$$

$$\text{Binary Tree Version2 TPGEO} = \text{Count} * \text{Weight} = 1.0 * 100\% = 1$$

$$\text{Binary Tree Version1 TPFDO} = \text{Count} * \text{Weight} = 1.0 * 60\% = 0.6$$

$$\text{Binary Tree Version2 TPFDO} = \text{Count} * \text{Weight} = 1.0 * 100\% = 1$$

$$\text{Binary Tree Version1 TPFMO} = \text{Count} * \text{Weight} = 1.0 * 60\% = 0.6$$

$$\text{Binary Tree Version2 TPFMO} = \text{Count} * \text{Weight} = 1.0 * 100\% = 1$$

$$\text{Binary Tree Version1 AOMD} = \frac{1}{6} (\text{TPDO} + \text{TPCO} + \text{TPGDO} + \text{TPGEO} + \text{TPFDO} + \text{TPFMO}) = \frac{1}{6} * (0.6+0.6+0.6+0.6+0.6+0.6) = 0.6$$

$$\text{Binary Tree Version2 AOMD} = \frac{1}{6} (\text{TPDO} + \text{TPCO} + \text{TPGDO} + \text{TPGEO} + \text{TPFDO} + \text{TPFMO}) = \frac{1}{6} * (1+1+1+1+1+1) = 1$$

### 5.3.3 Design Verification and Measurement for Controllability

Design verification and measurement of controllability refers to the ability of verifying and measuring the built-in-mechanism design and the API design by component controllability.

#### ● A Checklist to Verify Built-in-Mechanism Design for Component Controllability

- |  |  |
|--|--|
| ● If the requirements states having component behavior collection, does its corresponding design provide a consistent, complete, and correct solution?                                     | <input type="radio"/> Yes <input type="radio"/> No |
| ● If the requirements states having component feature customization, does its corresponding design provide a consistent, complete, and correct solution?                                   | <input type="radio"/> Yes <input type="radio"/> No |
| ● If the requirements states having installation and deployment mechanism, does its corresponding design provide a consistent, complete, and correct solution?                             | <input type="radio"/> Yes <input type="radio"/> No |
| ● If the requirements states having a component environment controller, does its corresponding design provide a consistent, complete, and correct solution?                                | <input type="radio"/> Yes <input type="radio"/> No |
| ● If the requirements states having a component execution controller, does its corresponding design provide a consistent, complete, and correct solution?                                  | <input type="radio"/> Yes <input type="radio"/> No |
| ● If the requirements states having a component state-based behavior controller, does its corresponding design provide a consistent, complete, and correct solution?                       | <input type="radio"/> Yes <input type="radio"/> No |
| ● If the requirements states having a component test controller through interfaces in the requirement, does its corresponding design provide a consistent, complete, and correct solution? | <input type="radio"/> Yes <input type="radio"/> No |
| ● If the requirements states having a component functional feature controller, does its corresponding design provide a consistent, complete, and correct solution?                         | <input type="radio"/> Yes <input type="radio"/> No |

● **A Comparison Tables to Measure Built-in Mechanism Design for Controllability**

Built-in Mechanism Design of Controllability	BT1	BT2	Built-in Mechanism Design of Controllability	BT1	BT2
Design for collect component behavior	0.125	0.125	Design for component execution controller	0	0.125
Design for customize component feature	0	0.125	Design for component state-based behavior controller	0	0.125
Design for installation and deployment mechanism	0	0.125	Design for component test controller through interfaces	0	0.125
Design for component environment controller	0	0.125	Design for component functional feature controller	0	0.125

**Table 48. Comparison Table for Built-in Mechanism Controllability in Design**

● **A Checklist to Verify API Design for Component Controllability**

● For the same input data, does it always get the same output data?	<input type="radio"/> Yes	<input type="radio"/> No
● For the same incoming message, does it always get the same outgoing message?	<input type="radio"/> Yes	<input type="radio"/> No
● For the same GUI input data, does it always get the same GUI output data?	<input type="radio"/> Yes	<input type="radio"/> No
● For the same input function call data, does it always get the same output function call data?	<input type="radio"/> Yes	<input type="radio"/> No
● For the same input function call message, does it always get the same output function call message?	<input type="radio"/> Yes	<input type="radio"/> No

● **A Comparison Tables to Measure API Design for Controllability**

API Design of Component Controllability	BT1	BT2
Same input data → Same output data	0.2	0.2
Same incoming message → Same outgoing message	0.2	0.2
Same GUI input data → Same GUI output data	0.2	0.2
Same input function call data → Same output function call data	0.2	0.2
Same input function call message → Same output function call message	0.2	0.2

**Table 49. Comparison Table for API Design Controllability in Design**



● **Average Controllability Metrics of Design (ACMD)**

Binary Tree Version1 TPBDC = Total Counts =  $(0.125)*(1) = 0.125$

Binary Tree Version2 TPBDC = Total Counts =  $(0.125)*(8) = 1$

Binary Tree Version1 TPADC = Total Counts =  $0.2*5 = 1$

Binary Tree Version2 TPADC = Total Counts =  $0.2*5 = 1$

Binary Tree Version1 ACMD =  $\frac{1}{2} * (TPBDC + TPADC) = \frac{1}{2} * (0.125+1) = 0.56$

Binary Tree Version2 ACMD =  $\frac{1}{2} * (TPBDC + TPADC) = \frac{1}{2} * (1+1) = 1$

**5.3.4 Design Verification and Measurement for Traceability**

Design verification and measurement of traceability refers to the ability of verifying and measuring the built-in-mechanism design and the API design by traceability.

