LEARNER MANUAL

TEST A COMPUTER PROGRAM AGAINST A GIVEN SPECIFICATION

US ID: 115384

NQF LEVEL: 5

CREDITS: 6

NOTIONAL HOURS: 60

Contents

This manual was compiled by SM Support on behalf of the training provider.

HOW TO USE THIS GUIDE

This workbook belongs to you. It is designed to serve as a guide for the duration of your training programme. It contains readings, activities, and application aids that will assist you in developing the knowledge and skills stipulated in the specific outcomes and assessment criteria. Follow along in the guide as the facilitator takes you through the material, and feel free to make notes and diagrams that will help you to clarify or retain information. Jot down things that work well or ideas that come from the group. Also, note any points you would like to explore further. Participate actively in the skill practice activities, as they will give you an opportunity to gain insights from other people's experiences and to practice the skills. Do not forget to share your own experiences so that others can learn from you too.

ICONS

HOW YOU WILL LEARN

The programme methodology includes facilitator presentations, readings, individual activities, group discussions, and skill application exercises.

PROGRAMME OVERVIEW

PURPOSE

People credited with this unit standard are able to:

- Test a computer program against given specifications according to test plans
- Record the results from testing a computer program
- Review the testing process for a computer program against organisation policy and procedures.

LEARNING ASSUMPTIONS

The credit value of this unit is based on a person having the prior knowledge and skills to:

- Demonstrate an understanding of fundamental mathematics (at least NQF level 3)
- Apply the Principles of Computer Programming

HOW YOU WILL LEARN

The programme methodology includes facilitator presentations, readings, individual activities, group discussions, and skill application exercises.

HOW YOU WILL BE ASSESSED

This programme has been aligned to registered unit standards. You will be assessed against the outcomes of the unit standards by completing a knowledge assignment that covers the essential embedded knowledge stipulated in the unit standards. When you are assessed as

competent against the unit standards, you will receive a certificate of competence and be awarded 6 credits towards a National Qualification.

SESSION 1: TESTING A COMPUTER PROGRAM

On completion of this section you will be able to test a computer program against given specifications according to test plans.

1. The testing executes operational steps identified in the test plan
2. The testing uses input data as specified in the test plan.
3. The testing outlines deviations from the test plan, with explanations.
4. The testing follows the standards and procedures specified in the test plan for testing and re-testing.

1.1 INTRODUCTION

Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not. In simple words, testing is executing a system in order to identify any gaps, errors, or missing requirements in contrary to the actual requirements.

According to ANSI/IEEE 1059 standard, Testing can be defined as - A process of analyzing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software item.

Who does Testing?

It depends on the process and the associated stakeholders of the project(s). In the IT industry, large companies have a team with responsibilities to evaluate the developed software in context of the given requirements. Moreover, developers also conduct testing which is called **Unit Testing**. In most cases, the following professionals are involved in testing a system within their respective capacities:

- Software Tester
- Software Developer
- Project Lead/Manager
- End User

Different companies have different designations for people who test the software on the basis of their experience and knowledge such as Software Tester, Software Quality Assurance Engineer, QA Analyst, etc.

It is not possible to test the software at any time during its cycle. The next two sections state when testing should be started and when to end it during the SDLC.

When to Start Testing?

An early start to testing reduces the cost and time to rework and produce error-free software that is delivered to the client. However in Software Development Life Cycle (SDLC), testing can be started from the Requirements Gathering phase and continued till the deployment of the software. It also depends on the development model that is being used. For example, in the Waterfall model, formal testing is conducted in the testing phase; but in the incremental model, testing is performed at the end of every increment/iteration and the whole application is tested at the end.

Testing is done in different forms at every phase of SDLC:

- During the requirement gathering phase, the analysis and verification of requirements are also considered as testing.
- Reviewing the design in the design phase with the intent to improve the design is also considered as testing.
- Testing performed by a developer on completion of the code is also categorized as testing.

When to Stop Testing?

It is difficult to determine when to stop testing, as testing is a never-ending process and no one can claim that a software is 100% tested. The following aspects are to be considered for stopping the testing process:

- Testing Deadlines
- Completion of test case execution
- Completion of functional and code coverage to a certain point
- Bug rate falls below a certain level and no high-priority bugs are identified
- Management decision

1.2 TYPES OF TESTS

This section describes the different types of testing that may be used to test a software during SDLC.

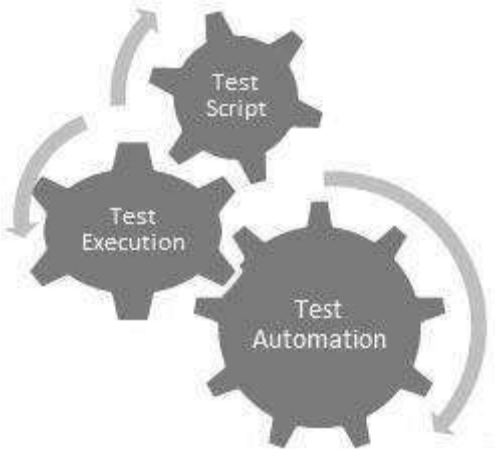
Manual Testing

Manual testing includes testing a software manually, i.e., without using any automated tool or any script. In this type, the tester takes over the role of an end-user and tests the software to identify any unexpected behavior or bug. There are different stages for manual testing such as unit testing, integration testing, system testing, and user acceptance testing.

Testers use test plans, test cases, or test scenarios to test a software to ensure the completeness of testing. Manual testing also includes exploratory testing, as testers explore the software to identify errors in it.

Automation Testing

Automation testing, which is also known as Test Automation, is when the tester writes scripts and uses another software to test the product. This process involves automation of a manual process. Automation Testing is used to re-run the test scenarios that were performed manually, quickly, and repeatedly.



Apart from regression testing, automation testing is also used to test the application from load, performance, and stress point of view. It increases the test coverage, improves accuracy, and saves time and money in comparison to manual testing.

What is Automate?

It is not possible to automate everything in a software. The areas at which a user can make transactions such as the login form or registration forms, any area where large number of users can access the software simultaneously should be automated.

Furthermore, all GUI items, connections with databases, field validations, etc. can be efficiently tested by automating the manual process.

When to Automate?

Test Automation should be used by considering the following aspects of a software:

- Large and critical projects
- Projects that require testing the same areas frequently
- Requirements not changing frequently
- Accessing the application for load and performance with many virtual users
- Stable software with respect to manual testing
- Availability of time

How to Automate?

