DOKUZ EYLÜL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

ANALYSIS AND IMPLEMENTATION OF SOFTWARE TEST CASE DESIGN TECHNIQUES

by

Berk BEKİROĞLU

June, 2011

İZMİR

ANALYSIS AND IMPLEMENTATION OF SOFTWARE TEST CASE DESIGN TECHNIQUES

A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Master of Science
in Computer Engineering, Computer Engineering Program

by
Berk BEKİROĞLU

June, 2011
İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis, entitled **"ANALYSIS AND IMPLEMENTATION OF SOFTWARE TEST CASE DESIGN TECHNIQUES"**, completed by **BERK BEKİROĞLU** under the supervision of **PROF. DR. YALÇIN ÇEBİ,** and we certify that in our opinion, it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Prof. Dr. Yalçın ÇEBİ

Supervisor

[illegible]

Yrd. Doç. Dr. Kökten Ulaş BİRANT

(Jury Member)

[illegible]

Prof. Dr. Mehmet SEZER

(Jury Member)

[illegible]

Prof.Dr. Mustafa SABUNCU
Director
Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

I am heartily thankful to my thesis advisor Instructor Dr. Kökten Ulaş Birant, whose encouragement, supervision and support from the preliminary to the concluding level enabled me to develop an understanding of the subject.

I would also like to thank my supervisor, Prof. Dr. Yalçın ÇEBİ for his help, suggestions and guidance.

Berk Bekiroğlu

ANALYSIS AND IMPLEMENTATION OF SOFTWARE TEST CASE DESIGN TECHNIQUES

ABSTRACT

Software testing is a process of finding bugs in a system, and it is an essential part of the software development process. Although, software testing is still an art, numerous techniques are developed to design and implement software testing.

Designing software testing requires determining software test conditions. Different software test case design techniques are used to translate test conditions into test cases. Moreover, each software test case design technique produces different test cases from the same test conditions. The test scripts, which are used during test execution, are directly formed from test conditions.

The main goal of this thesis is researching software test case design techniques and their implementations. The process of determining the test conditions is described. Moreover, formal documentation of test conditions, test cases and test scripts are denoted. The requirement-based software testing, which uses software specifications to generate test cases, is analyzed. The cause-effect software test tool, which uses a requirement-based software testing method, is implemented, and its advantages and disadvantages are stated. In addition, other types of software testing tools, such as code coverage analysis tools and software test data generator tools, are described.

The used software testing design and test case design techniques highly depend on the type of project. Different test design and test case design techniques can be used in each step of the software development process. Moreover, different types of tests can be applied at different test levels. In this thesis, these software test types and test levels are also mentioned.

Keywords: Software Testing, Software Test Case Design Techniques, Software Testing Tool

YAZILIM SINAMA VAKA TASARIM TEKNİKLERİNİN ANALİZİ VE UYGULAMASI

ÖZ

Yazılım sınaması bir sistemdeki hataları bulma işlemidir ve yazılım geliştirme sürecinin temel bir parçasıdır. Yazılım sınaması halen sanat olarak kalsa da, birçok yazılım sınama tasarım ve uygulama teknikleri geliştirilmiştir.

Yazılım sınama tasarımı, yazılım sınama vakalarının belirlenmesini gerektirir. Yazılım sınama koşullarından, farklı yazılım sınama vakaları türetebilmek için farklı yazılım sınama vaka tasarım teknikleri kullanılır. Ayrıca, her bir yazılım sınama vaka tasarım tekniği, aynı yazılım sınama koşullarından farklı yazılım sınama vakaları üretir. Yazılımlar sınanırken kullanılan yazılım sınama senaryoları direk olarak yazılım sınama vakalarından oluşturulur.

Tezin başlıca amacı, yazılım sınama vaka tasarım tekniklerini ve bunların uygulamalarını araştırmaktır. Ayrıca, yazılım sınama koşullarının belirlenmesi için gerekli işlemler anlatılmıştır. Bunun dışında, yazılım sınama koşuları, vakaları ve senaryolarının resmi bir şekilde raporlanması gösterilmiştir. Yazılım sınama vakası üretmek için yazılım isterlerini kullanan ihtiyaç tabanlı yazılım sınaması analiz edilmiştir. İhtiyaç tabanlı yazılım sınama tekniğini kullanan neden-sonuç yazılım sınama aracı geliştirilmiş ve bunun avantaj ve dezavantajları belirtilmiştir. Bunun dışında, yazılım kaynak kodu kapsam analiz aracı ve yazılım sınama veri üreteci gibi diğer yazılım sınama araçları anlatılmıştır.

Kullanılan yazılım sınama tasarımı ve yazılım sınama vaka tasarımı projenin türüne bağlıdır. Yazılım geliştirme sürecinin her aşamasında farklı yazılım sınama tasarımı ve yazılım sınama vaka tasarımı kullanılabilir. Ayrıca, farklı yazılım sınama düzeylerinde farklı yazılım sınama türleri uygulanabilir. Bu tezde ayrıca bu yazılım sınama türleri ve düzeylerinden bahsedilmiştir.
Anahtar sözcükler: Yazılım sınama, Yazılım Sınama Vaka Tasarım, Yazılım Sınama Araçları

CONTENTS

CHAPTER ONE
INTRODUCTION

1.1 Overview

Software testing is an essential part of the software development process. It should be performed in all types of software development methodologies. Moreover, in each step of the software development process, software testing should be performed. Software testing can be applied at different test levels such as component, integration, system and acceptance.

Three main activities are performed during the whole testing process. Firstly, test conditions should be determined from domain requirements, technical requirements, business processes and source code. Secondly, test cases are derived from test conditions. Different test case design techniques are used to generate test cases from test conditions. Finally, test scripts, which describe how the test should be executed, are generated directly from test cases. The most costly and important process in the testing process is generating test cases. Determining the test conditions and generating test scripts from test cases are easy tasks.

Black box and white box are two main formal software test case design techniques. Black box testing methods use software specifications to generate test cases. On the other hand, white box testing methods use the source code to generate test cases. Both black box and white box software test case generation methods can be used at every level of software testing. During the testing process, firstly, black box techniques are used. Then white box testing techniques are used to verify the black box testing techniques and generate new test cases, which are not possible to derive by black box testing techniques. In addition, experience-based software test methods can be used. However, experience-based software testing methods cannot be used alone. It should be performed after all formal black and white box techniques. It is effective when the time for testing is very limited.

A large number of software test tools are developed to facilitate the software testing process. Software test case generation tools are one of the most important software testing tools. The cause-effect graph test tool which uses requirement-based software testing is one of them. By this tool, test conditions can be systematically combined according to software specifications.

1.2 Aim of Thesis

The main goal of this study is to analyze the software test case design techniques, and to develop a software test case design tool which uses cause-effect graphing software test case design technique.

In this thesis, also, the requirement based software testing process, which cause-effect graphing software testing techniques are based on, is analyzed and the problems related to software specifications are denoted.

In addition to the cause-effect software test design tool, other types of subsidiary software testing tools, which are used to generate test data for test cases, are examined and a random Turkish test data generator is developed.

1.3 Thesis Organization

This thesis is divided into eight chapters. A general description and aim of thesis are given in Chapter 1. The second chapter covers definitions of software testing and software bugs as well as the history and principles of software testing.

In Chapter 3, test levels and types are described. Next, Chapter 4 reviews software test design processes and related software testing documents. Subsequently, Chapter 5 focuses on the review of software test case design techniques. The cause-effect graphing software testing method is also given in this chapter.

Chapter 6 presents the requirement based software testing. The chapter ends with a section dedicated to problems when specifying external specifications. The Chapter

7 contains software testing tools. Moreover, cause-effect graphing software testing tool is explained in this chapter. Finally, the thesis will conclude with some conclusions.

CHAPTER TWO
SOFTWARE TESTING

Software testing is an essential and important part of the software development process. That is why more than fifty percent of the cost is due to software testing. In fact, software testing is developed spontaneously with the development of software. It becomes the natural part of a software development process. All commercial products should be tested when they are produced. So, if we think of software as a product, it should be tested. Software developers should ensure the correctness of their software. Thus, they have to test it to verify that the software works as expected.

There are many definitions and goals of software testing depending on the type of the software project. Some software developers consider software testing more different than others in the terms of definitions and goals. On the other hand, all definitions and goals have common points. According to one definition, software testing is a process or series of processes that verify the software codes to show that they work as expected. Moreover, software testing verifies that unexpected behaviors do not occur during the execution of software testing. In this definition, software testing is a highly intensive technical task. However, the other sides of software testing are not considered in this definition. Software testing highly depends on economic and psychological factors. However, the most important mistake made by software developers is the wrong definition of software testing rather than economic and psychological factors. Some wrong definitions, according to Glenford J. Myers (2004), are:

- "Testing is the process of demonstrating that errors are not present."
- "The purpose of testing is to show that a program performs its intended function correctly."
- "Testing is the process of establishing confidence that a program does what it is supposed to do."

All these definitions denote that good testing must show the accuracy of software functions, not software errors. However, one of the most important goals of software testing is adding a value to software. For other commercial products, producers try to add a value and differentiate their products. Ensuring good quality is one of these methods. For software, quality can be provided by finding and removing bugs. That is why, software testing focuses on finding software bugs.

Software testing shows the existence of software bugs rather than the correctness of software. For this reason, the more correct definition of software testing is: "Software testing is the process of executing a program with the intent of finding bugs". Furthermore, Dijkstra (1978) says "Testing can only show the presence of errors, not their absence". This definition shows that testing cannot prove the absence of bugs.

The analogy is when you are sick, you go to hospital to see your doctor. Then your doctor asks you to do some tests in order for him to diagnose your illness. After these tests, if your doctor says that your test results are normal, you do not believe or rely on these tests anymore, because you are in pain, and you know that you are sick. You expect tests to show abnormal values, which cause your illness. In this case, the successful test should show abnormal values, not normal values.

During the software testing process, testers should assume that all the software has a lot of bugs. In fact, producing bug-free software is impossible because of human nature. The main goal of software testing is reducing these bugs as much as possible within a given budget and time constraints.

2.1 The History of Software Testing

The word "bug" came into existence in 1947 at Harvard University. When technicians were working for a new computer, it suddenly stopped working. Then they tried to find the cause of that. Eventually, they found the cause which was a moth trapped between the points of Relay #70, in Panel F.

Some events were milestones for software testing. These events caused millions of dollar and prestige lost for companies. Moreover, these events have increased the importance and necessity of software testing. Some of these important events are described in this section.

2.1.1 Disney's Lion King, 1994 - 1995

In the fall of 1994, the Disney Company achieved a new multimedia CD-ROM game. The name of the game was The Lion King Animated Storybook. This was the first CD-ROM game developed by the Disney Company. There were other companies in the market that produced multimedia CD-ROM games at the time. On the other hand, the Disney Company planned to enter this market by this multimedia game. Thus, this product was highly promoted and advertised by the Disney Company. A large amount of copies were sold during that Christmas. After Christmas, most customers complained by calling the support center. A lot of stories appeared in TV news.

The problem was that the game did not work on most PC models in the market. Developers did not test it with all PC models in the market. The game worked on only few PCs one of which was used by the Disney programmers. However, the game did not work on the most common PC model. The result of this incident was that software should be tested on all available common PC models and operational environments in the market.

2.1.2 Intel Pentium Floating-Point Division Bug, 1994

In the old Intel Pentium CPU, there was a floating-point division bug. This bug was invented by Dr. Thomas R. Nicely of Lynchburg (Virginia) in 1994. He got an unexpected result in his experiment. After his research, numerous people met this problem. Fortunately, this bug appeared only extremely math-intensive, scientific, and engineering calculations. Thus, only few people encountered this bug.

The importance of this story was not the bug, but the way the bug was handled. Intel found this before achieving the chip. However, Intel's management decided to launch this chip with the found bug. Fixing the bug was not cost effective in this stage, according to Intel's managers. After the appearance of this bug, the confidence in Intel was damaged.

2.1.3 NASA Mars Polar Lander, 1999

NASA's Mars Polar Lander was decayed when it was trying to land. Investigations showed that the accident occurred because of one bit. The landing process in theory was, when Lander approaches the surface, its parachute opens to slow it. After deploying the parachute, the legs of Lander take a landing position. However, to save money, NASA simplified the mechanism of shutdown of thrusters. Instead of using costly radars, which were used in other space crafts, they used an inexpensive contact switch. This switch shut off the fuel by using one bit command. The engine burned until the legs touched the surface. The consequences of this accident were very big, but the cause of that was very simple. Lander was tested by multiple test groups. The first software testing group tested only the leg opening process. Another software testing group tested the processes after opening the legs. The first team did not test the condition of the touch-down bit when the legs were not opened. The second team started testing by resetting the computer. Thus, the second team always cleared the bit. Both parts worked perfectly. However, they did not work together.

2.1.4 The Y2K (Year 2000) Bug, circa 1974

Early programmers worked with a very limited memory. Thus, they had to use each bit more effectively. At that time, programmers shortened date information and represented it by two digits. For example, to hold the 1970 year information, they would just hold the last two digits 70. They knew that it would be a problem in the year 2000. (00 digits refer to 1900 in their representation) However, they thought that the existing system would have been changed by the year 2000. Unfortunately, some systems had not changed, and the developers retired. So, most of the companies did not know that their system is Y2K compliant. Moreover, date information is very important for companies because all payroll systems include date information. Thus, they have to be sure that their system is Y2K compliant. Consequently, several hundred billion dollars were spent on replacing or updating the computer programs.

2.2 What is Bug?

Bug is not a simple term. Bugs can cause the loss of life or millions of dollars, or interrupt our games. Most companies define our own software bug terms. Defect, variance, fault, failure, problem, inconsistency, error, feature, incident, and anomaly might be used interchangeably within software companies and among programmers. However, they have quite different meanings. Fault, failure, and defect are usually used for important things, and they usually cause big problems. For example, the text color or background color will not be a fault. These words contain responsibility, and someone should be accountable for these problems. On the other hand, anomaly, incident, and variance are not so negative. They usually occur because of unintended operations or misunderstandings.

Most well-known software companies spend a large amount of time on determining terms which are related with software bugs. Consistency is very important within all software development processes. For this reason, these terms should be determined before starting software development. In this context, not the correctness of these terms but the consistent usages of these terms within the same organization that are more important. Therefore, some companies and software development teams spend hours and hours on arguing and debating which term to use.

Software bug occurs when one of the following five rules is true:

- The software does not do something that the product specification says it should do.
- The software does something that the product specification says it should not do.
- The software does something that the product specification does not mention.
- The software does not do something that the product specification does not mention but should.
- The software is difficult to understand, hard to use, slow, or in the software tester's eyes will be viewed by the end user as just plain not right.

2.3 Software Testing Principles

According to Glenford J. Myers (2004), some principles might be applied for more effective and efficient software testing. In the following section, these principles are explained.

Principle 1: A software test case should include expected outputs of the test. This mistake is made by most testers because of human psychology. When testers define software test cases, if they do not declare these software test cases formally and explicitly, they might not notice bugs. For this reason, expected outputs for each test case should be documented. The phenomenon of "the eye seeing what it wants to see" is sometimes true. Thus, testers should carefully compare outputs of software tests after the execution of test cases with the expected outputs of software test cases.

Principle 2: Programmers should not test their own programs. Programmers do not think that their programs have bugs. If a programmer thought that his/her program has bugs, he/she would inspect and amend problems during development process. Moreover, programmers usually work hard with energy to develop a program. Thus, they usually do not tend to change their programs. Another problem is that programmers may misunderstand the software specifications. Therefore, if programmers test their own programs, they will not be aware of the bugs which are related to software specifications. So, they carry the same misunderstandings into the software testing process. This principle does not imply that a programmer cannot test his/her own programs. However, if a programmer tests his/her own programs, the efficiency of software testing decreases. On the other hand, programmers should debug their own programs, because only the programmer knows the internal structure of his/her programs. Thus, it is easier for him/her to correct the detected bugs.

Principle 3: The organization which develops the software should not test its own software. This problem again is related to the issue in *principle 2*. If software testing is performed in the same organization, some psychology factors can affect the software testing. First of all, software development is a planned activity. Time constraints for each stage in the software development process are represented in the project schedule, before starting the project. If testers exceed the defined deadline for software testing because of plenty of bugs, the cost of the project automatically increases. Thus, testers might be responsible for this delay. If testers accuse programmers because of their poor programs, the problems can appear in the organization. Similarly, if programmers accuse testers because of their poor testing skills, the same problems can emerge in the organization. In addition, programmers and testers know each other in the same organization. They might be good friends. So, testers might skip bugs because of this relationship. They do not want to set down. For this reason, testers should not know the authors of programs.

Principle 4: Testers should thoroughly inspect the results of each test. It is the most obvious principle. However, it is often overlooked. Some bugs cannot be detected, even if they are so obvious. For example, output of function is displayed on screen. If the output is wrong, it should be very easy for tester to detect this bug. However, tester sometimes does not notice these kinds of bugs, if tester does not check each outputs of function one by one. At a later time, tester may notify these bugs. Experiments show that bugs that are found in later tests are often missed in the early tests.

Principle 5: Test cases cover not only invalid and unexpected situations, but also valid and expected situations. When a programmer tests their own program, he/she usually tests his/her program with valid inputs. However, a tester should develop test cases for invalid inputs as well. For example, for triangle drawing software, three edge lengths are required. First of all, testers develop test cases for valid inputs by determining integer values to each edge length. Then, testers have to check the output. If a triangle is drawn, this test case passes the test. After this test, testers have to try the same process with invalid inputs by giving string values to each edge length. The testers expect a warning message. If a warning message appears on the screen, the test will be successful. However, some conditions may be overlooked. For example, “1”, “2” and “5” are valid inputs for this program if only input types are checked. However, “1”, “2” and “5” values cannot constitute a triangle because of triangle properties. If this test case is considered, these values cause a bug. Thus, when selecting inputs, a comprehensive analysis is required.

Principle 6: Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do. It is the result of the previous principle. Software testing involves not only controlling the system with existing conditions, but also considering non-existent conditions. Discovering new possible conditions can cause new bugs.

Principle 7: Avoid throwaway test cases, unless the program is truly a throwaway program. When a tester sits on the computer and generates test cases, he/she has to keep these test cases by writing a test case document. Test cases are investment and retest should be required after each modification. If previous test cases are not kept, the tester has to generate test cases again. However, it takes too much time and resources. As a result, a re-test may not be performed because of that reason. If a re-test is not performed, new medications may cause different bugs in the system. Thus, a re-test is required after each modification.

Principle 8: Do not plan software testing under tacit assumptions. This principle emphasizes the definition of software testing. Software testing is a process of executing a program with the intent of finding errors. If you make an assumption that the software is bug-free, this will not count as a test. It will only count as a validation for a customer. Moreover, tacit assumptions may not last long. For this reason, any tacit assumption should be documented.

Principle 9: The probability of the existence of more bugs in a section of a program is proportional to the number of bugs already found in that section. Errors cause more errors. If there are two modules, classes or subroutines “A” and “B”, and if testers detect five errors in “A” and one error in “B”, the probability of the existence of more errors in “A” is more than “B”. There is no scientific explanation for this. However, it is a well-known phenomenon between testers.

Principle 10: Testing is an extremely creative and intellectually challenging task. Although numerous software testing principles are developed, they do not explicitly show how software testing should be performed. Every tester has their own software testing methods. Nobody can say that this particular method is wrong, if it finds a bug.

CHAPTER THREE

SOFTWARE DEVELOPMENT MODELS

The testing process highly depends on the type of the software and the software development methodology used. In some projects, time is very important for marketing. Thus, the testing process should be performed very quickly. On the other hand, in some projects, quality is very important, so very comprehensive test documents should be written.

