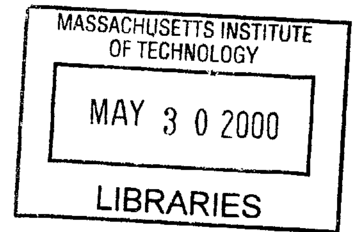
CONCEPTS OF TESTING FOR COLLOCATED AND DISTRIBUTED SOFTWARE DEVELOPMENT


MASSACHUSETTS INSTITUTE
OF TECHNOLOGY
MAY 3 0 2000
LIBRARIES

by

KENWARD MA

Bachelor of Science in Civil and Environmental Engineering
Cornell University, 1999.

ENG

Submitted to the Department of Civil and Environmental Engineering
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF ENGINEERING
IN CIVIL AND ENVIRONMENTAL ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

©2000 Massachusetts Institute of Technology. All rights reserved.

Signature of Author..
Kenward Ma
Department of Civil and Environmental Engineering
May 5, 2000

Certified by...
Feniosky Peña-Mora
Associate Professor, Department of Civil and Environmental Engineering
Thesis Supervisor

Accepted by...................................
Daniele Veneziano
Chairman, Departmental Committee on Graduate Studies

CONCEPTS OF TESTING FOR COLLOCATED AND DISTRIBUTED SOFTWARE DEVELOPMENT

by

KENWARD MA

Submitted to the
Department of Civil and Environmental Engineering
on May 5, 2000

In Partial Fulfillment of the Requirements for the Degree of

MASTER OF ENGINEERING

IN CIVIL AND ENVIRONMENTAL ENGINEERING

Abstract

Software testing is a significant process for software development because it provides the final verification of the product's quality. With the objective to discover errors encountered by the implementation of a product, testing is a very broad field of knowledge containing different principles, strategies, and techniques. Testing can be divided into different categories that describe the approaches we may use to test a product. Many testing tools have been developed to enhance the efficiency of test execution, and a great deal of research efforts has been put into this topic to upgrade the effectiveness of testing. Besides these technical issues, team concept is another important issue for software development and testing. Even though distributed development teams encounters many difficulties, they have become more popular for different industries. To allow full collaboration and to get the best outcome from distributed development teams, strong management and organization are required.

Thesis Supervisor: Feniosky Peña-Mora
Title: Associate Professor, Department of Civil and Environmental Engineering

Contents

Contents 3
List of Figures 6
List of Tables 7
Philosophies of Software Testing 8
1.1 Introduction and Overview 8
1.2 Background 9
1.3 Software Development Process 10
1.3.1 Software Development Models 10
1.3.2 Capability Maturity Model (CMM) 15
1.4 Objectives of Testing 17
1.5 Principles of Testing 18
1.6 Process of Testing 21
1.6.1 Test Planning 21
1.6.2 Test Case Design and Test Performing 22
1.6.3 Testing vs. Debugging 22
1.7 Summary 23
Concepts of Software Testing 24
2.1 Overview 24
2.2 Testing Categories 26
2.2.1 Static Testing 26
2.2.2 Dynamic Testing 28
2.2.3 Static Testing vs. Dynamic Testing 30
2.2.4 Symbolic Testing 30
2.3 Testing Strategies and Approaches 32
2.3.1 Unit testing 32
2.3.2 Black-Box (Functional) Testing 34
2.3.3 White-Box (Structural) Testing 35
2.3.4 Black-box vs. White-box 37
2.3.5 Integration Testing 37

2.3.6 Top-down Approach 39
2.3.7 Bottom-up Approach 40
2.3.8 System Testing 42
2.3.9 Validation Testing 44
2.4 Object-Oriented Testing 45
2.5 Summary 46
Software Testing Methodologies 48
3.1 Introduction 48
3.2 Testing Techniques 48
3.2.1 Input Space Partitioning and Path Testing 49
3.2.2 Control Flow Graph 51
3.2.3 Domain Testing 52
3.2.4 Program Instrumentation 53
3.2.5 Program Mutation 55
3.2.6 Functional Program Testing 57
3.2.7 Algebraic Program Testing 57
3.2.8 Data Flow Testing 58
3.2.9 Real-Time Testing 59
3.3 Testing and Code Analysis Tools 60
3.3.1 Overview 60
3.3.2 Scope and Implementation of Automated Testing Tools 61
3.3.3 Static Analysis Tools 61
3.3.4 Dynamic Analysis Tools 63
3.3.5 Automated Verification Systems (AVS) 63
3.3.6 Symbolic Evaluators 64
3.3.7 Program Instrumenters 64
3.3.8 Dynamic Assertion Processors 65
3.3.9 Automated Mutation System 65
3.3.10 Test Drivers 66
3.4 Test Data Generators 67
3.4.1 Pathwise Test Data Generators 67

3.4.2 Data Specification Systems 68
3.4.3 Random Test Data Generators 69
3.5 Summary 70
Collocated and Distributed Software Testing 71
4.1 Overview 71
4.2 Team Concepts for Testing 72
4.3 Collocated Development and Testing 74
4.4 Distributed Development and Testing 75
4.5 Case Study: ieCollab 77
4.5.1 Development of ieCollab 77
4.5.2 Testing for ieCollab 80
4.5.3 Details of ieCollab Testing 82
4.5.4 Lessons Learned from Testing ieCollab 83
4.6 Summary 85
Conclusion and Future Development 86
References 88
Bibliography 89
List of Appendices 92
Appendix A – Test Plan (modified) 93
Appendix B – Testing Specifications (modified) 95
Appendix C – Example of Test Log (modified) 110
Appendix D – Test Incident Report (modified) 111
Appendix E – Testing Report (modified) 116

List of Figures

Figure 1.1 Simplified Software Development Model 10
Figure 1.2 The Spiral Model 12
Figure 1.3 The Incremental Model 13
Figure 1.4 The Rapid Application Development Model 14
Figure 1.5 The Prototyping Model 15
Figure 1.6 Trade-off between Effort & Quality and Quality & Cost 20
Figure 1.7 Process of Testing 21
Figure 2.1 Cost as a function of Error Detection Time 25
Figure 2.2 Top-down Integration Testing 39
Figure 2.3 Flow Graph Elements 52
Figure 4.1 Schedule of ieCollab Development 79

List of Tables

Table 1.1 Main Issues of Test Plan Document...21
Table 2.1 Checklist for Common Coding Errors 28

Chapter 1

Philosophies of Software Testing

1.1 Introduction and Overview

Testing has always been treated as an insignificant activity that merely removes the minor flaws of the software. Since software products are quite often over time budget and sometimes cost budget, testing activity is usually left off or provided with a very small amount of time and resources in order to make the schedule. Developers always underestimate the impact of errors on the product because they tend to perceive that once the coding is complete, the product will work and can be released. Unfortunately, error-enriched products are more than disasters, especially when people have great reliance on them.

The purposes of this thesis are to unveil the importance of testing and to provide a detailed description of how testing can improve the software in terms of quality, integrity, and efficiency. This chapter describes the philosophies of testing, including the principles and process of testing. The second and third chapters get into the details of test execution and explain how we can approach and perform testing on software products. The fourth chapter reinforces the concepts in the first three chapters by delineating the case

study of ieCollab, a collaborative software product developed at the Massachusetts Institute of Technology (MIT). It also outlines the key issues of distributed and collocated software development. The final chapter will summarize and conclude the whole thesis.

1.2 Background

Testing has been a part of the software process since the birth of software development. In the early years of programming, software projects were generally small tasks that were developed by a few people. They were accepted as and expected to have flaws. Testing of the product was merely an informal post-coding activity performed by the programmers to see if the program would produce expected results when executed against a small set of *ad hoc* test cases. Little resources were allotted to this activity and it provided little confidence in the reliability of the software [Infotech, 1979].

With a tremendous development of the software industry in the last two decades, software products are now controlling most activities of our daily life, including many life-and-death applications. Imagine what will happen if a hospital's software system that keeps track of all the information of its patients breaks down. Imagine how the stock market will be affected if an on-line stockbroker site is down because of software errors. Imagine what a chaotic scene will occur as a result of a software failure in the registration office of a university. Since software unreliability can no longer be tolerated, traditional testing methods are by no means efficient enough to deal with the more complex and high-volume software.

Typically, the biggest part of software cost is the cost of errors or bugs: the cost of detecting them, correcting them, the cost of designing test cases that discover them, and the cost of running these tests [Beizer, 1990]. About half of the costs of nowadays software projects go to the effort on testing, and the cost of revealing and rectifying errors become larger with each successive step in the development cycle [Deutsch, 1982].

Over the past two decades, a great deal of research has been done to improve the efficiency and effectiveness of software testing. The concept of testing has become a highly rigorous function involving formalized plans and procedures [Deutsch, 1982]. In addition to formalizing and extending existing techniques, a rich variety of completely new approaches to test case design have evolved. These methods not only provide the developer with a more systematic approach to testing, they also provide a mechanism that can help ensure the completeness of tests and provide the highest likelihood for uncovering errors in the software [Pressman, 1997]. Instead of having the programmers perform the testing tasks, a separate testing group has emerged as an important entity of the development process.

1.3 Software Development Process

To understand the principles of testing, a general knowledge of the software development process and different software development models is necessary. Software development process is a very important part of the production because it affects the overall efficiency of the utilization of different resources to produce the software. Software models allow us to describe the process we implement a product from the very beginning until the release of a product.

1.3.1 Software Development Models

Figure 1.1 shows a simplified software development model. The first component is requirement analysis. The analyst specifies the information domain for the software, as

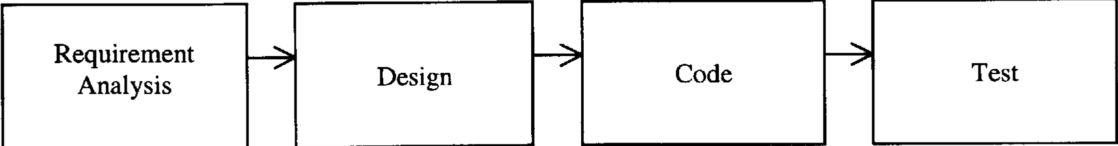


Figure 1.1 Simplified Software Development Model

well as the required function, behavior, performance, and interfacing. Requirements for both the system and the software are documented and reviewed with the customer. Software design follows, which involves a multi-step process that focuses on data structure, software architecture, interface representations, and procedural (algorithmic) detail. The design process translates requirements into a representation of the software that can be assessed by programmers. The third step is coding, which translates the design of the product into machine-readable code. This part is usually the most time consuming process and typically accounts for one-third of the project duration. The final stage is testing, which focuses on the logical internals of the software and assures that all the requirements have been met. The main purposes of testing are to uncover errors and to ensure that defined input will produce output that agrees with the required results [Pressman, 1997].

In real-life software development, however, the model is never as simple and sequential as Figure 1.1, which is also known as the *waterfall model* or *linear sequential model.* There are no clear division lines between different stages, and a product can possibly move to the next phase and then return the an earlier phase. For example, some requirements may not be specified clearly enough for the designers, or the design features are not complete enough for the programmers. These problems can possibly make the sequential model a cycle. Most of the time, different phases are in process concurrently. One particular example is the testing phase. In reality, testing tasks begin as early as the requirements are specified instead of waiting until the coding is complete. Since the scope of this thesis is the concepts of testing, we will focus on the testing phase of the development process.

Other software development models can be applied to different software projects. One of which is called the *spiral model*, as shown in Figure 1.2. This model couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model in Figure 1.1. The software is released incrementally, with earlier prototypes containing mainly the specifications of the product and later prototypes containing more complete versions. In this particular model, test planning begins after prototype 3 is released and

continues until the release of the operational prototype and the ultimate product. Other activities are in progress at the same time. The spiral model shows that all the development activities can be overlapped instead of clearly separated. The advantage of this model is that no personnel will be totally idled during the whole process, which could have resulted in inefficient use of human resources [Pressman, 1997].

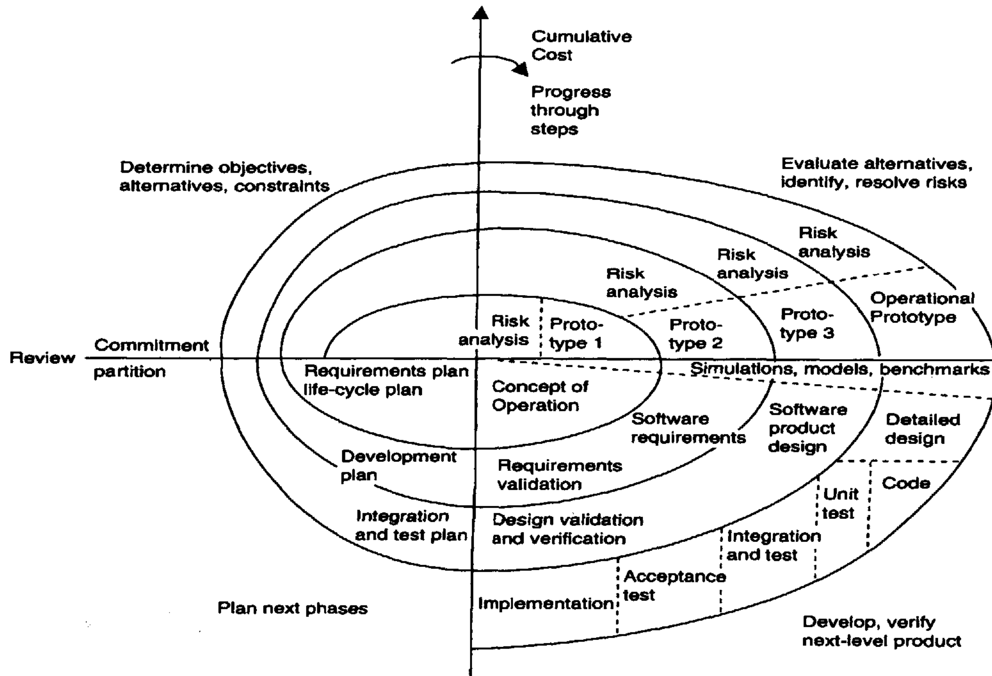


Figure 1.2 The Spiral Model

(adopted from Boehm, 1988)

Besides the spiral model, the *incremental model* is also a popular choice. Figure 1.3 depicts such a model. In this model, the product is broken into a number of increments or versions. During the process, the next increment begins before the current one ends. The main advantage of doing so is similar to that of the spiral model. For example, requirement analysts can begin working on the next increment while the designers are working on the current increment. For testing, test case design begins as soon as requirement specifications of the first increment are finished, and the process of testing continues until the release of the final increment, or the ultimate product. In such case, utilization of human resources is very efficient.

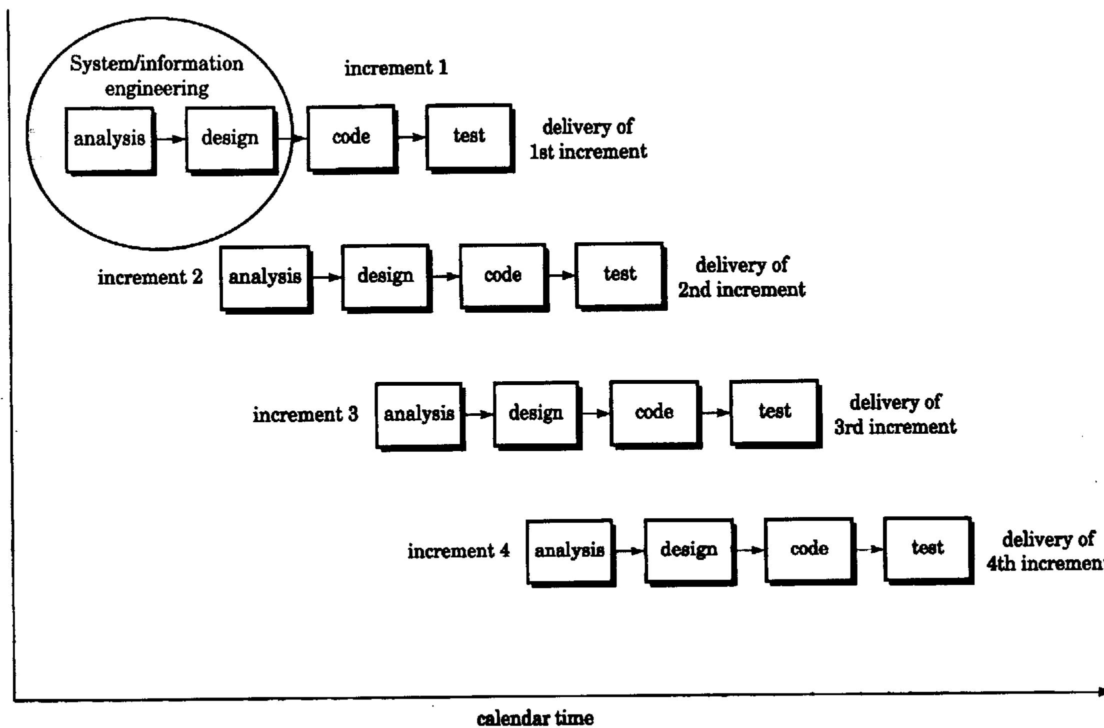


Figure 1.3 The Incremental Model
(adopted from Pressman, 1997)

Figures 1.4 shows another model named the Rapid Application Development (RAD) model [Pressman, 1997]. It is a "high-speed" version of linear sequential development process model (Figure 1.1) that emphasizes an extremely short development cycle. The rapid development is achieved by using a component-based construction approach. The RAD contains the following five phases [Kerr, 1994]:

- Business modeling: The information flow among business function is modeled in a way that answers the following questions: What information drives the business process? What information is generated? Who generates it? Where does the information go? Who processes it?
- Data modeling: The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business. The characteristics of each object are identified and the relationships between these objects are defined.

- Process modeling: the data objects defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.
- Application generation: RAD assumes the use of fourth generation techniques. Rather than creating software using conventional third generation programming languages, the RAD process works to reuse existing program components or create reusable components. In all cases, automated tools are used to facilitate construction of the software.
- Testing and Turnover: Since the RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time. However, new components must be tested and all interfaces must be fully exercised.

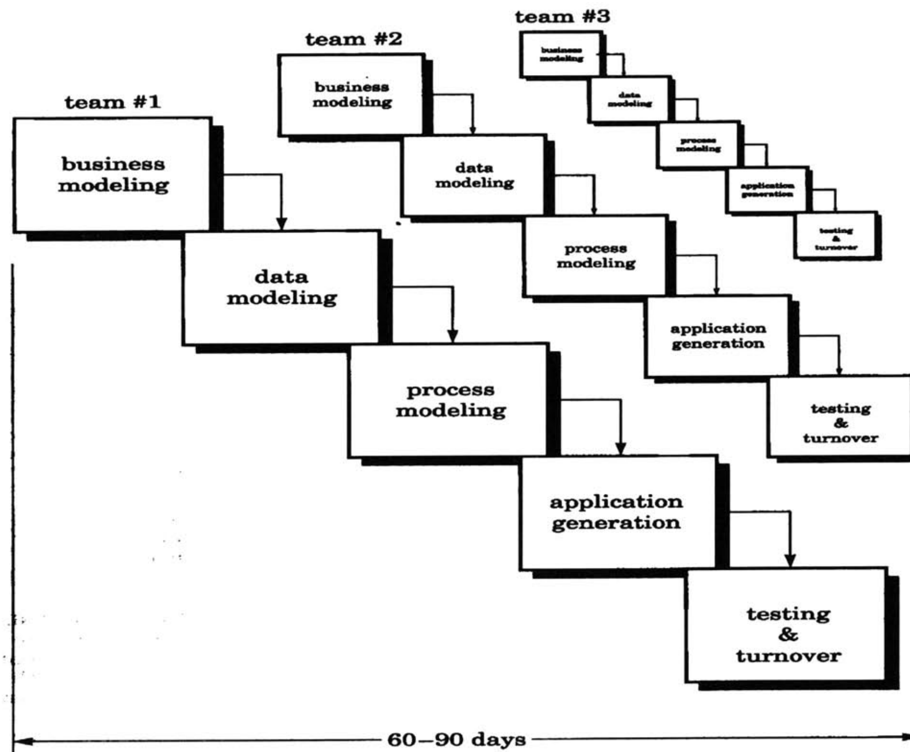


Figure 1.4 The Rapid Application Development Model
(adopted from Pressman, 1997)

The final model to be described is the prototyping model shown in Figure 1.5. This model begins with requirement gathering, during which the developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A quick design occurs and leads to the building of a prototype, which is to be revised by the customer. Iterations occur as the prototype is tuned to satisfy the needs of the customer. At the same time, it enables the developer to better understand what needs to be done. This model serves best when the customer does not identify detailed input, processing, or output requirements [Pressman, 1997].