● **A Checklist to Verify Built-in Mechanism Design for Component Traceability**

● If the requirements states having a defined trace format, does its corresponding design provide a consistent, complete, and correct trace format?	<input type="radio"/> Yes	<input type="radio"/> No
● If the requirements states having trace code insertion in the requirement, does its corresponding design provide a consistent, complete, and correct trace solution?	<input type="radio"/> Yes	<input type="radio"/> No
● If the requirements states having trace collection, does its corresponding design provide a consistent, complete, and correct trace solution?	<input type="radio"/> Yes	<input type="radio"/> No
● If the requirements states having trace storage, does its corresponding design provide a consistent, complete, and correct trace solution?	<input type="radio"/> Yes	<input type="radio"/> No
● If the requirements states having operational tracking, does its corresponding design provide a consistent, complete, and correct trace solution?	<input type="radio"/> Yes	<input type="radio"/> No
● If the requirements states having performance tracking in the requirement, does its corresponding design provide a consistent, complete, and correct trace solution?	<input type="radio"/> Yes	<input type="radio"/> No

- If the requirements states having state tracking, does its corresponding design provide a consistent, complete, and correct trace solution?  Yes  No
- If the requirements states having event tracking, does its corresponding design provide a consistent, complete, and correct trace solution?  Yes  No
- If the requirements states having error tracking in the requirement, does its corresponding design provide a consistent, complete, and correct trace solution?  Yes  No

● **A Comparison Tables to Measure Built-in-Mechanism Design for Traceability**

Built-in-Mechanism of Traceability	BT1	BT2	Built-in-Mechanism of Traceability	BT1	BT2
Design for trace format	0.11	0.11	Design for performance tracking	0	0
Design for trace code insertion	0	0.11	Design for state tracking	0	0.11
Design for trace collection	0.11	0.11	Design for event tracking	0	0.11
Design for trace storage	0.11	0.11	Design for error tracking	0.11	0.11
Design for operational tracking	0	0.11			

**Table 50. Comparison Table for Built-in-Mechanism Traceability in Design**

● **A Checklist to Verify API Design for Component Traceability**

- Does the design need source code in order to perform tracking?  Yes  No
- Does the design allowed Code Separation?  Yes  No
- Does the design consider Overhead?  Yes  No
- Does the design demonstrate with high/low complexity?  High  Low
- Does the design demonstrate with high/low flexibility?  High  Low
- Does the design provide applicability?  Yes  No  
 If so, what kind of applicability is it?  OP Trace  Performance Trace
- Does the design include applicable components?  Yes  No  
 If so, what kind of applicable components is it?  In-house components  COTS

● **A Comparison Table to Measure API Design for Traceability**

API Design of Traceability	TP		API Design of Traceability	TP	
	BT1	BT2		BT1	BT2
Tracking			Complexity		
Need Source Code(1.0)	1	1	High Complexity(1.0)	0.5	1
Without Source code(0.5)			Low Complexity(0.5)		
Code Separation			Flexibility		
Allow(1.0)	0.5	1	High Flexibility(1.0)	0.5	1
Not Allow(0.5)			Low Flexibility(0.5)		
Overhead			Applicability		
Consider Overhead(1.0)	0.5	1	OP Trace(1.0)	0	0
Without Overhead(0.5)			Performance Trace(0.8)		
			Applicable Components	BT1	BT2
			In-house Components(1.0)	0	0
			COTS(0.8)		

**Table 51. Comparison Table for API Traceability in Design**

● **Average Traceability Metrics for Design (ATMD)**

Binary Tree Version1 TPBDT = Total Counts =  $0.11 * 4 = 0.44$

Binary Tree Version2 TPBDT = Total Counts =  $0.11 * 8 = 0.88$

Binary Tree Version1 TPADT =  $\frac{1}{7}$  (Total Testing Points) =  $\frac{1}{7} * (1+0.5+0.5+0.5+0.5) = 0.43$

Binary Tree Version2 TPADT =  $\frac{1}{7}$  (Total Testing Points) =  $\frac{1}{7} * (1+1+1+1+1) = 0.71$

Binary Tree Version1 ATMD =  $\frac{1}{2}$  ( TPBDT + TPADT ) =  $\frac{1}{2} * (0.44+0.43) = 0.44$

Binary Tree Version2 ATMD =  $\frac{1}{2}$  ( TPBDT + TPADT ) =  $\frac{1}{2} * (0.88+0.71) = 0.8$

### 5.3.5 Design Verification and Measurement for Test Support Capability

Design verification and measurement of test support capability refers to the ability of verifying and measuring component design by test support capability.

#### ● A Checklist to Verify Design for Component Test Support Capability

● If the requirements states having a test generation mechanism, does its corresponding design provide a consistent, complete, and correct trace solution?	<input type="radio"/> Yes	<input type="radio"/> No
● If the requirements states having test management capability, does its corresponding design provide a consistent, complete, and correct trace solution?	<input type="radio"/> Yes	<input type="radio"/> No
● If the requirements states supporting test coverage, does its corresponding design provide a consistent, complete, and correct trace solution?	<input type="radio"/> Yes	<input type="radio"/> No
● If the requirements states having component test scripting capability, does its corresponding design provide a consistent, complete, and correct trace solution?	<input type="radio"/> Yes	<input type="radio"/> No

#### ● A Comparison Table to Measure Design for Test Support Capability

Test Support Capability	BT1	BT2	Test Support Capability	BT1	BT2
Design for test generation mechanism	0	0.25	Design for supporting test coverage	0.25	0.25
Design for test management capability	0	0.25	Design for component test scripting capability	0	0.25

**Table 52. Comparison Table for Component Test Support Capability in Design**

#### ● Average Test Support Capability Metrics for Design (ATSMD)

**Binary Tree Version1 ATSMD = Total Testing Points =  $0.25 * 1 = 0.25$**

**Binary Tree Version2 ATSMD = Total Testing Points =  $0.25 * 4 = 1$**

## 5.4 Final Results and Comparison

According to Table 53, the summary of chapter 5.3, two pentagram models can be draw as Figure 21 and 22. In addition, the testability of binary tree version1 and 2 can be obtained as following based on the testability metrics, calculated from chapter 4.3.