In every software development life cycle, one process focuses on verification and validation. The verification test ensures that the program works according to the software specifications. On the other hand, the validation test ensures the correctness of the software specifications.

3.1 V-Model

The V-model is an improved version of the waterfall model. Although testing is performed in every stage of the waterfall model, most of the bugs are detected just before the implementation phase. Moreover, if there is a feedback or a bug in the past phases, the iteration of a particular phase becomes very difficult or impossible. In the V-model, testing starts in the early stage of the software development process, and a lot of test activities are performed before testing the final program. The V-model explicitly determines the validation and verification tests of each software development phase. Especially, the validation testing starts in the early stages of the software development life cycle by reviewing the user requirements. In addition, the acceptance testing is usually performed in subsequent stages of the software development life cycle.

Although numerous variants of the V-model exist, common V-models use four test levels. These levels are component testing, integration testing, system testing and acceptance testing. These levels may increase or decrease according to the size and type of project. The test can be combined and re-organized according to the system architecture.

3.2 Iterative Life Cycle

Instead of a sequential life cycle, an iterative or incremental life cycle can be used. The common feature of this kind of development methods is defining increments and building additional functions in each increment. In each increment, current and previous increments should be tested. This is called regression of existing functions. For this reason, important functions are implemented first. Furthermore, each function can be tested at different levels in different iterations.

3.3 Agile Development

Extreme programming is one of the best well-known agile development life cycle models. Functions are rapidly developed from business stories. These functions are evaluated with continual feedback. Thus, the acceptance testing is performed in every stage of the software development life cycle. In this way, the risk of failure at the end of the project can be decreased.

In the agile development, software tests are dynamic, because requirements can change very quickly. In other words, the software test cases should be changed after each requirement modification. In addition, before the development of a component, its test cases and scripts should be produced. In this manner, the efficiency of software testing can be increased.

Numerous iterations may be required for each level of testing. In each change, an integration test is required to show that the current change is successfully integrated into the existing system.

3.4 Test Levels

Software tests are usually grouped by the level of specificity of the test. To prevent overlap and repetition of software tests, these test levels should be known. In this section, each test level is explained.

3.4.1 Component Testing

This is also called unit or module testing. The objective of this testing is finding the defects in software units or modules which are objects and classes in a program. Thus, each unit or module should be singly testable. Thus, components can be tested insulated from other components.

Component testing involves testing both functional and non-functional specifications of a component. For instance, testing the resource behavior, performance and robustness of a component is considered as component testing.

Software test cases can be derived from all produced software engineering products, such as a software design document or a data model. In a component testing, a source code is usually available. Thus, most component tests are white box software testing. A component is usually tested by its author. However, it is better when different testers or coders test the component.

In the Extreme Programming, component test cases should be prepared and automated before the implementation of components. This approach is also called test-driven development.

3.4.2 Integration Testing

Integration testing includes testing interfaces between components and interaction between other systems, such as operating systems, file systems and hardware systems. Integration testing is usually performed by an integrator. Integrator is usually a specific integration team or test team.

Integration testing can be divided as different levels according to the type and scale of a product. For example, integration between components is tested after the component testing to find defects between component interfaces. Moreover, the system integration testing can be performed after the system testing to test interaction between systems.

Different methods are developed to perform an integration test. One extreme method is the integration of all components and systems simultaneously. This is also called big-bang integration testing. The biggest advantage of this is that everything is finished before starting the integration test. Testers do not have to simulate the unfinished parts. On the other hand, it is very difficult and time-consuming when tracing the cause of the bug. Thus, the big-bang integration might be useful for unproblematic components. In addition, finding defects with this method is ineffective.

The other extreme integration technique is integrating all components one by one. Finding bugs with this method is relatively easy. However, it is very time-consuming. For this incremental technique, a wide range of methods, which depend on the system architecture, are developed.

The first method is top-down. In this approach, software testing is performed from top to bottom. Firstly, software testing starts with control flows such as the GUI or main menu. Then the components are substituted by stubs. The second method is bottom-up. In that approach, software testing is performed from the bottom to control flow upwards. Components are substituted by drivers. The final method is functional incremental. In this method, software testing is performed, based on functions or functionality. These functions can be identified from the software specification document.

There is no formal method to sequence or number components for integration. However, the most important and problematic components should be tested first. Moreover, testers should only concentrate on integration, not functional problems in a component. For example, if there are two components “X” and “Y”, testers should test only the communication between two components, not test the functionality of component “X” or “Y”.

Both black and white box software test case generation techniques can be used during integration testing. Furthermore, non-functional characteristics can also be tested during the integration test.

3.4.3 System Testing

In system testing, the whole system or product is tested. Software test cases for system testing can be generated from software specification documents, business processes, uses cases or other high level descriptions of system document.

System testing is usually the final test that is performed on the developer side. The testing can be performed by test specialists who are dedicated to system testing. In addition, they can work in an independent test team within the organization. In some organizations, a third party can be responsible for system testing.

Both functional and non-functional specifications can be tested by system testing. Besides this, both black and white box test techniques can also be used. However, usually black box test techniques are preferred for the system testing.

Sometimes, a controlled test environment may be required for system testing. Some specific bugs can only be detected when conditions of business environment are satisfied. In order to do this, the program should be tested on all common operating systems with the realistic test data.

3.4.4 Acceptance Testing

After the whole system is tested and most of the bugs are detected and corrected, the system is delivered to users or customers for acceptance. The question of "can the system be achieved?" is answered during the acceptance testing. The acceptance testing is usually performed by users or customers or other stakeholders. Furthermore, this test is performed in a test environment. In the test environment, most factors and conditions are the same as in real business environment.

The main purpose of the acceptance testing is not finding a defect in the program. However, the fundamental aim of acceptance testing is providing enough confidence to show that the system works as expected. The acceptance test may not be the final test. In large projects, integration tests may be applied after the acceptance testing. Acceptance testing may be performed at different levels. These are:

- Commercial off-the-shelf software product can be tested when it is installed or integrated.
- Usability of components can be tested during the component testing.
- Acceptance of new functionality can be tested before the system testing.

Acceptance tests are divided into two categories. The first one is operational acceptance testing. It is also called production acceptance testing. In these tests, users or application managers validate the system to confirm that the system meets the software requirements. In some organizations, system administrators can perform the acceptance testing before releasing the software. Operational acceptance tests include backup/restore testing, disaster recovery and maintenance testing. The second type of the acceptance test is compliance acceptance testing. This test is performed according to acceptance criteria, which are declared in a contract for custom software. Safety, legal and governmental regulations are tested with the compliance accepting testing. For commercial off-the-shelf software (COTS), testing with a single user or customer is not practical. The feedback should be collected from the market before selling the software. There are two steps to achieve this type of tests. The first stage is the alpha testing which is performed by developers. Potential users and members of the developer organization can involve in this test. Furthermore, an independent testing team can also perform the alpha testing. The second stage is beta testing or field testing, which is performed by real users who install the program to try. They use the program under the real-world environment conditions. At the end of this test, users send the incidents' logs and their feedbacks to the developer organization.

3.5 Test Types

At each level of testing, different types of tests can be applied according to the objectives of testing. For example, in a component testing, the functionality and performance of the component are tested in different ways. The test organization can also be changed according to the test objectives.

3.5.1 Functional Testing

The functions of a system or a component are tested in this type of test. The functions are described in requirement specifications, functional specifications or use cases. Moreover, there are implicit requirements that are not documented. These functions should also be tested. Thus, testers should be aware and consider this kind of implicit requirements.

At all testing levels, functional testing can be performed. Functional testing is often referred to as black-box testing. However, non-functional specifications can also be tested by block-box testing.

According to ISO 9126 standards, functional tests include suitability, interoperability, security, accuracy and compliance testing. For example, security testing covers the functions of a firewall which are related to detection of threats, such as virus and worms.

Functional test cases can be derived in two ways. First, test cases can be derived from requirement specifications. This is also called requirement-based testing. The table of content of a requirement specification document can be used to trace the requirements. These requirements should be prioritized, if this was not done. Software test cases should be primarily derived from important requirements, which are identified after the prioritization process. Thus, most of the test efforts should be dedicated to important and critical requirements. The second way of deriving functional test cases is business processes. In this method, test cases are derived from business scenarios. Moreover, use cases are very useful for identifying business processes.

Most of the functional test techniques are specification-based. However, experience-based techniques can also be used for functional testing.

3.5.2 Non-functional Testing

The quality characteristics or non-functional attributes of components or systems are tested in this type of tests. Like functional testing, non-functional testing can be performed at all test levels. Performance testing, load testing, stress testing, usability testing, maintainability testing, reliability testing and portability testing are included in non-functional testing.

Although many software quality characteristics and sub-characteristics are defined by companies and organizations, the International Organization for Standardization (ISO) has defined the quality characteristics of software [ISO/IEC 9126]. According to ISO 9126, six quality characteristics are defined. These quality characteristics are divided into sub-characteristics. ISO 1921 quality characteristics are stated in the following section:

- *Functionality:* This quality characteristic is related to functional testing. The functionality quality characteristic includes five sub-characteristics. These are suitability, accuracy, security, interoperability and compliance.
- *Reliability:* This quality characteristic includes four sub-characteristics. These are maturity, fault-tolerance, recoverability and compliance.

- *Usability:* Understandability, learnability, operability and attractiveness are sub-characteristics of the usability quality characteristic.
- *Efficiency:* Performance, resource utilization and compliance are sub-characteristics of the efficiency quality characteristic.
- *Maintainability:* Analyzability, changeability, stability, testability and compliance are sub-characteristics of the maintainability quality characteristic.
- *Portability:* Adaptability, installability, testability and compliance are sub-characteristics of the portability quality characteristic.

3.5.3 Structural Testing

The structure or architecture of a system or a component is tested by this kind of tests. This type of tests is also called white box or glass box test, because the inside of a component or a system can be analyzed.

Structural testing can be performed at all testing levels. The thoroughness of a component or a system is measured by structural testing methods. The techniques used in structure testing are called structure-based techniques. Usually control flow models are used to support structural testing.

3.5.4 Testing Related to Changes

The final type of tests is the testing of changes. This type of tests is different from the other types. The reason is that, when changing some parts of the software, functional, non-functional or structure of software can be changed. For these changes, different types of tests should be applied.

3.5.4.1 Confirmation Testing

When defects are found during the software testing, these defects should be reported. Then, the new version of software is expected. After these defects are fixed, the fixed parts should be tested again. In this test, only fixed parts are tested to ensure that the defects are fixed. This test is called confirmation testing or re-testing.

In the confirmation tests, all tests should be executed exactly in the same way and with the same inputs. The testing environment should also be the same as the first testing environment. If the confirmation test passes after the amendments, this does not mean the software is correct now. This test only shows the fixed parts are correct or not. The new changes may cause problems with the existing parts which are thought to be correct and passed the previous tests. The side effects of changes can only be tested by regression testing.

3.5.4.2 Regression Testing

Similar to confirmation testing, regression testing involves the execution of tests, which have already been performed. However, in the regression test, not only fixed parts, but also correct parts are tested. In this way, the side effects of changes can be found.

Most software companies and organizations establish a regression test suite or regression test pack to use these in the regression testing. These tests usually include the tests of the important functions. Moreover, these tests are not too detailed, because it takes too much time to cover all functions in each change. For this reason, automation should be required to perform the regression test especially for large-scale projects. It is also possible to execute a subset of test suites according to the project type.

Regression testing is not performed only when the software is changed, but also when the environment of the software is changed. It is a good practice to perform a regression test when some aspects of the software are altered. For example, after the operation system is updated or the database version is changed, regression testing can be performed.

The maintenance of the regression test suite is important, because the software evolves and some parts are changed over time. The new functionalities and removed functions should be covered in the regression test suites. In the long run, regression test cases may become very large. Thus, the manual execution of each test may not be possible. In that case, a subset of regression test suites may be executed. Another approach is that some test cases, which have not found a defect for a long time, can be removed from regression test suites. However, this approach is not usually recommended because some defects may be found.

3.6 Maintenance Testing

When the software is delivered and installed, that system may be used for years or even decades. During this time, the operation environment may change. Thus, the software should be tested periodically to ensure the system still works as expected. This type of tests is called maintenance testing.

Maintenance testing is not different than testing new software. Same methods and test cases can be used in maintenance testing at all test levels. According to the size and type of software, component testing, integration testing, system testing, and acceptance testing can be performed in the same way.

When performing a maintenance test, the latest changes should be tested first. After that, older changes and modifications should be tested. In addition to this, missing parts of software test documents can be completed in this stage. This is called a catching up operation. Missing test cases, which should be derived from software specifications, can be completed during the maintenance test.

Sometimes, there might be no document about the software. In this case, it is very difficult or almost impossible to trace the cause of bugs.

CHAPTER FOUR

TEST DESIGN TECHNIQUES

Software testing is a planned activity. Thus, before the execution of a test, testers should consider inputs and predicted outputs. Furthermore, testers should know how to get ready for the test and how to run the test. In the whole testing process, three main things should be performed and documented. These are the test conditions, test cases and test procedures (scripts).

The test conditions are documented in a test design specification document, the test cases are documented in a test case specification document, and the test procedures or scripts are documented in a test procedure specification document. The software test cases are derived from the test conditions and then translated into the test procedures or scripts.

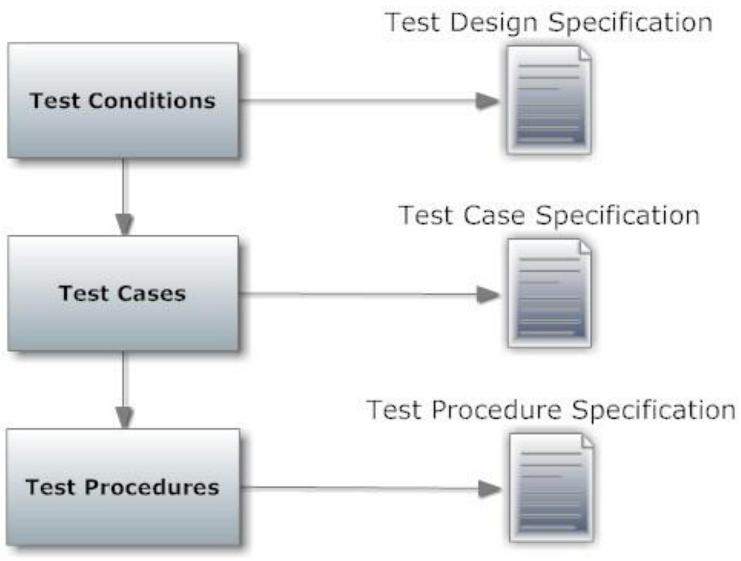


Figure 4.1 Software test documents

4.1 Formality of Test Document

The formality of test documents depends on the type of software, organization, and time and budget constraints. The very formal test involves detailed explanation of inputs and expected outputs. On the other hand, the very informal test may not involve any documentation. However, even in this case, the testers should keep some notes, and have an idea about the expected outputs. In practice, most testers are between two extreme points.

Organization is another factor that influences the formality of documents. The culture, employees' (developers) behavior, the maturity of the development process and the maturity of the testing process are the organization factors that influence the formality of test documents. Time constraints and deadline pressure also affect the formality and thoroughness of test documents.

4.2 Test Analysis: Identifying Test Conditions

Test analysis is a process of looking for the test information that is used to derive test cases. This process is also called a test basis. Test conditions can be found from system requirements, domain requirements, technical requirements, business processes and source codes. Furthermore, some conditions can be derived from experience which are not documented anywhere. Thus, a test basis includes everything that should be tested. For example, when deriving test conditions from a source code, decisions or branches are determined, and test conditions are listed as true or false. On the other hand, when deriving test conditions from a requirement specification document, the table of contents can constitute the list of test conditions.

Test conditions may include a large range of possibilities and may not include exact information. For example, in the university course registration system, when identifying test conditions for the student search facility, one condition includes students who register more than four courses and one condition includes students who are registered by engineering faculty. It may not include the exact name, surname, student number or department and faculty name. On the other hand, they should be stated when generating test cases.

A list of software test case conditions is formed with different names by different test experts. Some experts call it "test requirement" which refers to things that should be tested. Moreover, some experts call it “test inventory” which refers to things that could be tested. Another term which refers to this list is “test objectives”. It is a broader category of test inventory. Furthermore, it means the actual list which needs to be tested.

When identifying test conditions, not all conditions are identified. Only required conditions should be selected and combined into test conditions. This is also called test possibilities. Exhaustive testing, which is trying all input combinations, is also impractical and impossible. Thus, only subsets of all test cases can be applied. In practice, although this subset constitutes only the small part of all test cases, the probability of finding defects is very high. For this reason, the important point is finding the best subset. Some techniques can be used to find this subset. These techniques are called test techniques.

Software test techniques provide guidelines as well as rules to select a good set of tests from all possible tests for a specific system. Each test technique offers a different method of deriving test conditions and test cases. Each test technique handles the system from a different perspective. Thus, each test technique derives different test conditions and test cases. For this reason, more than one test technique should be used in a complementary manner. For example, according to the selected test technique, test conditions might be based on the system model, risk, likely failures, compliance requirement, expert advice or heuristic.

When deriving test conditions, the source of conditions should also be kept for traceability. Traceability can be provided either horizontally or vertically. Horizontal traceability can be achieved by all software test documents at the same test level. Vertical traceability can be performed at different levels in the same test document. Traceability of test conditions is very important, especially for the following reasons:

- If a function or a feature of a component is changed, tests for this function or component should be changed. Traceability can give information about the influenced tests which are caused by these changes.

- If the passed tests cause a problem in the future, the cause of the problem can be detected by finding tested functions, which can be achieved by traceability. Traceability between requirements and tested functions is required to detect affected functions or features.
- Traceability ensures that all specified requirements in the requirement specification are tested. This also provides a view for unimportant and useless test conditions.

After listing the test conditions, these conditions should be prioritized. Because producing a test case from test conditions is an expensive process, it is beneficial to derive the most important test conditions. Finding test conditions is not hard and does not take too much time. Spending extra time on the elimination of test conditions saves a lot of time and effort when generating test cases from test conditions.

Test data such as test inputs and outcomes should be identified from test conditions. For example, data types, number of records, file or database types and sizes are determined by test conditions.

The software test conditions are documented in the test design specification document. This document is defined in IEEE 829 standards. According to IEEE 829 standards, the design specification document includes the following parts:

- **Test design specification identifier:** This is the first part of the design specification document. A unique number may be given by a company or an organization to identify the document. This number is usually related to the software test case specification at the same test level. This part also includes a unique short name for the case. Version date and version number of the case as well as version author and contact information should be covered in this part. At the end of this part, revision history might be given.
- **Features to be tested:** This is the second part of the design specification document. Sets of test objectives are defined in this part. Groups of related

items, which are the overall purpose of documents, are mentioned in this part. Features which are attributes, characteristics and groupings of features are covered in this part. If this document includes more than one test level, it should be determined. Moreover, this document may include a reference to the original documentation, which this test objective (feature) is obtained.

- **Approach refinements:** This is the third part of the document. Necessary details are added to the original approach, which is defined in the test plan. Selected test techniques are defined in this part. Furthermore, the reason for this selection should be explained. An analysis of the used method's results and used tools should be stated in this section. The relationship between test items or features and test levels should be explained. In addition, common information about multiple test cases or procedures should be summarized in this section. Shared environment, common setup or recovery and case dependencies can also be explained in this part.
- **Test identification:** In this section, each test case is identified and shortly explained. This explanation may include the test levels and information about the relationship between software tests. Furthermore, each test procedure with a short description should be stated in this section.
- **Feature pass and fail criteria:** Success criteria for tests are explained to determine whether the feature or set of features pass the test or not.