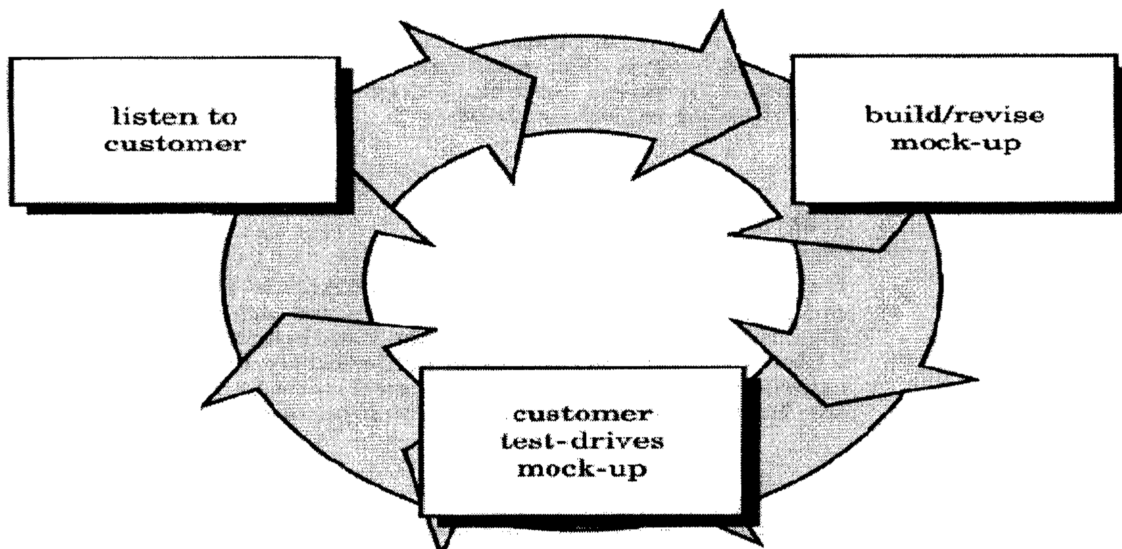


Figure 1.5 The Prototyping Model
(adopted from Pressman, 1997)

1.3.2 Capability Maturity Model (CMM)

Unlike the software models described in Section 1.3.1, a *capability maturity model (CMM)* defines the level of maturity of the software engineering practice as well as the activities carried out at different levels [Pressman, 1997]. The CMM was developed by the Software Engineering Institute (SEI) as a response to the significant emphasis on process maturity in recent years [Paulk, 1993]. Five levels are present for CMM, with each

higher level corresponds to a greater effectiveness in the development organization and practices. The CMM is "predicated on a set of software engineering capabilities that should be present as organizations reach different level of maturity" [Pressman, 1997]. The SEI approach provides a measurement of global effectiveness of a company's software engineering practices and the five levels are defined as follows [Pressman, 1997]:

Level 1: *Initial* – The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort.

Level 2: *Repeatable* – Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

Level 3: *Defined* – The software process for both management and engineering activities is documented, standardized, and integrated into an organization-wide software process. All projects use a documented and approved version of the organization's process for developing and maintaining software. This level includes all characteristics defined for Level 2.

Level 4: *Managed* – Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all characteristics defined for Level 3.

Level 5: *Optimizing* – Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics for Level 4.

Regarding the testing activity on the software development, software teams at Level 1 perform testing as an ad hoc activity, if any. Testing is insignificant and the main concern for the team is to have the product ready. At Level 2, quality assurance becomes a *key process area (KPA)* and more attention is put on the correctness of the product. At

Levels 3 and 4, the attention of testing has become organization-wide and a separate group of developers are dedicated to working on the quality of the product. Testing activities are more organized and are planned beforehand. The product's correctness has a higher insurance. At Level 5, defect prevention becomes a KPA and the complete version of testing can be seen. The testing team is systematically managed with tasks and responsibilities clearly identified [Pressman, 1997]. The remainder of this thesis will explore the details of a complete version of testing.

1.4 Objectives of Testing

As described in previous sections, the objective of testing is to seek errors, not to demonstrate that the program does not have them. It aims at ensuring the integrity of the software. By integrity, we mean that the program 1) provides evidence that it satisfies its mission, requirements, and specifications; 2) includes mechanisms to maintain appropriate levels of performances even under pressures which will occur from operator errors, procedural program flaws, and user initiated errors; and 3) is well documented, functionally simple, modular in nature, relatively short in length, integrated into as rigorously structuring practices and sensible programming standards [Infotech, 1979]. Testing can also be described as an examination of a program by executing it with a set of predesigned test inputs and observing its outputs. The latter are compared with the specifications of the intended behavior to decide whether the program is correct.

Software testing is an element of the broader topic: Verification and Validation (V & V). Many different interpretations of V &V exist, but generally verification refers to the set of activities that ensure the software correctly implements a specific function and validation refers to the set of activities that ensure the software meets the requirements specified by the customer [Pressman, 1997]. According to Boehm [Boehm 1976], verification means the examination whether a program is implemented correctly with respect to its specification, while validation means whether the correct program with respect to the user requirements is implemented.

To sum up, the objective of testing is to find answers to the following questions [Deutsch, 1982]:

1. Do the design and product contain all the required functions and have them implemented correctly?
2. Does the product produce the desired parameter accuracies, respond within the required time frame, and operate within the allocated computational resources?
3. Can the system be produced under cost and budget?
4. Can the product be modified easily or extended to accommodate changing user requirements?
5. Can the system operate efficiently in the user's operating environment?

1.5 Principles of Testing

Software testing is a critical part of the software development process, because it is the ultimate review of specification, design, and coding. It provides the final bastion from which quality can be assessed and errors can be detected. It is the process that decides whether the software has reached a desirable quality to be released. However, testing can never be exhaustive, and it will not improve the quality of the software if quality does not exist beforehand [Pressman, 1997].

The principles of software testing contain the following [Pressman, 1997]:

1. All tests should be traceable to customer requirements. A piece of software is most unacceptable from the customer's point of view if it fails to meet its requirements.
2. Testing should begin "in the small" and progress toward testing "in the large." As will be described in more details in Section 2.3, testing shall always begin with the smallest units of the program and shall move progressively towards the inte-

grated components and finally the whole system. Sometimes errors that occur in the "large" unit are a result of a number of errors in the "small" units.

3. A good test case is one that has a high probability of finding an error. The testers must stay involved in the whole development process so as to gain a thorough understanding of the program. A common misunderstanding is that testers do not have to know the program as much as designers and programmers do. This obviously is false, because without a complete knowledge of how the program works, testers will not be able to design good test cases.
4. A good test is not redundant. Having more than one test case that checks the same input is only a waste of resources. A good test design shall specify one input with an expected output, and only one.
5. Tests should be planned long before testing begins. As mentioned in Section 1.3 and will be described again in Section 1.6.1, test planning begins as early as after the completion of requirements analysis. Detailed design of test cases can begin as soon as the design is solidified.
6. To improve the effectiveness, a third party is designated for testing. By effectiveness, we mean the probability to find new errors by testing.

The final point needs to be elaborated. Every programmer must have experienced testing his own programs. A question arises: why do we need a separate third party to perform testing? The answer is that programs that can be completed by individuals are usually small in size containing no more than a few hundred lines of codes. A program of this size does not require a lot of testing, so it is feasible for the programmer to test it himself. However, the kind of software in interest is large products that require a number of developers to implement. Testing by the programmers becomes inefficient, and sometimes, impossible. The following are the main reasons:

1. The programmers have a tendency to demonstrate that the product is error-free and that it works according to customer requirements. With these thoughts on their minds, they may not be interested in performing a thorough testing [Pressman, 1997].

2. The more you know about the design, the likelier you are to eliminate useless tests, which may have the probability to discover unknown errors. Also, the likelier you will have the same misconceptions as the designers [Beizer, 1990].
3. Tests designed and executed by an independent testing team are bias-free and more objective.

The trade-off between time and effort spent on testing and the quality of the software is obvious. Figure 1.2 describes the relationships between effort and quality as well as quality and cost. If insufficient effort is spent in testing, the quality of the software is low, the reject rate will be high, and the net cost will be huge. Conversely, if so much effort is spent on testing that almost all faults are caught and removed, the quality will be excellent, but inspection costs will dominate and again the net cost will be high [Beizer, 1990]. Although there exists a point where the cost is minimized, deciding accurately how much effort is necessary to reach that point is difficult. Thus, the criterion for testing termination always remains a question. However, a practical way to decide when

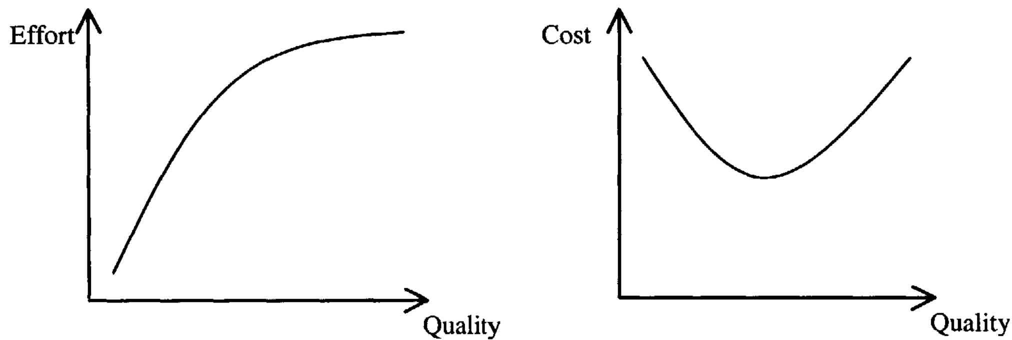


Figure 1.6 Trade-off between Effort & Quality and Quality & Cost

to terminate testing is to rely on statistical models and previous similar projects to approximate how much effort is necessary to reach the probability that a certain percentage of errors are uncovered.

1.6 Process of Testing

The process of testing does not only contain the design and running of test cases. It all begins with test planning which generally describes the estimated schedule and outlines the resources necessary. Upon completion of the testing process, the code is returned to the programmers for debugging. Figure 1.3 displays a road map for the process of testing.

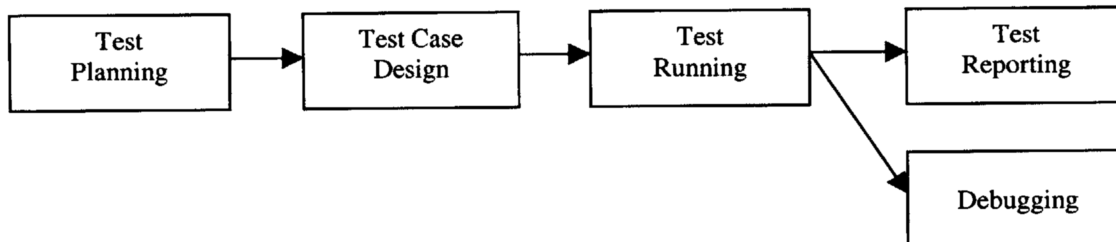


Figure 1.7 Process of Testing

1.6.1 Test Planning

As early as the project is defined, test planning begins. Table 1.1 describes the main issues that are discussed in the test plan, which usually is a formal document prepared by the testing team.

Table 1.1 Main Issues of Test Plan Document

Categories	**Description**
Test Items	The components of the software that can be tested
Features to be tested / not to be tested	Features of the software that are planned to be / not to be tested on
Approach	Types of testing to be performed
Pass / Fail Criteria	Definition of how the software passes / fails the test
Deliverables	The documents that will be generated by the testing team
Tasks	Work to be done by the testing team
Environmental Needs	Hardware and operating environment required for the product
Responsibilities	Individual tasks for the testing team
Staffing and Training	Special training, if necessary, for the testing staff
Schedules	The estimated schedule of the whole testing process

1.6.2 Test Case Design and Test Performing

After the plan is set up, relevant testing tasks such as the design of test cases will follow according to the schedule. A test oracle, which is any program, process, or body of data that specifies the expected outcome of a set of tests as applied to the tested object, may be required. The most common test oracle is an input / outcome oracle that specifies the expected outcome for a specified input [Beizer, 1990]. During the testing of the code, test log that records all the outputs corresponding to the specified inputs and any happenings that occur will be prepared. The whole testing process will be summarized in the final testing report, which marks an end of the testing process.

1.6.3 Testing vs. Debugging

Testing and debugging have always been treated as one single activity, and their roles have always been confusing. In fact, they differ in their goals, methods, and approaches.

1. Testing demonstrates errors and apparent correctness; debugging is a deductive process because no single rule applies to how an error can be found and corrected.
2. Testing can be planned, designed, and scheduled; debugging cannot be so constrained because errors are not foreseeable.
3. Much of testing can be done without design knowledge; debugging is impossible without detailed knowledge about the design.
4. Testing can be performed by an outsider; debugging must be done by an insider.
5. Much of test execution can be automated; debugging can yet to be automated.

The purpose of testing is to show that the program has bugs, and the purpose of debugging is to find the errors or misconceptions that led to the program's failure. It also aims at designing and implementing the program changes that correct the error [Beizer, 1990]. So, debugging can be described as an activity that follows testing.

1.7 Summary

This chapter describes what has to or should be done for testing, such as the procedures, the insights, and the general concepts, while the following two chapters will describe how testing is performed. The principles and objectives of testing outlined in this chapter define the correct attitude of carrying out testing activities. Without such knowledge, designing and executing test cases will be inefficient and sometimes chaotic. Thus, even though this chapter presents only the philosophies of testing, it has direct impact on how good the testing is.

Chapter 2

Concepts of Software Testing

2.1 Overview

The previous chapter outlines the basic ideas of testing, including what has to be completed before actual testing is performed. As stated in Sections 1.3 and 1.6, test planning can begin as early as requirement documents are completed. A number of additional issues, however, must be emphasized before we move into the concepts and methodologies of testing.

In every form of testing, we are comparing deliverables: specification with specification or implementation with specification [Ould and Unwin, 1986]. Since all testing ultimately concerns specifications, testing has a strong reliance on the quality of the documents delivered in earlier software development process. As mentioned in Section 1.5, testing can only be as high quality as the requirement specifications on which it is based. The product objectives, design plans, and code should be inspected and revised repeatedly until they are believed to meet their requirements and specifications [DeMillo et al., 1987]. A good specification will not only enable quality testing, it can also make test planning more straightforward and easier. More importantly, the later an error has been

detected, the more expensive and difficult it is to correct, because more changes may need to be made as a result of a single late modification and correction. Figure 2.1 depicts such relationship.

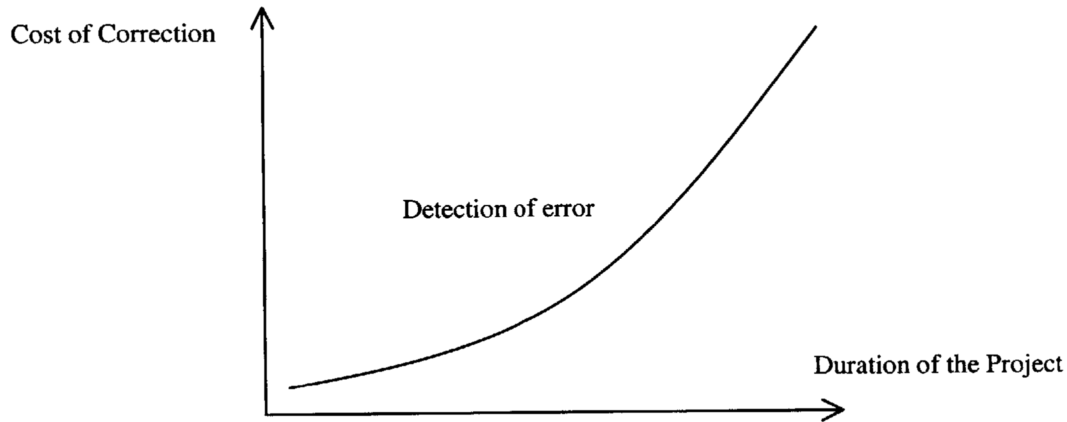


Figure 2.1 Cost as a function of Error Detection Time

Gilb [Gilb, 1995] states that the following issues must be addressed to implement a successful software testing strategy:

1. Specify product requirements in a quantifiable manner long before testing commences. A good testing strategy assesses other quality characteristics such as portability, maintainability, and usability.
2. State testing objectives explicitly and in measurable terms (see Table 1.1).
3. Understand the users of the software and develop a profile for each user category. Use cases that describe the interaction between each class of users can reduce the overall testing effort by focusing testing on actual use of the product.
4. Develop a testing plan that emphasizes "rapid cycle testing." A testing team should "learn to test in rapid cycles (2 percent of project effort) of customer-useful, at least field 'trialable,' increments of functionality and / or quality improvement" [Gilb, 1995].
5. Build "robust" software that is designed to test itself. Software should be capable of diagnosing certain classes of errors and the design should accommodate automated testing.

2.2 Testing Categories

The three main categories for testing are static testing, dynamic testing, and symbolic testing. Static testing inspects the correctness of the program without executing or running it, while dynamic testing actually runs the program to check its correctness. Symbolic testing can be treated as a transition between static and dynamic testing. Instead of using particular values, symbolic testing uses symbols to represent a range of values to test the code.

2.2.1 Static Testing

The term "static" refers to the status of the program. This testing category analyzes the requirement and design documents as well as the code, either manually or automatically, without actually executing the code [DeMillo et al, 1987]. It involves detailed inspection of specifications and code to see if an adequate standard has been reached in the implementation, and specifications are compared with each other to verify the correctness of the program. Described the other way, static testing checks the code by dry running. The testers execute the code mentally with sample test data, noting the values of variables as they are changed and the path through the program [Ould and Unwin, 1986]. The main advantage of such a testing strategy is that when the program is dry run, the actual paths or execution are noted and any deviation from the intended behavior can be discovered.

Static analysis can provide useful information for test planning, reduce the effort spent on dynamic testing (see Section 2.2.2), and increase confidence in test results. It is particularly useful for discovering logical errors and questionable code practices [Fairley, 1978]. However, static testing has a major practical limitation on programs containing array references and pointer variables. Array subscripts and pointers are mechanisms for selecting a data item at run time, but static analysis cannot evaluate subscripts or pointers. As a result, it is unable to distinguish between elements of an array or members of a list.

Static analysis is only accurate when individual program objects such as tasks and entries can be identified statically. Besides, since the analysis conducted is independent of the target environment, the implications of delay statements, non-zero execution times, and scheduler algorithms cannot be taken into account [Hausen, 1983].

Static analysis consists of four categories: requirement analysis, design analysis, code inspections and walkthroughs, and experimental evaluation.

Requirement analysis corresponds to analyzing the requirements and the needs of the customers. They are analyzed by using a checklist of correctness conditions, including properties such as consistency, their necessity to achieve the goals of the system, and the feasibility of their implementation with existing resources [Fuji, 1977].

Design Analysis refers to inspecting the elements of a software system design, such as the algorithms, the data flow diagrams, and the module interfaces. Similar to requirement analysis, they can be analyzed by creating a checklist. Each property specified in the checklist may be checked by a different method such as modeling, which can be used to determine if the design agrees with the performance requirements [Howden, 1981].