Requirement Metrics of Binary Tree Version1					Requirement Metrics of Binary Tree Version2				
AUMR	AOMR	ACMR	ATMR	ATSMR	AUMR	AOMR	ACMR	ATMR	ATSMR
0.52	0.6	0.03	0.01	0.08	0.75	1.0	0.78	0.46	0.66

Table 53. Summary Table of Requirements Metrics

### Testability of Requirements (Binary Tree Version 1)

$$=0.48*(ACMR*AOMR+AOMR*ACMR+AUMR*ATMR+ATMR*ATSMR+ACMR*ATSMR)$$

$$= 0.48 * ((0.52*0.6)+(0.6*0.03)+(0.52*0.01)+(0.01*0.08)+(0.03*0.08))=0.16$$

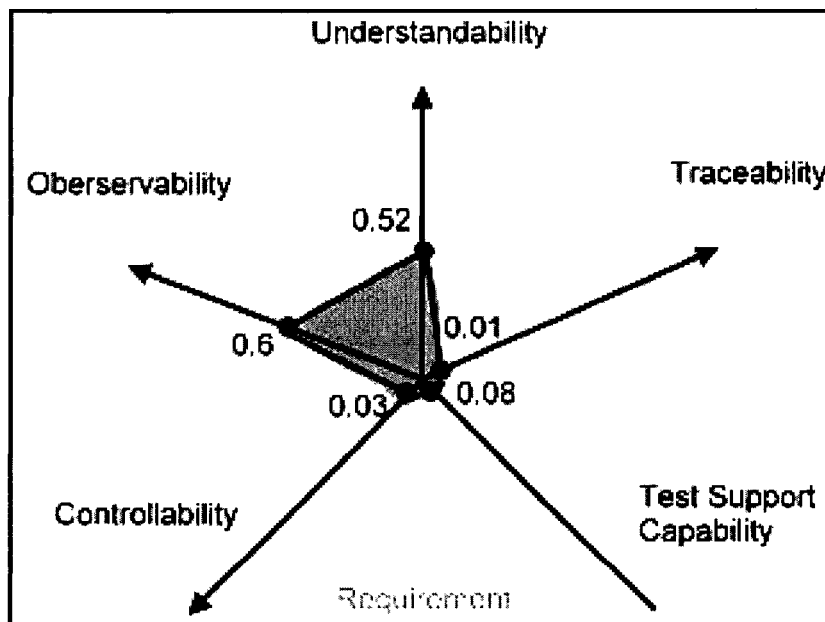
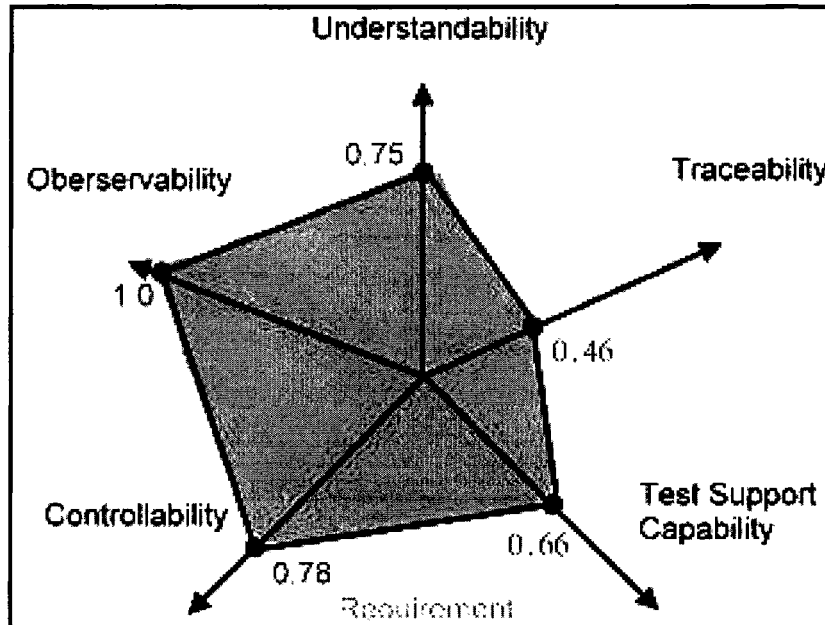


Figure 21. Pentagram of Requirements (Binary Tree - Version 1)

**Testability of Requirements (Binary Tree Version 2)**

$$= 0.48 * (ACMR * AOMR + AOMR * ACMR + AUMR * ATMR + ATMR * ATSMR + ACMR * ATSMR)$$

$$= 0.48 * (0.75 * 1 + 1 * 0.78 + 0.75 * 0.46 + 0.46 * 0.66 + 0.78 * 0.66) = 1.29$$



**Figure 22. Pentagram of Requirements (Binary Tree - Version 2)**

According to Table 54, the summary of chapter 5.4, two pentagram models can be draw as Figure 23 and 24. In addition, the testability of binary tree version1 and 2 can be obtained as following based on the testability metrics, calculated from chapter 4.3.

Design Metrics of Binary Tree Version1					Design Metrics of Binary Tree Version2				
AUMD	AOMD	ACMD	ATMD	ATSM D	AUMD	AOMD	ACMD	ATMD	ATSM D
0.08	0.6	0.56	0.44	0.25	0.66	1.0	1.0	0.8	1.0

**Table 54. Summary Table of Design Metrics**

### Testability of Design (Binary Tree Version 1)

$$=0.48*(ACMR*AOMR+AOMR*ACMR+AUMR*ATMR+ATMR*ATSMR+ACMR*ATSMR)$$

$$= 0.48 * ((0.6*0.6)+(0.6*0.56)+(0.6*0.44)+(0.44*0.25)+(0.56*0.25))=0.58$$

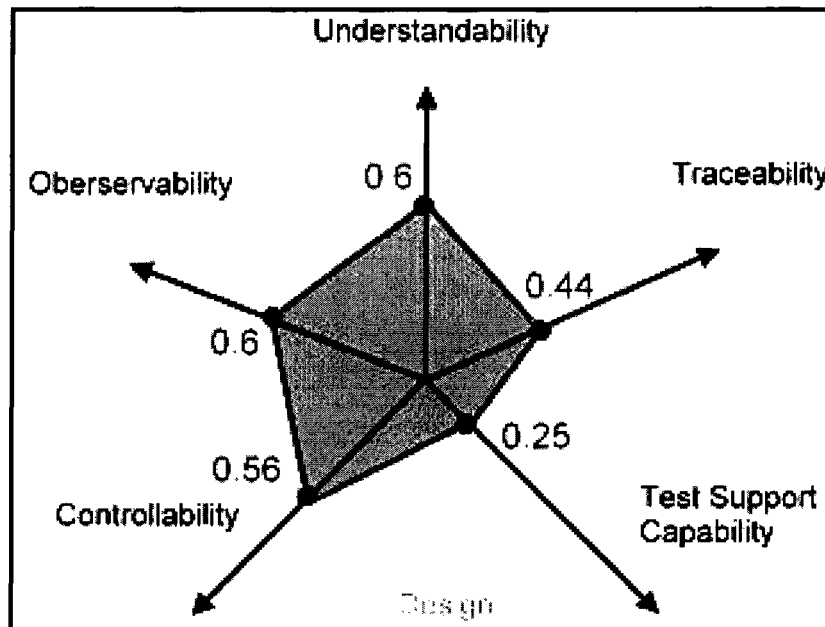


Figure 23. Pentagram of Design (Binary Tree - Version 1)