4.3 Test Design: Specifying Test Cases

Test conditions are sometimes ambiguous and include a large range of probabilities. However, when test cases are determined, the exact data should be known. For example, in the university course registration system, when identifying test cases for a student search facility, the exact inputs should be determined according to the test conditions. When generating test cases, more than one condition may be merged into a single test case.

Test conditions should have an input value or values. However, just determining input values is not enough. What the system is supposed to do with these inputs should also be known. Otherwise, the success or failure of the test cannot be identified. Moreover, test cases should be documented formally with inputs and expected outputs. On the other hand, documentation at all test levels is not mandatory, if the test team has a lot of experience in that field. In this case, only high-level test specifications may be documented.

The most important aspect of testing is determining what the system is supposed to do with given inputs. If the result of inputs is not known, it will not be a test. It shows only what the system does with given inputs. This is also called kiddie testing and cannot count as a test. In order to perform a test, first, information about the correct behavior of the system should be known. This is called an ancient Greek word ‘oracle’ or ‘test oracle'.

When inputs are described, their expected results should also be documented as part of the test case. Expected results may be a text message on the screen, change of data or states, printed document and data signal. If the expected results are not identified exactly, the correctness of outputs may not be identified. For example, if the tested function is a mathematical calculation, very small differences may not be detected. The correctness can only be traced by comparing the expected results with the actual results. In addition to this, the test becomes more objective when comparing the expected results and actual results for given inputs. However, sometimes, the expected results may not be calculated because of the complexity of the process. In these situations, the expected results can be estimated, and a reasonable check can be done. This is also called “partial oracle”. In these situations, wrong results can be identifiable, and the other values which look reasonable can be assumed as correct.

A software test case should also include the test environment, preconditions and post-conditions. The test environment may include the environmental values such as simulation time and temperature. Preconditions are things that should be performed before the test. On the other hand, post-conditions are things that should be performed after the test. Only when these conditions are met that the software test can be assessed correctly and objectively.

The software test cases are documented in the test case specification document. This document is defined in IEEE 829 standards. According to the IEEE 829 standards, the test case specification covers the following sections:

- **Test Case Specification Identifier:** This is a unique identifier which is assigned by the company or organization. A short unique name can be assigned for the test case. The version date and version number of the case as well as the version author and contact information can be declared in the identifier section. The revision history can also be included in this section.
- **Test items:** Tested items and their features in the test case are described. Items should also be referenced from different sources depending on the level of software specification.
- **Input specifications:** In this part, all input values should be identified based on the level of the test case. Inputs do not only include test data, such as values, ranges and sets, but also tables, human actions, conditions, files, and relationships. The file section covers databases, control files and transaction files. Furthermore, the relationship part includes the timing information. The input section can also be represented as table which simply the documentation. A table template can be used to represent input specifications. This template is depicted in Figure 4.2. In this table, "R1" is a valid value rule. "R2" is an invalid value rule. "R3" is a navigation rule between tests. "ATT" is a field for any specific attributes. "Proc" is a procedure to follow for the test case. "Dep" is any elements or test case dependencies.

Case Numbers	Test Element	Valid Values	Valid Response	Invalid Response	R 1	R 2	R 3	ATT	Proc	Dep	

Figure 4.2 Template for input specification in the test case specification document

- **Output specifications:** In this part, the output specifications which are required to verify the test case are described. In addition, the outputs are based on the level of test cases. Similarly, the outputs are not just data. Tables, human actions, conditions, files, relationships, and timing should be included in the output specifications. In the same way, the output specifications can be represented by a table. This table is the same as the input specification table. However, in this table, entries are output values.
- **Environmental needs:** Environmental needs are described in this section. Hardware, software and other applications can be declared in this part. The hardware environmental needs include configurations and limitations. On the other hand, the software environmental needs cover systems, such as compilers, operating systems and related tools.
- **Special procedural requirements:** Special constraints on test cases are defined in this part. This part is usually required especially when more than one test case are declared for these type requirements. Special setups, operational interventions, output locations and identifications, and special wrap-ups are identified in this part.
- **Inter-case dependencies:** Any prerequisite test cases are defined in this part. The relationship between test cases can be defined at the end of each test case.

4.4 Test Case Implementation: Specifying Test Procedures or Scripts

After defining the test cases, these test cases should be grouped and sequenced to run produced tests. For example, simple tests can be grouped as a regression suite. Moreover, the test of complicated risky functions and features can form another group.

Some software tests should be performed with a particular sequence. For example, without adding a student or students into the system delete or modify student function cannot be tested. If the proper order does not provide, the meaning of test can be lost.

After defining the test procedures or test scripts, a test execution schedule should be formed. This schedule specifies the sequence of tests, data and execution times. The tester name may also be declared in the test execution schedule. When developing a test execution schedule, logical and technical dependencies between test scripts should also be considered. For example, the regression test should be performed first, when testing a new realize of software.

Writing the test procedure provides another chance to prioritize software tests, if it has not done when producing the test design specification and the test case specification. When executing tests, risky and scary functions or features should be performed first. The famous rule between testers, which is “Find the scary stuff first”, shows the importance of prioritization. These scary functions or features highly depend on the type of project. The most important functions or features for business are usually counted as scary. Thus, they should be performed first. For example, in the e-commerce web application, money transaction from a bank is one of the most crucial functions, because it may cause loss of money.

Test procedures or scripts are documented in the test procedure specification document. This document is defined in IEEE 829 standards. According to the IEEE 829 standards, the test procedure specification document includes the following sections:

- **Test procedure specification identifier:** It is a unique identifier which is given by the company or organization. The unique short name, whose pattern is similar to other documents, can be given. The version date, version number of procedure, version author, and contact information and revision history are also covered in this part.
- **Purpose: A** list of all test cases covered by the procedure and their descriptions is explained.
- **Special requirements:** Additional requirements are defined in this section. If automation is provided, tool names should be stated. On the other hand, if the

test is executed manually, this should be declared. All test stages (pre-testing, repair and regression testing, future compliance testing, end-to-end testing, re-testing after certification, etc.) should also be defined in this section. In addition, test environments, special skills, required trainings and any prerequisite procedures should be determined and explained.

- **Steps:** Activities associated with the procedure are listed in this part. For example, log, set-up, start, proceed, measure, shutdown, stop and wrap-up are some activities that are used during the execution of the test.

CHAPTER FIVE

TEST CASE DESIGN TECHNIQUES

Exhaustive testing is impossible because of time and cost constraints. For this reason, test cases are always incomplete. The obvious strategy is selecting the subsets of all possible test cases, whose probability of revealing bugs is more than other test cases. Thus, designing a test case becomes more important.

There are three main methods available for test case design. These are specification-based (Black-box techniques), structure-based (White-box techniques) and experience-based techniques. Each method is successful in finding a specific type of errors. However, it may be hard or impossible when finding other types of errors, if only one type of the methods is used. For this reason, these methods are used in a complementary manner. Moreover, some methods are more suitable and effective in some stages of the software development life cycle.

One of the first approaches of test case design techniques is trying random inputs. This method is one type of black box testing. It is also called fuzz testing or fuzzing. In this method, unexpected and invalid inputs are selected randomly as an input. When the program fails with that random input, defects can be found. This method cannot be used as a formal test case design technique. However, it can be used in large projects to increase the quality of software. The probability of finding defects by using this method is very low. On the other hand, robustness testing can be achieved by this method. Especially, when testing file formats and network protocols, it can be effective. However, fuzz testing is not effective for just finding bugs.

In general, the software test case design techniques are divided into three parts as in the figure. Usually, black box test methods are applied first, and then white box test methods are applied.

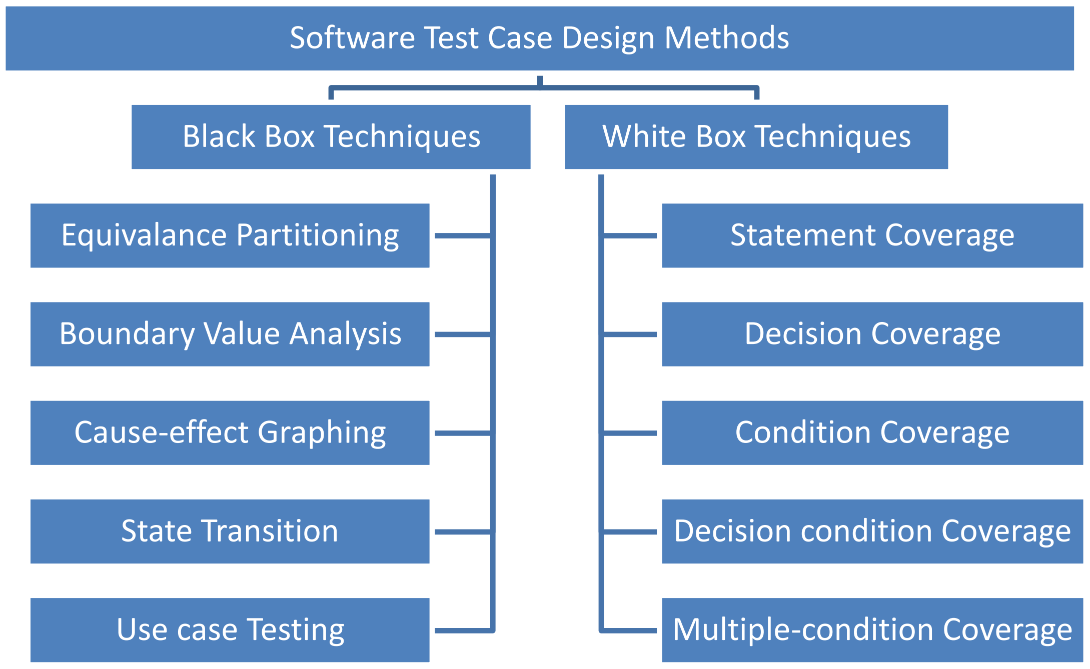


Figure 5.1 Formal software test case design methods

White box test methods are achieved with the knowledge of the internal structure of programs. It is also called glass box testing. The tester can design test cases by using the source code. For example, conditions and loops can be used to produce the test cases. In this case, the functionality of the software may not be considered.

The white box testing is also used at every level of testing like the black box testing. From component testing to integration testing, it can be successfully applied. Moreover, in the acceptance tests, white box methods can also be used. However, the structure is different. The higher level of structures can be used in the acceptance testing.

In the experience-based software test techniques, test cases and conditions are generated by experience. Formal software test case techniques are not used. It is only suitable for very experienced programmers and testers. Because the common defects usually occur in the same domains, previous experiences are very effective in finding common defects.

5.1 Black Box (Specification-based) Software Test Design Techniques

The black box testing methods are based on the software specifications. In this technique, the program is considered as a close box. In addition, it is assumed that the internal structure of the program is not known. Only the inputs and outputs of the systems are known from the requirement specifications. This means the tester focuses on what the system does, instead of how the system does. Both functional and non-functional requirement specifications are tested in this technique. Most formal methods in black box techniques are related with functional requirements. Non-functional requirements, such as performance, usability, portability, and maintainability, are tested less formally than functional requirements by using black box techniques.

The black box testing can be used at every level of software testing from component testing to acceptance testing whenever the software specifications are defined. Especially, the acceptance testing is based on the software specifications. So, the black box testing is very suitable for this kind of tests.

5.1.1 Equivalence Partitioning

This black box approach is one of the basic and well-known testing methods, and is usually performed informally. However, if it is performed formally and systematically, the benefits of this technique will increase dramatically. In this technique, input conditions are divided into equivalence classes. If one test case from one equivalence class has a bug, the other test cases in the same class should also include a bug. In the same way, if one test case from one equivalence class does not have any bugs, the other test cases in that class should not include any bugs. Furthermore, if one representative test case in one class includes a bug and other representatives in the same equivalence class do not include any bugs or vice versa, the equivalence classes are not defined correctly. In this case, the equivalence classes should be defined again or the current equivalence classes may be divided into smaller classes.

Test cases are derived from equivalence classes. So, the number of equivalence classes should be minimized to decrease the number of test cases. On the other hand, the test cases should cover all possible conditions. It is possible to generate more than one test case from one equivalence class, but because of time and cost constraints, usually only one representative is selected. More than one representative can be selected for ensuring the correctness of the equivalence classes, especially for the high-budget and critical systems. Generally, this selection is performed randomly. There is no specific rule developed for selecting representatives from equivalence classes.

Two main steps are required to perform this method. The first step is identifying the equivalence classes. Then software test cases are derived from the equivalence classes. To identify equivalence classes, each input condition should be considered. These conditions are available in the software specification document. At least two classes should be generated for each condition. These two classes are valid equivalence classes and invalid equivalence classes, which represent valid and invalid inputs respectively.

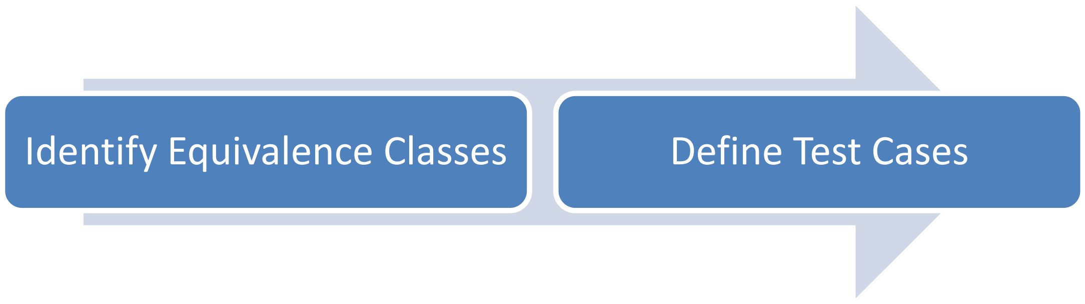


Figure 5.2 Steps of equivalence partitioning software test case technique

5.1.1.1 Identifying the Equivalence Classes

If the input and external conditions are known, finding the equivalence classes becomes a heuristic process. The guidelines for identifying equivalence classes are shown in Figure 5.3.

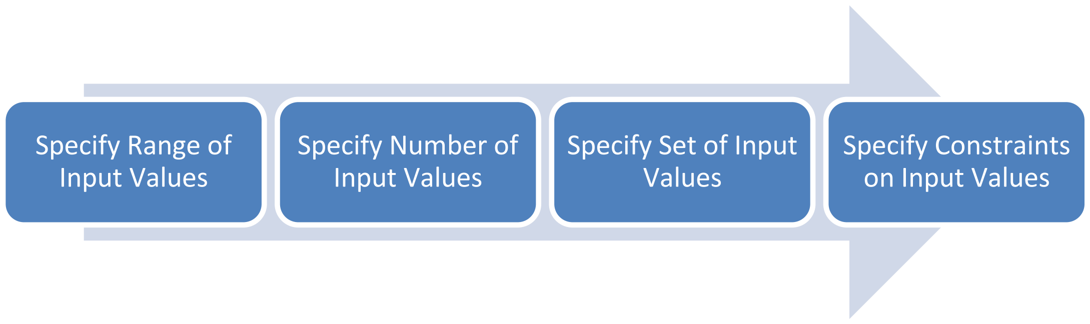

Specify Range of Input Values
Specify Number of Input Values
Specify Set of Input Values
Specify Constraints on Input Values

Figure 5.3 The guideline for identifying equivalence classes

First, the input's ranges are evaluated. If one input falls in a specific range, one equivalence class is identified from a valid range, and also two invalid equivalence classes should be identified, which are less and greater than the range values. For example, in a university course registration system, the system administrator should set the capacity of classes according to the physical capacity of the building and the professors' schedule. That class size input should have a value that is between the minimum and maximum capacity of classes. If the minimum class size is 10 and the maximum class size 50, then one equivalence class should include a valid class size input such as 15 or 25. Then two equivalence classes should be produced for invalid class sizes, one for less than the minimum class size such as 5 or 7 and one for greater than maximum class size such as 70 or 300. These values are selected randomly according to the ranges and conditions. There is no specific rule for selecting these values. On the other hand, in the boundary value analysis, these values are selected according to some rules, which are discussed in the boundary value analysis section.

Secondly, the number of values for a specific input should be considered. In this case, one equivalence class is identified for the valid number of values for a specific input. Moreover, two equivalence classes should be identified, one for no value and one for greater than the maximum number of value. In the university registration system example, each class has a capacity, and students register for classes according to class capacities, which are assigned by the system administrator. When testing student records that are registered for a specific class, one equivalence class should include the valid number of records which is less than the class capacity. The second equivalence class should be the situation of empty record that the number of registered student is zero. The last equivalence class should be the situation which the number of registered students exceeds the class capacity.

The next process is specifying a set of input values. If an input has predefined specific values, then create separate equivalence class for each possible input value. For example, in the university class registration system, users are classified as administrator, instructor and student. Thus, when defining the user type, this value should be one of them. For this reason, at least three different equivalence classes should be defined for administrator, instructor and student.

The final step is identified from the software specifications which are constraints on input values. In this situation, one test case is defined for a valid input which satisfies constraints on the specifications, and one test case is defined for an invalid input which does not satisfy constraints on the specifications. For example, in the university class registration system, students' names must start with a capital letter. So, in that case, one equivalence class should include names that start with a capital letter, and one equivalence class should include names that start with a lower case.

5.1.1.2 Defining Test Cases

After specifying the equivalence classes, test cases are derived from equivalence classes. Similar to identifying test equivalence classes, deriving test cases from equivalence classes is a heuristic process. The guidelines are depicted in Figure 5.4.

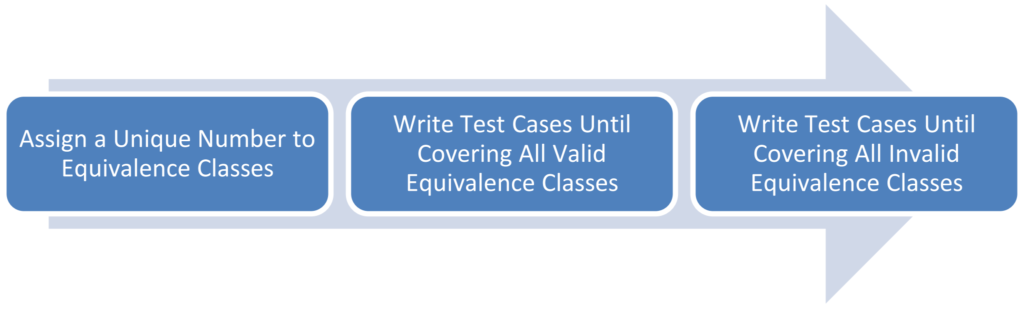

Assign a Unique Number to Equivalence Classes
Write Test Cases Until Covering All Valid Equivalence Classes
Write Test Cases Until Covering All Invalid Equivalence Classes

Figure 5.4 The guideline for identifying test cases

In this process, first of all, each equivalence class is assigned with a unique number. This assignment usually starts with one, and it is incremented for each equivalence class. A specific number may be assigned for specific equivalence classes to facilitate tracing the equivalence classes. Then each equivalence class is converted to test cases. First, valid test equivalence classes are converted to test cases. In this process, test cases are formed to cover more than one equivalence class, if it is possible. This can decrease the number of test cases. If equivalence classes cannot be merged with other equivalence classes, in this case, for each equivalence class, one test case should be generated. After all equivalence classes are covered, invalid equivalence classes are generated. For each invalid equivalence class, one test case should be formed. In testing the critical systems, the number of test cases can increase. When testing critical systems, merging the equivalence classes may not be suitable. Furthermore, more than one test case may be generated to increase the quality of testing.