Automation is done by using a supportive computer language like VB scripting and an automated software application. There are many tools available that can be used to write automation scripts. Before mentioning the tools, let us identify the process that can be used to automate the testing process:

- Identifying areas within a software for automation
- Selection of appropriate tool for test automation
- Writing test scripts
- Development of test suits
- Execution of scripts
- Create result reports
- Identify any potential bug or performance issues

Software Testing Tools

The following tools can be used for automation testing:

- HP Quick Test Professional
- Selenium
- IBM Rational Functional Tester
- SilkTest
- TestComplete
- Testing Anywhere
- WinRunner

- LoadRunner
- Visual Studio Test Professional
- WATIR

1.3 SOFTWARE TESTING METHODS

There are different methods that can be used for software testing. This chapter briefly describes the methods available.

1. Black-Box Testing

The technique of testing without having any knowledge of the interior workings of the application is called black-box testing. The tester is oblivious to the system architecture and does not have access to the source code. Typically, while performing a black-box test, a tester will interact with the system's user interface by providing inputs and examining outputs without knowing how and where the inputs are worked upon.

The following table lists the advantages and disadvantages of black-box testing.

Advantages	Disadvantages
• Well suited and efficient for large code segments. • Code access is not required. • Clearly separates user's perspective from the developer's perspective through visibly defined roles. • Large numbers of moderately skilled testers can test the application with no knowledge of implementation, programming language, or operating systems.	• Limited coverage, since only a selected number of test scenarios is actually performed. • Inefficient testing, due to the fact that the tester only has limited knowledge about an application. • Blind coverage, since the tester cannot target specific code segments or error-prone areas. • The test cases are difficult to design.

2. White-Box Testing

White-box testing is the detailed investigation of internal logic and structure of the code. White-box testing is also called **glass testing** or **open-box testing**. In order to perform **white-box** testing on an application, a tester needs to know the internal workings of the code.

The tester needs to have a look inside the source code and find out which unit/chunk of the code is behaving inappropriately.

The following table lists the advantages and disadvantages of white-box testing.

Advantages	Disadvantages
• As the tester has knowledge of the source code, it becomes very easy to find out which type of data can help in testing the application effectively. • It helps in optimizing the code. • Extra lines of code can be removed which can bring in hidden defects. • Due to the tester's knowledge about the code, maximum coverage is attained during test scenario writing.	• Due to the fact that a skilled tester is needed to perform white-box testing, the costs are increased. • Sometimes it is impossible to look into every nook and corner to find out hidden errors that may create problems, as many paths will go untested. • It is difficult to maintain white-box testing, as it requires specialized tools like code analyzers and debugging tools.

3. Grey-Box Testing

Grey-box testing is a technique to test the application with having a limited knowledge of the internal workings of an application. In software testing, the phrase the more you know, the better carries a lot of weight while testing an application.

Mastering the domain of a system always gives the tester an edge over someone with limited domain knowledge. Unlike black-box testing, where the tester only tests the application's user interface; in grey-box testing, the tester has access to design documents and the database. Having this knowledge, a tester can prepare better test data and test scenarios while making a test plan.

Advantages	Disadvantages
• Offers combined benefits of black-box and white-box testing wherever	• Since the access to source code is not available, the ability to go over

possible. • Grey box testers don't rely on the source code; instead they rely on interface definition and functional specifications. • Based on the limited information available, a grey-box tester can design excellent test scenarios especially around communication protocols and data type handling. • The test is done from the point of view of the user and not the designer.	the code and test coverage is limited. • The tests can be redundant if the software designer has already run a test case. • Testing every possible input stream is unrealistic because it would take an unreasonable amount of time; therefore, many program paths will go untested.

A Comparison of Testing Methods

The following table lists the points that differentiate black-box testing, grey-box testing, and white-box testing.

Black-Box Testing	Grey-Box Testing	White-Box Testing
The internal workings of an application need not be known.	The tester has limited knowledge of the internal workings of the application.	Tester has full knowledge of the internal workings of the application.
Also known as closed-box testing, data-driven testing, or functional testing.	Also known as translucent testing, as the tester has limited knowledge of the insides of the application.	Also known as clear-box testing, structural testing, or code-based testing.
Performed by end-users and also by testers and developers.	Performed by end-users and also by testers and developers.	Normally done by testers and developers.
Testing is based on external expectations - Internal	Testing is done on the basis of high-level database	Internal workings are fully known and the tester can

behaviour of the application is unknown.	diagrams and data flow diagrams.	design test data accordingly.
It is exhaustive and the least time-consuming.	Partly time-consuming and exhaustive.	The most exhaustive and time-consuming type of testing.
Not suited for algorithm testing.	Not suited for algorithm testing.	Suited for algorithm testing.
This can only be done by trial-and-error method.	Data domains and internal boundaries can be tested, if known.	Data domains and internal boundaries can be better tested.

1.4 TESTING LEVELS

There are different levels during the process of testing. In this chapter, a brief description is provided about these levels.

Levels of testing include different methodologies that can be used while conducting software testing. The main levels of software testing are:

- Functional Testing
- Non-functional Testing

Functional Testing

This is a type of black-box testing that is based on the specifications of the software that is to be tested. The application is tested by providing input and then the results are examined that need to conform to the functionality it was intended for. Functional testing of a software is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

There are five steps that are involved while testing an application for functionality.

Steps	Description
I	The determination of the functionality that the intended application is meant to perform.
II	The creation of test data based on the specifications of the application.
III	The output based on the test data and the specifications of the application.
IV	The writing of test scenarios and the execution of test cases.
V	The comparison of actual and expected results based on the executed test cases.

An effective testing practice will see the above steps applied to the testing policies of every organization and hence it will make sure that the organization maintains the strictest of standards when it comes to software quality.

Unit Testing

This type of testing is performed by developers before the setup is handed over to the testing team to formally execute the test cases. Unit testing is performed by the respective developers on the individual units of source code assigned areas. The developers use test data that is different from the test data of the quality assurance team.

The goal of unit testing is to isolate each part of the program and show that individual parts are correct in terms of requirements and functionality.

Limitations of Unit Testing

Testing cannot catch each and every bug in an application. It is impossible to evaluate every execution path in every software application. The same is the case with unit testing.

There is a limit to the number of scenarios and test data that a developer can use to verify a source code. After having exhausted all the options, there is no choice but to stop unit testing and merge the code segment with other units.

Integration Testing

Integration testing is defined as the testing of combined parts of an application to determine if they function correctly. Integration testing can be done in two ways: Bottom-up integration testing and Top-down integration testing.