### Design of Testability (Binary Tree Version 2)

$$=0.48*(ACMR*AOMR+AOMR*ACMR+AUMR*ATMR+ATMR*ATSMR+ACMR*ATSMR)$$

$$= 0.48 * ((1*1)+(1*1)+(1*0.8)+(0.8*1)+(1*1))=2.2$$

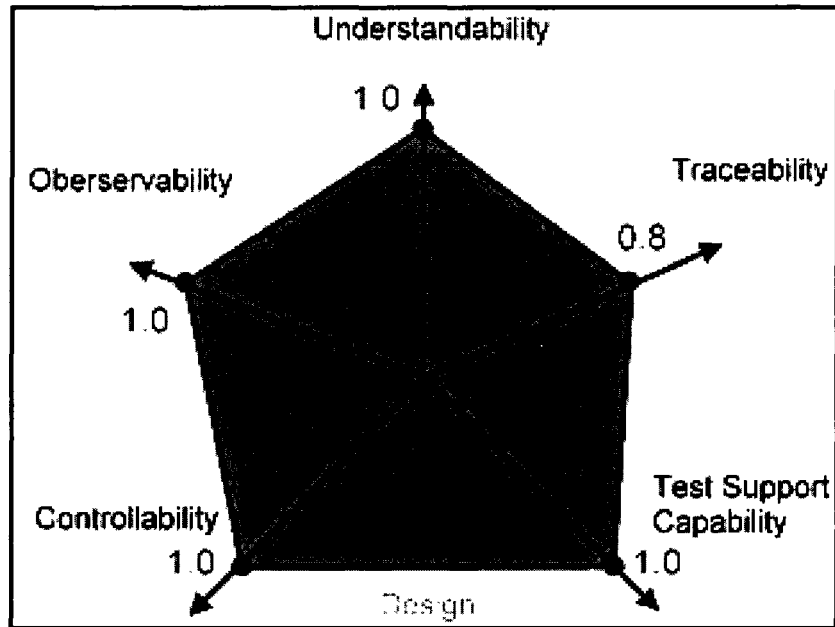


Figure 24. Pentagram of Design (Binary Tree - Version 2)

- Comparison of Final Results.

Inspecting the gain chart, shown in Figure 25, can easily illustrate a clear comparison between the two different versions of binary tree application.

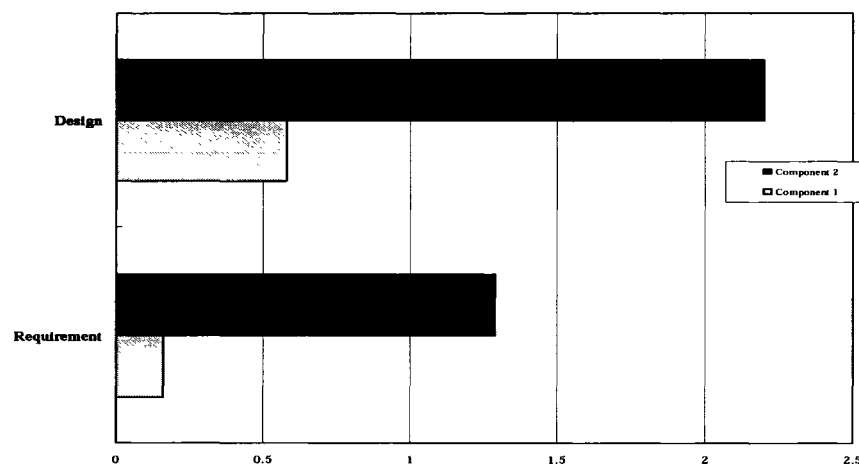


Figure 25. Comparing Testability of The Two Versions



## **6. Conclusions**

The work presented in this thesis proposes an approach to verifying and measuring software component testability. In this last chapter, the work is summarized and concluded. In addition, some topics and research areas are recommended for further study.

### **6.1 Summary**

The beginning of this thesis introduces concepts of software component testability and several definitions related to its verification and measurement. It is also understood that good testability of software components can reduce not only the testing cost, but also the effort expended in developing software components. From the point of view of this thesis, good software component testability needs five essential factors: understandability, observability, controllability, traceability, and test support capability.

By using these five factors, the testability of software components can be verified and measured. From the requirements to the design phase, these five factors ensure that the proposed systematic solution can be applied to verify testability of software components. Also, by applying the designed metrics, we can measure the testability of software components in each of these five factors and obtain five testing points in order to draw the pentagram. The area of the pentagram will be the final value for the testability of software components.

Finally, there is a case study to compare two different versions of the binary tree application. One version follows the traditional way, and the other is created through addressing the five factors. Comparing the two versions, we can see a huge improvement

in the testing points, which means better testability. This will prove that addressing these five factors did improve the testability of software components.

## **6.2 Conclusions and Recommendations for Future Research**

An ounce of prevention is worth a pound of cure. It also can be applied to the process of building software components. Every software manager and engineer would like to discover a better way of reducing testing effort and cost. This thesis introduces a more efficient way to reduce software defects and costs by enhancing the testability of software components in the requirements and design phases.

Several other areas suggest themselves for future research.

1. **Implementation for Testability:** In the software development cycle, implementation is the phase after the requirements and design phase. After achieving good testability of requirements and design, we can then create implementation by addressing these five factors. The development of self-testable, test-reusable, and run-time testable code, on the same platform of conventional OO programming, is also recommended.

2. **Solutions for Building Testable Software Components:** A systematic solution for adding testability features to existing software components. This will reduce cost and effort for existing software components that were built without considering testability.

3. **Applied Testability for Building Hardware:** Can the concept of testability also be applied for building hardware? Hardware can be viewed as a component, and it should be possible to add testability features.