5.1.2 Boundary Value Analysis

The boundary value analysis is similar to equivalence partitioning. However, when selecting representative in equivalence classes, the values are selected from the edge of equivalence classes. In addition, in the boundary value analysis, not only input space, but also output space should be considered. There is no specific way to achieve boundary analysis, because it requires a degree of creativity. Nevertheless, the general guidelines for boundary value analysis are shown in Figure 5.5.

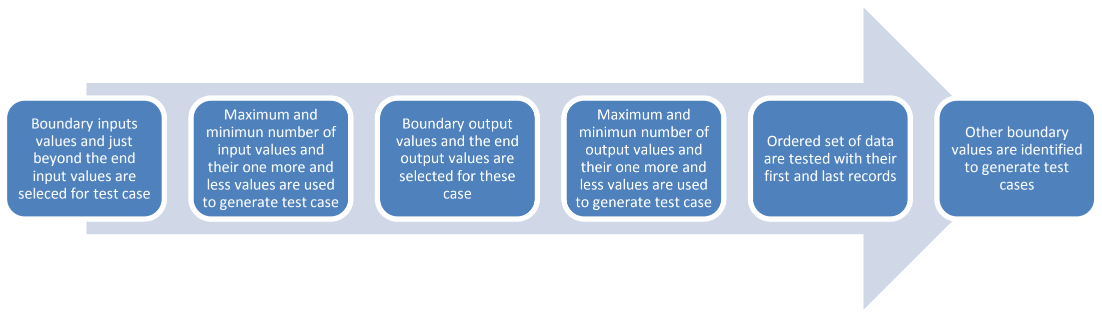

Boundary inputs values and just beyond the end input values are seleced for test case
Maximum and minimun number of input values and their one more and less values are used to generate test case
Boundary output values and the end output values are selected for these case
Maximum and minimun number of output values and their one more and less values are used to generate test case
Ordered set of data are tested with their first and last records
Other boundary values are identified to generate test cases

Figure 5.5 The guideline for boundary value analysis software test case technique

First of all, range values are considered. Then valid input test cases are formed by selecting the edge values. Furthermore, invalid input test cases are generated by selecting input values that are just beyond the boundaries. For example, in the university class registration system, system administrators should set the class sizes. The class sizes are between 10 and 50. In this situation, two test cases are generated for valid inputs that are the boundary values. Thus 10 and 50 values are selected for test cases. Then two invalid input values are selected just beyond the boundary values. In this situation, 11 and 51 are selected.

In the second process, the number of values for inputs is evaluated. The test cases are formed for the minimum and maximum number of values. Then two more test cases are generated, one for one more than the maximum number and one for one less than the minimum number. In our university class registration example, when testing student records that are registered for a specific class, two test cases take 10 and 50 as input values which are boundary values. Then one test case is produced for 9 students which is less than the minimum capacity, and also one test case is generated for 51 students which is greater than the maximum capacity.

In the next process, output ranges are identified. This process is similar to the first process, but this time the outputs' ranges are evaluated instead of the inputs' ranges. For example, in the university grading system, each numeric score has an equivalent letter score like in Table 5.1.

Table 5.1 University grading system, scores and their corresponding letter scores

SCORE	GRADE
90-100	AA
85-89	BA
80-84	BB
75-79	CB
70-74	CC
65-69	DC
60-64	DD
50-59	FD
49-0	FF

Each numeric score corresponds to a letter between AA to FF. The maximum score is AA and the minimum score is FF. When developing test cases, two test cases should cause to get output boundary values, which are AA and FF. Then two more test cases should be generated, one for greater than AA and one for less than FF. Although it is impossible, one greater than 100 and one negative score should be tried as inputs to test the output responses.

The next process is related with the number of output values. If inputs cause a change in the number of outputs, this step should proceed. Most information retrieval systems fall in this category. In these cases, two test cases are formed to produce the minimum and maximum number of records. Furthermore, one test case should be generated for the empty record, and one test case should be generated to produce erroneous output. For example, when system administrators try to display the class list of a specific class, the output will be a student list that should include 0 to 50 records. When developing test cases in this example, one class which nobody registered for is selected for the test case. Then the class, which 50 students registered for is selected for the test case to test the maximum value. Then an invalid class is selected to test erroneous outputs.

The other test cases are formed, when the inputs and outputs of the program are ordered set of data. This can be a table, list or sequential file. When generating a test case, the first and last elements of the data set are selected. For example, when a student list is added to the system, the first student and the last student in the list should be tested.

The last step is identifying the other boundaries which are usually domain-specific. They cannot be automatically identified. This can be derived from system and domain specifications.

5.1.3 Cause-effect Graphing

This method was developed because of the weaknesses of equivalence partitioning and boundary value analysis. Both methods do not cover the combination of input circumstances. Two test cases may not cause an error when performing individually. However, they may cause a problem when two test conditions are combined. In addition to this, non-functional constraints can be tested effectively by this method.

If the only combination of inputs is considered for test cases, huge numbers of combinations are produced even for a simple program. If other non-functional constraints are considered with the input conditions, combinations become astronomical. Thus, a systematic way of combination should be required to handle this process. Otherwise, arbitrarily selected combination of conditions will be ineffective. The cause-effect graphing method provides a systematic way to produce test cases from test conditions. The guidelines for the cause-effect graphing method are shown in Figure 5.6.

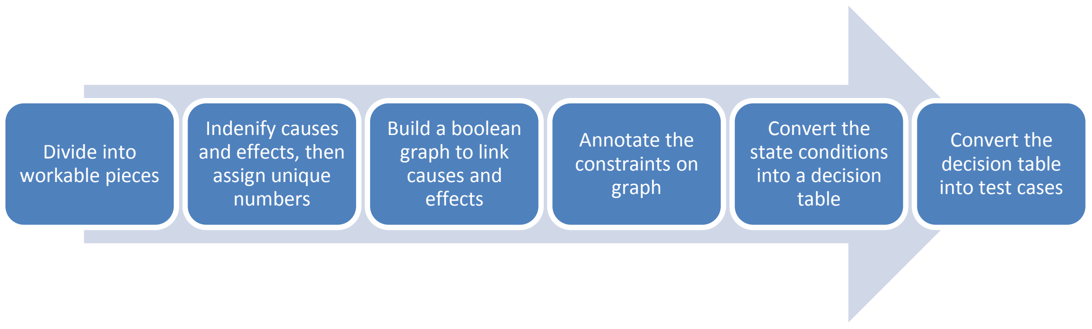

Divide into workable pieces
Indenify causes and effects, then assign unique numbers
Build a boolean graph to link causes and effects
Annotate the constraints on graph
Convert the state conditions into a decision table
Convert the decision table into test cases

Figure 5.6 The guideline for the cause-effect graphing software test case method

In the first process, the system is decomposed into logical sub-systems. The reason is that applying the cause-effect graph to the whole system is cumbersome and not manageable. Then each logical sub-system is decomposed into functions. Furthermore, it is also possible to decompose functions into sub-functions.

The second process is the identifying the causes and effects. The causes refer to inputs, and effects refer to outputs. The causes are distinct input conditions and equivalence classes. The effects are output conditions and system transformations. The causes are identified from the software specifications. Each key word and phrase in the software specification might be an input. The effects can include hardware events, API calls and return codes. The effects can include output actions, an equivalence class of output conditions and interaction messages, such as warning and error messages. Any outputs from an applicant can be an effect, such as a printed document or hardware command. Then a unique number is assigned for each cause and effect.

For example, in the university course registration system, when assigning buildings to courses, the faculty and the number of registered students should be considered. Assume that there are 4 buildings, which are A, B, C and D blocks; also there are two faculties, which are the Faculty of Engineering and the Faculty of Arts and Sciences. The following specifications are declared in the software specification document.

- R0101 If the number of registered students for a course is less than 10 in the Engineering Faculty, the course building will be A block
- R0102 If the number of registered students for a course is between 10 and 50 in Engineering Faculty, the course building will be B block
- R0103 If the number of registered students for a course is less than 10 in Arts and Sciences Faculty, the course building will be B block
- R0104 If the number of registered students for a course is between 10 and 50 in Arts and Sciences Faculty, the course building will be C block
- R0105 If the number of registered students for a course is greater than 50 both for the Engineering and Arts and Sciences Faculties, the course building will be D block

Table 5.2 Causes and effects and their assigned numbers for the university course registration system

Causes (input conditions)	Effects (output conditions)
1. Faculty is Engineering	100. Building is A block
2. Faculty is Art and Science	101. Building is B block
3. Registered student number < 10	102. Building is C block
4. Registered student number >= 10 and <50	103. Building is D block
5. Registered student number > 50	

According to these simple specifications, the causes and effects will be like in Table 5.2. The next process is building the Boolean graph to link causes and effects. The semantics of the software specifications should be analyzed to identify causes and effects. To a draw cause-effect graph, some conventional notations, which are depicted in Figure 5.7, are used.

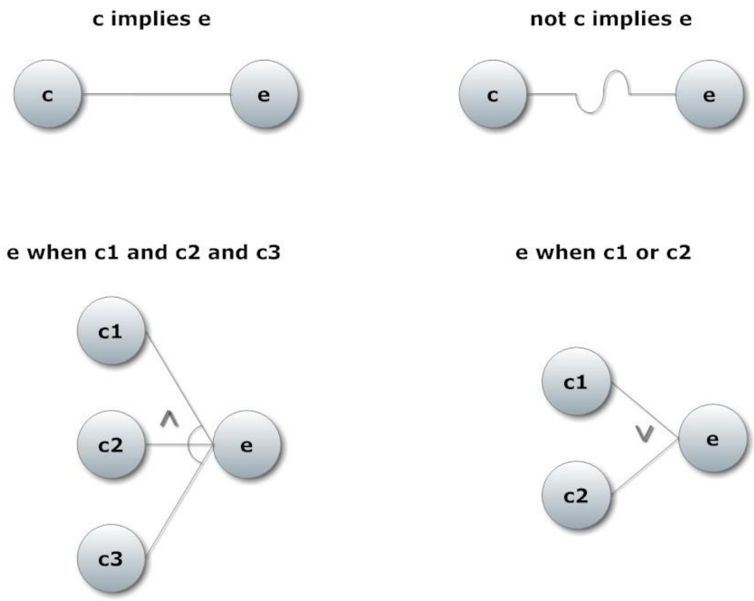


Figure 5.7 Notation of cause-effect graphing software test case design method

In Figure 5.8, separate cause-effect graphs are derived from previous software specifications. Although all nodes can be represented in one graph, they can be separated.

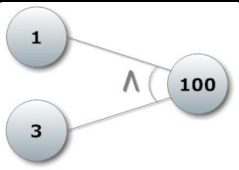
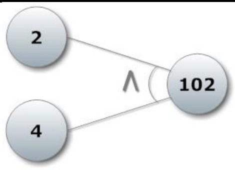
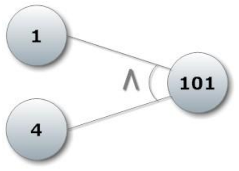
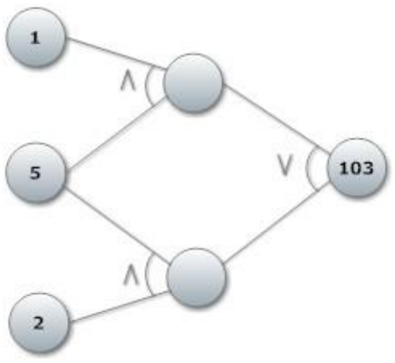
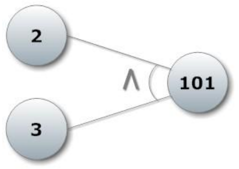
	Cause: 1. Faculty of Engineering And (∧) 3. Number of registered students < 10 Effect: 100. Building is A block		Cause: 2. Faculty of Art and Science And (∧) 4. Number of registered students >=10 and <50 Effect: 102. Building is C block
	Cause: 1. Faculty of Engineering And (∧) 4. Number of registered students >=10 and <50 Effect: 101. Building is B block		Cause: 1. Faculty of Engineering And (∧) 5. Number of registered students > 50 Or (V) 2. Faculty of Art and Science And (∧) 5. Number of registered students > 50 Effect: 103. Building is D block
	Cause: 2. Faculty of Art and Science And (∧) 3. Number of registered students < 10 Effect: 101. Building is B block		

Figure 5.8 The cause-effect graphs of the university course registration system

After building these Boolean graphs, constraints on them should be defined. Because some combinations are impossible, this should be stated in the graphs. For example, a student cannot be registered by two faculties simultaneously (Assuming that double major is not available in this university) or the class size cannot be greater than 10 or less than 10 simultaneously. The notations for cause-effect constraints and their descriptions are explained in Figure 5.9.

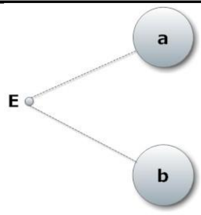
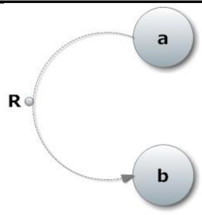
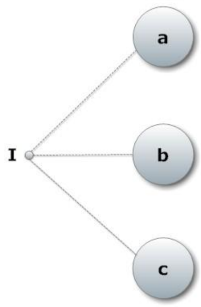
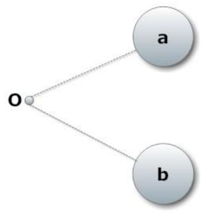
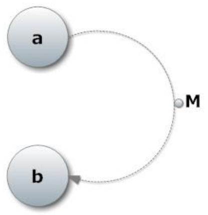
Constraint	Interpretation	Constraint	Interpretation
	‘E’ constraint stands for exclusive constraint. In this constraint, at most one of these nodes can be true. This means node ‘a’ and node ‘b’ cannot be true simultaneously.		‘R’ constraint stands for requires. If cause a is true, then cause b must be true. If node ‘a’ is true, node ‘b’ cannot be false
	‘I’ constraint stands for inclusive. In this constraint, at least one of causes is true. This means that all causes cannot be false simultaneously.		
	‘O’ constraint stands for one and only one. If node ‘a’ is true, then node ‘b’ cannot be true.		‘M’ constraint is for effects. M’ stands for mask. If node ‘a’ is true, then node ‘b’ must be false.

Figure 5.9 Constraint notations and their descriptions in the cause-effect graphing software test case design method

According to the previous cause-effect example, only E constraint can be applied to the fifth cause-effect graph. The following graph shows constraint on causes 1 and 2. According to the software specifications, one student cannot be a student at both Engineering and Arts and Sciences Faculties.

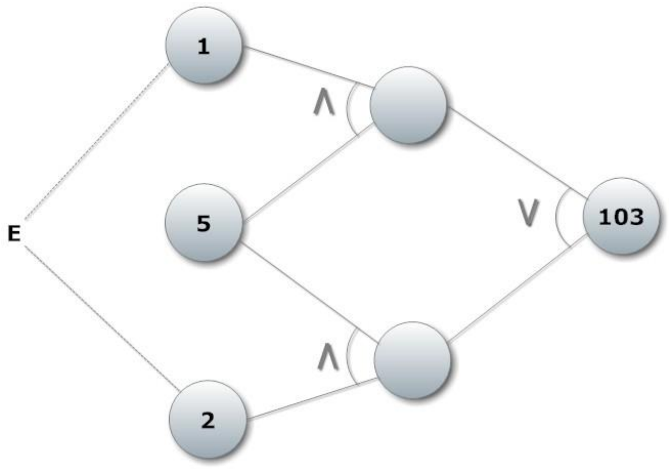


Figure 5.10 ‘E’ constraint on cause 1 and 2 in the university course registration system

In the next step, cause-effect graphs with their constraints are converted to a limited entry decision table. This is also a heuristic process. In order to do this, one effect is selected. Then, all causes, which make this effect true, are determined by back tracking. Each factor is determined in one column of the decision table. Thus, each column represents a different test case. For example, in the first cause-effect graph, the effect 100 is true, when both cause 1 and cause 3 are true. For this reason, in the first column, cause 1 and cause 3 are 1 and other causes are 0. Moreover, the effect 100 is 1 and other effects are 0. The full decision table is depicted in Table 5.3. Finally, in the last process, a limited decision table is converted to test cases. This is shown in Table 5.4.

Table 5.3 The decision table for the university course registration system

Test Case	**1**	**2**	**3**	**4**	**5**	**6**
Causes:						
1. Engineering	**1**	**1**	**0**	**0**	**1**	**0**
2. Art and Science	**0**	**0**	**1**	**1**	**0**	**1**
3. < 10	**1**	**0**	**1**	**0**	**0**	**0**
4. >= 10 and <50	**0**	**1**	**0**	**1**	**0**	**0**
5. > 50	**0**	**0**	**0**	**0**	**1**	**1**
Effects:						
100. A block	**1**	**0**	**0**	**0**	**0**	**0**
101. B block	**0**	**1**	**1**	**0**	**0**	**0**
102. C block	**0**	**0**	**0**	**1**	**0**	**0**
103. D block	**0**	**0**	**0**	**0**	**1**	**1**

Table 5.4 Test cases for the university course registration system

Test Cases	**Input (Causes)**		**Expected Output (Effects)**
	Faculty	Class Size	
1	Engineering	< 10	A block
2	Engineering	>= 10 and <50	B block
3	Art and Science	< 10	B block
4	Art and Science	>= 10 and <50	C block
5	Engineering	> 50	D block
6	Art and Science	> 50	D block

5.1.4 State Transition Testing

If a system can be defined as a finite state machine, this software test case design method can be used. Real-time systems usually fall in this category. This kind of systems can be defined with the finite number of different states. Transition from one state to another state is achieved by the rules of machine. Each state takes an input and performs some actions according to inputs; outputs may change according to previous states. State diagrams are usually used to show these systems. This model includes four main components. These are:

1. States of software
2. Transition between states
3. Events that cause transition
4. Actions that are results of transition

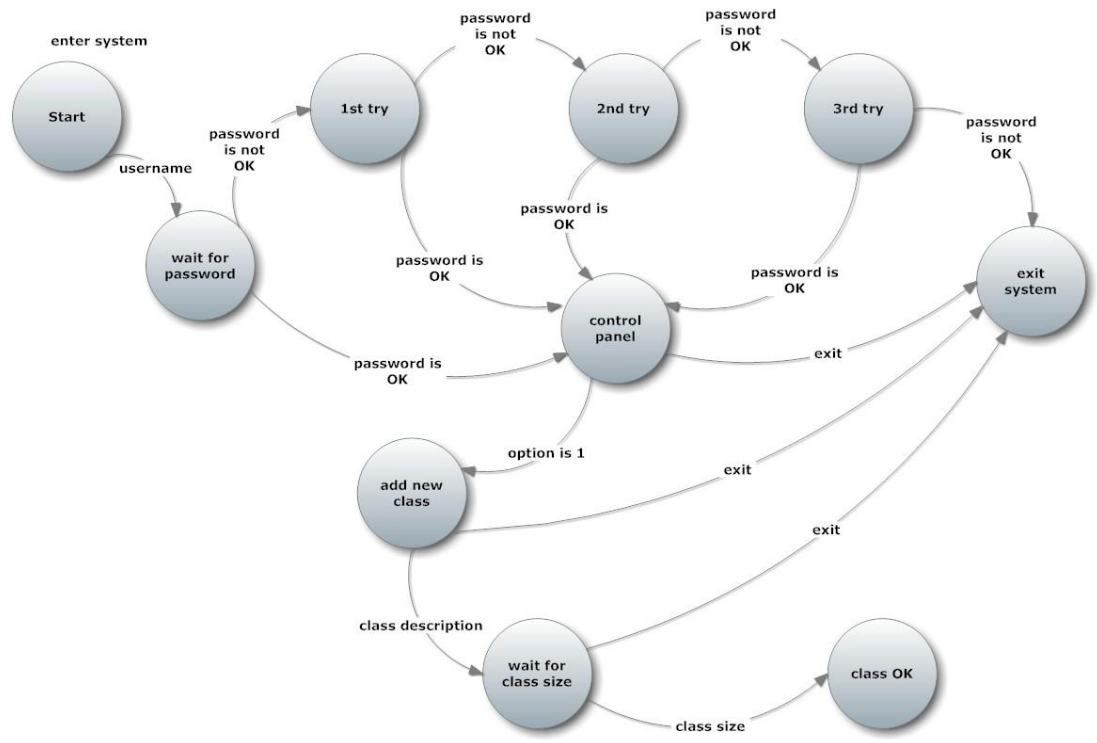


Figure 5.11 The state diagram of the university class registration system

For example, Figure 5.11 shows a part of the university class registration system. The system starts with the start state, which is the first interface of the system. In this state, the system administrator should enter the username. Then the system waits for password input. If a correct password is entered, then the system is transformed to a control panel. Otherwise, system is transformed 1st try. If in this state the correct password is entered, again the system is transformed to a control panel. On the other hand, if the password is not correct, the system is transformed to 2nd try. If the user enters a wrong password three times, the system is transformed to an exit system state, which is a dead-end. In the control panel state, the input has two possible values. The first option changes the current state to an "add new class" state, and the second option changes the current state to an "exit system" state. In the "add new class" state, when entering the class description such as the location and the name of the class, the system starts waiting for the class size input. When a valid class size is entered, the system is transformed to a "class OK" state, which is the end state for this state diagram.