S.N.	Integration Testing Method
1	Bottom-up integration This testing begins with unit testing, followed by tests of progressively higher-level combinations of units called modules or builds.
2	Top-down integration In this testing, the highest-level modules are tested first and progressively, lower-level modules are tested thereafter.

In a comprehensive software development environment, bottom-up testing is usually done first, followed by top-down testing. The process concludes with multiple tests of the complete application, preferably in scenarios designed to mimic actual situations.

System Testing

System testing tests the system as a whole. Once all the components are integrated, the application as a whole is tested rigorously to see that it meets the specified Quality Standards. This type of testing is performed by a specialized testing team.

System testing is important because of the following reasons:

- System testing is the first step in the Software Development Life Cycle, where the application is tested as a whole.
- The application is tested thoroughly to verify that it meets the functional and technical specifications.
- The application is tested in an environment that is very close to the production environment where the application will be deployed.
- System testing enables us to test, verify, and validate both the business requirements as well as the application architecture.

Regression Testing

Whenever a change in a software application is made, it is quite possible that other areas within the application have been affected by this change. Regression testing is performed to verify that a fixed bug hasn't resulted in another functionality or business rule violation. The intent of regression testing is to ensure that a change, such as a bug fix should not result in another fault being uncovered in the application.

Regression testing is important because of the following reasons:

- Minimize the gaps in testing when an application with changes made has to be tested.
- Testing the new changes to verify that the changes made did not affect any other area of the application.
- Mitigates risks when regression testing is performed on the application.
- Test coverage is increased without compromising timelines.
- Increase speed to market the product.

Acceptance Testing

This is arguably the most important type of testing, as it is conducted by the Quality Assurance Team who will gauge whether the application meets the intended specifications and satisfies the client's requirement. The QA team will have a set of pre-written scenarios and test cases that will be used to test the application.

More ideas will be shared about the application and more tests can be performed on it to gauge its accuracy and the reasons why the project was initiated. Acceptance tests are not only intended to point out simple spelling mistakes, cosmetic errors, or interface gaps, but also to point out any bugs in the application that will result in system crashes or major errors in the application.

By performing acceptance tests on an application, the testing team will deduce how the application will perform in production. There are also legal and contractual requirements for acceptance of the system.

Alpha Testing

This test is the first stage of testing and will be performed amongst the teams (developer and QA teams). Unit testing, integration testing and system testing when combined together is known as alpha testing. During this phase, the following aspects will be tested in the application:

- Spelling Mistakes
- Broken Links
- Cloudy Directions
- The Application will be tested on machines with the lowest specification to test loading times and any latency problems.

Beta Testing

This test is performed after alpha testing has been successfully performed. In beta testing, a sample of the intended audience tests the application. Beta testing is also known as **pre-release testing**. Beta test versions of software are ideally distributed to a wide audience on the Web, partly to give the program a "real-world" test and partly to provide a preview of the next release. In this phase, the audience will be testing the following:

- Users will install, run the application and send their feedback to the project team.
- Typographical errors, confusing application flow, and even crashes.

- Getting the feedback, the project team can fix the problems before releasing the software to the actual users.
- The more issues you fix that solve real user problems, the higher the quality of your application will be.
- Having a higher-quality application when you release it to the general public will increase customer satisfaction.

Non-Functional Testing

This section is based upon testing an application from its non-functional attributes. Non-functional testing involves testing a software from the requirements which are nonfunctional in nature but important such as performance, security, user interface, etc.

Some of the important and commonly used non-functional testing types are discussed below.

Performance Testing

It is mostly used to identify any bottlenecks or performance issues rather than finding bugs in a software. There are different causes that contribute in lowering the performance of a software:

- Network delay
- Client-side processing
- Database transaction processing
- Load balancing between servers
- Data rendering

Performance testing is considered as one of the important and mandatory testing type in terms of the following aspects:

- Speed (i.e. Response Time, data rendering and accessing)
- Capacity
- Stability
- Scalability

Performance testing can be either qualitative or quantitative and can be divided into different sub-types such as **Load testing** and **Stress testing**.

Load Testing

It is a process of testing the behavior of a software by applying maximum load in terms of software accessing and manipulating large input data. It can be done at both normal and peak load conditions. This type of testing identifies the maximum capacity of software and its behavior at peak time.

Most of the time, load testing is performed with the help of automated tools such as Load Runner, AppLoader, IBM Rational Performance Tester, Apache JMeter, Silk Performer, Visual Studio Load Test, etc.

Virtual users (VUsers) are defined in the automated testing tool and the script is executed to verify the load testing for the software. The number of users can be increased or decreased concurrently or incrementally based upon the requirements.

Stress Testing

Stress testing includes testing the behavior of a software under abnormal conditions. For example, it may include taking away some resources or applying a load beyond the actual load limit.

The aim of stress testing is to test the software by applying the load to the system and taking over the resources used by the software to identify the breaking point. This testing can be performed by testing different scenarios such as:

- Shutdown or restart of network ports randomly
- Turning the database on or off
- Running different processes that consume resources such as CPU, memory, server, etc.

Usability Testing

Usability testing is a black-box technique and is used to identify any error(s) and improvements in the software by observing the users through their usage and operation.

According to Nielsen, usability can be defined in terms of five factors, i.e. efficiency of use, learn-ability, memory-ability, errors/safety, and satisfaction. According to him, the usability of a product will be good and the system is usable if it possesses the above factors.

Nigel Bevan and Macleod considered that usability is the quality requirement that can be measured as the outcome of interactions with a computer system. This requirement can be fulfilled and the end-user will be satisfied if the intended goals are achieved effectively with the use of proper resources.

Molich in 2000 stated that a user-friendly system should fulfill the following five goals, i.e., easy to Learn, easy to remember, efficient to use, satisfactory to use, and easy to understand.

In addition to the different definitions of usability, there are some standards and quality models and methods that define usability in the form of attributes and sub-attributes such as ISO-9126, ISO-9241-11, ISO-13407, and IEEE std.610.12, etc.

UI vs Usability Testing

UI testing involves testing the Graphical User Interface of the Software. UI testing ensures that the GUI functions according to the requirements and tested in terms of color, alignment, size, and other properties.

On the other hand, usability testing ensures a good and user-friendly GUI that can be easily handled. UI testing can be considered as a sub-part of usability testing.