Code inspection and walkthroughs involve the visual inspection of a program by different individuals and then as a group. The main objective is to detect deviations from specifications and errors [DeMillo et al, 1987].

During code inspection, the programmers narrate the logic of the program statement by statement, and the program is analyzed with respect to a checklist for common programming errors similar to the one shown in Table 2.1. A walkthrough is similar to code inspection, except that they have different procedures and error detection techniques. Walkthroughs are conducted at a group meeting, at which the participants walk through a small set of inputs. Both code inspection and walkthrough can be analyzed with the assistance of static analyzer which analyzes the control and data flow of the program and

records in a database such problems as uninitialized variables, inconsistent interfaces among modules, and statements that can never be executed [DeMillo et al, 1987].

Table 2.1 Checklist for Common Coding Errors

(adopted from Ould and Unwin, 1986)

Poor or non-standard errors
Omission in coding of logic element
Wrong use of subscripts or indices
Improper address modification
Reference to undefined symbols
Multiply defined symbols or data names
Transcription of the characters in a name
Wrong initialization of fields, counters, and flags
Incorrect or missing saving and restoration of registers
Use of absolute addresses
Incorrect format of statements
Non-cleared buffers print lines, registers
Incorrect calling of subroutines
Incorrect setting of parameters or switches
Creation of endless loops
Overwriting of constants
Failure to allow for overflow
Wrong branches of logic
Poor organization of *if* or *case* constructions
Improper termination of loops
Wrong number of loop cycles
Incorrect loop initialization
Failure to open or close files
Failure to manage print line correctly
Poor cohesion of the logic
Poor coupling of the logic modules
Incorrect linkage to the operating system
Wrongly set variables
Use of alias or re-use of variables
Poor comments; not enough why or when

Experimental evaluation of code inspections and walkthroughs shows that static analysis techniques are very effective in finding from 30% to 70% of the logic design and coding errors in a typical program [Myers, 1978]. Walkthroughs and code inspections alone can detect an average of 38% of the total errors in the programs studied [Myers, 1979].

2.2.2 Dynamic Testing

In contrast to static analysis, dynamic testing actually executes the program to search for errors, and it serves to detect errors that cannot be found by static testing. These errors,

called dynamic errors, are usually those that can only be revealed when the software is executed [Infotech, 1979]. The general objective of dynamic testing is to verify the internal behavior of the program by running it with certain sample inputs and compare the outputs with the expected results.

Dynamic testing involves running the test cases that are designed prior to the execution of the program. One major difficulty for this test approach is the determination of the adequacy of test cases, which can be samples from the space of all possible execution sequences. Testing cannot guarantee correct operation of software under all operating conditions. However, a well-designed test plan that specifies the critical test cases and their expected outcomes can provide a significant level of confidence in the software [Fairley, 1978].

The three different dynamic techniques are path testing, branch testing, and functional testing [Howden, 1976]:

1. *Path testing* requires that every logical path through a program be tested at least once. However, programs that contain loops may have an infinite number of paths. A general approach to solving this problem is to group all paths through a program into a finite set of classes and then to require the testing of one path from each class.
2. *Branch testing* is a weaker technique than path testing because it does not cover all the paths of the program. Rather, it only requires that each branch of the program to be traversed at least once.
3. *Functional testing* involves testing of a system over each of the different possible classes of input and the testing of each function implemented by the system.

Further details, including strategies and approaches for dynamic testing, will be discussed in Section 2.3. Different testing techniques will be addressed in Chapter 3.

2.2.3 Static Testing vs. Dynamic Testing

Static testing and dynamic testing contrast in many aspects. The former is particularly useful in examining concurrent software development, because it does not require the execution of the code. A suitably constructed analyzer is able to investigate all possible sequences of events, under all circumstances. The test is independent of changing real-time conditions and the actions of an unpredictable scheduler. However, it makes several assumptions and often has several limitations that lessen its utility. For example, static analysis cannot even begin to address functional correctness, as it works on the shallower structural properties. Dynamic testing cannot be applied for concurrent software, because of its reliance on the execution of the code. However, it makes up for some of the deficiencies of static testing [Hausen, 1983].

In reality, static testing and dynamic testing are complementary approaches. The former concentrates on program structures, while the latter can be used to investigate run-time behavior of the program and its functionality. Static analysis provides definitive and informative results, but has limited applicability and is expensive. Dynamic analysis makes fewer assumptions and has fewer limitations, but its assurances are not as strong in that the meaning of the results obtained is not very clear unless an error is discovered. To provide a better scope of testing, these two approaches are usually employed jointly to eliminate some of the deficiencies. As mentioned in Section 2.2.1, static testing is performed before the execution of the code. Consequently, dynamic testing, which requires the execution of the program, can be used to further investigate results from static analysis. In such a case, static analysis results to help reduce the overhead incurred by the dynamic techniques.

2.2.4 Symbolic Testing

Symbolic testing has been the foundation for much of the current research on software testing [Hausen, 1983]. As its name suggests, symbolic testing utilizes symbolic expres-

sion to perform testing. Input data and program variable values are assigned symbolic names, which may be elementary symbolic values or expressions corresponding to a class of actual data values. For example, a symbolic input x may represent an actual data value of positive numbers smaller than 200. When testing symbolically, this particular input value is equivalent to a large number of tests with actual data. To run a symbolic test, a set of input values are executed, and the values of all variables are maintained as algebraic expressions in terms of the symbolic names. The output of the program is then examined by some external mechanism to determine the correctness of the program behavior. The execution is usually performed by a system called a *symbolic evaluator* whose major components are a *symbolic interpreter* and an *expression simplifier*. Since the input values are symbolic, the generated output values may consist of symbolic formula or symbolic predicates [DeMillo et al, 1987].

The advantage of using symbolic values is that it describes the relationship between the input data and the resulting values. Normal execution computes numeric values and loses information about the way these numeric values are derived, but symbolic analysis preserves this information. Moreover, it is more demanding than conventional testing in that it requires a symbolic interpreter and someone to certify that the symbolic results are correct. It provides much more confidence on the program's correctness, because one single symbolic test of a program is equivalent to an unbounded number of normal test runs [Darringer and King, 1978].

The main drawback for symbolic testing is that the symbolic representations can be too long and complex to be meaningful. Even if it is short, the tester may not detect an error in a representation. Other than this problem, a symbolic execution must be applied to completely specified program paths including the number of iterations for each loop. A large program may have an infinite number of program paths and symbolic testing may become unmanageable [Clarke, 1976]. In spite of these deficiencies, this type of testing is more powerful than static and dynamic testing in the evaluation of arithmetic expressions and mathematical formula.

2.3 Testing Strategies and Approaches

Section 1.5 mentions that testing should always begin with the smallest unit and progress towards the big units, because sometimes errors in the small units are the causes of those in the big units. Basically, we can divide testing into three levels. The lowest level, *unit testing*, concentrates on the modules of the program. The middle level, known as *integration testing*, checks whether these modules are compatible. The highest level, specified as *system testing*, looks at the program as a whole and acts as the final verification of the program. Testing starts with unit testing, and then is carried over to integration testing and finally to system testing.

2.3.1 Unit testing

The smallest units of a program are discrete, well-defined, and small components known as modules. After the programmers have removed errors that can be detected by the compiler, the modules are ready to undergo unit testing, which aims at uncovering errors within the boundary of the module. The principle is to ensure that each case handled by the module has been tested, so unit testing should be as exhaustive as possible. The advantage of such testing is that independent test cases have the virtue of easy repeatability. The smaller the component, the easier it is to understand, and the more efficient it is to test. If an error is found, only a small test, instead of a large component that consists of a sequence of hundreds of interdependent tests, needs be rerun to confirm that a test design bug has been fixed. Similarly, if the test has a bug, only the test needs be changed, not the whole test plan [Beizer, 1990].

During unit testing, each module is isolated from another in an artificial environment known as a "test harness," which consists of the driver and / or stub software. The reason of using such software is because a module is not a standalone program, thus cannot be run independently. In most applications a driver is nothing more than a "main program" that accepts test case data, passes such data to the module to be tested, and generates

relevant outputs. Stubs or "dummy programs" serve to replace other modules that are subordinate to the module in concern. They use the subordinate modules' interfaces, do minimal data manipulations, print verification of entries, and return [Pressman, 1997].

Pressman [Pressman, 1997] described local data structure for a module as a common source of errors. So, test cases should be designed to uncover errors in:

1. Improper or inconsistent typing
2. Erroneous initialization or default values
3. Incorrect (misspelled or truncated variable names)
4. Inconsistent data types
5. Underflow, overflow, and addressing exceptions

Other than local data structure, global data can also have a great impact on the integrity of the module. Test cases should also concentrate on uncovering errors due to erroneous computations, incorrect comparisons, or improper control flow. Common computation errors include:

1. Misunderstood or incorrect arithmetic precedence
2. Mixed mode operations
3. Incorrect initialization
4. Precision inaccuracy
5. Incorrect symbolic representation of an expression
6. Comparison of different data types
7. Incorrect logical operators or precedence
8. Expectation of equality when precision error makes equality unlikely
9. Incorrect comparison of variables
10. Improper or nonexistent loop termination
11. Failure to exit when divergent iteration is encountered
12. Improperly modified loop variables

The last task of unit testing step is boundary testing, because software quite often fails at its boundaries. Some commonly made mistakes include processing nth element of a n-dimensional array (n-dimensional array indexes from 0 to n-1), encountering wrong maximum or minimum value, and iterating the loop with an incorrect number of times (misusing "repeat when i < n" instead of "repeat when i <= n"). Boundary testing should be able to uncover these errors.

Unit testing produces a number of side effects, though. The obvious one is the overhead represented by drivers and stubs, which are software that must be developed for each unit test and will not be delivered as part of the final software package. If modules to be tested are complicated, drivers and stubs cannot be kept simple and the overhead costs will be high, making unit testing inefficient. Another disadvantage of unit testing is that every component has interfaces with other components, and usually all interfaces are sources of confusion. Furthermore, development of such drivers and stubs may introduce more bugs and errors, thus expanding the testing effort [Pressman, 1997].

Unit testing can be based on two different aspects. If the objective is to check whether the requirements for an object have been met, it is called *black-box* testing. If the testing is based on the inner details, it is called *white-box* testing.

2.3.2 Black-Box (Functional) Testing

Black-box testing is also known as functional testing. The testing object is viewed as a black box because inputs are pushed into it and its outputs are checked against the expected results that are constructed from the requirement specifications [Ould and Unwin, 1986]. The only concern for black box testing is the functionality or features, no matter how the implementation details and internal structures are. It only focuses on the software interface and the integrity of external information. Instead of just detecting errors, black-box testing can also be conducted to demonstrate that each function is operational and that input is properly accepted and output is correctly produced. Usually, black-box

testing is applied during later stages of testing when most functionality and features are implemented [Pressman, 1997].

To minimize the number of test cases, the input space of the program is partitioned into a number of equivalence classes with respect to the input specifications so that each equivalence class is actually a case covered by an input specification [DeMillo et al, 1987]. Generally, black-box testing attempts to find errors in the following categories [Pressman, 1997]:

1. Incorrect or missing functions
2. Interface errors
3. Errors in data structures or external database access
4. Performance errors
5. Initialization and termination errors

Random testing is one of the black-box testing strategies. The test is performed by randomly selecting some subset of all possible input [DeMillo et al, 1987]. Such testing strategy does not provide a strong evidence of the program's correctness, but it can be used for final testing of a program by selecting data by statistical means, such as the expected run-time distribution of inputs.

The major drawbacks for black-box testing are its strong reliance of the completeness of the requirement specifications (which usually is not the case) and the necessity of using every possible input as test case to ensure the correctness of the functions [Infotech, 1979].

2.3.3 White-Box (Structural) Testing

Also known as glass-box testing and structural testing, white-box testing concentrates on the contents of the box – the object, to see whether they fit together. In contrast to black-

box testing, which does not look at the implementation details, white-box testing focuses on programming style, control method, source language, database design, and coding details [Beizer, 1990]. The program structure and procedural structure are examined and logical paths through the software are tested by providing test cases that exercise specific sets of conditions and / or loops. The basic criterion for this testing approach is to ensure that every single line of code is executed at least once. However, this is far from sufficient because errors may go undetected [Meyers, 1979]. In addition, exhaustive testing presents logistical problems. For even small programs, the number of independent logical paths can be astronomical numbers. For example, consider a program of 100 lines written in C language containing two nested loops that execute from 1 to 20 times each, depending on the conditions specified at input. Inside the interior loop, four if-then-else constructs are required. In this case, approximately 10^{14} possible paths may be executed! Even so, white-box testing is still effective. The main point is to select a limited number of important logical paths so that important data structures can be validated [Pressman, 1997].

Similar to black-box testing, "white-box testing requires partitioning the input space of a program into path domains and constructing test cases by picking some test data from each of these path domains to exercise every path in a program at least once" [DeMillo et al, 1987]. Unfortunately, this technique is also not guaranteed to detect every possible error even if every path is exercised at least once.

An alternative, which provides a stronger criterion than statement coverage, is *branch coverage*, which requires every possible outcome of all decisions to be exercised at least once. Statement coverage is included in this criterion because every statement is executed if every branch in a program is exercised at least once. However, even branch statement is not perfect when the program contains decision statements. In such case, the program may contain multiple entry points to the branch, and some statements may only be executed if the program enters at a particular entry point [Meyers, 1979].

Yet another testing criterion, domain testing, can be used. This is a modified form of path testing. It attempts to reveal errors in a path domain by picking test data on and slightly off a given closed border [DeMillo et al, 1987].

2.3.4 Black-box vs. White-box

Black-box and white-box testing are usually used in conjunction. The former has the advantage of testing special test cases that may have been overlooked or incorrectly implemented, while the latter has the advantages of uncovering special cases that are not explicitly specified in the requirement specifications and concentrating on implementation problem areas [Clarke, 1976].

2.3.5 Integration Testing

By integration, we mean the process by which smaller components are aggregated to create larger components, also known as subsystems. Integration testing takes the modules that have passed the unit test as inputs, groups them into larger aggregates, and then demonstrates that even though they perform well individually, they are incorrect or inconsistent when combined [Beizer, 1990]. The objective of integration testing is to verify the functional, performance, and reliability requirements placed on the aggregates. Test cases are designed to test that components within an aggregate interact correctly [Ould and Unwin, 1986]. Examples of inconsistencies include improper call or return sequences, inconsistent data validation criteria, and inconsistent handling of data objects. For example, suppose we have two separate functions A and B. Function A calculates the sum s and the product p of two input values, namely x and y. Function B takes the sum s and calculates its square. If the product p is misused as an input to function B instead of the sum s, even though the two functions operate perfectly individually, the combined result is incorrect.

Between unit testing and integration testing, an interface exists. The unit of transfer across that interface may consist of a tested module, a tested component, or a tested sub-system [Ould and Unwin, 1986]. Usually, interface is the point where errors occur. For example, data can be lost across an interface; one module can have an inadvertent, adverse affect on another; sub-functions, when combined, may not produce the desired major function; individually acceptable imprecision may be magnified to unacceptable levels [Pressman, 1997].

Black-box testing techniques are the most prevalent for integration testing, but a limited amount of white-box testing techniques may be used to ensure coverage of major control paths. Every time a new module is added as part of the integration testing, new data flow paths are established, new I / O may occur, and new control logic is invoked. These changes may create problems for those functions that work flawlessly beforehand. To ensure that these changes have not propagated undesired side effects, *regression testing* is performed, which is the re-execution of some subset of tests that have already been conducted. This testing may be conducted manually, and since the number of regression tests can grow quite large as integration proceeds, the regression tests suite should be designed to include only those tests that address one more classes or errors in each of the major program function [Pressman, 1997].

After successful integrations, the product will undergo high-order tests, such as system and validation testing [Pressman, 1997]. These two testing approaches will be discussed in Sections 2.3.8 and 2.3.9 respectively.

Integration testing can be non-incremental or incremental. The former combines all the modules in advance and then test the entire program as a whole. This approach is chaotic, and a set of errors is encountered. Correction becomes more difficult because isolation of causes is complicated by the vast expanse of the entire program. Even if these errors are corrected, new ones may appear and the process becomes an infinite loop [Pressman, 1997]. As a result, incremental integration testing that constructs and tests the program in small segments progressively is suggested. Errors are easier to isolate and

correct, interfaces are more likely to be tested completely, and a systematic approach may be applied. Two commonly know strategies are *top-down* and *bottom-up* approaches, which will be discussed in the following two sections respectively.

2.3.6 Top-down Approach

A top-down testing strategy starts with top modules and moves downward progressively through the control hierarchy. The top-level modules are tested first, and they become the test harness for the immediate subordinate routines. Dummy modules called stub modules are constructed to simulate the functionality of the modules subordinate to the one being tested. A good practice of using top-down approach is to search for critical modules first, which may be modules that are suspected to be error prone. These modules should be added to the sequence as early as possible, and input-output modules should also be added to the sequence as early as possible [Meyers, 1979].

Modules subordinate to the main control module can be incorporated into the structure in either a *depth-first* or *breadth-first* manner. Figure 2.2 briefly depicts the concepts of integration testing. Each box represents a single module. Depth-first integration would integrate all modules on a major control path of the structure, which may be selected

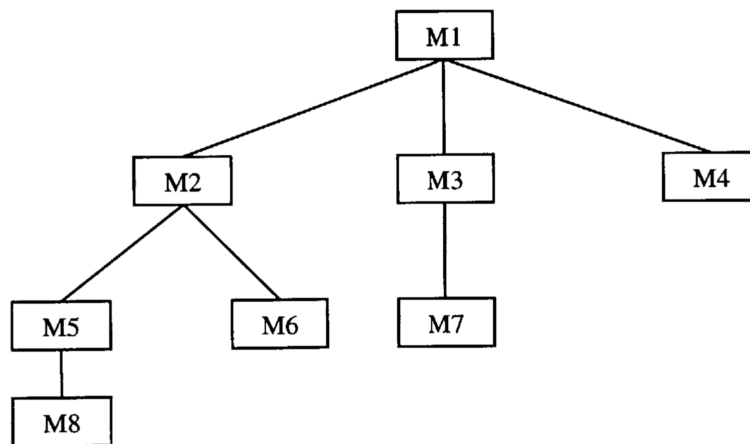


Figure 2.2 Top-down Integration Testing
(adopted from Pressman, 1997)

arbitrarily and depending on application-specific characteristics. For example, modules M1, M2, M5 would be integrated first, and then M8 or M6 would be integrated next. Then, the central and the right hand control paths are built using the same approach. Breadth-first integration incorporates all modules directly subordinate at each level, moving across the structure horizontally. For the structure in Figure 2.2, modules M2, M3, and M4 will be integrated first, followed by those on the next level, namely M5, M6, and M7. The process continues until the lowest level is tested, in this case, the level containing M8 [Pressman, 1997].

The main advantage of top-down approach is that it allows the tester to see the preliminary version of the system, which can serve as evidence that the overall design of the program is correct [Infotech, 1979]. However, the major disadvantages are that using top levels of the system as test harness may be expensive and that stub modules that represent overhead costs are required. In practice, the representation of test data in the stubs may be difficult until input-output modules are added, and some stub modules may be difficult to create [DeMillo et al, 1987]. Moreover, to adequately test upper levels, processing at lower levels in the hierarchy may be required. When stubs are used to replace lower level modules, no upward dataflow can occur. To solve this problem, either delay some tests until lower level stubs are replaced with actual modules or develop stubs that have limited functionality to simulate actual modules. However, delaying the test can lose some control over correspondence between specific tests and incorporation of specific modules, while developing complicated stubs may contain significant overhead, making the test inefficient [Pressman, 1997]. The best solution is to use another approach, the bottom-up integration.

2.3.7 Bottom-up Approach

In direct contrast to top-down integration, bottom-up approach starts with modules at the lowest level of the hierarchy and progresses upwards. Since modules subordinate to a given level is always available in this approach, stub modules are not required. However,

driver modules are needed to simulate the function of a module super-ordinate to the one being tested.