5.1.4.1 Testing for valid transitions

When developing test cases by using the state transition method, valid transitions are executed first. In addition, when deriving test cases, typical scenarios are selected first. For the current example, if correct a password is entered and then the valid class description and class size are provided, the system is ended with a "class OK" state. Then the second test case is derived by entering an incorrect password three times to reach an "exit system" state. Although all states are executed, not all conditions are tested. In order to do this, the correct password is entered in the 1st try state and other transitions are achieved like in the first case. This process is repeated for 2nd and 3rd try states. Furthermore, each exit condition after the "control panel" state should be tested. Thus, separate test cases should be created for each condition.

Many methods are developed to derive test cases from a state transition graph. Each state as well as transition can cause a test condition. One technique is used to coverage set of test cases in the terms of transition. According to this technique, individual transitions are 0-switch coverage. Pairs of transitions are 1-switch coverage and triple transitions are 2-switch coverage.

With the state transition method, the system can be modeled as abstract as needed. Testers can elaborate the model according to needs. For example, some parts of systems are not important as others. Thus, an unimportant part may be modeled as more abstract, and an important or critical part may be modeled as more detailed.

5.1.4.2 Testing for Invalid Transitions

Deriving test cases from a state transition graph is easy for valid transitions. However, when considering invalid transitions, they are not easy as valid transitions. For this reason, a state table is used both for valid and invalid transitions to see the total number of combinations.

In the state table, all states are listed on one side (first column) and events that cause a transition are listed on the top side (first row). The intersection cells show the state when the event is applied to a corresponding state. In the university class registration example, the state table is derived from a state diagram like in Table 5.5.

Table 5.5 The state table for the university class registration system

		Username	**Password is OK**	**Password is not OK**	**Option is 1**	**Class description**	**Class size**	**Exit**
S1	Start	S2	-	-	-	-	-	-
S2	Wait for password	-	S6	S3	-	-	-	-
S3	1st try	-	S6	S4	-	-	-	-
S4	2nd try	-	S6	S5	-	-	-	-
S5	3rd try	-	S6	S10	-	-	-	-
S6	Control panel	-	-	-	S7	-	-	S10
S7	Add new class	-	-	-	-	S8	-	S10
S8	Wait for class size	-	-	-	-	-	S9	S10
S9	Class OK	-	-	-	-	-	-	-
S10	Exit system	-	-	-	-	-	-	-

In Table 5.5, both valid and invalid (negative) transitions can be seen. Although, most of the transitions are impossible, it provides a systematic way of deriving these transitions as well as test cases.

5.1.5 Use Case Testing

Use cases can show the system transaction by transaction from start to finish. Moreover, they show the interaction between actors and systems. The actors are usually user, but systems or subsystem may be an actor.

In the use case, systems are defined in terms of actors, not systems. Thus, inputs and outputs are not involved in use cases. It only shows the interaction of actors with the system. Moreover, usually business language and terms are used rather than complex terms. For this reason, it facilitates the understandability of the system.

Generating test cases from use cases is very practical, because use cases show the real-world business processes. In this way, defects in most used parts can be identified. Furthermore, use case testing is usually used at system and acceptance testing levels. Incorrect interaction between components can also be tested at the integration test level.

5.2 White Box (Structure-based) Software Test Design Techniques

The white box testing techniques have two main purposes. The first one is measuring the coverage of specification-based (black box) test techniques. The second one is developing new test cases, which are not covered by specification-based test techniques.

The white box software test design techniques provide a very effective way to generate test cases, which are sometimes impossible when using only black box software test design techniques. Impossible or hardly possible conditions can be tested by white box testing. By applying both white and black box techniques in a complementary manner, almost all documents and source codes can be covered.

The most important disadvantage of the white box software test techniques is that they cannot find the deficits which are written in software specifications or requirements documents. With white box testing, only existing documents or source codes can be tested. Thus, non-implemented documents and source codes cannot be tested. On the other hand, non-implemented documents and source codes can be tested by specification-based techniques.

Most white box techniques use the coverage term to measure the success of the test and its efficiency. Moreover, not only white box techniques, but also black box techniques can use the coverage term to measure the success of the test.

5.2.1 What is Test Coverage?

Test coverage shows the amount of testing performed according to specific test criteria. When counting things, such as statements or decisions, to determine whether these things are tested or not, the coverage is measured by finding the rate of tested things to all things. In practice, all countable things, such as statements, conditions, decisions or interfaces, can be a coverage item. The basic test coverage can be measured by the following formula:

$$Coverage = \frac{\text{Number of coverage items exercised}}{\text{Total number of coverage items}}\, x\ \%100$$

It is important to note that %100 coverage does not mean %100 tested. Furthermore, different test cases can achieve the same coverage by finding different bugs. Also %100 coverage may not find any bugs, although there are many obvious errors.

Test coverage can be measured at all test levels. For example, at the integration level, coverage measures can include interfaces and specific interactions. On the other hand, at the acceptance level coverage measures can include screens, menu options and business transactions.

Besides the structure-based techniques, test coverage for specification-based techniques can also be measured. For example, EP (Equivalence Partitioning) coverage shows the percentage of exercised equivalence classes from all equivalence classes. In addition to this, EP coverage can be measured both for valid and invalid equivalence classes. In the same way, BVA (Boundary Value Analysis) coverage represents the percentage of exercised boundary values, and decision table coverage shows the percentage of exercised business rules or decision table columns. Moreover, for the state transition testing, there are numerous possible coverage measures. The percentage of visited states, the percentage of exercised valid transitions (Chows's 0-switch coverage), the percentage of the pair of exercised valid transitions (Chows's 1-switch coverage) and the percentage of exercised invalid transitions are some coverage measures for the state transition testing.

According to business analysts, system testers and users, test coverage refers to the percentage of requirements that has been tested by a set of test. However, test coverage mostly refers to the coverage of code among programmers. For this reason, in the context of white box testing, mostly code coverage techniques are discussed. Moreover, a lot of code coverage metrics are developed and used. Furthermore, some of them are formally defined and standardized by organizations. For example, the U.S. Department of Transportation Federal Aviation Administration (FAA) has formal requirements for the structural coverage in the certification of safety-critical airborne systems. (Steve Cornett, 2010)

Test coverage is usually calculated by special software test tools. These tools increase both the quality and productivity of software tests. Most of the defects on independent paths can be found by these tools. Especially for safety-critical systems, code coverage tools should be applied, after the structure-based and experience-based software testing.

5.2.2 Statement-Coverage Testing

The white box text method involves analyzing the logic or source code of the program. The ultimate goal of white box testing is testing each path of the program. However, due to the complexity of programs, testing each path is almost impossible even for a very small program. For this reason, testers try to execute every statement at least once instead of executing all possible paths. The basic statement coverage can be measured by the following formula:

$$Statement\ coverage = \frac{\text{Number of statements exercised}}{\text{Total number of statements}} x\ \%100$$

In this statement coverage metric, each line might count as a statement or one statement might be spread in several lines. In addition, one line might include more than one statement. For this reason, statement coverage is sometimes called line coverage or segment coverage (Ntafos, 1988). Besides this, basic block coverage methods sometimes refer to statement coverage methods. However, in this case, all branches should be executed at least once. For example, in the “if-else” statement code block, two branches should be executed.

In this approach, all executable statements which are considered as declarative statements should be executed at least once. If a statement or statements are covered by control flow statements such as “if” or a “switch” statement, test cases should be generated to cover all conditionally covered statements. For example, for C programming language, test cases should be generated to satisfy the “if”, “while” and “for” condition statements. Moreover, for the “if-else” statement, at least one more test case must be generated to satisfy %100 coverage. The reason is that, if the test condition satisfies the “if” condition, the “else” statement is not executed. In the same way, the “else” condition can be executed only when the “if” condition is false. Thus, at least two test cases should be generated. One satisfies the “if” condition and another does not. Furthermore, each “case” statement in a “switch” statement causes the generation of a new test case. Because only one “case” statement satisfies the “switch” condition at a time, different test cases should be generated for each “case” condition to achieve %100 coverage. Moreover, if the “switch” condition includes the “default” statement, an extra test case, which does not satisfy all the “case” conditions in the “switch” statement, should be generated. If a condition statement does not include any executable statements, an extra test case may not be generated for this condition.

Statement coverage does not check whether loops are terminated or not. Only one time execution of a body part of a loop is enough for statement coverage. Some condition statements check the condition after the execution of code. This means that the conditionally covered code is executed at least once, regardless of the conditions’ outcome. For example, the “do-while” statement falls in this category. For this reason, when measuring statement coverage, “do-while” statements can be omitted. Lastly, the logical “AND” and “OR” operators in the condition statements are insensitive to statement coverage measures.

Testing all statements usually does not figure out important bugs. For this reason, coverage analysis may be considered useless for some types of projects. For example, in the following C# code, all statements can be executed with only one test case. If 5, 8 and 9 integer values are given to ‘a’, ‘b’ and ‘c’ respectively, %100 statement coverage is provided.

Test Case: a = 5, b = 8 and c =9.

```
public funct (int a, int b, int c)
            {
               if ( a>2 || b<10)
               {
                   c = c + 2;
               }
               if ( a>4 && c<10)
               {
                   b = b + 2;
               }
            }
```

With this test case, all statements are executed once, and it achieves %100 coverage. However, if there is an error in these condition statements, such as “or” condition should be “and” in the first “if” condition, this error cannot be detected. Moreover, the second “if” condition depends on the ‘c’ variable whose value can change in the first “if” condition. So, the variation of ‘c’ values should be tested. Furthermore, there is a condition which ‘b’ and ‘c’ are not changed. This condition should also be tested. As a result, the most important conditions cannot be tested by this method. Thus, the strategy of executing every statement is useless.

5.2.3 Decision-Coverage Testing

This technique is stronger than statement coverage testing. In this method, each decision condition should be tested. Thus, different test cases should be generated for each condition output. All branches of the “if-else”, “do-while”, “switch” and multiple “go” statements can be tested by this method. For example, for the “if” statement, one test case is generated for the true output and also one test case is generated for the false output.

The decision coverage testing usually covers the statement coverage testing. When executing all branch conditions, all statements are executed. There are some exceptions. For example, if there is no decision in the program or if the program has a multiple entry point and program is called from one of these entry points, decision coverage testing may not cover all statements in the program. Furthermore, statements in all ON-units (statements that are executed before the condition) cannot be executed when traversing all branch directions. The basic decision coverage can be calculated by the following formula:

$$Decision\ coverage = \frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} x\ \%100$$

For two-way decisions or branches, two test cases are derived both for true and false outcomes. For example, for a simple “if” condition, there are only two results. If the result is true, the statements in the “if” condition are executed. Otherwise, the statements in the “if” condition are not executed. For multi-way decisions or branches such as select (case) statements, test cases are generated for each branch. For example, according to the following C# “switch” condition, at least four test cases should be generated. The test value for the ‘a’ variable should be 1, 2, 3 and another value, which is different from 1, 2 and 3, to satisfy the “default” condition. To achieve %100 decision coverage, the following test cases might be applied:

Test Case 1: a = 1, b = 1
Test Case 2: a = 2, b = 1
Test Case 3: a = 3, b = 1
Test Case 4: a = 10, b = 1

```
public funct(int a , int b)
        {
            switch(a)
            {
              case 1:
                  b = b + 1;
                  break;

              case 2:
                  b = b + 2;
                  break;

              case 3:
                 b = b + 3;
                 break;

             default:
                 b = b + 4;
                 break;
            }
        }
```

It is possible that different test cases can satisfy the %100 decision coverage. In this case, the test value for the ‘b’ variable can be randomly selected with its valid range.

In the decision coverage, control structures are evaluated as true or false. The logical operators are not separately considered. Moreover, the definition FAA, the Boolean expressions in the control structures should be evaluated both for true and false outcomes, whether the control structures affect the control flows or not (Steve Cornett, 2010).

Another disadvantage of decision coverage is that a branch in Boolean expressions may be omitted because of short-circuit operators. For example, if one term is false in ANDed Boolean expression, the outcome becomes false, regardless of the other ANDed terms. Similarly, if one term is true an in ORed Boolean expression, the outcome becomes true, regardless of the other ORed terms. Thus, errors may be overlooked in the omitted parts.

5.2.4 Condition-Coverage Testing

Condition coverage testing is stronger than decision coverage, because errors in condition statements cannot be detected by decision coverage. In this method, a sufficient number of test cases are generated for each condition in a decision. However, in that situation, all statements may not be traced. For this reason, each entry point of the program, subroutine and on-units should be invoked at least once.

For example, according the following C# code, in the first "if" statement, there are two conditions. These conditions are "a > 2" and "b < 10". In the second "if" decision, again there are two decisions which are "a > 4" and "c < 10".

```
public funct (int a, int b, int c)
         {
            if ( a>2 || b<10)
            {
                c = c + 2;
            }
            if ( a>4 && c<10)
            {
                b = b + 2;
            }
         }
```

When deriving test cases, each decision condition should be considered. Moreover, all test conditions should be covered in the test cases. For this reason, in the first "if" decision "a >2", "a <=2", "b <10" and "b >=10" conditions should be covered. Furthermore, in the second "if" decision "a >4", "a <=4", "c <10" and "c>=10" conditions should also be covered. At least two test cases should be required to cover all conditions. For example, the following test cases can satisfy all conditions:

Test Case 1: a = 5, b = 9, c = 8
Test Case 2: a = 1, b = 12, c = 20

The first test case satisfies the conditions of "a >2", "a >4", "b <10" and "c<10". The second test case satisfies the conditions of "a <=2", "a <=4", "b >=10" and "c>=10". In this example, all statements are executed, if only test case 1 is used.

However, all statements may not be executed in the condition coverage testing. For example, in the following code, the decision has two conditions.

```
if ( a && b)
{
  c = c + 2;
}
```

The ‘a’ and ‘b’ have Boolean values. Thus, the value of the ‘a’ has two possible values, which are true and false. Likewise, the ‘b’ has two possible values. If the following test cases are selected, statements in the “if” condition are not executed.

Test Case 1: a = true, b = false
Test Case 2: a = false, b = true

In the test case 1, when the ‘a’ is true and the ‘b’ is false, the output of decision will be false. Similarly, when the ‘a’ is false and the ‘b’ is true, the output “if” decision will be false. Thus, in both cases, all statements are not executed. As a result, the condition coverage criterion does not always satisfy the decision condition coverage.

5.2.5 Decision / Condition Coverage Testing

Because of the weaknesses of decision coverage and condition coverage, a decision/condition coverage method is developed. In the decision coverage, sufficient test cases are generated, so that all conditions in the decisions as well as all outputs of decisions should be tested. Furthermore, all entry points should be invoked at least once.

The weakness of decision/condition coverage is that certain conditions may mask other conditions. For example, in the AND operation, one false term makes the output false. Likewise, in the OR operation, one true term makes the output true. For example, compilers break multiple decisions into individual decisions, because usually multiple conditions cannot be represented as a single instruction in some machines. For instance, the logic diagram of the following C# code is represented in Figure 5.12.

```
public funct (int a, int b, int c)
            {
               if ( a>2 || b<10)
               {
                   c = c + 2;
               }
               if ( a>4 && c<10)
               {
                   b = b + 2;
               }
            }
```

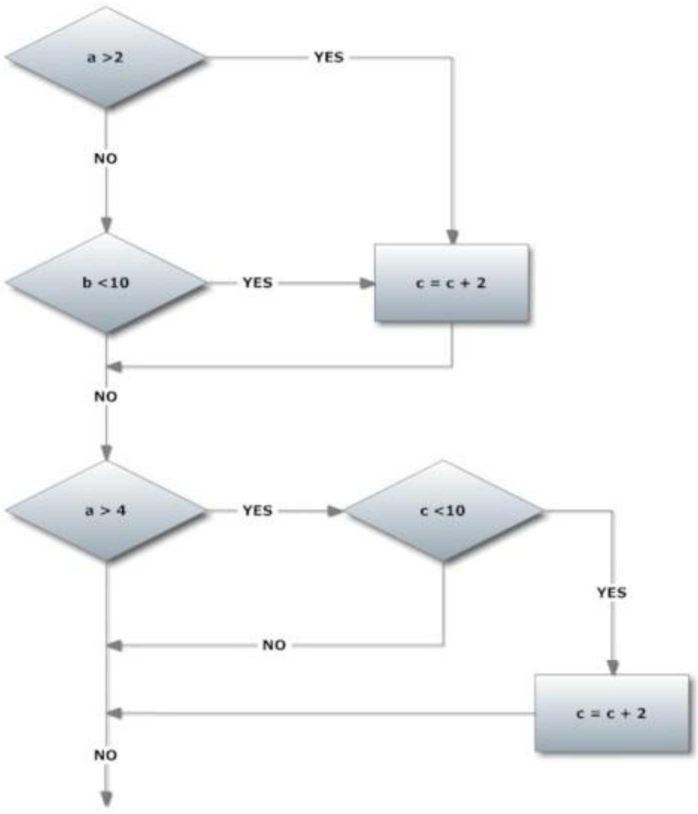


Figure 5.12 The logic diagram of the C# code which describes the logic of compilers

5.2.6 Multiple Condition Coverage

The problem of masking conditions is solved in the multiple condition coverage. In this test, a sufficient number of test cases are generated to handle all combinations of conditions in each decision. In addition to this, all multiple entry points should be tested at least once.

According to the previous C# source code, test cases are generated from the combination of conditions in each decision. Thus, at least the following test cases should be generated.

For the first decision:

Test Case 1: a > 2, b <10 → a = 3, b=9

Test Case 2: a > 2, b >=10 → a = 3, b=11

Test Case 3: a <= 2, b <10 → a = 1, b=9

Test Case 4: a <= 2, b >=10 → a = 1, b=11

For the second decision:

Test Case 5: a > 4, c <10 → a = 5, c=9

Test Case 6: a > 4, c >=10 → a = 5, c=11

Test Case 7: a <= 4, c <10 → a = 1, c=9

Test Case 8: a <= 4, c >=10 → a = 1, c=1

5.3 Experience-based Software Test Design Techniques

The aim of software testing design and test case design techniques is a providing systematic way to derive test cases and find software bugs. However, experience-based testing techniques do not provide systematic methods. They usually depend on the person's experience, knowledge, intuition and imagination.