Security Testing

Security testing involves testing a software in order to identify any flaws and gaps from security and vulnerability point of view. Listed below are the main aspects that security testing should ensure:

- Confidentiality
- Integrity
- Authentication
- Availability
- Authorization
- Non-repudiation
- Software is secure against known and unknown vulnerabilities
- Software data is secure
- Software is according to all security regulations
- Input checking and validation
- SQL insertion attacks
- Injection flaws
- Session management issues
- Cross-site scripting attacks
- Buffer overflows vulnerabilities
- Directory traversal attacks

Portability Testing

Portability testing includes testing a software with the aim to ensure its reusability and that it can be moved from another software as well. Following are the strategies that can be used for portability testing:

- Transferring an installed software from one computer to another.
- Building executable (.exe) to run the software on different platforms.

Portability testing can be considered as one of the sub-parts of system testing, as this testing type includes overall testing of a software with respect to its usage over different environments. Computer hardware, operating systems, and browsers are the major focus of portability testing. Some of the pre-conditions for portability testing are as follows:

- Software should be designed and coded, keeping in mind the portability requirements.
- Unit testing has been performed on the associated components.
- Integration testing has been performed.
- Test environment has been established.

1.5 OPERATIONAL STEPS AND THE TESTING PLAN

The testing process must execute operational steps identified in the test plan. Test planning should be done throughout the development cycle, especially early in the development cycle.

A test plan is a document describing the scope, approach, resources, and schedule of intended test activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency plans. An important component of the test plan is the individual test cases.

A test case is a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

NOTE

Write the test plan early in the development cycle when things are generally still going pretty smoothly and calmly.

This allows you to think through a thorough set of test cases. If you wait until the end of the cycle to write and execute test cases, you might be in a very chaotic, hurried time period. Often good test cases are not written in this hurried environment, and ad hoc testing takes place. With ad hoc testing, people just start trying anything they can think of without any rational roadmap through the customer requirements. The tests done in this manner are not repeatable.

1.6 TESTING PROCESS

It is also very important to consider test planning and test execution as iterative processes. As soon as requirements documentation is available, it is best to begin to write functional and system test cases. When requirements change, revise the test cases. As soon as some code is available, execute test cases. When code changes, run the test cases again. By knowing how many and which test cases actually run you can accurately track the progress of the project. All in all, testing should be considered an iterative and essential part of the entire development process.

1.6.1 PERFORMING BLACK BOX TESTING

Black box testing, also called functional testing and behavioural testing, focuses on determining whether or not a program does what it is supposed to do based on its functional requirements.

Black box testing attempts to find errors in the external behaviour of the code in the following categories:

(1) Incorrect or missing functionality;
(2) Interface errors;
(3) Errors in data structures used by interfaces;
(4) Behaviour or performance errors; and
(5) Initialization and termination errors.

Through this testing, we can determine if the functions appear to work according to specifications. However, it is important to note that no amount of testing can unequivocally demonstrate the absence of errors and defects in your code. It is best if the person who plans and executes black box tests is not the programmer of the code and does not know anything about the structure of the code. The programmers of the code are innately biased and are likely to test that the program does what they programmed it to do. What are needed are tests to make sure that the program does what the customer wants it to do. As a result, most organizations have independent testing groups to perform black box testing. These testers are not the developers and are often referred to as third-party testers. Testers should just be able to understand and specify what the desired output should be for a given input into the program, as shown in Figure.

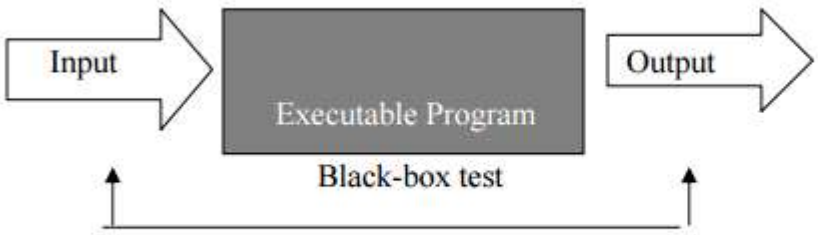


Figure 3: Black Box Testing. A black-box test takes into account only the input and output of the software without regard to the internal code of the program.

Strategies for Black Box Testing

Ideally, we'd like to test every possible thing that can be done with our program. But, as we said, writing and executing test cases is expensive. We want to make sure that we definitely write test cases for the kinds of things that the customer will do most often or even fairly often. Our objective is to find as many defects as possible in as few test cases as possible. To accomplish this objective, we use some strategies that will be discussed in this subsection. We want to avoid writing redundant test cases that won't tell us anything new (because they have similar conditions to other test cases we already wrote). Each test case should probe a different mode of failure. We also want to design the simplest test cases that could possibly reveal this mode of failure – test cases themselves can be error-prone if we don't keep this in mind.

1. Tests of Customer Requirements

   Black box test cases are based on customer requirements. We begin by looking at each customer requirement. To start, we want to make sure that every single customer requirement has been tested at least once. As a result, we can trace every requirement to its test case(s) and every test case back to its stated customer requirement. The first test case we'd write for any given requirement is the most-used success path for that requirement. By success path, we mean that we want to execute some desirable functionality (something the customer wants to work) without any error conditions. We proceed by planning more success path test cases, based on other ways the customer wants to use the functionality and some test cases that execute failure paths. Intuitively, failure paths intentionally have some kind of errors in them, such as errors that users can accidentally input. We must make sure that the program behaves predictably and gracefully in the face of these errors. Finally, we should plan the execution of our tests out so that the most troublesome, risky requirements are tested first. This would allow more time for fixing problems before delivering the product to the customer. It would be devastating to find a critical flaw right before the product is due to be delivered. We'll start with one basic requirement. We can write

many test cases based on this one requirement, which follows below. As we've said before, it is impossible to test every single possible combination of input. We'll outline an incomplete sampling of test cases and reason about them in this section.

> *Requirement: When a user lands on the "Go to Jail" cell, the player goes directly to jail, does not pass go, does not collect $200. On the next turn, the player must pay $50 to get out of jail and does not roll the dice or advance. If the player does not have enough money, he or she is out of the game.*

There are many things to test in this short requirement above, including:

1. Does the player get sent to jail after landing on "Go to Jail"?
2. Does the player receive $200 if "Go" is between the current space and jail?
3. Is $50 correctly decremented if the player has more than $50? 4. Is the player out of the game if he or she has less than $50?

At first it is good to start out by testing some input that you know should definitely pass or definitely fail. If these kinds of tests don't work properly, you know you should just quit testing and put the code back into development. We can start with a two obvious passing test case, as shown in Table 5.