## References

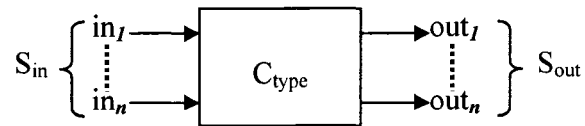
- Albrecht, A. J. (1979). *Measuring Application Development Productivity*. Proceedings of IBM Applications Development Symposium, 83-92.
- Baldini, A. & Prinetto, P. (2003, April). Design for Testability for Highly Reconfigurable Component-Based Systems: Proceedings of the International Workshop on Testing and Analysis of Component Based Systems. *Elsevier Science B. V.* Retrieved from <http://www1.elsevier.com/gej-ng/31/29/23/133/50/41/82.6.020.pdf>
- Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., & Wallnau, K. (2000, May). *Volume I: Market Assessment of Component Based Software Engineering*. Retrieved from <http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tn007.pdf>
- Beckert, B., and Schlager, S. (2004). Software Verification with Integrated Data Type Refinement for Integer Arithmetic. *IFM*, 207-226.
- Blackburn, M. R., Busser, R. D., & Nauman, A. M. (2001, May). Removing Requirement Defects and Automating Test. *STAREAST*. Retrieved from <http://www.software.org/pub/externalpapers/RemovingRqmtDefects.doc>
- Cem Kaner, J. D. (2001, March). *Measurement Issues and Software Testing*. Retrieved from [http://www.kaner.com/pdfs/measurement\\_segue.pdf](http://www.kaner.com/pdfs/measurement_segue.pdf)
- Freedman, R. S. (1991, June). Testability of Software Components. *IEEE Transactions on Software Engineering*, 17(6), 553-563.
- Fenton, N. E., Krause, P., & Neil, M. (2002). Software Measurement: Uncertainty and Causal Modelling, *IEEE Software* 10(4), 116-122.

- Gao, J., Tsao, J. & Wu, Y. (2003, September). *Testing and Quality Assurance for Component-Based Software*, MA: Artech House.
- Hetzel, W. C. (1988). *The Complete Guide to Software Testing*. Mass:Wellesley.
- IEEE Standard Glossary of Software Engineering Terminology. (1990). New York: IEEE Press.
- Jungmayr, S. (2002, October). *Testability Measurement and Software Dependencies: Proceedings of the 12th International Workshop on Software Measurement*. Retrieved from <http://www.dasma.org/contray/media/iwsm-inhalt.pdf>
- Kelvin, L.(1894). *Popular Lectures and Addresses*. London: Macmillan.
- McCall, J. A., Richards, P. K. & Walters, G. F. (1977). *Factors in Software Quality (Vol. 1-3)*. Springfield, VA: National Technical Information Service.
- Myers, G.,(1979). *The Art of Software Testing*. U.S.: John Wiley and Sons
- Robach, C., Traon Y. (1995). Testability Analysis of Co-Design Systems. *Asian Test Symposium, 4th*, 206-213.
- Robertson, J. and Robertson, S. (1997, August). *Requirements: Made to Measure. American Programmer, 10 (8)*.
- Royce, W. W. (1970, November) *Managing the development of large software systems*. International Conference on Software Engineering, 9th , 328-338.
- Veryard, R. (1996, January) *IFIP WG 5.4, AQUIS 96: Third International Conference on Achieving Quality in Software*, Florence, Italy.
- Voas, J. M., and Miller, K.W. (1995, May). Software Testability: The New Verification. *IEEE Software 12 (3)*, 17-28.

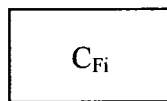
## Appendices

### Appendix A. Notation Page

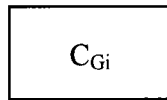
#### Notation Page



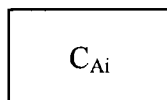
#### Framework Component



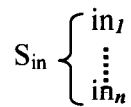
#### GUI Component



#### Application Component



Input list:  $\langle \text{Input List} \rangle$ :  $\langle \text{Data types allowed on port } , \{in_1 \dots in_n\} \rangle$

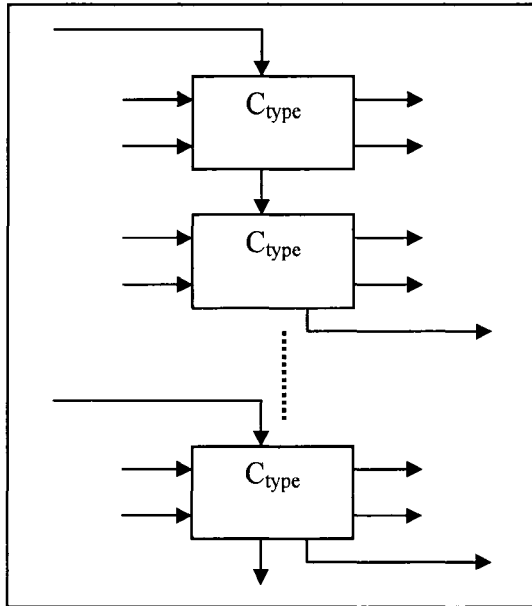


Output list: <Output List>: <Data types allowed on port, {out<sub>1</sub>...out<sub>n</sub>}>

out<sub>1</sub>  
⋮  
out<sub>n</sub> } S<sub>out</sub>

Direction flow : →

Component inside view:



## Appendix B. Binary Tree

### ● Requirements of Binary Tree Version 1

<p><b>Non-Functional Requirements (NFReq)</b></p>
<p><b>Environment Requirements:</b></p> <p><b>NFReq1:</b> The application will be written in Java.</p> <p><b>NFReq2:</b> The system has a minimum of Java 1.3 SDK installed.</p> <p><b>NFReq3:</b> The minimum PC specification that the system must be run on is a 200 MHz Pentium PC with 100 Mb of spare disk space and 60 Mb of memory.</p> <p><b>NFReq4:</b> The system must run under MS Windows technology.</p>
<p><b>Availability Requirements:</b></p> <p><b>NFReq5:</b> After the binary tree application has been run, users can start to access the binary tree.</p> <p><b>NFReq6:</b> Users can get a response from the binary tree in 99% of the cases within three days.</p> <p><b>NFReq7:</b> When users have finished the binary tree application, they can no longer access the binary tree.</p>
<p><b>Performance Requirements:</b></p> <p><b>NFReq8:</b> Over a three-day period of running the binary tree application, the system response to each user action must be less than 1 second in 99% of the cases.</p>
<p><b>Scalability Requirements:</b></p> <p><b>NFReq9:</b> The file size that can be loaded must be limited to no more than 64 nodes. If the file contains more than 64 nodes, the loading will function to terminate at the 64<sup>th</sup> node and will display a message stating that the tree is full.</p> <p><b>NFReq10:</b> The tree will hold no more than 64 nodes. If a user tries to insert an element after the limit has been reached, a message will be displayed stating that the tree is full.</p>
<p><b>Throughput Requirements:</b> None for this binary tree.</p>
<p><b>Reliability Requirements:</b></p> <p><b>NFReq11:</b> Over a three-day period of running the binary tree application, users must get a 99% correct action when inserting, deleting, or searching a node from the binary tree.</p> <p><b>NFReq12:</b> While the application is being run, all data will be held in memory. If there is a system crash, the data will automatically be recovered.</p>
<p><b>Utilization Requirements:</b></p> <p><b>NFReq11:</b> The utilization of system resources can be monitored.</p>