Most of the time, experience-based techniques are very useful, because some defects may not be found easily by systematic methods. In addition, some defects can only be known by experienced testers and developers who had worked on a similar type of projects or domains. For this reason, non-systematic methods should be performed in a complementary manner with other systematic methods.

There are two main approaches or informal techniques, which are based on experience-based techniques. These are error guessing and exploratory software testing.

5.3.1 Error Guessing

It is a software testing method that is always achieved after all software testing methods are performed. There is no formal rule for error guessing. The tester guesses situations that cause bugs in the system. For example, the division of zero, entering empty values for inputs, and wrong or empty file inputs can cause a bug. Thus, the tester might try and consider these situations. Moreover, some conditions that are thought as never occurring should be considered as a good test case candidate.

Some people are naturally good at testing and some people become a good tester by gaining experience. For this reason, error guessing should be performed after all formal software testing methods. By applying the formal testing methods, testers gain a better understanding of the system as well as its weaknesses. So, they can produce better test cases to reveal bugs.

It is possible to perform structured error guessing. Software testing requires explicit documentation of inputs and expected outputs. Thus, arbitrary actions to find bugs cannot count as software testing. If all found bugs by error guessing are listed with their inputs and expected outputs, the error guessing method becomes more effective and formal. Moreover, the generated test cases become reusable.

5.3.2 Exploratory Testing

It is a hands-on approach and involves more test execution than test planning. In this method, test cases with inputs, expected outputs and test scripts are not formally written. The aim of this method is learning the system. What works and what does not work can be explored by this method. The test logs may be written during the test execution. If any bugs are found during the execution, these should be noted.

Although exploratory testing is not formal testing, formal testing methods, such as boundary value analysis, can be used during the software testing. Exploratory testing is usually used when there is no, or there is poor, software specification, and the time for software testing is very limited. Furthermore, it can be used in a complementary manner to increase the confidence in the software. So, it might be thought as checking formal testing methods.

CHAPTER SIX

REQUIREMENT-BASED SOFTWARE TESTING FOR CAUSE-EFFECT GRAPH SOFTWARE TEST TECHNIQUE

The objective of requirement-based software testing is determining the problems, when translating the external specifications into internal specifications, which are implemented by source codes or software documents. The quality of software highly depends on the quality of external specifications. More than fifty percent of bugs are caused by poorly specified requirements. Moreover, the cost of fixing these bugs is much more than any other bugs, which occur in the other stages of the software development life cycle.

In most projects, the software specification document is used in the design process without formal testing or with little testing because of time or budget constraints. This may cause problems in the later stages of the software development life cycle.

6.1 Cause-effect Graphing Objectives

The cause-effect graph has two main objectives. The first objective is providing a formal approach to test external specifications. Completeness, consistency and lack of ambiguity are some important factors to test external specifications. The second objective is providing a formal way to design test cases from software specifications. Because the cause-effect graphing method is a black box testing, functions are monitored by executing a source code.

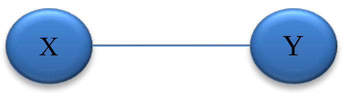
The cause-effect graph software test technique provides a problem solving method which is similar to the decision table. In addition, this technique emphasizes the importance of writing correct external specifications.

6.2 Logical operators in Cause-Effect Graph Software Test Technique

Seven different operators are used in the cause-effect method. These operators are simple, OR, XOR, AND, NOR, NAND and Negation.

6.2.1 Simple

The simple relation denotes that if “cause X” is present, then “effect Y” will happen. Similarly, if “cause X” is absent, then “effect Y” will not happen.



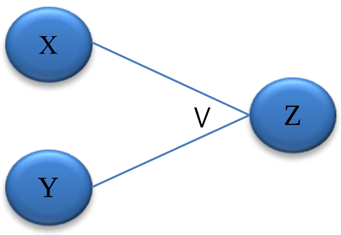
X	Y
T	**T**
F	**F**

Figure 6.1 The notation and truth table of simple relation

In Figure 6.1, X and Y are nodes and the connection line between X and Y is called vector. The logic between X and Y can be considered as reverse. In order to present Y, X must occur. Moreover, the truth table for this relation is represented in Figure 6.1.

6.2.2 OR Operator

OR operator states that if either node X and Y or both X and Y are present, then effect Z will happen. The representation and truth table of OR operator are shown in Figure 6.2.

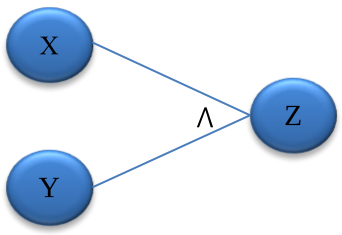


X	Y	Z
T	T	T
T	F	T
F	T	T
F	F	F

Figure 6.2 The notation and truth table of OR operator

6.2.3 AND Operator

AND Operator indicates that if all causes are present, then the effect will happen. The representation and truth table of AND operator are shown in Figure 6.3.

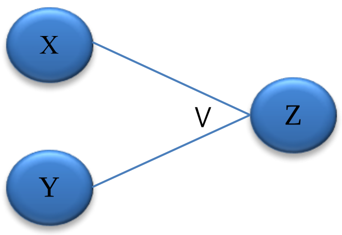


X	Y	Z
T	T	T
T	F	F
F	T	F
F	F	F

Figure 6.3 The notation and truth table of AND operator

6.2.4 NOR Operator

NOR operator is the opposite of the OR operator. The effect will happen, if all causes are absent. The NOR operator is usually used in editing data. If the effect is a warning message, it occurs only when valid values are missing. The truth table and NOR operator are depicted in Figure 6.4.

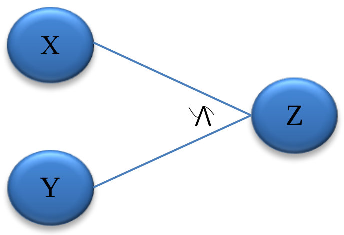


X	Y	Z
T	T	F
T	F	F
F	T	F
F	F	T

Figure 6.4 The notation and truth table of NOR operator

6.2.5 NAND Operator

NAND operator is the opposite of the AND operator. The effect will happen, if all causes are present. The representation and truth table of NAND operator are shown in Figure 6.5.

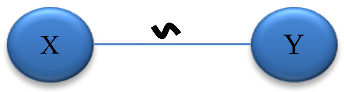


X	Y	Z
T	T	F
T	F	T
F	T	T
F	F	T

Figure 6.5 The notation and truth table of NAND operator

6.2.6 Negation

Negation denotes that if the cause is absent, then the effect will be present. Tilda symbol (~) is used on the connection to represent negation, which is depicted in Figure 6.6.



X	Y
T	**F**
F	**T**

Figure 6.6 The notation and truth table of negation operator

NOR and NAND logics can be represented by OR and AND with negation respectively. Both notations are convenient in cause-effect graphs. If more than two nodes are connected in cause-effect graphs, arc can be used and the operator can be written inside the arc like in Figure 6.7.

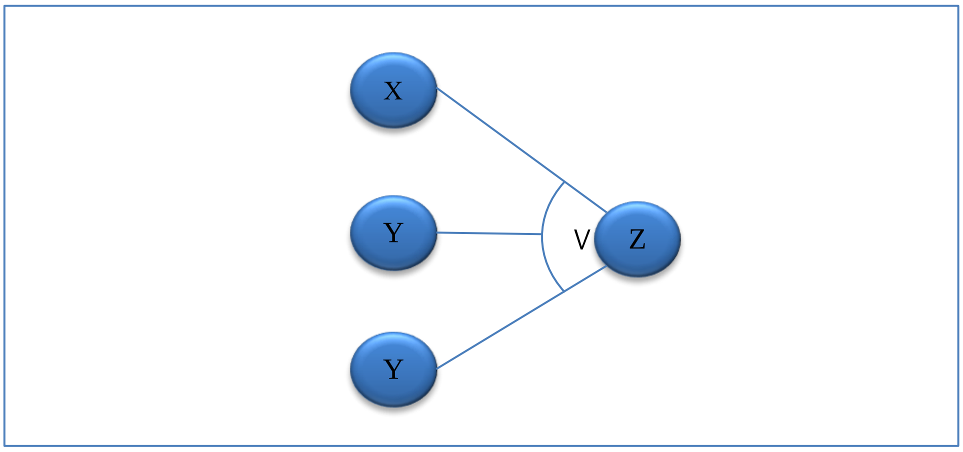


Figure 6.7 The arc symbol when combining more than two nodes in the cause-effect graph software test method

Some outputs of operations will be an input for intermediate nodes. When drawing a cause-effect graph, each time one sentence of requirements should be implemented. Moreover, nodes should not be duplicated. If the same nodes are already defined, they should be tracked, and defined nodes must be used in graphs.

6.3 Environmental Constraints

One of the most important purposes of cause-effect graph software test method is determining the combinations of causes to test all available functions. However, sometimes some combinations cannot be physically implemented. Thus, these constraints should be identified to decrease the number of combinations. Each extra constraint decreases the number of total test cases. Five main constraints can be applied to nodes. “One and only one”, “exclusive”, “inclusive”, “requires”, and “mask” are the main constraints that can be applied to nodes. Constraints on nodes are represented by dotted lines as shown in Figure 6.8.

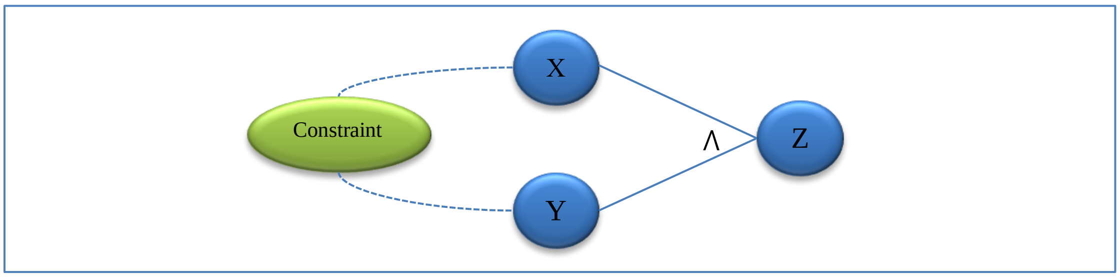


Figure 6.8 The representation of constraints on nodes in the cause-effect graphs

6.4 Natural Language Problems

The external specification is the key factor to provide the overall quality of the system. However, most of the bugs are caused by poor external specifications. Ambiguity of natural languages can cause a lot of problems when specifying external requirements. They might be misunderstood in the later stages of the software development life cycle. Some types of common ambiguities are listed in this section.

6.4.1 Dangling Else

Software specifications usually focus on direct paths. They do not mention what happens when something goes wrong. For example, if there are three conditions X, Y and Z, specifications usually state the events for each condition, but they may not mention what happens if the three conditions do not happen. Thus, each "must be" statement in software specifications should include an "else" clause to determine the situation of what happens if this "must be" statement does not satisfy. If the data is already controlled and validated before the condition, then the event should be defined by a "will be" statement. In that case, the "else" clause does not have to be used. For example, in the university course registration example, the following specification is given:

R0101: If the number of registered students for a course is less than 10 in the Engineering Faculty, the course building will be "A block".

So, what happens when registered students are more than ten? Can we infer, if the number of students is more than ten, then the assigned building is not A? All these types of questions may be interpreted differently in the other software development stages. To avoid these misunderstandings, the "else" clause should be used.

6.4.2 Ambiguity of Reference

In the natural language, pronouns are usually used to refer to nouns. However, programmatically, it is not possible to use pronouns instead of variables. Thus, all variables must be explicitly defined. Similarly, when writing specifications, pronouns should not be used, and all nouns should be explicitly stated. For instance, in the following R1 specification, there is a reference problem.

R1: The input X and Y will be a string. The input will start with a capital letter.

In this example, which input is started with a capital letter is not clear. It might refer to X or Y. Because this specification is ambiguous, it must be amended during the requirement-based software testing.

6.4.3 Scope of Action

Sometimes series of actions in the software specification can be misunderstood because of connecting adverbs such as "however". For example, if condition X or Y occurs, then action A, B, and C occur and M1 message is displayed. However, if only X condition is satisfied, M2 message is displayed. In this specification, the actions, when only X condition occurs, are not clear. It might be interpreted in two ways. The A, B and C actions are preceded, and M2 message is displayed or only M2 message is displayed. Usually the specification is interpreted as second one. However, it might be interpreted wrongly. These misunderstandings cause bugs in the system. For these reasons, all conditions and their actions should be stated in such a way that they are not misunderstood.

Another issue in this situation is the type of messages. The messages in specifications can be interpreted in different ways. Thus, there should be a separate message section which describes the type of message. The requirement-based test software technique should identify these types of weaknesses in the software specification document.

6.4.4 Omission

Finding the missing causes and effects in the software specification is not easy, because it is not written anywhere. Thus, functional and non-functional knowledge should be required to find the missing causes and effects. In addition to this, there might be causes which do not have effects. Similarly, there might be effects which do not have causes. If these causes and effects are not clearly determined, the cause-effect graph cannot be drawn. Consequently, they will not be tested.

6.4.5 Ambiguous Logical Operators

The software specifications can be structured in many ways. This means there is no one way to design the system. However, the written specifications are not structured. For this reason, the connection between actions may not be inferred correctly. Moreover, there might be some implicit connectors, which can be generated from software specifications. Thus, the software specifications should be written in a structured manner.

6.4.6 Negation

Negative statements may not be understandable as positive statements. For this reason, some negative statements in the specifications may not be comprehended. To solve these problems, some rules are used when writing software specifications. Firstly, if there are two statements, and one of these statements is negative and the other statement is positive, then when writing this specification, positive statements should be written first to avoid misunderstandings. For instance, the specification "If not X and Y, then Z” can be interpreted in two different ways. The “not” may affect only X or both X and Y. Thus, to get rid of this misunderstanding, the statement should be written as “If Y and not X, then Z”.

The second problem is that sometimes unnecessary negation might be used in the software specifications by system analysts. So, these unnecessary negations should be changed. For example, the specification “The X will not be less than Y” should be amended as “The X will be greater than Y”.

The last problem about the negation issue is double negation. Some words have negative meaning themselves. Thus, if these words are used with negative statements, the meaning of specifications might be complicated. For this reason, another word which refers to double negation should be preferred. For example, instead of “not present”, “absent” might be used. In the same way, instead of “not on time”, “late” might be used.

6.4.7 Ambiguous statements

Software specifications should be clear and precise. Thus, ambiguous words should not be used. The most common problem in software specifications is ambiguous verbs. The verbs in the statements may not exactly explain the actual process. For example, after clicking the X button, the Y value will be calculated. The calculation may be achieved with several algorithms. Without exact definition of the algorithm, the specification is incomplete.

The second important problem is ambiguous variables. If there is no data dictionary in the software specification document, the used variables might become ambiguous. Although most variable names reflect the meaning of variables, the similar variable names may cause ambiguity. For this reason, all variables should be explicitly stated in the software specification documents.

Another problem is unnecessary aliases. In some poor projects, several nick-names are used for the same things. Although it is acceptable for novels, it is not unacceptable for software specifications. For instance, in software specifications, “faculty student” and “university student” can be used interchangeably. However, it might cause a problem when mentioning specifically the faculty student. To avoid this problem, all names should be identified and explained in data dictionaries.

The last problem in this context is ambiguous adjectives and adverbs. When using an adjective and an adverb in the software specifications, it should be clear. For instance, one of the most used words in the software specification is “usually”. However, “usually” does not show the exact occurrence of something. Similarly, the adverbs such as “quickly” in specifications do not determine how fast the system or process is. For this reason, when using adverbs and adjectives, exact and clear information should be used.

6.4.8 Random Organization

Writing software specifications is different than a normal essay. The specifications should be simple and understandable. In addition, implicit words should not be used. Moreover, in the normal essay, different words are used to avoid iteration. On the other hand, iteration is acceptable to provide consistency in the software specifications. For instance, in the following R2 specification, two different verbs are used.

R2: If condition X is satisfied, then produce data Y. If condition Z is satisfied, then create data W

“Produce” and “create” verbs are used for the same process. For this reason, instead of two different verbs, the same verb should be used to avoid misunderstanding.

The other issue is the organization of cause and effects. When writing specifications, causes should be determined first, and then related effects should be stated. Thus, the structure of specification should be like “if, then, else”.

6.4.9 Other Problems

The other problems are related to built-in assumptions. The system analyst usually knows the system very well, so that he/she does not need to state the process explicitly. As a result, these assumptions may not be understandable by inexperienced developers or testers. Thus, each process should be explicitly stated, even if the process is well known among developers and testers.

During specification testing, the structured English should be used. This can minimize the number of faults in specifications. It is important to note that specification testing is not a correction process. It only shows the problems in software specifications. The detected problems should be informed to system analysts or requirement engineers who write software specifications. After the amendments, the testers should test the software specifications again to ensure the correctness of software specifications.

CHAPTER SEVEN

SOFTWARE TEST TOOLS

Today, there are more than 600 (six hundred) tools available for software testing. These tools are related test design tools, GUI test drivers, load and performance tools, test management tools, test implementation tools, test evaluation tools, and analysis tools. Moreover, each testing tool group can be divided into subgroups such as component, integration, regression and coverage tool.

In this thesis, three test tools are explained and analyzed. Firstly, the cause-effect graph testing tool, which is based on specification testing, is analyzed. Then the test data generator tool is explained. Finally, the statement coverage tools, which use white box testing methods, are analyzed.

7.1 Cause-effect Test Tool

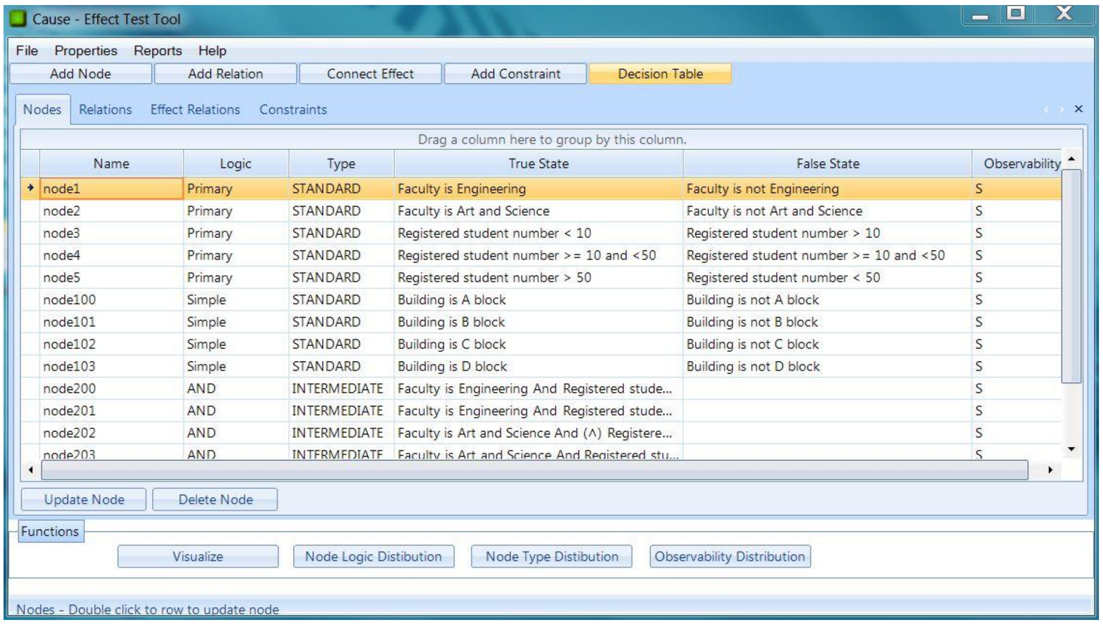
Software test case design tools are test design tools that are used to generate test cases automatically. If black box test case generation methods are used, then test cases can be generated from software specifications or any documents, which are related software design. On the other hand, if white box test case generation methods are used, in this situation, test cases are produced from a source code. The cause-effect graph software test design method is a black box method. Thus, it uses software specifications to generate test cases.