Test ID	Description	Expected Results	Actual Results
5	Precondition: Game is in test mode. Number of players: 1 Money for player 1: $1200 Player 1 dice roll: 3 Player 1 clicks "End Turn" button.		
		Player 1 is sent to jail Only "Get Out of Jail" button is enabled for Player 1.	
	Player 1 clicks "Get Out of Jail" button.		
		Money for Player 1: $1150	
6	Precondition: Game is in test mode. Number of players: 2 Money for player 1: $1200 Money for player 2: $1200 Player 1 dice roll: 3 Player 1 clicks "End Turn" button.		
		Player 1 is sent to jail	
	Player 2 dice roll: 2 Player 2 clicks "End Turn" button.		
		Only "Get Out of Jail" button is enabled for Player 1.	
	Player 1 clicks "Get out of Jail" button.		
		Money for Player 1: $1150	

Table 5: Test Plan #1 for the Jail Requirement

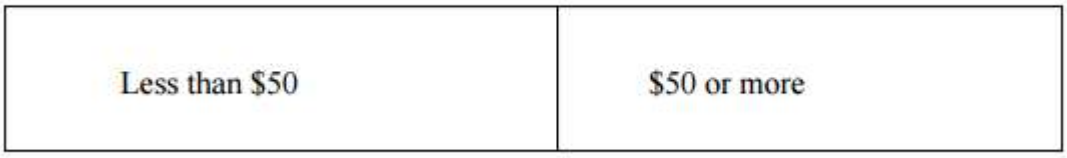
You will also note that we should test the simplest possible means to force the condition we are trying to achieve. For example, in Test Case 5, we only have one player so we temporarily didn't have to spend our time with Player 2. We add Player 2 in Test Case 6 so we can observe that the loss of $50 and dice roll occurs on the next turn (after Player 2 goes). We could go on and test many more aspects of the above requirement. We will now discuss some strategies to consider in creating more test cases.

2. Equivalence partitioning

To keep down our testing costs, we don't want to write several test cases that test the same aspect of our program. A good test case uncovers a different class of errors (e.g., incorrect processing of all character data) than has been uncovered by prior test cases. Equivalence partitioning is a strategy that can be used to reduce the number of test cases that need to be developed. Equivalence partitioning divides the input domain of a program into classes. For

each of these equivalence classes, the set of data should be treated the same by the module under test and should produce the same answer. Test cases should be designed so the inputs lie within these equivalence classes. For example, for tests of “Go to Jail” the most important thing is whether the player has enough money to pay the $50 fine. Therefore, the two equivalence classes can be partitioned, as shown in Figure 4.

Figure 4: Equivalence Classes for Player Money

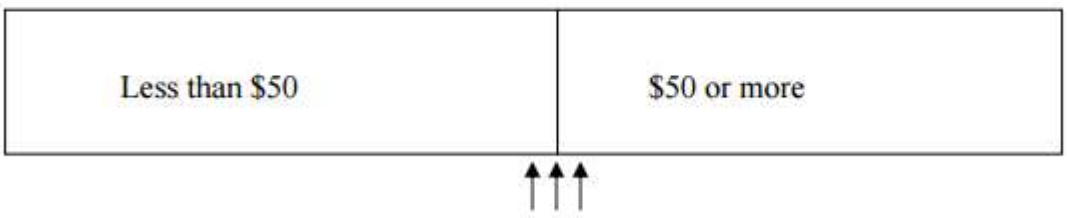


Once you have identified these partitions, you choose test cases from each partition. To start, choose a typical value somewhere in the middle of (or well into) each of these two ranges.

3. Boundary Value Analysis

Boris Beizer, well-known author of testing book advises, “Bugs lurk in corners and congregate at boundaries.” Programmers often make mistakes on the boundaries of the equivalence classes/input domain. As a result, we need to focus testing at these boundaries. This type of testing is called Boundary Value Analysis (BVA) and guides you to create test cases at the “edge” of the equivalence classes. Boundary value is defined as a data value that corresponds to a minimum or maximum input, internal, or output value specified for a system or component . In our above example, the boundary of the class is at 50, as shown in Figure 5. We should create test cases for the Player 1 having $49, $50, and $51. These test cases will help to find common off-by-one errors, caused by errors like using >= when you mean to use >.

Figure 5: Boundary Value Analysis. Test cases should be created for the boundaries (arrows) between equivalence classes.



When creating BVA test cases, consider the following [17]:

1. If input conditions have a range from a to b (such as a=100 to b=300), create test cases:
    - Immediately below a (99)
    - at a (100)
    - Immediately above a (101)
    - immediately below b (299)
    - At b (300)
    - immediately above b (301)
2. If input conditions specify a number of values that are allowed, test these limits. For example, input conditions specify that only one train is allowed to start in each direction on each station. In testing, try to add a second train to the same station/same direction. If (somehow) three trains could start on one station/direction, try to add two trains (pass), three trains (pass), and four trains (fail).

Black Box Test Case Automation

By their nature, black box test cases are designed and run by people who do not see the inner workings of the code. Ultimately, system and acceptance cases are intended to be run through the product user interface (UI) to show that the whole product really works.

Test automation can be difficult because the developer has no knowledge of the inner workings of the software and because system and acceptance cases must be run through the UI. However, the more automated testing can be, the easier it is to run the test cases and to re-run them again and again. The simpler it is to run a suite of tests, the more often those tests will be run. The more the tests are run, the faster any deviation from those tests will be found. If your role on the team is as a software developer, it is always good to consider the types of black box test cases (functional, system, and acceptance) that will ultimately be run on your code and to automate test cases to test the logic (separate from the UI logic) behind these black box test cases. Automated test cases can be run often with minimal time investment once they are written. By automating the testing of the logic behind the black box test cases, (1) you are ensuring that the logic “behind the scenes” is working properly so that the inevitable black box test cases can run smoothly through the UI by the testers and the customers; and (2) you are more motivated to decouple program/business logic separate from the UI logic (which is always a good design technique).

1.6.2 WHITE BOX TESTING PROCESS

White-box testing is testing that takes into account the internal mechanism of a system or component (IEEE, 1990). White-box testing is also known as structural testing, clear box testing, and glass box testing (Beizer, 1995). The connotations of “clear box” and “glass box” appropriately indicate that you have full visibility of the internal workings of the software product, specifically, the logic and the structure of the code.