Pressman [Pressman, 97] outlines the bottom-up strategy as an implementation of the following steps:

1. Low-level modules are combined into clusters or builds that perform a specific software sub-function.
2. A driver is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

The advantage of bottom-up testing is that the construction of test data is no longer difficult because driver modules can simulate all the calling parameters even if the data flow is not organized into a directed acyclic graph. The major disadvantages of bottom-up integration are that a preliminary version of the system does not exist until the last module is tested, and design and testing of a system cannot overlap because testing cannot begin until the lowest level modules are designed [Meyers, 1979]. In addition, it requires a test harness for each module and subsystem that can be as much as half of the code written [Fairley, 1978].

The best approach to perform integration testing is to combine the advantages of both top-down and bottom-up testing strategies, sometimes known as *sandwich testing*. For example, if the top two levels of the program structure are integrated top-down, they can be used as the test harness for the testing of the lower level modules, reducing the number of test drivers needed [Pressman, 1997]. In other cases, it may not be possible to use stub modules to simulate modules subordinate to those being tested. If a certain critical low-level modules are tested first, stub modules are no longer necessary to replace them [Fairley, 1978].

2.3.8 System Testing

A system is the biggest component of the whole program. After all the aggregates have been integrated, the final batch is integrated and incorporated with other system elements such as hardware subsystems. System testing is then carried out with the objective of revealing bugs due to inter-aggregate inconsistency when exercised by the complete system. Same as integration testing, system testing is also a form of black-box testing.

System testing is virtually a series of different tests whose purpose is to fully exercise the computer-based system. Each test has a different purpose and scope, but the main idea is to verify that all system elements have been properly integrated and perform allocated functions [Pressman, 1997]. Test data are usually exaggerated and are unlikely to occur in practice. However, they serve to ensure the system's performance even under such extreme conditions. Some of these tests that will be discussed in this section include limit testing, volume testing, recovery testing, security testing, stress testing, performance testing, compatibility testing, storage testing, and installation testing [Ould and Unwin, 1986].

Limit testing is developed to investigate how the system reacts to data that is maximal or minimal. The system should also be tested beyond the limits specified for it so as to discover any situations where insufficient safety margins have been built in.

Volume testing submits the system to large volumes of data, such as a large source file with a large number of identifiers for a compiler and a large edit for an editor. This test serves to detect errors in function when an unexpectedly large job is being processed. As with other types of system testing, volume testing pushes the system beyond the levels specified to observe if any inadequate safety margins exists.

Recovery testing is a vital area of testing for safety critical systems, which must recover from faults and resume processing within a pre-specified amount of time. In some cases, the system has to be fault tolerant, that is, processing faults must not cause overall system

function to cease [Pressman, 1997]. Recovery testing forces the system to fail in a variety of ways. The system's reactions to all sorts of failures should be tested under situations that simulate the types of errors the system is designed to handle. Failure conditions should be determined before testing and the system's responses to them are recorded. If recovery is automatic, re-initialization, data recovery, and restart are each evaluated for correctness. If recovery requires human intervention, the mean time to repair is evaluated to determine whether it is within acceptable limits [Pressman, 1997].

Security testing should be performed on any system that manages sensitive information or causes actions that can improperly harm or benefit individuals is a target for improper or illegal penetration by hackers. The purpose of this test is to verify that the protection mechanisms built into a system have the ability to protect the system from improper penetration. During the test, the testers act as unauthorized individuals who attempt to break into the system using different techniques, such as acquiring passwords through external clerical means, attacking the system with custom software designed to break down any defenses that have been constructed, overwhelming the system to deny service to others, and causing system errors purposefully [Pressman, 1997].

Stress testing involves running the system under heavy loading. This differs from volume testing in that it does not have to reach the prescribed limits of the system. Instead, it executes the program with abnormal conditions that are above average but below the limits. For example, a database system may be stressed with large numbers of queries and updates that may be two times the expected numbers on average. Stress tests are expected to find errors that only show themselves under certain combinations of events or data values. For instance, a database system may fail unexpectedly when certain queries and updates occur simultaneously.

Performance testing probes the system's performance against prescribed timings. For most real-time systems, even if the software meets all the functional requirements, it is considered unacceptable if it fails to conform to performance requirements. Performance

testing is usually coupled with stress testing to detect the system's response under extreme conditions.

Compatibility testing is especially useful when a new system is devised. This test aims at detecting compatibility errors for the new system such as failure to subsume the facilities and modes of use of the old system where it is intended to do so.

Storage testing tests whether the product exceeds the specified storage limits, such as a maximum amount of main or backing storage occupancy. Systems may fail such test when processed or supplied with large amounts of data. The testing technique is similar to that for volume testing.

Installation testing concerns about whether the software can be installed on a variety of machines or models, or allows a number of installation options such as different peripheral devices. At least the most commonly expected options are tested thoroughly, if developing tests for all possible combinations of options is impossible.

2.3.9 Validation Testing

Validation testing provides the final assurance that the product meets all functional, behavioral, and performance requirements. The simplest definition for this testing strategy is that validation succeeds when the software functions in a manner that can be reasonably expected by the customer. Black-box techniques are commonly used to demonstrate conformity with requirements [Pressman, 1997].

Acceptance tests are a series of tests that check whether the software enable the customer to validate all requirements and accept the product. Usually, acceptance testing are conducted over a period of weeks or months to uncover cumulative errors that might degrade the system over time. Two types of such test are *alpha testing* and *beta testing*. The former is conducted at the developer's site in a controlled environment. The software is

used in a natural setting with the developers recording errors and usage problems. Beta testing is conducted by the end user(s) of the product, so the environment cannot be controlled. A beta version of the product is like a "live" application. The customers are responsible for recording all problems that are encountered during execution of the product and reporting these problems to the developers. The software developers will then make final modifications and release the product to the entire customer base [Pressman, 1997].

2.4 Object-Oriented Testing

Object-oriented concepts have changed the way we code. Since this new generation of programming language has become more common than conventional procedural programming, specific testing concepts need to be applied to such programs.

When testing object-oriented programs, some concepts outlined above have to be modified. First, the definition of unit is no longer the individual module. Rather, the smallest testable unit is now the encapsulated class or object. A class may contain different operations, and a particular operation may exist as a part of a number of different classes by means of inheritance. Therefore, the meaning of unit testing is not exactly the same as the conventional one [Pressman, 1997].

In object-oriented testing, *class testing* is equivalent to unit testing for conventional software. This test is driven by the operations encapsulated by the class and the state behavior of the class [Pressman, 1997].

Integration testing strategies are different also. Since we do not have a hierarchical control structure, conventional top-down and bottom-up approaches no longer apply. In object-oriented programming, unlike procedural programming, units cannot be integrated one at a time to form an aggregate, because direct and indirect interactions of the components that make up the class cannot be separated in the same manner as we do for con-

ventional programs. Two different strategies, *thread-based testing* and *use-based testing*, are common for integration testing [Pressman, 1997].

Thread-based testing integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested separately. Same as conventional methods, regression testing is applied to ensure no side effects occur as a result of the integration.

Use-based testing begins the construction of the system by testing the independent classes that use very few, if any, of server classes. Then, dependent classes are tested, using independent classes as test drivers. This sequence of testing layer continues until the whole system is constructed. Notice that test drivers and stubs and other replacement operations are to be avoided whenever possible [Pressman, 1997].

System testing for object-oriented programs is more or less the same. Recall Section 2.3.8, this kind of testing checks whether or not bugs will occur due to inter-aggregate inconsistency when the final batch of the software portion is integrated with the system hardware. Since object-oriented programs do not require a change in the system hardware, the same types of tests are applicable.

Basically, white-box and black-box testing strategies are applicable to object-oriented programs as well. However, the former is suggested to apply on tests at a class level because of the complexity of the program structure [Pressman, 1997]. As for procedural testing, black-box testing constitutes the major part of testing approaches for object-oriented software products.

2.5 Summary

The concepts and approaches of testing outlined in this chapter provide valuable insight for test case design. Different types of tests are performed based on the nature of the

project or the product. However, all of the categories, from unit testing to integration testing to system testing, are recommended so as to provide a thorough coverage of the test. While these approaches enable testers to prepare for testing, actual techniques are required to execute the tests. The following chapter will discuss the techniques and tools available to perform testing.

Chapter 3

Software Testing Methodologies

3.1 Introduction

The most difficult and important part of testing is the execution of test cases, because this is the centerpiece of testing. The previous chapter describes different testing concepts to approach the program, but to achieve the testing objective of uncovering errors, efficient and effective testing techniques are the key. This chapter will explore the methodologies of testing as well as different testing tools that can enhance the quality and efficiency of test execution.

3.2 Testing Techniques

Testing techniques can be described as the way and method we may use to implement the testing strategies discussed in the previous chapter. Many different techniques are available, including program instrumentation, program mutation, input space partitioning, path testing, branch testing, loop testing, domain testing, functional program testing, algebraic program testing, data flow testing, and real-time testing.

3.2.1 Input Space Partitioning and Path Testing

Input space partitioning is a technique that divides the program's input space into path domains such that all inputs within a domain path are equivalent in the sense that any input represents all inputs in that domain path. That means, if the selected input is shown to be correct by a test, then processing is presumed correct, and all inputs within that domain are expected to be correct [Beizer, 1990]. Test cases are derived based on an evaluation of equivalence classes for an input condition. Partitioning aims at defining a test case that uncovers classes or errors, thereby reduces the number of test cases that must be developed [Pressman, 1997].

Path testing is a structural technique most applicable to new software for unit testing. It involves the selection of test data to execute chosen paths and requires complete knowledge of the program's source code. A path in a program consists of some possible flows of control and is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or the same, junction, decision, or exit. Paths consist of links, which are the smallest entity of segments. An independent path is any path through the program that introduces at least one new set of processing statements or a new condition [Pressman, 1997]. The test case construction refers to choosing one test point from each path domain so that each path through a program is executed at least once. Path testing is meant to detect computation, path selection, and missing path errors. A computation error occurs when a computation statement along a path is computed incorrectly. Path selection errors occur when the branching predicates are incorrect. Missing path errors are those in which the required branch predicate does not exist in a program [Howden, 1976]. In addition, path testing can detect infinite loops so long as the program can accept an input that will cause the infinite loop to occur [Infotech, 1979]. If the set of paths is properly chosen, then the testing can be described as thorough and complete.

Since a program usually has a huge number of paths, a practical path testing technique has to use a procedure to select a subset of the total set of paths [Howden, 1976]. *Branch testing*, a weaker technique than path testing, that executes tests to assure that every

branch alternative has been exercised at least once can be an alternative for path testing if the number of paths is too large [Beizer, 1990]. *Conditioning testing* can be another alternative. It focuses on testing each condition in the program. The advantages are that measurement of test coverage of a condition is simple, and that the test coverage of conditions in a program provides guidance for the generation of additional tests for the program [Pressman, 1997].

A program that has loops will generally have an infinite number of paths, especially when it contains nested loops. Unfortunately, loops are the cornerstone for the vast majority of all algorithms implemented in software, so more effort should be put on them. *Loop testing* is a special path testing technique that exclusively deals with loops. Beizer [Beizer, 1990] outlines approaches to deal with simple nested loops:

For simple loops:

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where $m < n$.
5. n-1, n, n+1 passes through the loop.

For nested loops:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding outer loops at their minimum iteration parameter values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to typical values.
4. Continue until all loops have been tested.

As other testing techniques do, path testing has a number of limitations [Beizer, 1990]:

1. Path testing may not reveal totally wrong or missing functions.
2. Interface errors, particularly at the interface with other routines, may not be caught by unit-level path testing.
3. Database and data-flow errors may not be caught.
4. The routine can pass all of its tests at the unit level, but the possibility that it interferes with or perturbs other routines cannot be determined by unit-level path tests.
5. Not all initialization errors are caught by path testing.
6. Specification errors can't be caught.

Partitioning testing is also applicable to object-oriented testing mentioned in Section 2.4. Same as conventional testing, partitioning can reduce the number of test cases required to exercise the class. Three general categories are derived [Pressman, 1997]:

1. *State-based partitioning* categorizes class operations based on their ability to change the state of the class.
2. *Attribute-based partitioning* categorizes class operations based on the attributes that they use.
3. *Category-based partitioning* categorizes class operations based on the generic function that each performs.

3.2.2 Control Flow Graph

Control flow graph is not a testing technique; it is a graphical representation of a program's control logic. It contains process blocks, decisions, and junctions to delineate the structure of a program. A flow graph differs from a flow chart in a sense that a flow graph does not describe the details of the statements; it puts them all into a process block. On the other hand, a flow chart describes every single detail.

Figure 2.3 depicts the elements of a flow graph. A process block is a sequence of program statements uninterrupted by either decisions or junctions. If one statement of the

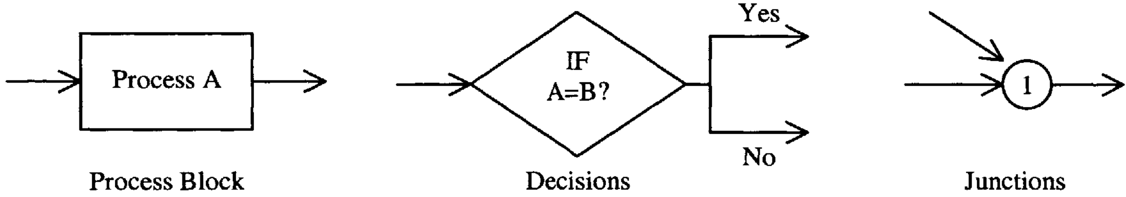


Figure 2.3 Flow Graph Elements
(adopted from Beizer, 1990)

block is executed, then all the statements within that block are executed. A process block has one entry point and one exit. A decision is the point at which the control flow can diverge, and a junction is a point in the program where the control flow can merge.

Flow graphs can be used to draw conclusions about the program's correctness directly from the inspection of the logical structure of the program, because some logic and coding errors are immediately visible from the flow graph, such as structurally unreachable code, non-initialized variables, variables set but not used later, and isolated code segments [Infotech, 1979].

3.2.3 Domain Testing

Domain testing is based on the program's input domain structure. It detects many path selection errors by selecting test data on and near the boundary of a path domain. The program is executed on the selected test data and the output is checked for correctness [DeMillo et al, 1987]. All inputs to a program can be considered as if they were numbers, which makes domain testing a favorable technique for testing mathematical programs [Beizer, 1990].

Domains are defined by their boundaries, the place where most domain bugs occur. An interior point is a point in the domain such that all points within an arbitrarily small distance, called an *epsilon neighborhood*, are also in the domain. A boundary point is one such that within an epsilon neighborhood there are points both in the domain and not in the domain. An extreme point is a point that does not lie between any two other arbitrary but distinct points of a domain. Some common errors include [Beizer, 1990]:

1. Double-Zero Representation – boundary errors for negative zero is common.
2. Floating-Point Zero Check – a floating point number can equal zero only if the previous definition of that number set it to zero or if it is subtracted from itself, multiplied by zero, or created by some operation that forces a zero value.
3. Contradictory Domains – an implemented domain can never be ambiguous or contradictory / inconsistent, which means that at least two supposedly distinct domains overlap.

The major drawbacks of domain testing are the difficulty of selecting test cases for a program that has a large number of input variables and its concentration on path selection errors only. Other testing techniques must be used to thoroughly test a program [Hassell et al, 1982]. Besides, domain testing has a major restriction: it is not good at finding bugs for which the outcome is correct for the wrong reasons, which is also known as *co-incidental correctness* [Beizer, 1990].

3.2.4 Program Instrumentation

Every time the software being tested performs a significant event, the occurrence of that event is recorded. This is known as program instrumentation. The nature of the recording process depends on the information desired and the type of event performed [DeMillo et al, 1987]. The recording processes are called "probes," "monitors," or "software instruments" [Huang, 1979]. The main purpose of instrumentation is to confirm that the outcome of a path is achieved by the intended path.

One type of recording process is called "history-collecting subroutine." The instrumented program is executed on a number of test cases selected, and a history of the behavior of the program is recorded in a database. The behavior information can then be used to construct new sets of test data and to determine the compatibility of the program's behavior with its specifications [Fairley, 1978].

Basically, all instrumentation methods are a variation on a theme of an interpretive trace, which executes every statement in order and records the intermediate values such as all the calculations and all the statement labels traversed. Using such information, we can confirm whether the path that leads to the outcome is the intended one [Beizer, 1990].

A number of methods can be used to label the path. *Line marker* or *traversal marker* names every link by a lowercase letter, while *link counters* increments itself when a link is traversed, and the path length is recorded. One can also mark each link by a unique prime number and multiply the link name into a central register. The path name is a unique number and recaptures the links traversed by factoring [Beizer, 1990].

A simple but well-known program instrumentation method is *assertion*, which inserts additional output statements in the program. It allows the testers to check whether the test executions are proceeding as predicted. The general format of the assertion statement is a Boolean type. If the Boolean expression evaluates to false, the associated statement is executed and the program is terminated [Meyers, 1979]. Monitors are assertions that check whether the value of a variable is within the range specified by assertion.

A tedious job for using the assertion method is that the additional statements need to be removed after the program is recompiled and is confirmed to have passed the test. Therefore, automated tools such as instrumentation compilers and interpreters are devised to perform the assertion and the removal of inserted statements automatically. In some languages, a pre-processor called a "dynamic assertion processor" is used to implement the assertions [Infotech, 1979]. The idea is to place required commands in the form of a comment. To instrument a program, the dynamic pre-processor is called to recognize and

translate these commands (comments) into appropriate statements before the program is compiled. These additional statements can be removed easily by recompiling the program through a regular compiler that will ignore all these commands [Huang, 1979].

An instrumentation compiler instruments the program with history-collecting subroutines to record statement execution counts, ranges of variables, and timing estimates such as CPU time and relative time in each routine. An execution history, the sequence of changes in the computation states of an instrumented program, is recorded in the database. This entire execution history can be analyzed after the program is terminated [Fairley, 1978]. The advantage of such a process is that the analysis and history collection can be performed separately. However, the main disadvantage is the noise introduced by history collecting subroutines, which affects the obtaining of appropriate timing information. Moreover, the execution history of the program is potentially large in size [DeMillo et al, 1987].

Instrumentation interpreter does things differently. Instead of storing the execution history in a database, it maintains and updates the current computation state at each step as the execution advances. As a result, no history is available [Fairley, 1979]. However, it has the advantage of having an interaction with the executing program.

If used effectively, program instrumentation can reduce debugging time for complex programs and allows us to obtain additional information as a by-product.

3.2.5 Program Mutation

Strictly speaking, program mutation is not a testing technique. Rather, it is a mere technique to measure the test data adequacy, which refers to the ability of the data to ensure that certain errors are not present in the product under test [DeMillo et al, 1987]. However, since classes of errors can be found by mutation testing, we will regard it as a kind of testing technique.

In more details, a test data set is adequate if the correct program runs successfully on the data set and all incorrect programs fail in at least one point in the test set. More precisely, if a program "temp" is to be tested using test data D, then a set of mutants of temp is a set of programs that differ from temp in containing a single error chosen from a given list of error types. A mutation score is defined as the fraction of the nonequivalent mutants of temp that are distinguished by the test set D. A mutation score is a number between 0 and 1. A score closer to 1 means that D is very close to being adequate for temp relative to the set of mutants; a score close to 0 indicates that D is weak: the test data does not distinguish "temp" from a mutant that contains an error [DeMillo et al, 1987].

Mutation testing is error-based, which means its goal is to construct test cases that reveal the presence or absence of specific errors. The test is carried out in an automated test harness where mutant programs are generated by applying mutant operators on the program. The mutation operators are derived from actual observations of programmer errors, and a mutant is obtained by introducing exactly one of these errors into the original program. Then, the original program and the mutants are executed on the test data and the results are compared to obtain the mutation score [DeMillo et al, 1987].