Software test cases are very important assets, and they are not easy to generate. However, when software test cases are generated, they can be used over and over for the same project. On the other hand, software specifications are dynamic, and they tend to change during all stages of the software development process. Moreover, the software specifications may change after delivering the software. For this reason, test cases must be updated for each modification of the software specifications. Manually tracking the software specifications and producing test cases is sometimes impossible or infeasible. Thus, automatic test case generation tools are needed to track software specifications and also to generate software test cases. Especially, the cause-effect graph software test method involves a large number of combinations. Therefore, automation should be required to handle all possible combinations.

For all requirement-based software tests, the process is more important than the tool itself. Thus, in order to perform cause-effect graph testing, firstly, the overall requirement-based testing process must be known.

One of the first known cause-effect tools was developed by Richard Bender. His program was BenderRBT. Today, it is still used by many companies.

Our cause-effect software test tool provides a systematic way to develop a cause-effect graph and produce test cases according to the cause-effect graph. Our tool also provides extra features, which are coverage analysis of effects and distribution graphs of nodes according to their types. The tool can also provide basic functions of cause-effect tools, such as reporting, data exporting and importing. For this reason, this tool can also be used for requirements tracing. If this tool is used in the requirement engineering process, then the test cases can be automatically produced after each modification. The interface of our cause-effect test tool is depicted in Figure 7.1.

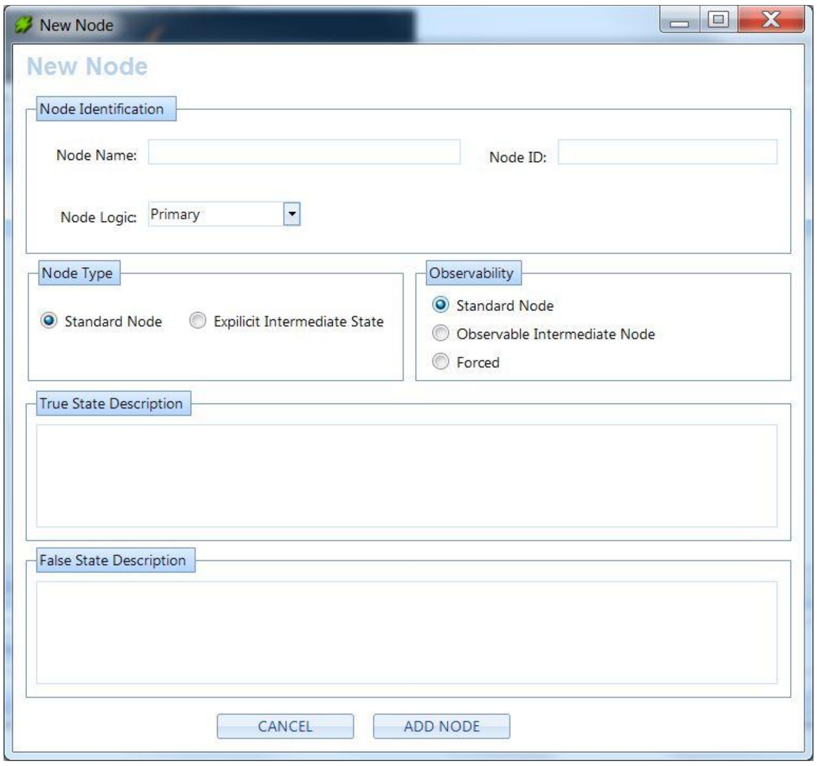


Name	Logic	Type	True State	False State	Observability
node1	Primary	STANDARD	Faculty is Engineering	Faculty is not Engineering	S
node2	Primary	STANDARD	Faculty is Art and Science	Faculty is not Art and Science	S
node3	Primary	STANDARD	Registered student number < 10	Registered student number > 10	S
node4	Primary	STANDARD	Registered student number >= 10 and <50	Registered student number >= 10 and <50	S
node5	Primary	STANDARD	Registered student number > 50	Registered student number < 50	S
node100	Simple	STANDARD	Building is A block	Building is not A block	S
node101	Simple	STANDARD	Building is B block	Building is not B block	S
node102	Simple	STANDARD	Building is C block	Building is not C block	S
node103	Simple	STANDARD	Building is D block	Building is not D block	S
node200	AND	INTERMEDIATE	Faculty is Engineering And Registered stude...		S
node201	AND	INTERMEDIATE	Faculty is Engineering And Registered stude...		S
node202	AND	INTERMEDIATE	Faculty is Art and Science And (∧) Registere...		S
node203	AND	INTERMEDIATE	Faculty is Art and Science And Registered stu...		S

Nodes - Double click to row to update node

Figure 7.1 The main menu of the cause-effect graph software test tool

In our cause-effect graph test tool, first of all, all causes, effects and the relations between them should be determined. All causes, effects and their combinations are defined as a node in the system. In the tool, these nodes are defined in a “new node” section, which is depicted in Figure 7.2.



Node Name:
Node ID:
Node Logic: Primary

Node Type: [✓ Standard Node] Expilicit Intermediate State

Observability: [✓ Standard Node] Observable Intermediate Node Forced

Figure 7.2 Add Node interface of the cause-effect graph software test tool

Special terminology is used in cause-effect graph tools. Firstly, the node name should be unique in the system, and all relations are defined by this unique name. The node's logic identifies the type of the node. Causes are defined as "Primary" and effects are defined as "Simple" nodes. The other middle nodes, which are the combination of causes, are defined according to their logic such as "AND" or "OR". If the node is a cause or an effect, the node type will be Standard. On the other hand, if the node is a middle node, the node type will be "Explicit Intermediate State". The true state description determines what will be displayed when this node is true. Likewise, the false state description determines what will be displayed when this node is false. Usually, if a node is an explicit intermediate state, the false state description will be empty. The Observability determines the availability of the test state during the execution of the test. For example, all objects on screens, database transections and objects on reports are considered as observable nodes. In addition to this, sounds and movements can also be considered as additional observable objects. In the context of requirement specification testing, all causes are assumed to be observable and all effects are assumed to be not observable. Thus, during the testing, if a defect is found in an intermediate node, the cause of the defect can only be found by tracking an observable cause node.

In our tool, if the middle node is not observable, the Observability must be set as Standard node. However, if the middle node is actually observable, then the Observability should be set as Observable Intermediate Node. Sometimes, the nodes cannot be testable because of constraints, although the node itself is observable. In that case, the Observability should be set as "Forced". In our tool, there is no negation logic. For this reason, if the negation of a node is required, a different node should be created for negation logic.

After identifying and defining nodes, the connections of nodes should be identified. In our tool, there are two types of connections. The first connection type connects cause nodes to middle nodes or middle nodes to other middle nodes. The second connection type connects cause nodes to effect nodes or middle nodes to effect nodes. In our tool, the first connection can be defined from the “Add Relation” section, and the second connection can be determined from the “Connect Effect” section. These connection interfaces are represented in Figure 7.3.

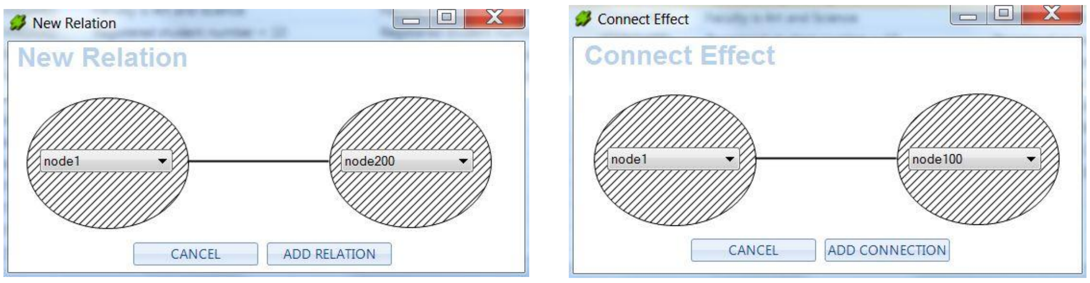


Figure 7.3 The add relation of cause-effect graph software test tool

The next process is defining constraints on nodes. In our tool, constraints are divided into two categories. The first category includes exclusive, inclusive and one and only one constraints that are applicable to more than two nodes. The second category covers require and mask constraints, which involve only two nodes. Furthermore, the constraint direction determines which node is required or masked. The constraints are defined in the “Add Constraint” section in the tool.

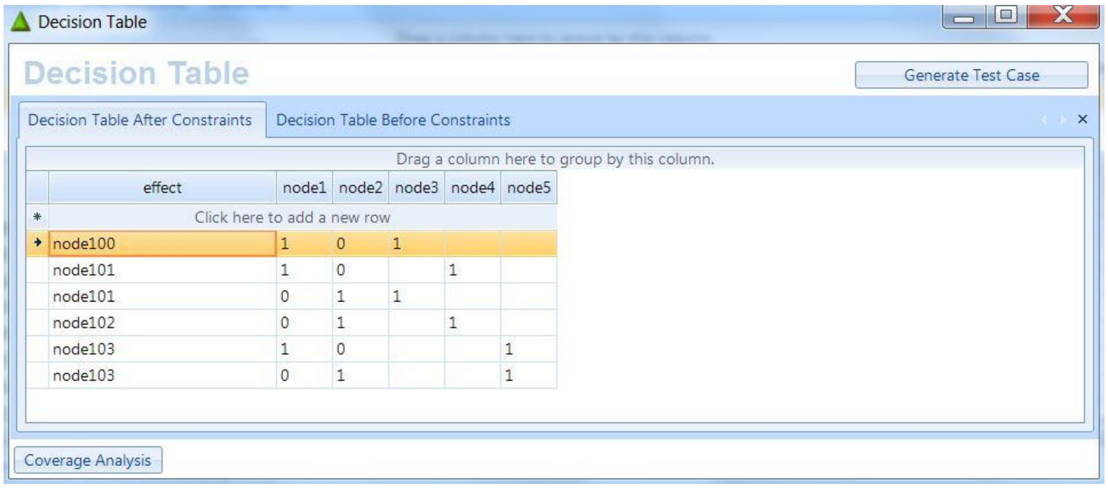
If all nodes, connections and constraints are defined, the decision table can be generated by using a “Decision Table” module. The following algorithm is used when generating a decision table.

```
Declare empty decision Table

For each row in effect node table
Do
  related node -> getEffectRelation (current row effect )

  If related node is Primary
    Declare new Row for decision Table
    Decision Table Row [ related node ] -> 1
    InsertDecisonTable(Decision Table Row)

  Else
     Declare accumulated condition list
     Propagate (list, related node, effect node)

For each row in constraint table
Do
     Switch constraint
```

```
Case: EXCL
      Parse(EXCL statement into nodes )
      For each row in decision table
      Do
         If more than one EXCL node is 1
             Delete (Current row)


         If only one EXCL node is 1
            For each other EXCL nodes
            Do
              Decision Table Current Row [EXCL node] -> 0

Case: INCL
      Parse(INCL statement into nodes )
      For each row in decision table
      Do
        If all INCL nodes are 0
          Delete (Current row)

Case: ONE
      Parse(ONE statement into nodes )
      For each row in decision table
      Do
         If number of true ONE node is not 1
           Delete (Current row)
         Else
             For each other ONE nodes
             Do
               Decision Table Current Row [ONE node] -> 0

Case: REQ
      Parse(REQ statement into node1 and node2 )
      For each row in decision table
      Do
         If Decision Table Current Row [node1] is 1
           If Decision Table Current Row [node2] is not 1
             Delete (Current row)



Case: MASK
      Parse(MASK statement into effect1 and effect2 )
      For each row in decision table
      Do
         If current row effect is effect2
             For each row in decision table
             Do
               If current row effect is effect 1
                 If effect 1 nodes are same as effect2
                    Delete (Current row)
```

```
Propogate (condition list, related node, effect node) {

Declare primaryConditionList
Declare intermediateConditionList

   If current row [ related node ] is Primary
       primaryConditionList.Insert(related node)
   Else
       intermediateConditionList.Insert(related node)

       primaryConditionList.CombineWith(condition list)


   If related node is intermediate AND node

      For each row in relation node table for the selected effect
       Do
          Declare new Row for decision Table

         For each node in primaryConditionList
         Do
            Decision Table Row [primaryConditionList.currentNode] -> 1

         InsertDecisonTable(Decision Table Row)

         For each node in intermediateConditionList
         Do
            Propogate (primaryConditionList, related node, effect node)


  If related node is intermediate OR node

      For each row in relation node table for the selected effect
       Do
         For each node in primaryConditionList
         Do
            Declare new Row for decision Table
            Decision Table Row [primaryConditionList.currentNode] -> 1
            InsertDecisonTable(Decision Table Row)

          For each node in intermediateConditionList
          Do
            Propogate (primaryConditionList, related node, effect node)
}
```

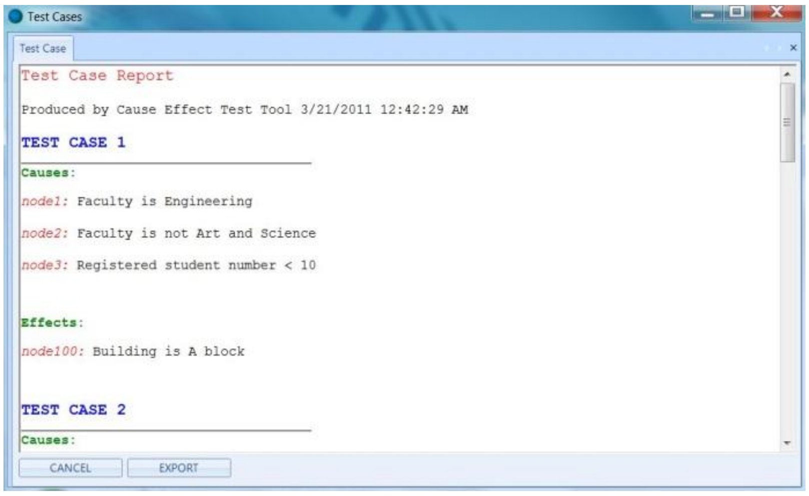
Two decision tables are produced. The first table represents the decision table where all constraints are applied. The second table shows the decision table without constraints. In this way, effects, which are canceled by constraints, can be detected. The decision table for the university course registration system is represented in Figure 7.4.


Decision Table
Decision Table
Generate Test Case
Decision Table After Constraints
Decision Table Before Constraints
Drag a column here to group by this column.
Click here to add a new row
Coverage Analysis

effect	node1	node2	node3	node4	node5
node100	1	0	1		
node101	1	0		1	
node101	0	1	1		
node102	0	1		1	
node103	1	0			1
node103	0	1			1

Figure 7.4 The decision table interface for university course registration system in the cause-effect graph software test tool

In the decision table, each row represents a test case. The first column shows the effect and other columns represent causes. If the first column is “#”, this means the effect cannot be performed with stated causes because of constraints. After generating a decision table, test cases can be directly generated from a decision table. Each row constitutes a different test case. Test cases can be produced by analyzing each column in a row. For example, for node100 in Figure 7.4, the node1 and node3 is 1 and node2 is 0. So, the test case will be, if statements in node1 and node3 are true and node2 is false, then statement in node100 must be true. In our system, test cases can be produced by the “Generate Test Case” section. Figure 7.5 shows the test cases which are produced from the previous decision table.



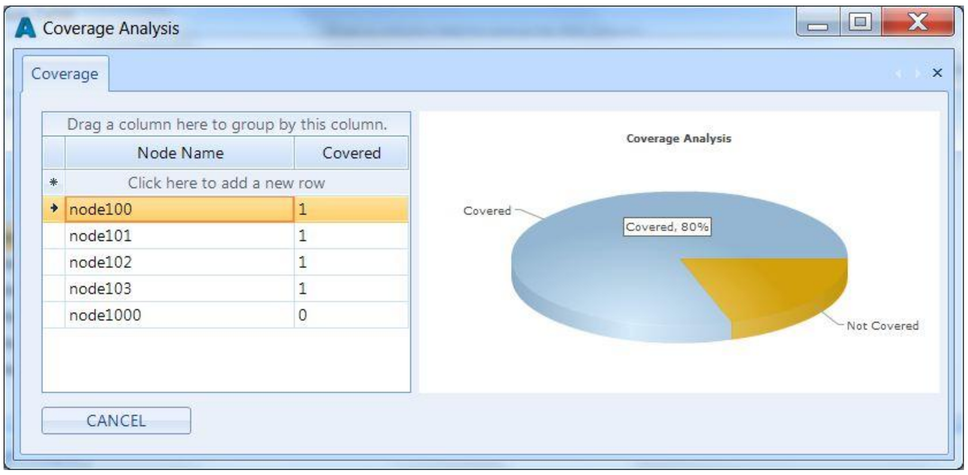
```
Test Case Report

Produced by Cause Effect Test Tool 3/21/2011 12:42:29 AM

TEST CASE 1
Causes:
node1: Faculty is Engineering
node2: Faculty is not Art and Science
node3: Registered student number < 10

Effects:
node100: Building is A block

TEST CASE 2
Causes:
```

Figure 7.5 The test case report interface for university course registration system in the cause-effect graph software test tool

Some effects may not be covered in software test cases, because constraints may cancel the test conditions or the effect relation may not be defined. So, effect coverage can be calculated by the following formula:

$$Effect\ Coverage = \frac{Number\ of\ different\ effects\ in\ test\ cases}{Total\ number\ of\ defined\ effects}\ x\ 100$$

One of the goals of cause-effect test tools is reaching %100 effect coverage. Figure 6.6 shows the interface of coverage analysis.



Node Name	Covered
node100	1
node101	1
node102	1
node103	1
node1000	0

Figure 7.6. The coverage analysis in the cause-effect graph software test tool

In this coverage analysis example, five effects are defined, but only four of them are covered. Thus, the coverage is %80. The “node1000”, which covered section is 0, is canceled, because constraints may be applied or the relation of effect may not be defined. For this reason, before starting to produce test cases, %100 coverage should be provided.