White-Box Testing by Stubs and Drivers

With white-box testing, you must run the code with predetermined input and check to make sure that the code produces predetermined outputs. Often programmers write stubs and drivers for white-box testing. A driver is a software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results (IEEE, 1990) or most simplistically a line of code that calls a method and passes that method a value. For example, if you wanted to move a Player instance,Player1, two spaces on the board, the driver code would be

Move Player (Player1, 2);

This driver code would likely be called from the main method. A white-box test case would execute this driver line of code and check Player get Position () to make sure the player is now on the expected cell on the board. A stub is a computer program statement substituting for the body of a software module that is or will be defined elsewhere (IEEE, 1990) or a dummy component or object used to simulate the behavior of a real component (Beizer, 1990) until that component has been developed. For example, if the movePlayer method has not been written yet, a stub such as the one below might be used temporarily – which moves any player to position 1.

```
public void movePlayer(Player player, int diceValue) {
        player.setPosition(1);
}
```

Ultimately, the dummy method would be completed with the proper program logic. However, developing the stub allows the programmer to call a method in the code being developed, even if the method does not yet have the desired behavior. Stubs and drivers are often viewed as throwaway code (Kaner, Falk et al., 1999). However, they do not have to be

thrown away: Stubs can be “filled in” to form the actual method. Drivers can become automated test cases.

DERIVING TEST CASES

In the following sections, we will discuss various methods for devising a thorough set of white-box test cases.

1. Basis Path Testing Basis path testing (McCabe, 1976) is a means for ensuring that all independent paths through a code module have been tested. An independent path is any path through the code that introduces at least one new set of processing statements or a new condition. (Pressman, 2001) Basis path testing provides a minimum, lower-bound on the number of test cases that need to be written. To introduce the basis path method, we will draw a flowgraph of a code segment. Once you understand basis path testing, it may not be necessary to draw the flowgraph – though you may always find a quick sketch helpful. If you test incrementally and the modules you test are small enough, you can consider having a mental picture of the flow graph. As you will see, the main objective is to identify the number of decision points in the module and you may be able to identify them without a written representation. A flowgraph of purchasing property appears in Figure 1. The flowgraph is intended to depict the following requirement.

If a player lands on a property owned by other players, he or she needs to pay the rent. If the player does not have enough money, he or she is out of the game. If the property is not owned by any players, and the player has enough money buying the property, he or she may buy the property with the price associated with the property.

In the simple flowgraph in Figure 2, a rectangle shows a sequence of processing steps that are executed unconditionally. A diamond represents a logic conditional or predicate. Some examples of logical conditionals are if-then, if-then-else, selection, or loops. The head of the arrow indicates the flow of control. For a rectangle, there will be one arrow heading out. For a predicate, there will be two arrows heading out – one for a true/positive result and the other for a false/negative result.

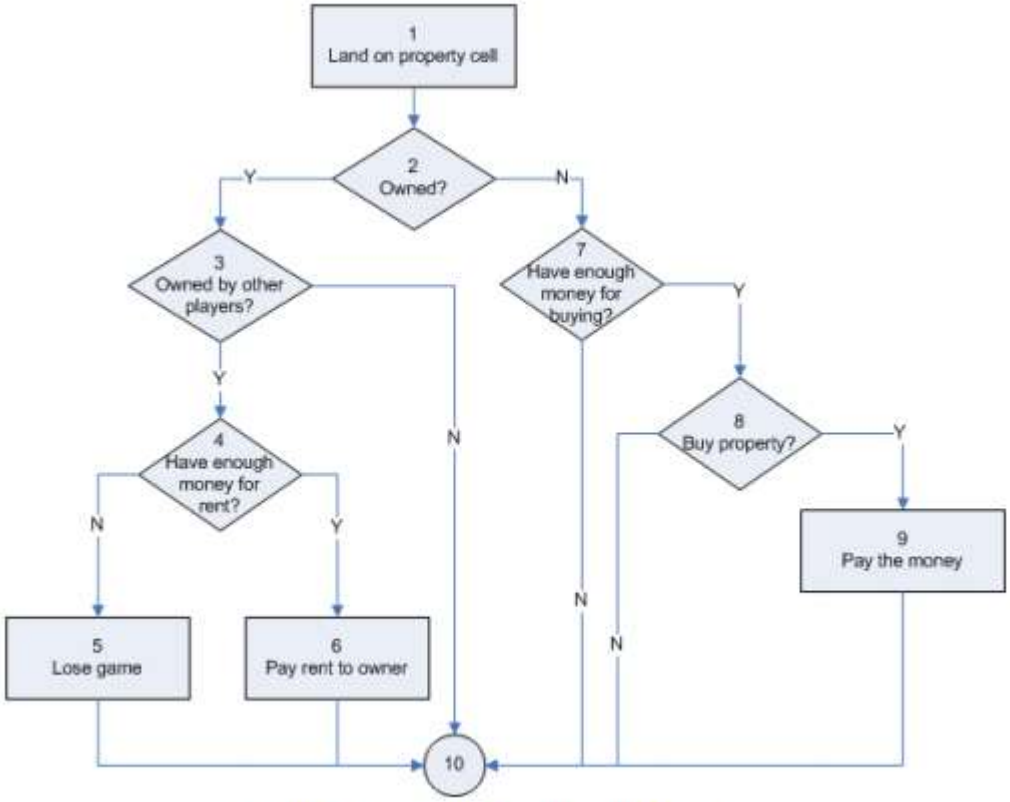


Figure 1: Flowgraph of purchasing property

Using this flow graph, we can compute the number of independent paths through the code. We do this using a metric called the cyclomatic number (McCabe, 1976), which is based on graph theory. The easiest way to compute the cyclomatic number is to count the number of conditionals/predicates (diamonds) and add 1. In our example above, there are five conditionals. Therefore, our cyclomatic number is 6, and we have six independent paths through the code. We can now enumerate them:

1. 1-2-3-4-5-10 (property owned by others, no money for rent)
2. 1-2-3-4-6-10 (property owned by others, pay rent)
3. 1-2-3-10 (property owned by the player)
4. 1-2-7-10 (property available, don't have enough money)
5. 1-2-7-8-10 (property available, have money, don't want to buy it)
6. 1-2-7-8-9-10 (property available, have money, and buy it)

We would want to write a test case to ensure that each of these paths is tested at least once. As said above, the cyclomatic number is the lower bound on the number of test cases we will write. The test cases that are determined this way are the ones we use in basis path testing.