A variation of mutation testing is called *weak mutation testing*. Instead of applying mutation operators to the whole program, they are applied to simple components of the program. The mutation score is defined as the fraction of the nonequivalent mutants for which the mutated component computes a value differing from the corresponding component of the program under test [Howden, 1982]. Therefore, a high mutation score does not guarantee that the classes of errors represented by the mutant operators can be avoided. Even so, weak mutation testing has an advantage over traditional mutation testing: a priori test cases that cause mutated components to fail can be described independently of the rest of the program [DeMillo et al, 1987]. Another variation of mutation testing called *trace mutation testing* applies mutation operators to the program as usual, but compares the program traces rather than the outputs [Brooks, 1980].

3.2.6 Functional Program Testing

Functional program testing is a design-based technique that views the design of the program as an abstract description of its design and requirements specifications [Howden, 1980]. The program is viewed as a hierarchy of abstract functions, which is used to identify the functions to be tested. Two kinds of functions, *requirement functions* and *design functions*, may be implemented. The former describes the expected behavior of a program and can be identified by examining the requirements specifications of the program. Design functions are supplements to functions program and provide more details [DeMillo et al, 1987].

The first step of functional program testing is to decompose the program into functional units based on the functional abstraction methods used for program design. Then, data is generated to test the functional units independently using data abstraction as a guide [DeMillo et al, 1987].

3.2.7 Algebraic Program Testing

Algebraic program testing shows that a program is correct if it is equivalent to a hypothetical correct version of itself. Since the equivalence problem for powerful programming languages is difficult to decide, the power of the language must be restricted. For example, branch statements are removed and only integer type variables are allowed. A small set of data suffices to prove the program equivalence [DeMillo et al, 1987].

Two major problems exist for algebraic program testing. First, defining sufficiently general classes of the program and related classes of test cases is difficult. The second concerns the assumption about the properties of the hypothetical correct version. Generating test cases that satisfy the computational properties of the hypothetical version is possible, and this does not prove that this version is hypothetically correct with regard to the original program [Howden, 1978].

3.2.8 Data Flow Testing

Data flow testing selects test paths of a program according to the locations of definitions and uses of variables in the program. This technique is useful for selecting test paths of a program containing nested *if* and *loop* statements and exploring the unreasonable things that can happen to data [Pressman, 1997]. In this approach, a program is considered as establishing meaningful relationships among program variables. A strategy may then be defined in terms of data transformation paths for some or all program variables [Laski, 1982].

A variable in an instruction is described as defined when it is referenced or assigned a new value. A block in a program is a sequence of instructions executed together. For this technique, dealing with blocks is more convenient than dealing with instructions [DeMillo et al, 1987].

One strategy for data flow testing is block testing, in which each tuple of definitions from the data context of every block in the program is to be tested at least once [Laski, 1982]. A data flow analyzer can be used to perform a complete block test, which requires a set of paths that activate all elementary data contexts of every instruction to be exercised. The major drawback for this technique is that it exercises blocks of a program independently, thus failing to force the control flow to activate more complex use-definition chains" [DeMillo et al, 1987].

To overcome the weakness of block testing, a *definition-tree* strategy has been proposed. The programmer can specify a set of variables whose final values rather than the instructions that define them are of interest. The definitions of output variables are traced backwards from the exit instruction until either an input, first block context is reached or a cyclic use of a context appears. Test data are generated to exercise each tuple in the tree at least once [DeMillo et al, 1987]. Another strategy, the *data-space testing*, is similar to definition-tree, but it traces the definitions of all data items such as variables and constants instead of outputs [Paige, 1981].

3.2.9 Real-Time Testing

Real-time software drives a computer that interacts with functioning external devices or objects occurring in an ongoing process [Glass, 1980]. A large number of modules have to be integrated and tested. Difficulties arise because the same sequence of test cases, when input at slightly different times, may produce different outputs. Isolation of problems is also difficult because a large number of decision statements and many modules may share the computer at the same time, and many modules may access the memory randomly [DeMillo et al, 1987].

Typically, real-time software system consists of host and target computers. The latter is the real-time computer that controls the activities of an ongoing process. A host computer is used to construct programs for the target and is usually a commercially available computer that contains a cross compiler or a cross assembler or both, a linking loader for the target, and an instruction level simulator, which allows a host to simulate the behavior of the target [Glass, 1980].

Real-time testing can be characterized as host and target computer testing. The objective of host computer testing is to reveal errors in the modules of the program. Techniques that work well for non-real-time software are used for host computer testing. A full system testing is rarely done, but if it is required, an environment simulator is necessary, which is an automated replication of the external system world constructed for testing. Some target dependent errors may be detected on the host by using an instruction level simulator [Glass, 1980].

For target computer testing, module testing is conducted first, followed by software system integration testing using an environment simulator to drive the target computer. Full system testing is conducted last, often in an operational environment that uses real data as the input source [Glass, 1980]. Since real-time software requires a large number of test cases, random generation of test data is more cost effective.

3.3 Testing and Code Analysis Tools

Just as we use computers to facilitate and automate many of our daily life activities, we can use automated testing tools to assist testing activities. Complicated software programs require the application of a large amount of test cases to thoroughly exercise the code. The generation of the test cases and analysis of the paths exercised in order to determine the extent of the resulting testing coverage would therefore quickly exceed human capacity [Deutsch, 1982].

3.3.1 Overview

The motivation for developing testing tools comes from the necessity of more efficient and lower cost methods to test the more complex and larger scale software systems. As these software products continue to grow in both size and complexity, the effort required to test these software systems manually becomes too tedious and error-prone, and the cost becomes too high to accept. Several testing techniques have considerable overhead in the number and size of test cases. Executing dozens, or even hundreds, of test cases manually is so labor-intensive that it is more feasible to develop special software that retrieves test cases, initiates program executions, and logs test results. In some cases, comparing generated outputs with expected outcomes is not feasible by hand, such as dealing with large output files [DeMillo et al, 1987]. Moreover, manual error checking itself is an error-prone process; the reliability of the test results is indeed in doubt. Therefore, tremendous research effort has been put on developing automatic testing tools that assist in the production of effective tests and analyze the test results.

Automatic testing tools have taken much of the time-consuming, mechanical aspects of testing out of human hands. They provide some attributes that are not easily attained by manual testing, such as improved organization of testing, measurement of testing coverage, and improved reliability. Automatic tools enable a higher volume of testing than would be achievable manually for the same cost, and they can bring rigor and engineer-

ing discipline to the testing process. More importantly, use of automated tools makes possible the enforcement and acceptance of exhaustive testing goals [Deutsch, 1982].

3.3.2 Scope and Implementation of Automated Testing Tools

Testing tools can be classified into static and dynamic analysis tools. The former includes tools for code analysis, program structure checks, proper module interface checks, and event-sequence checking. Dynamic analysis tools include tools for monitoring program run-time behavior, automated test case generation, checking assertions, and inserting software defenses [Deutsch, 1982].

The tool of choice for transition from a manual environment to automated test environment is the *capture / replay* test tool because of its ease to use and understand [Beizer, 1990]. The tester executes the test manually from the keyboard and the software tool captures the keystrokes and the responses that are sent to the screen. The tester can then specify which tests are to be replayed, and the system will compare the new outcomes the stored outcomes and reports any discrepancies. Because of its simplicity, capture / replay test tool is the most popular commercial testing tool [Beizer, 1990].

3.3.3 Static Analysis Tools

As mentioned in Section 2.2.1, static analysis focuses on requirements and design documents and on structural aspects of programs without actually executing it. Static analysis tools analyze characteristics obtained from the program structure without regard to whether the program under test can be executed [DeMillo et al, 1987].

Static analyzers are programs that analyze source code to reveal structural errors, syntactic errors, coding styles, interface consistency, and global aspects of program logic. They generally consist of a front end language processor, a database, an error analyzer, and a

report generator. The following are the basic functions of static analyzers [DeMillo et al, 1987]:

- Code Auditing: the examination of source code to determine whether or not specified programming practices and rules have been followed.
- Consistency Checking: determines whether units of program text are internally consistent in the sense that they implement a uniform notation or terminology and are consistent with the specification.
- Dataflow Analysis: consists of graphical analysis of collections of data definition and reference patterns to determine constraints that can be placed on data values at various points of execution of the source program.
- Error Checking: determines discrepancies, their importance, and causes.
- Type Analysis: involves the determination of correct use of named data items and operations.

Many different static analysis tools are commercially available and like all other testing tools, they are identified by an acronym tool name. Some of them are [DeMillo et al, 1987]:

- DAVE – Documentation, Analysis, Validation, and Error detection developed by the University of Colorado at Boulder. Features include diagnostics, data flow analysis, interface analysis, cross reference, standard enforcer, and documentation aid.
- FACES – Fortran Automated Code Evaluation System developed by COSMIC, University of Georgia. Features include data flow analysis, diagnostics, variables analyzer (inter-module), interface checker, standards enforcer, and reachability analyzer.
- JOYCE – implemented by McDonnell Douglas Automation Company. Features include path structure analysis, symbol cross reference, variable analyzer / interface checking, and documentation aid.

3.3.4 Dynamic Analysis Tools

Dynamic testing tools execute the program being tested directly. The main operations include [DeMillo et al, 1987]:

- Coverage Analysis: determines and assesses measures associated with the invocation of program structural elements to determine the adequacy of a test run.
- Tracing: tracing the history of execution of a program.
- Tuning: determines what parts of a program are being executed the most.
- Simulation: represents certain features of the behavior of a physical or abstract component by means of operations performed by a computer.
- Timing: reports actual CPU times associated with a program or its parts.
- Resource Utilization: analyzes resource utilization associated with system hardware or software.
- Assertion Checking: checks user-embedded statements that assert relationships between elements of a program.
- Constraint Evaluation: generates and / or solves path input or output constraints for determining test input or for proving programs correct.

3.3.5 Automated Verification Systems (AVS)

One set of dynamic testing tools is called *automated verification systems* (AVS). These are tools that measure the coverage of test cases and assist in preparation of test cases. AVS performs five basic functions [Deutsch, 1982]:

1. Analysis of source code and creation of a database.
2. Generation of reports based on static analysis of the source code that reveal existing or potential problems in the code and identify the software control and data structures.

3. Insertion of software probes into the source code that permit data collection on code segments executed and values computed and set in storage.
4. Analysis of test results and generation of reports.
5. Generation of test assistance reports to aid in organizing testing and deriving input sets for particular tests.

One example of AVS is the *coverage certifier* or *coverage monitor.* It records the amount and the type of coverage achieved by a set of test cases over the same component [Deutsch, 1982].

3.3.6 Symbolic Evaluators

Symbolic evaluators are programs that accept symbolic values and execute them according to the expression in which they appear in the program. They are used for symbolic testing mentioned in Section 2.2.4 [DeMillo, et al, 1987].

Some of the well-known systems include [DeMillo et al, 1987]:

- DISSECT – developer unknown. Features include symbolic execution, static analysis, assertion checking, path structure analysis, and documentation.
- EFFIGY – developed by IBM. Features include interactive symbolic execution, assertion checking, proof of correctness, and standard interactive debug tools.

3.3.7 Program Instrumenters

Program instrumenters are systems that insert software probes into source code in order to reveal its internal behavior and performance. The main functions are coverage analysis, assertion checking, and detection of dataflow anomalies. Unfortunately, instrumenters have high overheads [Infotech, 1979].

Some examples of program instrumenters are FAVS – Fortran Automated Verification System developed by the General Research Corporation and PACE – Product Assurance Confidence Evaluator developed by TRW, SEID Software Product Assurance [DeMillo et al, 1987].

3.3.8 Dynamic Assertion Processors

Dynamic assertion processors insert extra statements in the code. These extra statements are assertions that describe some logical condition of the program. The dynamic assertion processor examines each program statement and processes each assertion individually [Infotech, 1979].

Two examples for dynamic assertion processors are JAVS – Jovial Automated Verification System developed by the General Research Corporation [DeMillo et al, 1987] and PET – Program Evaluator and Tester developed by McDonnell-Douglas Corporation [DeMillo et al, 1987].

3.3.9 Automated Mutation System

Automated mutation system is a test entry, execution, and data evaluation system that evaluates the quality of test data based on the results of program mutation. The system computes the mutation score that indicates the adequacy of the test data and provides an interactive test environment and reporting and debugging operations that are useful for locating and removing errors [DeMillo et al, 1987].

Some commercially available mutation systems are Portable Fortran Mutation System and TEC/1 – Fortran Mutation System [DeMillo et al, 1987].

3.3.10 Test Drivers

Recall that a stub is a simulator of a called subroutine, while a driver is the opposite of a stub in that it simulates the calling component or the entire environment for running module tests. A *test driver*, in addition to simulating the calling component, provides the following services [Beizer, 1990]:

- Test Case Initialization: initializes the environment to the state specified for each test and starts the test going.
- Input Simulation: provides simulated inputs to the tested unit, including operator keystrokes and I / O responses.
- Outcome Comparison: compares the actual outcome to the expected outcome specified by the test designer.
- Path Instrumentation and Verification: verifies that the specified path is achieved.
- Reporting: reports by exception for failed test cases, but has controls to allow complete documentation of call cases run, pass or fail.
- Passage to Next Test: provides automatic resetting of initial conditions and execution of the next test case in series.
- Debugging Support: captures the environment for crashes and / or passes control directly to debug package.

Automatic test drivers are significant to software testing because they provide a standard notation for specifying software tests, a standard set-up for verifying software tests, automate the verification of test results, and eliminate the need for writing separate drivers and stubs for unit and integration testing [Panzl, 1978].

DeMillo et al [DeMillo et al, 1987] outline the benefits and drawbacks of test drivers as follows:

1. Reduction in testing effort.
2. Standardization of test cases.

3. Ease of regression testing.
4. The automatic verification of results forces the programmer to state explicitly the expected outputs.
5. Additional work and difficulty associated with learning a specific testing language.
6. Test procedures of some drivers are lengthy and difficult to read. The programmers may feel that writing test procedures is tedious, lacking the challenge and interest of coding a program.
7. The language dependent nature of the test drivers that operate on source-level code makes it difficult to test multiple target languages.

3.4 Test Data Generators

The efficiency of testing totally depends on how well the test data are selected. Important factors affecting test data selection include the ease with which test cases can be constructed, the ease of checking output for correctness, the cost of executing test cases, the types of errors that occur, and the implications of successful test case executions for the behavior of the program for data not executed [Goodenough and Gerhart, 1975]. To achieve these goals, an automated test data generator is the best means.

A test generator is a tool that assists the user in the construction of test data. Generally, the three types of test data generators are pathwise test data generators, data specification systems, and random test data generator [DeMillo et al, 1987].

3.4.1 Pathwise Test Data Generators

Pathwise test data is generated from the input domains associated with program paths from a comprehensive representation of the input space. Pathwise test data generator is the commonest type of test data generators. The four basic operations for pathwise test

data generators are program digraph construction, path selection, symbolic execution, and test data generation [DeMillo et al, 1987].

Program digraph construction corresponds to the preprocessing of the source program to create a digraph representation of control flow in the program.

Path selection concerns with selecting program paths that satisfy testing criteria. The path selection process can be static or dynamic. In static selection, paths are automatically selected prior to symbolic execution. This method is based on the graph structure of the program. For dynamic selection, all feasible paths are automatically selected at the decision points.

Symbolic execution is used to generate path constraints that consist of equalities and inequalities describing program input variables. Input data that satisfy these constraints will result in the execution of that path.

Test data generation involves selecting test data that will cause the execution of the selected paths. Most systems use linear programming algorithms to find numerical solutions to the inequalities of path constraints.

An example of pathwise test data generator is SELECT – Symbolic Execution Language to Enable Comprehensive Testing.

3.4.2 Data Specification Systems

A data specification system assists the tester by providing a data specification language to describe the input data, which will be used by the system to generate the desired input data [Meyers, 1979]. One of the examples is file generators, because they generate test files by using special command languages to describe the data structure of the files.

Some of such data generators are GENTEXTS developed by IRISA, University of Rennes, France and DATAMACS. The former can be used as input to produce a program, while the latter can generate test file to enable file structure testing of the program [DeMillo et al, 1987].

3.4.3 Random Test Data Generators

One way of generating test data is by selecting a random point from the domain of each input variable of a program. For the results to be meaningful, it must be applied to both the selection of data within a path domain and the selection of different path domains [DeMillo et al, 1987].

Random test generation is an easy way to generate test cases because it does not rely on a deep analysis of the program's structure. It provides the tester a great supplement to other testing strategies. However, it sometimes is more stressful to the testers because the test data do not give high confidence on the coverage of all possible paths. The results are not highly reliable, and it is not acceptable to be the sole testing strategy. The following are the main difficulties and weaknesses of random test data [Beizer, 1990]:

1. Random data produce a statistically insignificant sample of the possible paths through most routines. Even copious tests based on random data may not allow you to produce a statistically valid prediction of the routine's reliability.
2. There is no assurance of coverage. Running the generator to the point of 100% coverage could take centuries.
3. If the data are generated in accordance with statistics that reflect expected data characteristics, the test cases will be biased to the normal paths – the paths that are least likely to have bugs.
4. There may be no rational basis for predicting the statistical characteristics of the users. In such cases it is not even random testing; it is arbitrary testing.

5. It may be difficult or impossible to predict the desired outputs and therefore to verify that the routine is working properly. In many cases, the only way to produce the output against which to make a comparison is to run the equivalent of the routine, but the equivalent is as likely to have bugs as the routine being tested.

3.5 Summary

This chapter completes the concepts and details of testing. Different testing techniques not only make testing flexible, but also make the whole testing process more efficient. Sophisticated testing tools can reduce the effort spent on different types of testing. For example, static analyzer makes static testing easier and more cost effective. More and more researches are being done on how to implement more efficient testing techniques and tools.

The previous three chapters describe in details the structure of testing, but have not explained how different testing teams can affect the results of testing. The following chapter will discover this final issue of testing with a relevant case study.

Chapter 4

Collocated and Distributed Software Testing

4.1 Overview

Collocated and distributed developments refer to how the members of the production teams are divided and located. The former usually refers to team members who are at the same location and work together for the same project, while the latter refers to team members who are collaborating at different geographical locations for the same goal. Different tasks may be assigned to different individuals or sub-teams, but their separate achievement combine to create one single product ultimately.

Improvements in telecommunications and computer technologies have made distributed development possible and at an acceptable cost. Decades ago, when electronic mail and Internet were still under development, the only communication medium that allowed geographically distributed teams to collaborate synchronously was telephone, but the cost for long distance calls was high. On the other hand, the efficiency of scheduling phone meetings was low and documentation and repository could not be shared by phone. Most likely, members of distributed teams had to travel to meet with each other for business

purpose, which increases its overhead costs significantly. With such constraints, collocated teams provided the most efficient structure for most projects.

In the contemporary era, a rich variety of communication tools have made the world closer than ever. Almost instantaneously, people from thousands of miles away can see each other; talk to each other, share documentation and repository, and most importantly, at a much lower cost than before. With the relief of geographical constraints, having development teams distributed all over the world is a common practice by different industries and businesses, a practice also known as *globalization*. The remaining issue is: why do we need to develop a project using distributed teams when collocated teams are just fine? To answer this question, we need to look at both the advantages and disadvantages of collocated and distributed development. As the scope of this thesis is software testing, this chapter will be dedicated to discussing issues around software development and testing by collocated and distributed teams.

4.2 Team Concepts for Testing

Sophisticated software projects are typically developed by teams because of their high complexity and volume. Section 1.3 describes different software development processes. Each of these processes is highly intensive and requires dedicated individual(s) to focus on. Take testing as an example. Testing requires a thorough understanding of the whole product and the rich knowledge of preparing test activities and items. Recall that one of the principles of software testing is to assign individuals other than the programmers to perform testing in order to achieve the best results. Therefore, teamwork is mandatory for software projects. In fact, having a successful team structure is the key to software development.