---

**Functional Requirement(FReq)**

---

**Function Feature: Load a Tree Category : Evident**

**FReq1.1:** A pre-saved tree can be loaded into the program for viewing.

**FReq1.2:** Only one tree can be loaded, as the user can save only one tree.

---

**Function Feature: Save a Tree Category : Evident**

**Freq2.1:** User can save the latest binary tree.

**Freq2.2:** Only one tree can be saved.

**Freq2.3:** The tree gets a default name specified by the program.

**FReq2.4:** All data have to be in ascending order before saving.

---

**Functional Feature: Insert a Node Category : Evident**

**FReq3.1:** Users can add a node in the binary tree.

**FReq3.2:** The number entered must be an integer.

**FReq3.3:** The number cannot be repeated.

**FReq3.4:** After users insert a node, the binary tree will be still in ascending order.

---

**Functional Feature: Delete a Node Category : Evident**

**FReq4.1:** Users can delete a node in the binary tree.

**FReq4.2:** The number entered must be an integer.

**FReq4.3:** An error message will appear when input is not an integer.

**FReq4.4:** The number that a user deletes must exist in the binary tree.

**FReq4.5:** An error message will appear when a value is not present in the binary tree.

---

**Functional Feature: Find a Node Category : Evident**

**FReq5.1:** Users can find the position of a node in the binary tree.

**FReq5.2:** The number entered must be an integer.

**FReq5.3:** An error message will appear when input is not an integer.

**FReq5.4:** The number entered must exist in the binary tree.

**FReq5.5:** An error message will appear when a value is not present in the binary tree.

---

**Functional Feature: Ascending Order Category : Evident**

**FReq6.1:** This function rearranges all nodes by using in the in-order algorithm.

---

**Functional Feature: Paint a Tree Category : Evident**

**FReq7.1:** This function displays the binary tree.

**FReq7.2:** A warning message will be displayed if the binary tree is empty.

---



---

## **Other Requirements**

---

### **Observability Requirements (OReq)**

- OReq1:** Users can observe input/output data manually.
- OReq2:** Users can allow users to observe communication data manually.
- OReq3:** Users can allow users to observe GUI events data manually.
- OReq4:** Users can allow users to observe function operation data manually.

---

### **Controllability Requirements (CReq)**

- CReq1:** A system log collects all component behavior manually.
- CReq2:** A component environment controller interface allows users to install and setup environment.

---

### **Traceability Requirements (TReq)**

- TReq1:** A tracking mechanism can monitor and trace users/system behavior.
- TReq2:** A tracking mechanism has a consistent well-define format.
- TReq3:** A tracking mechanism collects the tracking message.
- TReq4:** A tracking mechanism can save all messages into a file.
- TReq5:** A tracking mechanism can provide performance tracking which includes throughput, reliability, available, scalability and utilization.
- TReq6:** It supports warning messages tracking.

---

### **Test Support Requirements (TSReq)**

- TSReq1:** The Binary-Tree has well-defined test coverage.
-

- Requirements of Binary Tree Version 2

The bold face part shows the difference from version 1.

---

### **Non-Functional Requirements (NFReq)**

#### **Environment Requirements:**

**NFReq1:** The application will be written in Java.

**NFReq2:** The system has a minimum of Java 1.3 SDK installed.

**NFReq3:** The minimum PC specification that the system must be run on is a 200 MHz Pentium PC with 100 Mb of spare disk space and 60 Mb of memory.

**NFReq4:** The system must run under MS Windows technology.

**NFReq5:** **The Binary Tree application will check the environment to ensure that the environment meets the minimum requirements before the application is run. A warning message will be displayed when the environment is not suitable for the binary tree application.**

**NFReq6:** **A specifically designed monitor will identically record the time and other details of the checking procedure, and will record them in the system log.**

---

#### **Availability Requirements:**

**NFReq7:** After the binary tree application has been run, users can start to access the binary tree.

**NFReq8:** Users can get a response from the binary tree in 99% of the cases within three days.

**NFReq9:** When users have finished the binary tree application, they can no longer access the binary tree.

**NFReq10:** **A specifically designed monitor will record the errors and other details when users cannot get any response from system, and will record them identically in the system log.**

---

---

**Performance Requirements:**

**NFReq11:** Over a three-day period of running the binary tree application, the system response to each user action must be less than 1 second in 99% of the cases.

**NFReq12:** A specifically designed monitor will record the time of each action and save it in the system log.

---

**Scalability Requirements:**

**NFReq13:** The file size that can be loaded must be limited to no more than 64 nodes. If the file contains more than 64 nodes, the loading will function to terminate at the 64<sup>th</sup> node and will display a message stating that the tree is full.

**NFReq14:** The tree will hold no more than 64 nodes. If a user tries to insert an element after the limit has been reached, a message will be displayed stating that the tree is full.

**NFReq15:** A specifically designed monitor will record every error into the system log.

---

**Throughput Requirements:** None for this binary tree.

---

**Reliability Requirements:**

**NFReq16:** Over a three-day period of running the binary tree application, users must get a 99% correct action when inserting, deleting, or searching a node from the binary tree.

**NFReq17:** While the application is being run, all data will be held in memory. If there is a system crash, the data will automatically be recovered.