Control-flow/Coverage

Testing another way to devise a good set of white-box test cases is to consider the control flow of the program. The control flow of the program is represented in a flow graph, as shown in Figure 1. We consider various aspects of this flowgraph in order to ensure that we have an adequate set of test cases. The adequacy of the test cases is often measured with a metric called coverage. Coverage is a measure of the completeness of the set of test cases. To demonstrate the various kinds of coverage, we will use the simple code example shown in Figure 2 as a basis of discussion as we take up the next five topics.

```
1    int foo (int a, int b, int c, int d, float e)  {
2          float e;
3          if (a == 0)  {
4               return 0;
5          }
6          int x = 0;
7          if ((a==b) OR ((c == d) AND bug(a) )) {
8               x=1;
9          }
10          e = 1/x;
11    return e;
12     }
```

Figure 2: Sample Code for Coverage Analysis

Keeping with a proper testing technique, we write methods to ensure they are testable – most simply by having the method return a value. Additionally, we predetermine the "answer" that is returned when the method is called with certain parameters so that our testing returns that predetermined value. Another good testing technique is to use the simplest set of input that could possibly test your situation – it's better not to input values that cause complex, error-prone calculations when you are predetermining the values. We'll illustrate this principle as we go through the next items.

1. Method Coverage Method coverage is a measure of the percentage of methods that have been executed by test cases. Undoubtedly, your tests should call 100% of your methods. It seems irresponsible to deliver methods in your product when your testing never used these methods. As a result, you need to ensure you have 100% method coverage. In the code shown in Figure 3, we attain 100% method coverage by calling the foo method. Consider Test Case 1: the method call foo(0, 0, 0, 0, 0.), expected return value of 0. If you look at the code, you see that if a has a value of 0, it doesn't matter what the values of the other parameters are – so we'll make it really easy and make them all 0. Through this one call we attain 100% method coverage.
2. Statement Coverage Statement coverage is a measure of the percentage of statements that have been executed by test cases. Your objective should be to

achieve 100% statement coverage through your testing. Identifying your cyclomatic number and executing this minimum set of test cases will make this statement coverage achievable. In Test Case 1, we executed the program statements on lines 1-5 out of 12 lines of code. As a result, we had 42% (5/12) statement coverage from Test Case 1. We can attain 100% statement coverage by one additional test case, Test Case 2: the method call foo(1, 1, 1, 1, 1.), expected return value of 1. With this method call, we have achieved 100% statement coverage because we have now executed the program statements on lines 6-12.

3. Branch Coverage Branch coverage is a measure of the percentage of the decision points (Boolean expressions) of the program have been evaluated as both true and false in test cases.
4. Condition Coverage We will go one step deeper and examine condition coverage. Condition coverage is a measure of percentage of Boolean sub-expressions of the program that have been evaluated as both true or false outcome [applies to compound predicate] in test cases. Notice that in line 7 there are three sub-Boolean expressions to the larger statement (a==b), (c==d), and bug(a). Condition coverage measures the outcome of each of these sub-expressions independently of each other. With condition coverage, you ensure that each of these subexpressions has independently been tested as both true and false. We consider our progress thus far in Table 2

Table 2: Condition coverage

Predicate	True	False
(a==b)	Test Case 2 **foo(1, 1, x, x, 1)** **return value 0**	Test Case 3 **foo(1, 2, 1, 2, 1)** **division by zero!**
(c==d)		Test Case 3 **foo(1, 2, 1, 2, 1)** **division by zero!**
bug(a)		

At this point, our condition coverage is only 50%. The true condition (c==d) has never been tested. Additionally, short-circuit Boolean has prevented the method bug(int) from ever being executed. We examine our available information on the bug method and determine that is should return a value of true when passed a value of a=1. We write Test Case 4 to address test (c==d) as true: foo(1, 2, 1, 1, 1), expected return value 1. However, when we actually run the test case, the function bug(a) actually returns false, which causes our actual return value (division by zero) to not match our expected return value. This allows us to detect an error in

the bug method. Without the addition of condition coverage, this error would not have been revealed. To finalize our condition coverage, we must force bug(a) to be false. We again examine our bug() information, which informs us that the bug method should return a false value if fed any integer greater than 1. So we create Test Case 5, foo(3, 2, 1, 1, 1), expected return value “division by error”. The condition coverage thus far is shown in Table 15.3.

Table 3: Condition Coverage Continued

Predicate	True	False
(a==b)	Test Case 2 **foo(1, 1, 1, 1, 1)** **return value 0**	Test Case 3 **foo(1, 2, 1, 2, 1)** **division by zero!**
(c==d)	Test Case 4 **foo(1, 2, 1, 1, 1)** return value 1	Test Case 3 **foo(1, 2, 1, 2, 1)** **division by zero!**
bug(a)	Test Case 4 **foo(1, 2, 1, 1, 1)** return value 1	Test Case 5 **foo(3, 2, 1, 1, 1)** **division by zero!**

There are no industry standard objectives for condition coverage, but we suggest that you keep condition coverage in mind as you develop your test cases. You have seen that our condition coverage revealed that some additional test cases were needed. There are commercial tools available, called coverage monitors, that can report the coverage metrics for your test case execution. Often these tools only report method and statement coverage. Some tools report decision/branch and/or condition coverage. These tools often also will color code the lines of code that have not been executed during your test efforts. It is recommended that coverage analysis is automated using such a tool because manual coverage analysis is unreliable and uneconomical (IEEE, 1987).

Data Flow Testing

In data flow-based testing, the control flowgraph is annotated with information about how the program variables are defined and used. Different criteria exercise with varying degrees of precision how a value assigned to a variable is used along different control flow paths. A reference notation is a definition-use pair, which is a triple of (d, u, V) such that V is a variable, d is a note in which V is defined, and us is a node in which V is used. There exists a path between d and u in which the definition of V in d is used in u.

Failure (“Dirty”) Test Cases

As with black-box test cases, you must think diabolically about the kinds of things users might do with your program. Look at the structure of your code and think about every possible way

a user might break it. These devious ways may not be uncovered by the previously mentioned methods for forming test cases. You need to be smart enough to think of your particular code and how people might outsmart it (accidentally or intentionally). Augment your test cases to handle these cases. Some suggestions follow: • Look at every input into the code you are testing. Do you handle each input if it is incorrect, the wrong font, or too large (or too small)? • Look at code from a security point of view. Can a user overflow a buffer, causing a security problem? • Look at every calculation. Could it possible create an overflow? Have you protected from possible division by zero?

SESSION 2: RECORDING RESULTS FROM TESTING A COMPUTER PROGRAM

On completion of this section you will be able to record the results from testing a computer program.

1. The records are provided for all tests executed.
2. The records identify variations from expected test results and gives reason where available.
3. The recorded results are reproduced if the tests are repeated under the same conditions.
4. The recorded results are recorded in a way that allows the results to be reviewed

2.1 INTRODUCTION

Test documentation is the vital element which raises any "try out" or "experimental" activities to the level of a proper test. Even if the way something is tested is good, it is worthless if it is not documented. How test documentation should look like is e.g. specified in the IEEE Standard 829. The IEEE standard allows scaling the set of documents by a certain degree. But basically this standard boils down to:

Test plan

The scope of the test activities, the methods and tools, the schedule and sequence of all test activities related to the SW have to be stated and defined in this plan. The test objects have to be identified as well as the attributes which have to be tested and the related end of test criteria must be fixed. Responsibilities and risks have to be identified and documented.