Production of software by geographically distributed teams is more of a managerial issue. Even if the teams are collocated, poor management can lower the efficiency of the human resources. Testing especially needs more attention when teaming is required, because

this activity has long been perceived as insignificant and unattractive. Most engineers have a perception that testing is the most boring and the least rewarding among all software activities. Outstanding programmers may feel that testing roles should be left for less efficient engineers as they view this task in the following ways [Infotech, 1979]:

1. Testing is a dirty business, involving cleaning up someone else's mess and not involving any of the fun of creating programs.
2. Testing is unimaginative and lacks the sense of adventure and opportunity for self expression ordinarily associated with software production.
3. Testing is menial work.
4. Testing is not important, is under funded and generally is too much work for too little time.
5. The tools associated with testing are terrible.
6. Testing has a negative reward structure, in that finding mistakes requires a critic's mentality.

If an engineer is totally unmotivated for being a tester, no matter how smart and efficient he actually is, he will not give you high efficiency and satisfactory performance. If this person is teamed up with other individuals in the testing team, the whole team will become disastrous. In order to have a successful testing team, all the members must not have any of the attitudes mentioned above.

For a project to be successful, one of the most important concerns is human resources. A good and motivated team will definitely provide better integration of individual efforts and more efficient use of human resources. It usually ends up with faster response to requirement changes and fewer design errors because of continuous verification and validation. For every part of software development, having a *jelled* team is the definition of success. DeMarco and Lister [DeMarco and Lister, 1987] state that "a jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts... The team can become unstoppable, a juggernaut for success... They don't need to be man-

aged in the traditional way, and they certainly don't need to be motivated. They've got momentum."

4.3 Collocated Development and Testing

Even though distributed development has become more and more common, collocated teams still have a number of key advantages over teams separated by distance:

1. The cost of communication by distributed teams creates additional overhead compared with collocated teams, especially when teleconferencing is still an expensive mode.
2. Collocated teams allow face-to-face collaboration at all times during work, but distributed teams usually have to make use of asynchronous methods to communicate.
3. Collocated teams usually have higher team spirits, because they contain more communication and interaction opportunities for team members.
4. Collocated teams have less misunderstanding because any confusions or misinterpretations can be explained and solved relatively instantaneously.
5. Errors usually occur during translation and transmission of information. Since distributed teams rely on asynchronous communication, transferring of information between team members is more error-prone.
6. Changes of requirements or plans may be unaware by distributed teams.
7. Distributed teams are harder to motivate and control because of distance factor.
8. Time and cultural differences can be significant issues among distributed teams.

Point 7 is especially important because of the misconceptions mentioned in the last section. The situation is even worse when distributed teams have different number of team members. Team members of the smaller teams may feel themselves even more insignificant when the main control is over the larger teams, because they usually are given the

relatively less important tasks in testing. This undeniably will lower their morale and efficiency. For the worst situation, they may even become unproductive.

Static testing, especially code inspections and walkthroughs, becomes more inefficient as a result of distributed testing, because they require interaction between programmers and testers. With the geographical constraint, team members cannot discuss the implementation issues face to face. This hindrance may not only lower the efficiency of information flow, but may also make the members reluctant to go through the details in depth. Hence, when the testers and programmers are apart from each other, the efficiency and effectiveness of the walkthroughs will decline.

One particular difficulty for distributed testing teams is the design and execution of test cases. Some of the cases and items cannot be clearly divided into parts, that is, they have to be achieved and performed together. For example, integration testing requires interfacing modules together. If the work is separated, duplications of testing may occur. Furthermore, two sets of drivers and stubs may be required to implement the testing. Since the overhead cost for drivers and stubs are high, duplicating them is highly undesired and unacceptable.

4.4 Distributed Development and Testing

Despite all the deficiencies outlined in the previous section, geographically distributed teams will become more and more popular and they may even dominate the way teams are structured in the future. The reason is because of globalization. More and more businesses have expanded their markets globally, which requires them to have offices distributed all over the world. To cope with the change of industry structures, software industry needs to expand geographically to provide quicker and more customized products. In many senses, distributed development is preferred to collocated teams:

1. Distributed teams allow globalization. A firm in Asia may want to deal with suppliers in Asia. If a software company has development teams in America only, it will not be an option for that Asian company if there are alternatives in Asia.
2. Distributed teams provide greater "reach" to the customers. Different development offices become local offices for customers within that region.
3. Division of labor provides more specialization and expertise in particular development areas. A distributed team may make use of engineers of different talent located in different places.
4. The part of testing that requires certain kind of testing tools may be performed at a location where costs of purchasing those tools are lower.

To fully utilize the advantages of distributed development, the problems mentioned in the previous section must be dealt with. In general, miscommunication is the main concern for distributed teams. Lack of interaction and communication between distributed team members make them less motivated, less clear about the progress of the project, and have lower team spirit. The straightforward solution to this problem is to improve interaction.

The team leader for distributed testing team has a significant impact for the whole team. To avoid duplication of work performed or confusion of responsibility of each member, the team leader should assign each member different but clearly divided tasks after the formation of the team. This can allow each member to focus on his part and have a better picture of the whole project. Task assignments should be based on the ability of the team members. Smaller distributed teams should be given equivalent importance during the assignment so that the members will have higher commitment.

The most important task for the team leader is to maintain the relationship of the whole testing team. Even though synchronous collaboration is too costly, electronic mails and other electronic collaboration tools can enable members at different locations to communicate asynchronously more easily and less expensive than telephone and fax machines. The leader should act as a bridge between the members of differently located teams. Whenever something is going to happen, all the members, no matter collocated or dis-

tributed team members, should be informed. This can enhance the spirit of all the distributed teams and create a "jelled" team.

In summary, if distributed development will give better results for a particular project, more attention should be paid to make sure the whole team would perform as if they were one collocated team. Tasks should be assigned clearly in the test plan and everyone should be informed of any activities going on. Test cases should be designed interactively so that everyone agrees with the result and no duplications exist. More time should be scheduled for distributed development because of time overhead due to information flow.

4.5 Case Study: ieCollab

ieCollab stands for **i**ntelligent **e**lectronic **Collab**oration. It is an Internet-based collaborative application service provider for communicating information and sharing software applications in a protocol-rich Java meeting environment. The main objective of this product is to solve the problem of personnel relocation with reliable forums for communication among distributed teams to enhance more efficient and cost effective distributed collaboration.

4.5.1 Development of ieCollab

The product itself is developed collaboratively by distributed development teams. The whole project is a joint effort by the Massachusetts Institute of Technology (MIT) in Cambridge, Massachusetts, USA, Pontificia Universidad Catolica de Chile (PUC) in Santiago, Chile, and Centro de Investigacion Cientifica y Estudios Superiores de Ensenada (CICESE) in Ensenada, Mexico. Twenty-four students from MIT, five from PUC, and five from CICESE worked collaboratively from November 1999 to April 2000 for the project. Each of these students is pursuing degrees in Information Technology

and Computer Science with undergraduate and professional backgrounds ranging from engineering to business.

The main characteristics of ieCollab are Internet-based collaborative application, document sharing, application sharing, Java meeting environment, and meeting management facilities. The whole product is divided into four versions:

1. Meeting Management Environment:
   This version allows distributed users to set up and manage online meetings. This includes functionality to keep track of meeting agenda, meeting participants, user profiles, and meeting styles.

2. Transaction Management:
   This version allows the ieCollab server to track users' usage of ieCollab's meeting management services and charge fees on a per-transaction basis. It also allows other application service providers (ASP) to provide meeting management services to their clients transparently.

3. Collaboration Server:
   This version supports a set of interactive collaboration tools such as chat tools and whiteboards for group communications. Users should be able to pay for the use of these collaboration tools on a per-use basis.

4. Application Server:
   This version allows multiple distributed meeting participants to work on the same documents concurrently using third-party applications, such as computer-aided-design (CAD) tools and spreadsheet applications. Users should be able to pay for these third-party software applications on a per-use basis.

During the period from November 1999 to April 2000, the first two versions were scheduled to be implemented. However, due to unexpected delay of processes, the implementation of the second version was postponed in February 2000 to next academic year.

At the beginning of the project in November, the thirty-four students aforementioned were divided into different software development teams, including business management, marketing management, project management, quality assurance, requirement analysis, design, programming, testing, configuration management, and knowledge management. Each of these teams is responsible for different part of the software development process, and all of them work collaboratively for the common goal.

Figure 4.1 shows the schedule of ieCollab in brief. After the teams were formed, business and marketing managers defined the product to be created. Then, the requirement

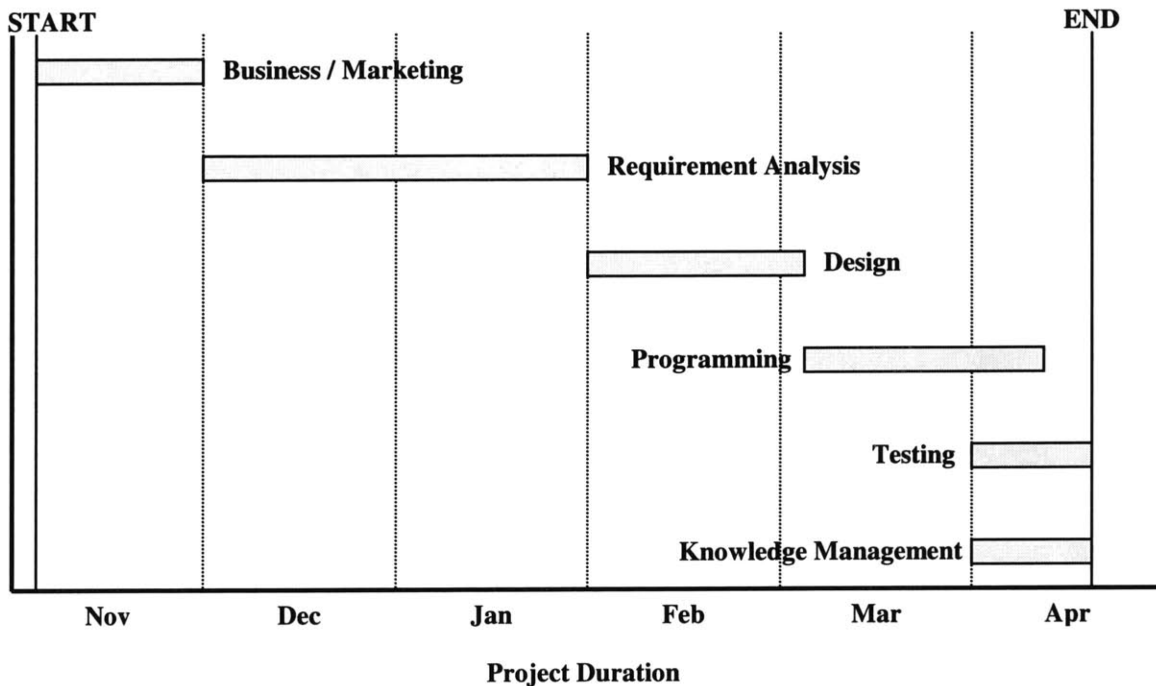


Figure 4.1 Schedule of ieCollab Development

analysts established and defined the standards and specifications of the product. Following their work were the designers, who transformed the requirements and specifica-

tions into formal language describing how the program should be structured and implemented. Afterwards, the programmers developed machine-readable languages corresponding to the design features and functionality using Java 2, Corba, and Oracle database SQL [Project Management Plan for ieCollab, 2000]. Finally, the testers performed testing on the code generated and the knowledge managers prepared user's guide for product release. Quality assurance and configuration management teams are responsible for ensuring the smooth running and the quality of the project. Their job functions span through the whole project period.

Communication among the three geographically distributed units were achieved by different telecommunication and computer technologies. Video conferencing, telephone calls, Microsoft NetMeeting, and CAIRO, a collaborative software product developed by the DiSEL team at MIT in 1999, were used to allow all the members to attend real-time meetings twice a week. For the rest of the time, members communicate mainly through emails and ICQ and occasionally by phone.

This project provides an excellent example for collocated and distributed software development. While most of the development teams are distributed, a few of them consist of only collocated members. Collaboration was enforced within and across each development team.

4.5.2 Testing for ieCollab

The testing team for ieCollab was originally composed of four members: three from MIT and one from CICESE. Unfortunately, due to the reasons outlined in Section 4.2, great miscommunication problems occurred and the testing team ended up with three members in MIT in early March. Testing activities were not affected, but further problems due to lack of communication among different development teams caused considerable delay of the project. The problems will be discussed in Section 4.5.4.

During the first four months, distributed testing team members were given separate tasks on the preparation of the test plan. Each member was assigned a few categories to work on and the team leader combined all the efforts into one single document. Collaboration was achieved by electronic mails.

Other than the compilation of the test plan, each member was given a set of reference materials on testing so as to prepare for the following activities. When test case design, which was scheduled to start in February, finally began in early March, lack of communication and motivation caused the team to break up.

The nature of ieCollab project did not allow testing activities to be performed in exactly the way outlined in this whole thesis. Even though the testing specifications clearly divided the test cases into unit, functional, integration, and system testing, the test cases were in fact a mixture of these different strategies and no clear division line could be made. The reason is due to the limited programming experience of the testers and time constraints. The testers were not exposed to the code and the details of the program. In addition, no testing tools and systems such as test drivers and test stubs were available to the testers, which added more difficulties during test execution. White-box testing, which requires knowledge of the structure of the code, and unit testing, which isolates the program into separate units, became difficult to perform as a result of these reasons. Consequently, the only testing strategies that could be performed were basically static testing and black box testing, which did not require much knowledge and execution of the internal structure of the program.

Eventually, only the client side graphical user interface of the product was handed to the testers, thus testing was only performed on this part. However, test cases were designed for all the features of Version 1 as well as those for Version 2, which might be implemented in the future.

4.5.3 Details of ieCollab Testing

The testing team generated the following deliverables for ieCollab:

- Test Plan (Appendix A)
- Testing Specifications and Test Cases Design (Appendix B)
- Test Log (Appendix C)
- Test Incident Report (Appendix D)
- Testing Report (Appendix E)

Test plan was prepared in December after the formation of the testing team in late November. The test plan was prepared based on all the items mentioned in Section 1.6.1, and the details were planned according to the information given by the business and marketing managers. The sample test plan for ieCollab is in Appendix A.

Testing specifications including all the test cases design began after the requirement specifications and design documents were ready, which provided the source of information for the test case design. The sample testing specifications for ieCollab Meeting Management version is in Appendix B.

Albeit most of the test cases mentioned above were not tested, one test execution was performed and the results were recorded in the **test log** (see Appendix C) and **test incident report** (see Appendix D).

A final **testing report** that summarizes all the activities and occurrences of the test execution was also prepared upon completion of the project. It basically is a composite document of all the deliverables just described. The testing report was one of the deliverables for the whole project. Appendix E contains the final testing report for ieCollab.

4.5.4 Lessons Learned from Testing ieCollab

The development of ieCollab revealed many classical and typical problems for software development, especially by distributed teams. Looking at the testing team in particular, the following points are disastrous:

- Lack of knowledge about testing. The testing team members had virtually no knowledge about details of testing. When the test plan was prepared, some issues were incorrect. For example, in the preliminary version of test plan, "Features to be tested / not to be tested" were misunderstood as "Testing approaches to be tested / not to be tested."
- Lack of consistency. Due to the incorrect generation of some earlier documents, later documents contained a lot of information that was different from the pre-defined issues. One particular example was the generation of test cases. Changes in the use case diagrams of the design documents caused the testing team to re-design many new test cases and discard some of the old ones.
- Lack of communication between distributed team members. Effort was attempted to enhance the relationship between the members at MIT and the member at CICESE. However, time difference and the significant difference in team size between the two teams made the collaboration unsuccessful. Eventually, the testing team became only a collocated team.
- Lack of communication between different development teams. Confusions of design documents affected the smooth running of the testing team, but insufficient effort was established to bind the two teams to work together. At one point, the testers were unable to continue the test case design due to confusions of the design documents, but the designers could not be reached at that time. This affected the efficiency of the testing activity.
- Late change of team structure. The leader of the testing team was changed midway through the project. Difficulties occurred because no time was allowed to ensure a smooth transition. The new leader had to pick up all the remaining work as a team leader without knowing much beforehand.

- Unstable schedule. The project schedule was unstable after the first delay experienced by the requirement analysts. Insufficient effort was put to cope with the change in schedule, which made all the activities following that point chaotic. For example, deadline for design documents was three weeks late, causing coding to begin three weeks late also. Test execution was more than one month late.
- Insufficient support from the programming team. Not enough details were given to the testers before and during test execution, which made life more difficult for the testers.

All these problems are unfortunate issues, but they happen again and again in software development process. Many of these classical mistakes can be avoided if more attention is paid to enhance the team spirit. The following are recommendations for similar projects:

- Encourage different team members to participant in activities of other teams. Doing so will not only allow each member to gain knowledge on different part of the process, it will also prevent a lot of avoidable confusions due to misconceptions and misinterpretations.
- Balance the number of team members in distributed teams. Miscommunication is usually the result of imbalance teams.
- Provide more knowledge to the team members. Sufficient background and skills are absolutely necessary to develop a team. Past sample documents should be given to the team members as early as the team is formed, and more details should be given to the team because such documents usually provide only brief insight into the work involved.

If the whole development process is nicely managed, the efficiency and quality of the product will be much higher and the result will be more motivated and intense individuals.

4.6 Summary

This chapter is the key of this thesis – testing for collocated and distributed development teams. The main issue described is team concepts, which has been regarded as a key managerial issue for developing large-scale software products. A detailed coverage of advantages and disadvantages of collocated and distributed teams are included, and these concepts guide to the case study of ieCollab, a collaborative software tool implemented at MIT. Lessons learned and experiences encountered in this case study reinforce the concepts described earlier in this chapter.

Chapter 5

Conclusion and Future Development

The whole thesis gives a detailed description of how software testing is performed. Even though software testing has always been treated as insignificant, its importance is equivalent to any other software development activities. Same as other roles, testers are expected to be professional, highly motivated, and most significantly, knowledgeable about their work. Unlike most people think, the demand for testing is indeed very high and sometimes the best programmers may be required to perform testing.

The first chapter of this thesis provides general information and background about testing. Principles and objectives of testing are described, and the process of having a good testing on the product is outlined. This chapter provides the foundation and fundamentals of what testing actually is, and what the requirements for testing are.

Chapters two and three give detailed information about testing and test execution. Different test categories, strategies, techniques, and tools are described. Just as testing can never be exhaustive, details about testing are evolving and different strategies and techniques come out momentarily. A good amount of research effort has been put on testing, which will result in more efficient and effective testing materials in the future. Readers

having no knowledge of software testing are encouraged to read these two chapters thoroughly.

The key of this thesis, however, is chapter four. The main issue of nowadays software industry is distributed development, especially when distributed systems that contain one component of the system in one location and the other at another location get more and more popular. Distributed team efforts are irreplaceable if distributed computing is the architecture of the product. To allow successful collaboration between team members, no matter collocated or distributed, great management is a basic requirement. The concern is even greater for distributed teams, when team members are unable to interact face to face. Many issues for collocated teams may evolve further when teams are distributed. The whole project can become disastrous if human resources are not properly managed.

Future research on testing will basically concentrate on the development of more effective testing tools. New approaches that concentrate on object-oriented programs or even next generation programs that will be developed in the future may be devised. Development of collaborative tools, such as ieCollab mentioned in the case study of Chapter 4, will definitely have great impacts on distributed collaboration and will make distributed development more desirable. Testing teams will undeniably benefit from such advantages.

In conclusion, in order to develop a product that has high quality and cost effectiveness, every tiny detail of the development process must be taken care of. Even though testing does not appear significant, it undeniably is a mandatory process to software development, so testers should never be ignored.

References

Business Plan for ieCollab, Information Technology Project 2000 – ieCollab of the Department of Civil and Environment Engineering, MIT, November 1999.

Project Management Plan version 1.2, Information Technology Project 2000 – ieCollab of the Department of Civil and Environment Engineering, MIT, January 2000.

Requirements Specification for ieCollab Version 1, Meeting Management, Specification Version 1.4, Information Technology Project 2000 – ieCollab of the Department of Civil and Environment Engineering, MIT, February 2000.