**NFReq18:** A specifically designed monitor will record every error into the system log.

---

**Utilization Requirements:**

**NFReq19:** The utilization of system resources can be monitored.

**NFReq20:** When system resources are running out, a warning message will be displayed.

---

---

**Functional Requirements (FReq)**

---

**Function Feature: Load a Tree Category : Evident**

**FReq1.1:** A pre-saved tree can be loaded into the program for viewing.

**FReq1.2:** Only one tree can be loaded, as the user can save only one tree.

**Function Feature: Save a Tree Category : Evident**

**Freq2.1:** Users can save the latest binary tree.

**Freq2.2:** Only one tree can be saved.

**Freq2.3:** The tree gets a default name specified by the program.

**FReq2.4:** All data have to be in ascending order before being saved.

**Functional Feature: Insert a Node Category : Evident**

**FReq3.1:** Users can add a node in the binary tree.

**FReq3.2:** The number entered must be an integer.

**FReq3.3:** The number cannot be repeated.

**FReq3.4:** After users insert a node, the binary tree will be still in ascending order.

**Functional Feature: Delete a Node Category : Evident**

**FReq4.1:** Users can delete a node in the binary tree.

**FReq4.2:** The number entered by a user must be an integer.

**FReq4.3:** An error message will appear when input is not an integer.

**FReq4.4:** The number that a user deletes must exist in the binary tree.

**FReq4.5:** An error message will appear when a value is not present in the binary tree.

**Functional Feature: Find a Node Category : Evident**

**FReq5.1:** Users can find the position of a node in the binary tree.

**FReq5.2:** The number entered by a user must be an integer.

**FReq5.3:** An error message will appear when input is not an integer.

**FReq5.4:** The number entered must exist in the binary tree.

**FReq5.5:** An error message will appear when a value is not present in the binary tree.

**Functional Feature: Ascending Category : Evident**

**FReq6.1:** This function rearranges all nodes by using in the in-order algorithm.

**Functional Feature: Paint a Tree Category : Evident**

**FReq7.1:** This function displays the binary tree.

**FReq7.2:** A warning message will be displayed if the binary tree is empty.

---

---

## **Other Requirements**

---

### **Observability Requirements (OReq)**

- OReq1.1:** A designed monitor must allow users to observe input/output data **automatically**.
- OReq1.2:** A designed monitor must allow users to observe communication data **automatically**.
- OReq1.3:** A designed monitor must allow users to observe GUI events data **automatically**.
- OReq1.4:** A designed monitor must allow users to observe function operation data **automatically**.

---

### **Controllability Requirements (CReq)**

- CReq1:** A system log collects all component behavior **automatically**.
- CReq2:** A designed interface can customize component features.
- CReq3:** A designed feature will install and deploy components automatically.
- CReq4:** A component environment controller, having a setup feature and additionally managing deployment, is installed automatically
- CReq5:** A component execution controller has three control modes: test mode, control mode, and normal function mode.
- CReq6:** A component feature controller has an enable/disable function feature.
- CReq7:** A component state-based behavior controller has a transition control, state control, and reset control function.
- CReq8:** A component test controller has an interface allowing users to have five features: function invocation control, input data setup, output data monitor, incoming message setup, and outgoing message monitor.

---

### **Traceability Requirements (TReq)**

- TReq1:** A tracking mechanism can monitor and trace users/system behavior.
  - TReq2:** A tracking mechanism has consistent, well-defined format.
  - TReq3:** A tracking mechanism allows automatic code insertion.
  - TReq4:** A tracking mechanism generates trace code automatically.
  - TReq5:** A tracking mechanism provides automatic trace collection.
  - TReq6:** A tracking mechanism provides system, component and class/function tracking.
  - TReq7:** A tracking mechanism provides performance tracking, which includes reliability, availability, scalability, and utilization.
  - TReq8:** The tracking mechanism provides system, component, and object state tracking.
  - TReq9:** The tracking mechanism provides GUI event tracking.
  - TReq10:** The tracking mechanism provides warning message tracking.
-

---

### **Test Support Requirements (TSReq)**

**TSReq1: The Binary-Tree generates component white-box and black-box testing.**

**TSReq2: The Binary-Tree provides test management capability, which includes Problem Management, Test Management, and Suit Management.**

**TSReq3: The Binary-Tree has well-defined test coverage.**

**TSReq4: The Binary-Tree provides test coverage analysis tools, which provide measuring and monitoring functions, and generate a report.**

**TSReq5: The Binary-Tree has Component Test Scripting Capability.**

**TSReq6: The Binary-Tree has Component Test Scripting Capability, which has Test Scripts, Test stubs, and Test Drivers.**

---

● Design of Binary Tree Version 1

### Interface Summary

Vector	<b>readData</b> (BufferedReader in, Vector element) Reads the data line by line from an external file
void	<b>writeData</b> (PrintWriter out, Vector values) Writes stored data to an external file line by line in a comma delimited format
void	<b>Insert</b> (int nKey) Inserts the passed integer into the tree
BSTree	<b>Delete</b> (BSTree value) Deletes the passed value from the tree and returns a pointer to the tree node
BSTree	<b>Find</b> (int nKey) Searches the tree for the passed integer and returns a pointer to the tree node
Vector	<b>Clear</b> (Vector stored, BSTree node) Takes the passed empty vector and the tree's root node and returns a Vector with all of the tree's contents stored using the in-order algorithm
Vector	<b>Ascend</b> (Vector stored, BSTree node) Takes the passed empty vector and the tree's root node and returns a Vector with all of the tree's contents stored using the in-order algorithm
Vector	<b>Store</b> (Vector stored, BSTree node) Takes the passed empty vector and the tree's root node and returns a Vector with all of the tree's contents stored using the pre-order algorithm
void	<b>PaintTree</b> (BSTree subtree, Graphics g, int x, int y, int Width, int Height) Takes the passed parameters and draws the current tree
void	<b>Non_Functional_Checking()</b> Checks and monitors Environment, Availability, Performance, Scalability, Utilization requirements.
void	<b>Observe_Behavior()</b> Observes all component behaviors and saves the following data into system log. 1. Input/output data. 2.Communcation Data 3. GUI events data      4.Function operation Data