The tool also provides distribution graphs of nodes according to their logic, type and observability. The distribution graphs provide a way to compare projects. The tester can compare projects according to the distribution of causes and effects as well as their observability. The interface of the bar and pie chart of node distribution according to node logic is depicted in the following figure:

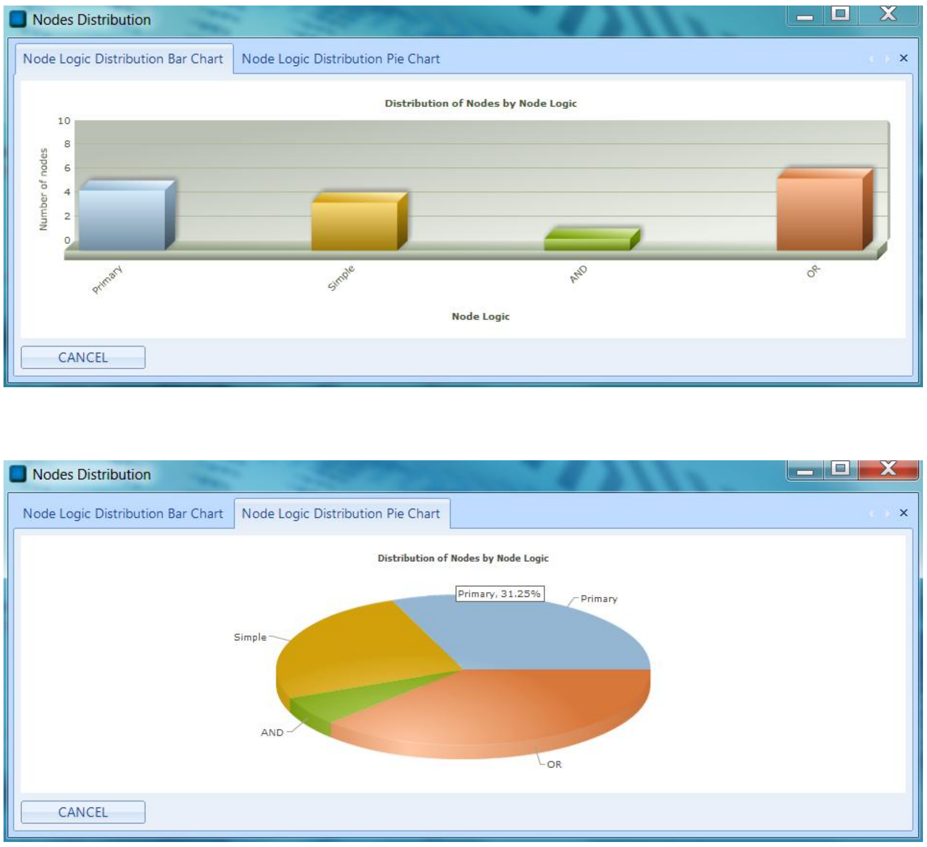

Nodes Distribution
Node Logic Distribution Bar Chart
Node Logic Distribution Pie Chart
Distribution of Nodes by Node Logic
Number of nodes
10
8
6
4
2
0
Primary
Simple
AND
OR
Node Logic
CANCEL
Nodes Distribution
Node Logic Distribution Bar Chart
Node Logic Distribution Pie Chart
Distribution of Nodes by Node Logic
Primary, 31.25%
Primary
Simple
AND
OR
CANCEL

Figure 7.7 The interface of the bar and pie chart of node distribution according to node logic in the cause-effect graph software test tool

7.2 Test Data Generator

Although test cases are very important for software testing, the used test data is also crucial for software testing. When testing software, usually dummy test data is used by developers. However, this may cause a problem when real data is used. In addition, the behavior of the system, when there are large numbers of records in the database, should be tested. There might be a problem because of a memory constraint or the program may not be working as expected. To test these situations, real data should be provided to the system. However, it is usually impossible to find real data to test programs, especially for the new type of projects. For this reason, test data tools are developed. These tools generate random test data, which usually has the same pattern as real data. Moreover, an unlimited number of data can be generated by these tools.

These tools are language depended and changed according to the country. For instance, the address pattern in Turkey is different than the address pattern in the US. Similarly, names and their character sets are also different.

The other important point for test data generator is the random function. Most programming languages use system time for random function. However, when generating a large number of data, same or similar data can be generated. To avoid this problem, different random functions can be used, and also the used random function will be determined randomly. Another alternative solution for this problem is using system sleep. In this way, more random test data can be produced.

In our web-based Turkish test data generator, we identified common used data types, such as name, telephone number, address, date, number and text. Moreover, we determined common used patterns of these data types. For example, for the "name" data type, six different patterns are defined. These are:

- Name (Male Name)
- Name (Female Name)

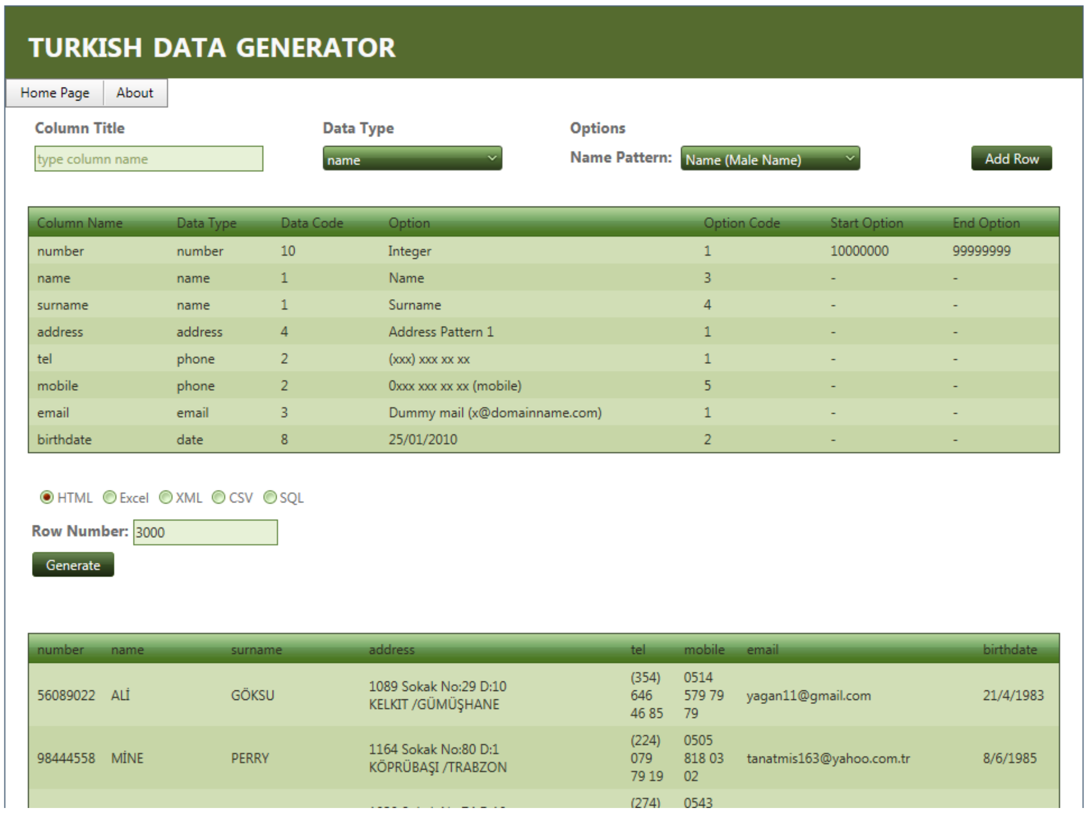
- Name (Male and Female)
- Surname
- Name Surname
- Surname Name

For instance, for the “text” data type, there are two options, which are a fixed number of words and a random number of words.

When a tester generates test data, firstly, he/she defines the database table by using predefined data types. Then the tester states the required number of rows. Finally, the tester declares the exporting type. The tester can export the generated data to HTML, Exel, XML, CSV and SQL according to his/her needs. For example, in the student registration system, the database includes the following information:

- Student Number (8 digits)
- Student name, surname
- Student address
- Student telephone number
- Student mobile phone number
- Student email address
- Student birth date

After columns are defined, the required number of records should be determined. If there is a specification for student capacity, such as “The system should record up to 3,000 students”, then the row number should be this boundary value. Figure 7.8 shows the result of the example above.


TURKISH DATA GENERATOR

Column Name	Data Type	Data Code	Option	Option Code	Start Option	End Option
number	number	10	Integer	1	10000000	99999999
name	name	1	Name	3	-	-
surname	name	1	Surname	4	-	-
address	address	4	Address Pattern 1	1	-	-
tel	phone	2	(xxx) xxx xx xx	1	-	-
mobile	phone	2	0xxx xxx xx xx (mobile)	5	-	-
email	email	3	Dummy mail (x@domainname.com)	1	-	-
birthdate	date	8	25/01/2010	2	-	-

[✓ HTML] Excel XML CSV SQL

Row Number: 3000

number	name	surname	address	tel	mobile	email	birthdate
56089022	ALİ	GÖKSU	1089 Sokak No:29 D:10 KELKIT /GÜMÜŞHANE	(354) 646 46 85	0514 579 79 79	yagan11@gmail.com	21/4/1983
98444558	MİNE	PERRY	1164 Sokak No:80 D:1 KÖPRÜBAŞI /TRABZON	(224) 079 79 19	0505 818 03 02	tanatmis163@yahoo.com.tr	8/6/1985
				(274)	0543		

Figure 7.8 The interface of Turkish test data generator

7.3 Code Coverage Tools

Coverage tools are usually used to test unimportant parts of software. Moreover, they provide a measure about completeness of test. These tools use the source code to test programs. For this reason, different code coverage tools must be developed for each programming language. There are five common code coverage tools. They are functional coverage, statement or line coverage, condition coverage or decision coverage, path coverage, and entry and exit coverage.

Code coverage is a white box technique. Thus, the coverage tools have to understand the source code. However, some codes may not be recognized or may not work correctly. The reason is that different tools use different methods to calculate the code coverage. To solve this problem, extra code is added to the source code, before executing the code coverage tool. However, these modified codes can produce different results, which differ from the original source code results.

All code coverage tools have limitations. Thus, the tester should know the limitations of selected tools to maximize the efficiency and effectiveness of software testing. Furthermore, testers should know how to interpret reports of code coverage tools. Otherwise, problematic codes cannot be detected.

Code coverage tools analyze the source code, and especially focus on condition statements. By analyzing the terms and variables in the condition statements, different test suites are generated. In a condition statement, there might be functions or mathematical expressions. Moreover, a function in the condition statement requires file input or output operations. For this reason, each condition statement should be stated as true and false. According to the type coverage analysis, test cases are generated by assigning input values, which make the condition statements true. In addition, for each test case, covered statements or conditions or any coverage measure should be kept. The process of generating test cases is iterated until %100 coverage is provided. On the other hand, %100 coverage may not be provided because of unreachable codes or because of a large number of iterations in loops. For all these reasons, code coverage tools should confine the number of test suites. This limitation may prevent %100 coverage, although the source code is %100 coverable.

After the execution of code coverage tools, they usually provide a report which tells which parts of the code were not exercised. After correcting the bugs which are found by the code coverage tool, the quality of the program is not guaranteed. In other words, code coverage tools just improve the quality of the program. For this reason, these tools are used in a complementary manner. Moreover, code coverage tools are not based on software specifications. Thus, specification-based bugs are not found by these tools.

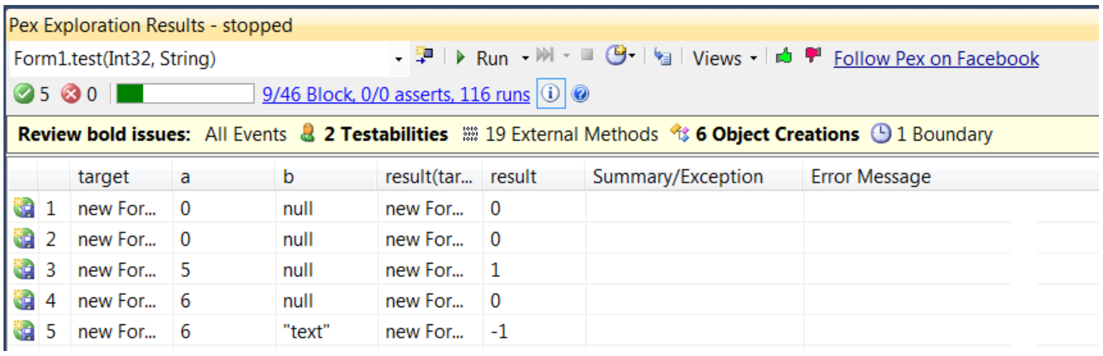
One of the most famous code coverage tools is Pex, which was developed by Microsoft Corporation. Pex is integrated to Visual Studio and automatically generates test suites with high code coverage. Pex understands the input and output behavior of the source code. In addition, Pex produces test cases by analyzing the source code. It creates inputs in order to reach all statements by analyzing each conditional branch.

Pex combines dynamic and static analysis by using a constraint solver. The goal of Pex is generating a set of input values, which execute as many statements as possible. When Pex starts, it chooses the possible simplest inputs. For example, the “null” value is selected for a string or an array. Then Pex executes the code with these simple inputs. After that, all conditions that are checked are monitored. Then, Pex negates the checked conditions to find a solution for negated conditions. If a solution exists, then another test input is produced for this solution. All these steps iteratively proceed.

For example, the following simple C# code is executed by Pex tool. The test function takes one integer and one string inputs and returns an integer value. According to the “a” and “b” variables, the result is changed. Moreover, there is an unreachable code which is caused by “if(unreachable == 2)” condition statement.

```
public int  test(int a, string b)
    {
    int result = 0;
    int unreachable = 1;

    if(a == 5 )
    {
      result = result + 1;
    }

    if( a == 6 && b == "text")
    {
      result = result - 1;
    }

    if(unreachable == 2)
    {
      result = 100;
    }

    return result;
    }
```

The Pex result is depicted in the following figure. Five different test cases are generated, and the source code is executed 116 times. The ‘a’ and ‘b’ columns represent the test functions inputs, and the result column shows the corresponding outputs.


Pex Exploration Results - stopped

Form1.test(Int32, String) Run Views Follow Pex on Facebook

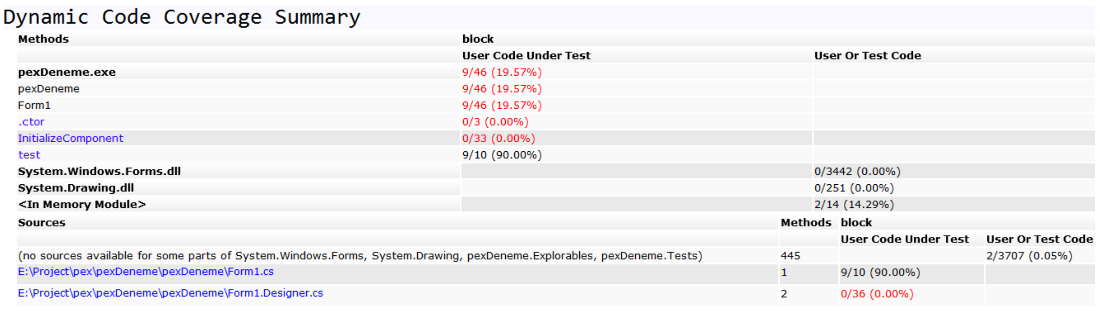
5 0 9/46 Block, 0/0 asserts, 116 runs

Review bold issues: All Events 2 Testabilities 19 External Methods 6 Object Creations 1 Boundary

	target	a	b	result(tar...	result	Summary/Exception	Error Message
1	new For...	0	null	new For...	0		
2	new For...	0	null	new For...	0		
3	new For...	5	null	new For...	1		
4	new For...	6	null	new For...	0		
5	new For...	6	"text"	new For...	-1		

Figure 7.9 After the execution of Pex tool for the test function

Like all code coverage tools, the Pex tool produces a test report to show the code coverage analysis. The Pex report is denoted in the following figure. According to this report, %90 of the code is covered.


Dynamic Code Coverage Summary

Methods	block	
	User Code Under Test	User Or Test Code
pexDeneme.exe	9/46 (19.57%)	
pexDeneme	9/46 (19.57%)	
Form1	9/46 (19.57%)	
.ctor	0/3 (0.00%)	
InitializeComponent	0/33 (0.00%)	
test	9/10 (90.00%)	
System.Windows.Forms.dll		0/3442 (0.00%)
System.Drawing.dll		0/251 (0.00%)
<In Memory Module>		2/14 (14.29%)

Sources	Methods	block	
		User Code Under Test	User Or Test Code
(no sources available for some parts of System.Windows.Forms, System.Drawing, pexDeneme.Explorables, pexDeneme.Tests)	445		2/3707 (0.05%)
E:\Project\pex\pexDeneme\pexDeneme\Form1.cs	1	9/10 (90.00%)	
E:\Project\pex\pexDeneme\pexDeneme\Form1.Designer.cs	2	0/36 (0.00%)	

Figure 7.10 Pex code coverage summary for the test function

Like all coverage tools, Pex tool has limitations. For instance, conditions in a large number of iterations in a loop and some string comparisons in a condition statement cannot be handled by Pex tool. The following simple C# code denotes these problems.

```
public int test(string a ,string b)
        {
            int i = 0;
            int result = 0;
            while (i < 10000)
            {
                if (i == 9000)
                {
                    result = 1;
                }
                i++;
            }
            if (a + b == "coverageTool")
            {
                result = 1;
            }
            return result;
       }
```

The yellow parts are not covered by Pex tool. However, it is very obvious that the code can be %100 coverable. In the 9000th iteration, the first “if condition” is satisfied and the second condition can be satisfied when the “a” is “coverage” and the “b” is “Tool”. Although Pex tool does not denote the source code %100 coverable, the source code is obviously %100 coverable.

CHAPTER EIGHT

CONCLUSION

In order to perform software testing, firstly test conditions should be identified. Then software test cases should be generated from software test conditions. Different software test case design techniques are used to produce test cases from test conditions. Cause-effect graphing software testing is one of them. With this technique, combination of test conditions can be tested.

The main goal of this study is to develop a cause-effect graphing software testing tool which is based on requirement-based software testing. In the cause-effect graph method, software specifications are used to generate test cases. Thus, the specification-based testing process should be known to implement this technique. In addition to this, the cause-effect graph provides a systematic way to combine test conditions. The combination of test cases can reveal new bugs, which cannot be found in the traditional software testing methods.

To produce test cases from requirements, all causes and effects, as well as the relations between them, should be defined. As a large number of combinations are produced even for a small program, some nodes may be discarded to decrease the number of test cases. However, if the program is a safety-critical or business-critical system, these kind of eliminations should be avoided.

Cause-effect tools are not usually preferred by small sized software. The reason is the complexity of processes and tools. All requirements should be defined in the system. This process requires huge resources in terms of tester labor. However, cause-effect graph method is just found some part of the bugs in the system. For this reason, it must be used complementarily with other testing methods.

REFERENCES

Beizer, B. (1990). *Software Testing Techniques* (2nd ed.), NY: International Thomson Computer Press

Bender, R. (2008). *Cause-Effect Graphing User Guide* (1st ed.), NY: Bender RBT Inc.

Burnstein, I. (2002). *Practical software Testing: a process-oriented approach* (1st ed.), NY: Springer-Verlag Inc.

Cornett, S. (1996). *Code coverage analysis.* Retrieved March 15, 2011, from http://www.bullseye.com/coverage.html#basic_condition

Graham, D., Veenendaal, E., Evans, I., & Black, R. (2008). *Foundations of software testing* (2nd ed.), Canada: Nelson Education Ltd.

How Does Pex Work? (n.d.). Retrieved March 16, 2011, from http://pexforfun.com/Page.aspx#3fb3105f-905b-4b5d-a368-d1924a9d1e71

IEEE 829. (n.d.). Retrieved March 20, 2011, from http://online.gerrardconsulting.com/iseb/otherdocs/ieee829mtp.pdf

Lewis, W. E. (2008). *Software testing and continuous quality improvement* (3rd ed.), USA:CRC Press

Mayers, G. J. (2004). *The art of software testing* (2nd ed.), USA: John Wiley & Sons Inc.

Nook, R. (2008). *Advanced software testing* (1st ed.), USA: Rock Nook Inc. Ammann, P., & Offutt J. (2008). *Introduction to software testing* (1st ed.), NY: Cambridge University Press

Patton, R. (2005). *Software testing* (1st ed.), USA: Sams Publishing

Srivastava, P. R., Patel, P., & Chatrola S. (2009). Cause effect graph to decision table generation. *SIGSOFT Software Engineering Notes, Volume 34 Number 2*

Test Case Specification Template IEEE 829. (n.d). Retrieved March 20, 2011, from http://www.testexpert.com.br/files/SQE%20Test%20Case%20Specification%20Template.pdf

Test Design Specification Template IEEE 829. (n.d). Retrieved March 20, 2011, from http://www.testexpert.com.br/files/SQE%20Test%20Design%20Specification%20Template.pdf