Bibliography

The ANSI/IEEE Standards 1008-1987, Standard for Software Unit Testing, American National Institute.

Beizer, B., *Software Testing Techniques*, 2nd Edition, Van Nostrand Reinhold, New York, 1990.

Boehm, B., "A Spiral Model for Software Development and Enhancement," *Computer,* vol. 21, no. 5, May 1988, pp. 61-72.

Brooks, M., *Testing, Tracing, and Debugging Recursive Programs Having Simple Errors,* Ph.D. Thesis, Stanford University, 1980.

Clarke, L., "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, Vol. SE-2 (3), September 1976.

Darringer, J. and King, J., "Applications of Symbolic Execution to Program Testing," *Computer*, April 1978.

DeMarco, T. and Lister, T., *Peopleware,* Dorset House, 1987.

DeMillo, R., McCracken, W., Martin, R., and Passafiume, J., *Software Testing and Evaluation*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1987.

Deutsch, M., *Software Verification and Validation*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

Fairley, R., "Tutorial: Static Analysis and Dynamic Testing of Computer Software," *Computer,* April 1978, pp. 14-23

Fuji, M., "Independent Verification of Highly Reliable Programs," *Proceedings of COMPSAC 77*, IEEE, 1977, pp. 38-44

Gilb, T., "What We Fail To Do In Our Current Testing Culture," *Testing Techniques Newsletter,* Software Research Inc., San Francisco, January 1995.

Glass, R., "Real-Time: The "Lost World" of Software Debugging and Testing," *Communications of the ACM*, Vol. 23 (5), May 1980, pp. 264-271.

Goodenough, J. B. and Gerhart, S. L., "Toward a Theory of Test Data Selection," *IEEE Transactions on Software Engineering,* Vol.SE-1 (2), June 1975, pp. 156-173.

Hassell, J., Clarke, L., and Richardson, D., "A Close Look at Domain Testing," *IEEE Transactions on Software Engineering,* Vol. SE-8 (4), July 1982, pp. 380-390.

Hausen, H., *Software Validation*, Elsevier Science Publishers B.V., Amsterdam, The Netherlands, September 1983.

Howden, W., "Reliability of the Path Analysis Testing Strategy," *IEEE Transactions on Software Engineering,* Vol. SE-2 (3), September 1976, pp. 208-214.

Howden, W., "Algebraic Program Testing," *Acta Informatica* (Germany), Vol.10 (1), 1978, pp. 53-66.

Howden, W., "Functional Program Testing," *IEEE Transaction on Software Engineering,* Vol.SE-6 (2), March 1980, pp. 162-169.

Howden, W., "A Survey of Static Analysis Methods," *Tutorial: Software Testing & Validation Techniques,* E. Miller and W. E. Howden, Editors, IEEE, 1981, pp. 101-115.

Howden, W., "Weak Mutation Testing and Completeness of Test Sets," *IEEE Transactions on Software Engineering*, Vol. SE-8 (4), July 1982, pp. 371-379.

Huang, J. C., "Program Instrumentation," *Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography,* Infotech International, 1979, pp. 144-150.

Infotech, *Software Testing, Volume 1: Analysis and Bibliography*, Infotech International Limited, Maidenhead, Berkshire, England, 1979.

Infotech, *Software Testing, Volume 2: Invited Papers*, Infotech International Limited, Maidenhead, Berkshire, England, 1979.

Jack, O., *Software Testing for Conventional and Logic Programming*, Walter de Gruyter, Berlin, New York, February 1996.

Kerr, J., and R. Hunter, *Inside RAD,* McGraw-Hill, 1994.

Kuyumcu, C., *Distributed Testing in Collaborative Software Development*, M.Eng Thesis, MIT, May 1999.

Laski, J., "On Data Flow guided Program Testing," *SIGPLAN Notices,* Vol. 17 (9), September 1982.

Miller, E., "Proceedings," *Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography,* Infotech International, 1979.

Myers, G., *The Art of Software Testing*, John Wiley & Sons, New York, 1979.

Paige, M., "Data Space Testing," *Performance Evaluation Review*, vol. 19 (1), spring 1981, pp. 117-127.

Panzl, D., "A Language for Specifying Software Tests," *AFIPS National Computer Conference Proceedings,* Vol. 47, 1978, pp. 609-619.

Paulk, M. et al., *Capability maturity Model for Software,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

Pressman, R., *Software Engineering: A Practitioner's Approach*, 4th Edition, The McGraw-Hill Companies, Inc., North America, 1997.

Ould, M. and Unwin, C., *Testing in Software Development*, The Press Syndicate of the University of Cambridge, Cambridge, United Kingdom, 1986.

List of Appendices

Appendix A – Test Plan (modified) 93

Appendix B – Testing Specifications (modified).. 95

Appendix C – Example of Test Log (modified) 110

Appendix D – Test Incident Report (modified) 111

Appendix E – Testing Report (modified) 116

Appendix A – Test Plan (modified)

Test Plan
version 0.1
ieCollab Testing Group

Updated by: Chang Kuang
Date: Wednesday, December 2, 1999
Participants on Modification

- online Session
- offline Session (11/30/1999-12/2/1999):
  - Kenward (MIT), Hermawan(MIT), Cesar(CICESE)

References and Links
IEEE Standard for Software Test Documentation (IEEE Std 829-1998)

Test Items:

All items that make up the ieCollab system will be tested during the system test. The configuration manager should control changes to the versions under test and notify the test group when new versions are available.

Features to be tested / not to be tested:

The basic features will be tested.
Security, recovery, and performance testing will be conducted.

Approaches:

- Unit Testing
- Functional Testing
- Integration Testing
- Regression Test (It is assumed that several iterations of the system test will be done in order to test program modification made during the system test period. A regression test will be performed for each new version of the system to detect un-expected impact resulting from program modification)
- System Testing

- Recovery Testing (Performance will be evaluated by halting the power of the machine and then following the recovery procedure)

Item pass / fail criteria:

The system must satisfy the following criteria to pass the tests:

- Memory requirements must not be greater than 64MB of real storage.
- Consistency of user procedures with other related procedures and requirements set by the analysts and designers.

Test Deliverables:

- Test Plan
- Test Specification
- Test Log
- Test Incident Report
- Test Summary Report

Testing Tasks:

Preparation of test plan, preparation of test specifications, execution of test cases, and preparation of test reports.

Environmental Needs:

Hardware: we may require that a processor as slow as 133MHz to work well.

Operating systems: different platforms (Window NT/95/98, Mac OS 8, Suns Solaris, etc.) should be able to use it.

Communication and system software: we may test whether the product is running well under Netscape Navigator 4.0+ and Internet Explorer 5.0+. It should also include the mode of usage and any other supplies or testing tools needed to support the test.

Responsibilities:

To be assigned as the work progresses.

Staffing and training needs:

No special staffing and training needs.

Schedule:

To be provided by project managers.

Appendix B – Testing Specifications (modified)

Testing Specifications
for ieCollab Meeting Management
version 2.0
Testing Team

Prepared by: Hermawan KAMILI, Kenward MA
Date: March 7, 2000 (Tuesday)
Participants on Modification

- online Session
  - Hermawan KAMILI (MIT), Kenward MA (MIT)
- offline Session:
  - Kenward MA (MIT)

References and Links
IEEE Standard for Software Test Documentation (IEEE Std 829-1998)
Design Specification for ieCollab Version 2 (Meeting Management) version 0.2
Requirements Specification for ieCollab Version 1 (Meeting Management) version 1.4
Testing Specifications for ieCollab Meeting Management version 1.0
Test Plan version 0.2

A. Test Cases for Unit Testing

A.1. Meeting

A.1.1. Create Meeting

1. The user shall be able to create a new meeting by clicking the "create meeting" button, and a new window shall pop up and ask the user to input the meeting details.
   Test Input: *click "create meeting" button*
   Expected Output: *the corresponding window pops up*

2. The user shall be able to fill in the meeting name, the meeting description, the meeting time, and the meeting finish time.
   Test Input: *(temp, temporary meeting, start and finish times), (urgent, urgent meeting, start and finish times), (casual, casual meeting, start and finish times)*
   Expected Output: *The inputs are accepted in the corresponding boxes*

3. If the user clicks "cancel," the system must close the window and return to the original state.
   Test Input: *click cancel*
   Expected Output: *as mentioned*

4. The system shall return the next available meeting ID from the database and return it to the user.
   Test Input: *click ok*
   Expected Output: *receive meeting ID*

5. The system shall save the meeting leader ID, the meeting name, the meeting description, the meeting time, and the meeting finish time to the database.
   Test Input: *check meeting info*
   Expected Output: *correct details listed*

6. If the user has a workgroup ID, then the system shall add the workgroup ID to the database under the list of workgroups.
   Test Input: *check list of workgroups*
   Expected Output: *correct workgroups appear*

A.1.2 Update Meeting

1. The user shall be allowed to update the meeting by clicking the "update meeting" button. The system shall pop up the update meeting screen and allow the user to enter the new details of the meeting.
   Test Input: *as mentioned*
   Expected Output: *as mentioned*

2. The user shall be allowed to fill in the new information of the following items: meeting name, meeting leader ID, meeting template, meeting time, meeting finish time, and meeting description.
   Test Input: *fill in new details (temp2, useless meeting...)*
   Expected Output: *details accepted*

3. If the user clicks the "cancel" button, the system must close the window and return to the original state.
   Test Input: *as mentioned*
   Expected Output: *as mentioned*

4. If the user is not the meeting leader, the system must not allow him/her to update the meeting.
   Test Input: *as mentioned*
   Expected Output: *as mentioned*

5. If the user is the meeting leader, then the system shall update the new information in the database.

Test Input: *click ok*
Expected Output: *details recorded*

6. If the user is the leader and he/she enters a new leader ID, then the system shall check if the new leader ID is among the list of users. If so, the system shall update all the information provided (the new leader ID as the leader of the meeting, remove it from the list of users, and move the original meeting leader ID into the list of users). If the new leader ID is not among the list of users, the update shall not be allowed.
   Test Input: *as mentioned*
   Expected Output: *as mentioned*

A.1.3. Join Meeting

1. The user shall be able to join a meeting by clicking the "join meeting" button.
   Test Input: *click "join meeting" button*
   Expected Output: *enters meeting*

2. If the user who requests to join the meeting is in the list of users of that meeting, or the user's workgroup is among the list of workgroups of that meeting, then the system shall display an error message to the user.
   Test Input: *as mentioned*
   Expected Output: *as mentioned*

3. If the user requests to join the meeting 30 minutes before the meeting start time or after the meeting is over and no one is online, then the system shall display an error message to the user.
   Test Input: *try to join meeting as mentioned*
   Expected Output: *display correct error message*

4. If the user is already on the list of online users, then the system shall display an error message. Otherwise, it shall add the user ID to the list of online users and also add the meeting ID to the user's history of meetings.
   Test Input: *try to join meeting as mentioned*
   Expected Output: *display correct error message / allow the user to enter the meeting*

A.1.4. Leave Meeting

1. The user shall be able to leave a meeting by clicking the "leave meeting" button.
   Test Input: *click "leave meeting" button*
   Expected Output: *quits the meeting and return to the original screen*

2. The system shall delete the user ID or the user's workgroup ID from the list of online users. It shall also refresh the list of online and offline users on the screen(s) of other user(s).
   Test Input: *check other users*
   Expected Output: *as mentioned*

A.1.5. Request Membership

1. The user shall be able to request for membership of a meeting by clicking the "request membership" button.
   Test Input: *click "request membership" button*
   Expected Output: *send request*

2. If the user does not belong to any workgroup, the system shall add the user ID to the database under the list of users requesting membership.
   Test Input: *check meeting leader's list*
   Expected Output: *user appears on the correct list*

3. If the user has a workgroup, the system shall check whether he/she is the workgroup leader. If so, the system shall add the user ID to the list of workgroups requesting membership. If not, the system shall add the user ID to the list of users requesting membership.
   Test Input: *as mentioned*
   Expected Output: *as mentioned*

A.1.6. Add member

1. The meeting leader shall be able to add member to the meeting he/she calls for. If the user is not the leader, the system shall display error message to the user.
   Test Input: *try to add member*
   Expected Output: *display corresponding information / error message*

2. If the meeting leader requests to add members to the meeting, the system shall add the parties requesting membership to the list of users / workgroups, depending on the party types.
   Test Input: *add member*
   Expected Output: *update information at the correct place*

3. The system shall acknowledge the parties requesting membership that the request(s) is/are accepted, and add the meeting to the list of meetings of the accepted user(s).
   Test Input: *no*
   Expected Output: *send acknowledgment*

A.1.7. Revoke Membership

1. The user shall be able to revoke his/her membership from a meeting by clicking the "revoke" button.
   Test Input: *click "revoke" button*
   Expected Output: *membership revoked*

2. If the user is the meeting leader, the system shall remove the meeting from the list of meetings.

Test Input: *click "revoke" as the leader*
Expected Output: *meeting removed*

3. If the user wants to revoke a member other than him-/her-self, the system shall display the members to be deleted and ask the leader to make a confirmation. Then, the system shall update the information in the database.
   Test Input: *revoke other members*
   Expected Output: *display confirmation screen and update information*

A.1.8. Invite Me

1. The meeting leader shall be able to invite another user to join the meeting he/she calls for. If the user is not the leader, the system shall prompt an error message.
   Test Input: *invite other users as leader and non-leader*
   Expected Output: *send invitation or display error message*

2. If the leader invites other users to join the meeting, the system shall add the invitees' information (user ID or workgroup ID) to the database.
   Test Input: *invite users*
   Expected Output: *receive relevant information*

A.1.9. Remove Meeting

1. The meeting leader shall be allowed to remove the meeting. If the user is not the meeting leader, the system shall display an error message and disallow the removal.
   Test Input: *click "remove" button as the leader and non-leader*
   Expected Output: *remove the meeting or display error message*

2. If the meeting is to be removed, the system shall save the changes in the database. The system shall also inform all the participants of the meeting that the meeting is removed.
   Test Input: *no*
   Expected Output: *update changes and sending information to relevant users*

A.2. Workgroup

A.2.1. Create Workgroup

1. The user shall be able to create a workgroup by clicking the "create workgroup" button.
   Test Input: *click "create workgroup" button*
   Expected Output: *display the corresponding window*

2. The user shall be able to fill in the workgroup name.
   Test Input: *fill in thesis, project, engineering*

Expected Output: accept fillings in the boxes

3. If the user clicks the "cancel" button, the system must close the window and return to the original state.
   Test Input: click "cancel" button
   Expected Output: as mentioned

4. The system shall check for the workgroup name if the name has been in the database. If so, the system shall give an error message.
   Test Input: fill in the same workgroup name
   Expected Output: display correct error message

5. If the workgroup name is not in the database, then the system shall create a workgroup by obtaining the next workgroup ID from the database and saving the user ID as the workgroup leader.
   Test Input: no
   Expected Output: receive workgroup ID

A.2.2. Update Workgroup

1. The user shall be allowed to update the workgroup by clicking the "update workgroup" button. The system shall pop up the update workgroup screen and allow the user to enter the new details of the workgroup.
   Test Input: click "update workgroup" button
   Expected Output: a corresponding window pops up

2. If the user clicks "cancel," the system must close the window and return to the original state.
   Test Input: click "cancel"
   Expected Output: as mentioned

3. The user shall be allowed to fill in the new information of the following items: workgroup name and workgroup leader ID.
   Test Input: fill in a new name and/or new leader ID
   Expected Output: accept inputs

4. If the user is not the workgroup leader, the system must not allow him/her to update the meeting.
   Test Input: try to update as a non-leader
   Expected Output: disallow update

5. If the user is the workgroup leader, then the system shall update the new information in the database.
   Test Input: try to update as the leader
   Expected Output: information updated

6. If the user is the leader and he/she enters a new leader ID, then the system shall check if the new leader ID is among the list of users. If so, the system shall update all the information provided (the new leader ID as the leader of the workgroup, remove it from the list of users, and move the original workgroup leader ID into the list of users). If the new leader ID is not among the list of users, the update shall not be allowed.
   Test Input: *as mentioned*
   Expected Output: *as mentioned*

A.2.3. Delete Workgroup

1. The workgroup leader shall be allowed to delete the workgroup. If the user is not the workgroup leader, the system shall display an error message and disallow the deletion.
   Test Input: *delete workgroup as a leader and non-leader*
   Expected Output: *delete the workgroup or display correct error message*

2. If the workgroup is to be deleted, the system shall save the changes in the database. The system shall also revoke all the users from that particular workgroup.
   Test Input: *no*
   Expected Output: *update deletion and send notice to relevant users*

A.2.4. Request Membership

1. The user shall be able to request for membership of a workgroup by clicking the "request membership" button.
   Test Input: *click "request membership"*
   Expected Output: *send request*

2. The system shall add the user ID to the list of users requesting membership in the database.
   Test Input: *no*
   Expected Output: *user ID appears in leader's corresponding list*

A.2.5. Add member

1. The workgroup leader shall be able to add member to the workgroup under his/her leadership. If the user is not the leader, the system shall display error message to the user.
   Test Input: *add member as a leader and non-leader*
   Expected Output: *accept new user or display correct error message*

2. If the workgroup leader requests to add members to the workgroup, the system shall add the users requesting membership to the list of users and get the information of the users.
   Test Input: *no*

Expected Output: *update information and send new member's info to leader*

3. The system shall acknowledge the users requesting membership that the request(s) is/are accepted, and add the workgroup to the list of workgroups of the corresponding accepted users.
   Test Input: *no*
   Expected Output: *receive acknowledgement*

A.2.6. Revoke Membership

1. The user shall be able to revoke his/her membership from a workgroup.
   Test Input: *click "revoke" button*
   Expected Output: *revoke user*

2. If the user is the workgroup leader, the system shall remove the workgroup from the list of workgroups.
   Test Input: *revoke membership as a leader*
   Expected Output: *remove workgroup*

3. If the user wants to revoke a member other than him- /her-self, the system shall display the members to be deleted and ask the leader to make a confirmation. Then, the system shall update the information in the database.
   Test Input: *revoke other members*
   Expected Output: *display confirmation window and update information if confirmed*

A.2.7. Invite Me

1. The meeting leader shall be able to invite another user to join the workgroup under his/her leadership. If he/she is not the leader, the system shall prompt an error message.
   Test Input: *invite other users as a leader and non-leader*
   Expected Output: *send invitation or display error message*

2. If the leader invites other users to join the workgroup, the system shall add the invitees' information (user ID) to the database.
   Test Input: *no*
   Expected Output: *as mentioned*

B. Test Cases for Functional Testing

B.1. Registration

1. The system shall allow the user to register an account that does not exist in the database. A new user profile shall also be created.
   Test Inputs: *Vincent Lee (lee, vlee), Hank Koffman (koffman, hkoffman), Horacio Camblong (camblong, hcamblong)*

Expected Output: *Accounts created.*

2. The system shall not allow an account that already exists in the database to be registered again.
   Test Input: *Any of the above.*
   Expected Output: *Registration disallowed.*

B.2. Login

1. The system shall allow the user with a valid account to log in with the correct password.
   Test Input: *lee, vlee; koffman, hkoffman; camblong, hcamblong;*
   Expected Output: *Login successful.*

2. If the account does not exist in the database, then the system shall prompt that the account is not available.
   Test Input: *smith, jsmith*
   Expected Output: *Login unsuccessful.*

3. If the account is correct but the password is not valid, then the system shall prompt to re-input the password.
   Test Input: *lee, lee; koffman, koff...etc.*
   Expected Output: *Login unsuccessful.*

B.3. Logout

When the user logs out, the system shall remove all the information of the user in the system memory. Besides, this user shall be removed from other participants' screen if the user has been listed as workgroup participants.
Test Input: *log out of the system*
Expected Output: *mentioned above.*

B.4. Search

The system shall search the ieCollab database and list user information or relevant workgroup information when the user requests.
Test Input: *search for user and workgroup information*
Expected Output: *display corresponding results.*