## Interface Summary

void	<b>Obersevability()</b> Makes sure that the numbers of input equals to numbers of output.
void	<b>Collect_Behavior()</b> Collects and saves all component behaviors into system log.
void	<b>Environmetn_Controller()</b> Allows users to install and setup environment.
void	<b>Controllability()</b> Makes sure that the same input always gets the same output.
Void	<b>Trace()</b> A tracking mechanism: <ol style="list-style-type: none"> <li>1. monitors and traces users/system behavior</li> <li>2. has a consistent, well-defined format</li> <li>3. collects tracking messages manually.</li> <li>4. saves all messages into a file</li> <li>5. needs Source Code to track</li> <li>6. does not allow code separation</li> <li>7. is without overhead</li> <li>8. designs with high complexity</li> <li>9. designs with low flexibility</li> </ol>
void	<b>Error_Tracking()</b> Allows warning message tracking.
void	<b>Support()</b> Supports well-defined test coverage



<b>Button Action Summary</b>	
void <b>mouseClicked</b> (MouseEvent e) - <b>loadAction</b>	Functions Called: readData()                      Find()                      Insert()                      PaintTree()
void <b>mouseClicked</b> (MouseEvent e) - <b>saveAction</b>	Functions Called: Store()                                      writeData()                                      PaintTree()
void <b>mouseClicked</b> (MouseEvent e) - <b>insertAction</b>	Functions Called: Search()                                      Insert()                      Ascend()                      PaintTree ()
void <b>mouseClicked</b> (MouseEvent e) - <b>deleteAction</b>	Functions Called: Search()                                      Delete()                                      PaintTree ()
void <b>mouseClicked</b> (MouseEvent e) - <b>findAction</b>	Functions Called: Search()                                      PaintTree ()                                      Trace()
void <b>mouseClicked</b> (MouseEvent e) - <b>ascendingAction</b>	Functions Called: Ascend()                                      PaintTree ()                                      Trace()

- Design of Binary Tree Version 2

### Interface Summary

Vector	<b>readData</b> (BufferedReader in, Vector element) Reads the data line by line from an external file
void	<b>writeData</b> (PrintWriter out, Vector values) Writes stored data to an external file line by line in a comma delimited format
void	<b>Insert</b> (int nKey) Inserts the passed integer into the tree
BSTree	<b>Delete</b> (BSTree value) Deletes the passed value from the tree and returns a pointer to the tree node
BSTree	<b>Find</b> (int nKey) Searches the tree for the passed integer and returns a pointer to the tree node
Vector	<b>Clear</b> (Vector stored, BSTree node) Takes the passed empty vector and the tree's root node and returns a Vector with all of the tree's contents stored using the in-order algorithm
Vector	<b>Ascend</b> (Vector stored, BSTree node) Takes the passed empty vector and the tree's root node and returns a Vector with all of the tree's contents stored using the in-order algorithm
Vector	<b>Store</b> (Vector stored, BSTree node) Takes the passed empty vector and the tree's root node and returns a Vector with all of the tree's contents stored using the pre-order algorithm
void	<b>PaintTree</b> (BSTree subtree, Graphics g, int x, int y, int Width, int Height) Takes the passed parameters and draws the current tree
void	<b>Non_Functional_Checking()</b> Checks and monitors Environment, Availability, Performance, Scalability, Utilization requirements.
void	<b>Observe_Behavior()</b> Observes all component behaviors and saves the following data into system log. 1. Input/output data. 2.Communcation Data 3. GUI events data      4.Function operation Data

## Interface Summary

void	<b>Obersevability()</b> Makes sure that the numbers of input equals to numbers of output.
void	<b>Collect_Behavior()</b> Collects and saves all component behaviors into system log automatically.
void	<b>Customize_Feature()</b> A designed interface can customize component feature.
void	<b>Install_Deployment()</b> A designed feature for install and deploy component automatically.
void	<b>Environment_Controller()</b> Allows users to install and setup environment.
void	<b>Execution_Controller()</b> A component execution controller has three control modes, which are test mode, control mode, and normal function mode.
void	<b>State_Based_Cotroller()</b> A component state-based behavior controller has transition control, state control, and reset control function.
void	<b>Test_Controller()</b> A component test controller has an interface that allow users to have the function invocation control, input data setup, output data monitor, incoming message setup, and outgoing message monitor.
void	<b>Feature_Controller()</b> A component feature controller can enable or disable function feature.
void	<b>Controllability()</b> Makes sure that the same input always gets the same output.
void	<b>Trace()</b> A tracking mechanism: <ol style="list-style-type: none"> <li>1. monitors and traces users/system behavior</li> <li>2. has a consistent well-defined format</li> <li>3. allows automatic code insertion</li> <li>4. generates trace code automatically</li> <li>5. provides automatic trace collection</li> </ol>

## Interface Summary

	6. needs source code 7. allows code separation 8. considers overhead 9. designs with low complexity 10. designs with high flexibility
void	<b>Operation_Tracking()</b> Provides system, component and class function tracking.
void	<b>Performance_Tracking()</b> Provides performance tracking which includes reliability, available, scalability and utilization.
void	<b>State_Tracking()</b> Provides system, component and object state tracking
void	<b>Event_Tracking()</b> Provides GUI event tracking
void	<b>Error_Tracking()</b> Allows warning message tracking.
void	<b>Test_Generate()</b> Generate component white-box and black-box Test
void	<b>Test_Management()</b> Includes problem Management, Test Management and Suit Management
void	<b>Test_Coverage()</b> Well-define test coverage which can provide measure and monitor function, and generate a report
void	<b>Test_Script()</b> Has Test Scripts, Test stubs and Test Drivers.

<b>Button Action Summary</b>	
void <b>mouseClicked</b> (MouseEvent e) - <b>loadAction</b>	Functions Called: readData()                      Find()    Insert()    PaintTree()
void <b>mouseClicked</b> (MouseEvent e) - <b>saveAction</b>	Functions Called: Store()                              writeData()                      PaintTree()
void <b>mouseClicked</b> (MouseEvent e) - <b>insertAction</b>	Functions Called: Search()                              Insert()    Ascend()    PaintTree ()
void <b>mouseClicked</b> (MouseEvent e) - <b>deleteAction</b>	Functions Called: Search()                              Delete()                              PaintTree ()
void <b>mouseClicked</b> (MouseEvent e) - <b>findAction</b>	Functions Called: Search()                              PaintTree ()                              Trace()
void <b>mouseClicked</b> (MouseEvent e) - <b>ascendingAction</b>	Functions Called: Ascend()                              PaintTree ()                              Trace()