B.5. Edit User Profile

The system shall let the user change his / her profile information when requested and store the changes in the database.
Test Input: *change profile.*
Expected Output: *profile updated.*

B.6. Workgroup management for Normal User

1. The system shall list accessible workgroups when requested.
   Test Input: *search for all workgroups.*
   Expected Output: *list all accessible workgroups.*

2. The system shall allow the user to search for workgroup(s) by name or by description.
   Test Input: *thesis, project, engineering*
   Expected Output: *list the details of the workgroups chosen*

3. The system shall allow the user to request for workgroup membership to the workgroup(s) chosen.
   Test Input: *request membership to the above workgroup.*
   Expected Output: *request allowed.*

4. The system shall allow the user to accept or decline a request to join a workgroup.
   Test Input: *as mentioned.*
   Expected Output: *as mentioned.*

5. The system shall allow the user to create a new workgroup.
   Test Input: *create a new workgroup.*
   Expected Output: *ask for detailed information of the new workgroup.*

B.7. Workgroup management for Normal Workgroup Member

1. The system shall allow the user to relinquish workgroup membership.
   Test Input: *revoke workgroup membership.*
   Expected Output: *user revokes from the workgroup.*

2. The system shall list all the members of the workgroup the user asks for.
   Test Input: *request to list members of thesis, project, engineering.*
   Expected Output: *display all members of the selected workgroups.*

3. The system shall list all the scheduled workgroups for the workgroup(s) the user is in.
   Test Input: *list workgroups*
   Expected Output: *as mentioned*

4. The system shall list and access previous workgroup logs for the workgroup(s) the user has joined.
   Test Input: *access workgroup logs*
   Expected Output: *display log details*

5. The system shall allow the user to request for workgroup membership to other workgroup(s).
   Test Input: *request for membership of any workgroup*
   Expected Output: *membership request sent*

B.8. Workgroup management for Workgroup Leader

1. The system shall list all the members of the workgroup the user asks for.
   Test Input: *list workgroup members*
   Expected Output: *display all members of the workgroup*

2. The system shall list all the scheduled workgroups for the workgroup(s) the user is in.
   Test Input: *as mentioned*
   Expected Output: *as mentioned*

3. The system shall list and access previous workgroup logs for the workgroup(s) the user has joined.
   Test Input: *access workgroup logs*
   Expected Output: *display workgroup logs*

4. The system shall allow the user to add new member(s) to the workgroup under his/her leadership.
   Test Input: *add members*
   Expected Output: *members added*

5. The system shall allow the user to accept or deny any request for membership of the workgroup under his/her leadership.
   Test Input: *as mentioned*
   Expected Output: *as mentioned*

6. The system shall allow the user to invite another user to join the workgroup he/she is leading.
   Test Input: *invite any of the users to join workgroup*
   Expected Output: *invitation sent*

7. The system shall allow the user to change workgroup leader.
   Test Input: *request for a new leader*
   Expected Output: *new leader information processed*

8. The system shall allow the user to revoke his/her workgroup membership after the user has requested to change the workgroup leader.
   Test Input: *revoke membership*
   Expected Output: *request approved*

9. The system shall allow the user to list or remove old workgroup logs.
   Test Input: *as mentioned*
   Expected Output: *as mentioned*

10. The system shall allow the user to remove the workgroup he/she is leading.
    Test Input: *remove workgroup*
    Expected Output: *workgroup deleted*

11. The system shall allow the user to request for workgroup membership to other workgroup(s).
    Test Input: *request other workgroup's membership*
    Expected Output: *as mentioned*

B.9. Meeting management

1. The user shall be able to schedule a new meeting.
   Test Input: *call for a new meeting (temp, urgent, casual)*
   Expected Output: *request processed*

2. The user shall be able to set meeting agenda if he/she is scheduling the meeting.
   Test Input: *enter agenda*
   Expected Output: *information recorded*

3. The user shall be able to set meeting roles he/she is scheduling the meeting.
   Test Input: *set meeting roles*
   Expected Output: *information recorded*

4. The user shall be able to select a meeting template if he/she is scheduling the meeting.
   Test Input: *select template*
   Expected Output: *template saved*

5. The user shall be able to invite other users to attend the meeting he/she schedules.
   Test Input: *invite any of the available users*
   Expected Output: *invitation sent*

6. The user shall be able to cancel a meeting he/she schedules, and the system must send a cancellation message to all the invited users. The cancellation message must appear in the Invitation List and shall be clicked away.
   Test Input: *cancel a meeting*
   Expected Output: *as mentioned*

7. The meeting leader shall be able to change the agenda, participants, and the scheduled time for the meeting.
   Test Input: *change meeting details*
   Expected Output: *change allowed and changes recorded*

8. The meeting leader shall have the option to save meeting logs.
   Test Input: *save meeting logs*
   Expected Output: *meeting logs saved in the database*

9. The system shall allow the user to accept or decline invitation to join a meeting.
   Test Input: *as mentioned*
   Expected Output: *as mentioned*

10. The user shall be able to request to join a meeting scheduled by someone else.
    Test Input: *request membership of an existing meeting*
    Expected Output: *request sent*

11. Any user invited to join a meeting shall be able to start a scheduled meeting within a small interval of time before the scheduled start time.
    Test Input: *start the meeting*
    Expected Output: *meeting begins*

B.10. System maintenance

The system shall remove meeting logs from the database after checking that the all links to them have been deleted.
Test Input: *delete meeting logs*
Expected Output: *meeting logs removed*

C. Test Cases for Integration Testing

C.1. Edit Workgroup

1. When the user attempts to edit the workgroup profile, the system shall pop up a window that displays the existing information and allow the user to type in the new details.
   Test Input: *edit the workgroup details*
   Expected Output: *as mentioned*

2. If the user enters new information, the system must update all the information to the server. The system shall display the new information to the user, and shall confirm the changes.
   Test Input: *enter new information*
   Expected Output: *display confirmation*

C.2. Edit User Profile

1. When the user attempts to edit the user profile, the system shall pop up a window that displays the existing information and allow the user to type in the new details.
   Test Input: *edit the user profile*
   Expected Output: *as mentioned*

2. If the user enters new information, the system must update all the information to the server. The system shall display the new information to the user and shall confirm the changes.
   Test Input: *enter new information*
   Expected Output: *display confirmation*

C.3. Edit Meeting

1. When the user attempts to edit the meeting profile, the system shall pop up a window that displays the existing information and allow the user to type in the new details.
   Test Input: *edit the meeting details*
   Expected Output: *as mentioned*

2. If the user enters new information, the system must update all the information to the server. The system shall display the new information to the user, and shall confirm the changes.
   Test Input: *enter new information*
   Expected Output: *display confirmation*

C.4. User Login

1. If the user attempts to log in, the system shall check whether the login name is stored in the database and if so, whether the password is valid. If any or both of them are false, the system must display an error message and prompts to re-login or register. If both are correct, then the system must display the main window of the program.
   Test Inputs: *(johnson, mjohnson), (lee, vvlee)*
   Expected Output: *as mentioned*

2. If the user attempts to add him/herself as a new member, the system shall display the registration window. In registration window the user shall be able to input his information.
   Test Inputs: *All user profile information*
   Expected Output: *notification of data saving*

3. If the user attempts to register, the system must pop up a registration window and allow the user to enter his/her profile information.
   Test Input: *register for a new account*
   Expected Output: *display registration window*

4. After the user enters the information, the system must store them in the database and then return the user ID and shall confirm the storing of information.
   Test Input: *as mentioned*
   Expected Output: *display confirmation window*

D. System Testing

D.1. Volume testing

1. We will run about 20 machines to join a meeting to see whether the system can handle the user request and respond quickly. This is to simulate a real-world meeting that has about 20 participants.

Test input: *n/a*
Expected output: *the system should respond with ease.*

D.2. Load/stress testing

1. When there are 20 people in a meeting, the testing scripts will open 100 simultaneous connections to try to login the system with random user account and password.
   Test input: *n/a*
   Expected output: *the system shall respond to the login request while maintaining the meeting process.*

2. When there are 20 people in a meeting, the testing scripts will add 100 valid users to the meeting at the rate of one person per second.
   Test input: *n/a*
   Expected output: *the system should be able to add all the users to the meeting while still maintaining the current meeting process*

D.3. Configuration testing

1. Determine whether the program operates properly when the software or hardware is configured in a required manner.

D.4. Compatibility/conversion testing

Since this is a new system, it is not necessary to test its compatibility and conversion. Besides, the requirement analysis document says nothing about compatibility/conversion functionality.

D.5. Recovery testing

1. When the meeting is in progress, power off one computer that a participant is using. The system shall be able to detect that this user is not connected to the meeting, and remove him from the meeting participant list. When the user reboots his machine, he shall be able to return to the meeting (if it is still active) by logging the system.
   Test input: *n/a*
   Expected output: *the user is removed from the participant list when he is not connected to ASP server; the system shall be able to return the meeting after he reboots the machine and re-logins.*

Appendix C – Example of Test Log (modified)

Test Log

A test log describes the activities that occurred during the whole testing process. It only outlines the activities, not the process or the incidents. Details of the test results are described in the test incident report.

Criterion

Since only the user interface is available, the testing will concentrate on the appearance and the basic functionality of the program. The sections described refer to sections in a separate test incident report.

Activities

March 30, 2000 (Thursday)

18:30 Testing began by Kenward Ma

18:50 Minor problem discovered in Login Window, see 2.1, Login Button

19:00 Minor problems discovered in Register Window, see 2.2

19:35 Minor problem discovered in Main Window, see 2.3, Create Workgroup Button

19:45 Minor problems discovered in Edit Workgroup Window, see 2.5, Edit Workgroup Window

19:55 Suggestion for User's Public Information, see 2.11, User's Public Info Window

20:00 Suggestion for Create Workgroup window, see 2.6, Create Workgroup Window

20:10 Suggestion for Edit Meeting window, see 2.8, Edit Meeting Window.

20:20 Suggestion for Schedule Meeting window, see 2.10, Schedule Meeting Window

20:30 Finished testing

20:45 Test log and test incident reports generated.

Appendix D – Test Incident Report (modified)

Test Incident Report
for ieCollab Client Interface
Testing Team

Prepared by: Kenward MA
Date: March 30, 2000 (Thursday)
Participants on Modification

- online Session
  - Kenward MA (MIT)
- offline Session:
  - None

References and Links
IEEE Standard for Software Test Documentation (IEEE Std 829-1998)
Testing Specifications for ieCollab (System) version 1.0
Testing Specifications for ieCollab (Meeting Management) version 2.0
Test Plan version 0.2

1. Login Window

Login window pops up upon execution of the program.

Test Item	**Description**
Appearance	OK
Title Bar	OK
Labels	OK
Entry Boxes	OK
Exit Button	OK
Close Window Button	OK
Register Button	OK, see 2
Login Button	Main Window pops up, but login window doesn't go off. See 3

2. Register Window

Register window pops up when the register button of Login window is pressed.

Test Item	Description
Appearance	Labels "Login Name", "Password", Confirm Password" are not centralized
Title Bar	OK
Labels	"Confirmed Password" is not completely shown. Consider widening the window or changing to "Confirmed".
Entry Boxes	OK*
Close Window Button	OK
Register Button	OK**

*MI, Zip Code, and Telephone should accept only corresponding valid entry.
** Same data pops up after closing the register window and re-open it again.

3. Main Window

Main window pops up when the login button of Login window is pressed.

Test Item	Description
Appearance	OK
Title Bar	OK
Labels	OK
Scroll Bars	OK
Menu Bar	OK
Exit Button	OK
Help Button	OK, see 4
Edit Workgroup Button	OK, see 5
Create Workgroup Button	Rollover text should be "Create Workgroup" instead of "Create Work Group" to be consistent with Edit Workgroup Button see 6
Edit User Profile Button	OK, see 7
Edit Meeting Button	OK, see 8
Search Button	OK, see 9
Schedule Meeting	OK, see 10

4. Help Window

Help window pops up when help button of the Main window is pressed.

Test Item	Description
Appearance	OK
Title Bar	OK
Scroll Bars	OK
Close Button	OK

Close Window Button	OK

5. Edit Workgroup Window

Edit workgroup window pops up when the edit workgroup button of the Main window is pressed.

Test Item	**Description**
Appearance	OK
Title Bar	To be consistent with rollover text, consider changing it to "Edit Workgroup" instead of "Edit WorkGroup".
Labels	1. Consider changing to "User(s) Requesting Membership" instead of "User Requesting Membership". 2. Consider changing to "Invited User(s)" instead of "Invited User".
Entry Boxes	OK
Scroll Bars	OK
Exit Button	OK
Exit Window Button	OK
Grant Button	OK
Neglect Button (top)	OK
Profile Button	OK, see 11
Accept Button	OK
Neglect Button (bottom)	OK
Details Button	OK
Update Button	OK

6. Create Workgroup Window

Create workgroup window pops up when the create workgroup button of the Main window is pressed.

Test Item	**Description**
Appearance	OK
Title Bar	OK
Labels	Consider adding the label "Description" for the lower text box.
Entry Boxes	OK
Exit Window Button	OK
Submit Button	1. Null Entry for Name: Error window pops up, OK 2. Correct Entry for Name: OK

7. Profile Window

Profile window pops up when the edit user profile button of the Main window is pressed.

Test Item	Description
Appearance	OK
Title Bar	OK
Labels	OK
Entry Boxes	OK
Submit Button	OK

8. Edit Meeting Window

Edit meeting window pops up when the edit meeting button of the Main window is pressed.

Test Item	Description
Appearance	OK
Title Bar	OK
Labels	1. Consider changing to "User(s) Requesting Membership" instead of "User Requesting Membership". 2. Consider changing to "Invited User(s)" instead of "Invited User".
Entry Boxes	OK
Scroll Bars	OK
Close Button	OK
Close Window Button	OK
Grant Buttons	OK
Neglect Buttons	OK
Profile Button (User)	OK, see 11
Profile Button (Workgroup)	OK
Invite User Button	OK

9. Search Window

Search window pops up when the search window button of the Main window is pressed.

Test Item	Description
Appearance	OK
Title Bar	OK
Labels	OK
Entry Boxes	OK
Scroll Bar	OK

Submit Button	OK
Close Button	OK
Close Window Button	OK

10. Schedule Meeting Window

Schedule meeting window pops up when the schedule meeting button of the Main window is pressed.

Test Item	**Description**
Appearance	OK
Title Bar	To be consistent with rollover text, consider changing it to "Schedule a Meeting" instead of "Create a Meeting".
Labels	Consider adding the label "Description" for the lower text box.
Entry Boxes	OK
Close Window Button	OK
Submit Button	1. Null Entry for Name: Error window pops up, OK 2. Correct Entry for Name: OK

11. User's Public Info Window

User's Public Info window pops up when "Profile" button is pressed.

Test Item	**Description**
Appearance	OK
Title Bar	OK
Labels	Consider putting First and Last Names before MI
Close Button	OK
Close Window Button	OK

Appendix E – Testing Report (modified)

Testing Report
for ieCollab 2000
Testing Team

Prepared by: Kenward MA, Leader, Testing Team
Date: April 21, 2000 (Friday)

References and Links
IEEE Standard for Software Test Documentation (IEEE Std 829-1998)
Testing Specifications for ieCollab (System) version 1.0
Testing Specifications for ieCollab (Meeting Management) version 2.0
Testing Specifications for ieCollab (Transaction Management) version 2.0
Test Plan version 0.2
Test Log and Test Incident Report for ieCollab Client Interface

---

This testing specification was written in accordance with the ANSI/IEEE Std. 829-1998 Standard for Software Documentation.

1 Overview

The purpose of this report is to describe the testing activities for ieCollab for the project period November 1999 to April 2000 and to serve as one of the deliverables of the final product. This report summarizes all the occurrences as well as events encountered by the testing team during the whole period.

1.1 Team Structure and Personnel

The testing team was formed in November 1999. Chang Kuang (MIT) was elected as the leader, with Kenward Ma (MIT), Hermawan Kamili (MIT), and Cisar Guerra (CICESE) as the team members. Every team member was expected to work on every testing task, and job responsibilities were assigned during each activity according to the schedule and competency of each member

In February 2000, a change of team leader took place because the programming team needed the expertise of some of our team members for their programming tasks. Kenward Ma was promoted to be the team leader, while the rest of the members remained tester as a secondary role for the project. In addition to the transition of leadership, due

to communication problems, CICESE withdrew from the project. So, by the end of February, the testing team contained only the three members from MIT.

1.2 Schedule and Process

After the testing team was formed, test plan was prepared accordingly. The final version of test plan came out in early December, and the testing team was idled until mid-February when the requirement specifications were fixed and the design documents were in progress. Even though no specific activities were assigned for the testers, all the members of the testing team attended all the meetings of the project and were involved in the inspection of the requirement specification documents and design documents.

Testing Specification documents that outline the test cases for testing ieCollab came out in March, after the design documents were ready. At the same time the programmers were working on the project, the testers revised the test cases and prepared for the test execution.

Implementation of the first two versions of ieCollab was scheduled during this project period, but due to unexpected delay of the requirement analysis, version 2, transaction management, was postponed in February. Even so, test cases for version 2 were prepared. All the testing specifications document are attached in the appendix of this report.

The first testing of the program was performed in late March, when the programmers handed the client graphical user interface code portion to the testers. Test log and test incident report was prepared correspondingly, and further testing was standing by for other portions of the code. Unfortunately, the programmers failed to provide further codes to the testers, and the above test execution was the only test performed by the testers.

1.3 Approaches

Black box testing was the only available testing approach to testing ieCollab. Due to insufficient knowledge of programming details as well as insufficient time allowed, the testers were not given the contents of the code. So, white box testing could not be performed. In addition, since no testing tools were available, the strategy for testing ieCollab was a mixed one. Instead of having unit testing, integration testing, and system testing separately, the test cases generally involved a mixture of these strategies. Furthermore, when the test was carried out, since most features and functionality were not ready, the test cases did not completely match with those specified in the testing specifications.

2 Testing Specifications

See Appendix B.

3 Test Log

See Appendix C.

4 Test Incident Report

See Appendix D.

5 Conclusion

The development of ieCollab revealed many classical and typical problems for software development, especially by distributed teams. Looking at the testing team in particular, the following points are disastrous:

- Lack of knowledge about testing. The testing team members had virtually no knowledge about details of testing. When the test plan was prepared, some issues were incorrect.
- Lack of consistency. Due to the incorrect generation of some earlier documents, later documents contained a lot of information that was different from the predefined issues.
- Lack of communication between distributed team members. Effort was attempted to enhance the relationship between the members at MIT and the member at CICESE. However, time difference and the significant different in team size between two teams made the collaboration unsuccessful.
- Lack of communication between different development teams. Confusions of design documents affected the smooth running of the testing team, but insufficient effort was established to bind the two teams to work together.
- Late change of team structure. The leader of the testing team was changed midway through the project. Difficulties occurred because no time was allowed to ensure a smooth transition. The new leader had to pick up all the remaining work as a team leader without knowing much beforehand.
- Unstable schedule. The project schedule was unstable when the first delay experienced by the requirement analysts. Insufficient effort was put to cope with the change in schedule, which made all the activities following that point chaotic.
- Insufficient support from the programming team. Not much detail was given to the testers before and during test execution, which made life more difficult for the testers.

All these problems are unfortunate issues, but they happen again and again in software development process. Many of these classical mistakes can be avoided if more attention is paid to enhance the team spirit. The following are recommendations for similar projects:

- Encourage different team members to participant in activities of other teams. Doing so will not only allow each member to gain knowledge on different part of the process, it will also prevent a lot of avoidable confusions due to misconceptions and misinterpretations.
- Balance the number of team members in distributed teams. Miscommunication is usually the result of imbalance teams.
- Provide more knowledge to the team members. Sufficient background and skills are absolutely necessary to develop a team. Past sample documents should be given to the team members as early as the team is formed, and more details should be given to the team because such documents usually provide only brief insight into the work involved.