Test design specification

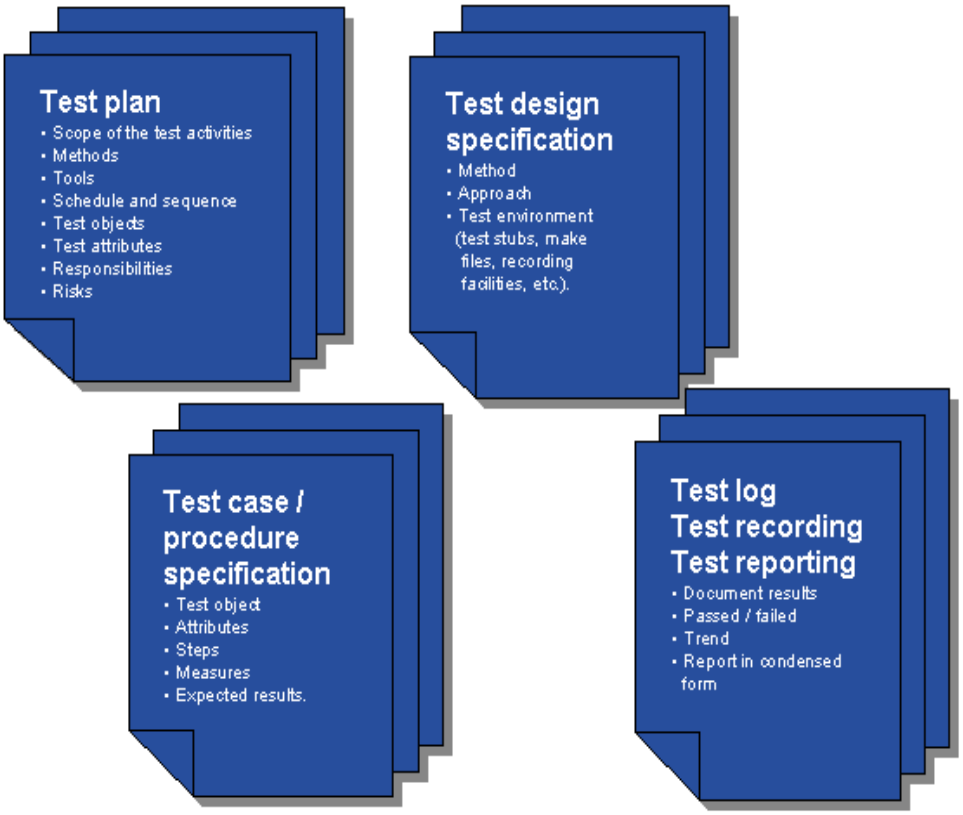
The method and approach for the tests has to be defined. In some cases it may be necessary to design a sophisticated test environment (test stubs, make files, recording facilities, etc.). All the technical details have to be specified, designed and documented.

Test case specification / Test procedure specification

In the test case specification the test object has to be identified as well as the attributes which have to be tested. It has to be made clear which steps and measures have to be applied to execute the test cases and which results are expected.

Test log / Test recording / Test reporting

The test results have to be documented and it has to be identified if the test ended with the expected results i.e. if they passed or failed. The test recording strongly depends on the test environment. In some cases this can be an automatic printout. In other cases this may be a check list which is ticked by the tester (may be even included in the test procedure / test case specification!). It may be even necessary to apply different methods within the same project, depending on the kind of test object and the kind of test employed. A preparation of the test results in a report is required. Test logs may be voluminous and have to be condensed to have their contents prepared for a quick overview and reference as well as for management or customer presentations.

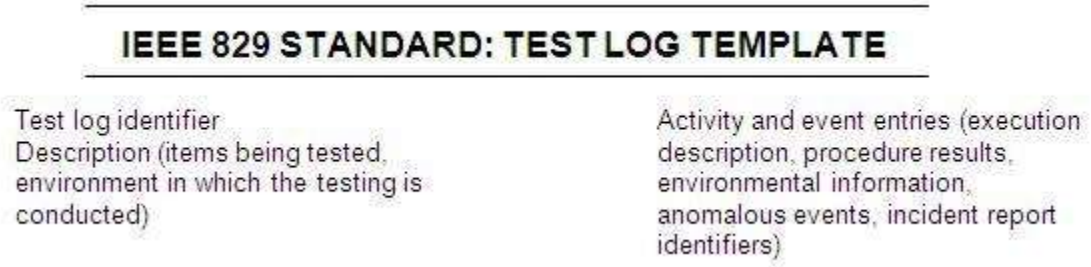


2.2 TEST DOCUMENTATION

There are a range of documentations that must be kept inorder to record and keep test results in computer programs.

2.2.1 TEST LOG

The IEEE Std. 829-1998 defines a test log as a chronological record of relevant details about the execution of test cases. It's a detailed view of activity and events given in chronological manner.



Let us take an example as shown in Figure 5.1, columns A and B show the test ID and the test case or test suite name. The state of the test case is shown in column C ('Warn' indicates a test that resulted in a minor failure). Column D shows the tested configuration, where the codes A, B and C correspond to test environments described in detail in the test plan. Columns E and F show the defect (or bug) ID number (from the defect-tracking database) and the risk priority number of the defect (ranging from 1, the worst, to 25, the least risky). Column G shows the initials of the tester who ran the test. Columns H through L capture data for each test related to dates, effort and duration (in hours). We have metrics for planned and actual effort and dates completed which would allow us to summarize progress against the planned schedule and budget. This spreadsheet can also be summarized in terms of the percentage of tests which have been run and the percentage of tests which have passed and failed.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1		System Test Case Summary											
2		Cycle One											
3													
4	Test			System	Bug	Bug	Run	Plan	Act	Plan	Actual	Test	
5	ID	Test Suite/Case	Status	Config	ID	RPN	By	Date	Date	Effort	Effort	Duration	Comment
6													
7	1.000	*Functionality*											
8	1.001	File	Fail	A	701	1	LTW	1/8	1/8	4	6	6	
9	1.002	Edit	Fail	A	709	1	LTW	1/9	1/10	4	8	8	
10					710	5							
11					718	3							
12					722	4							
13	1.003	Font	Pass	B			IHB	1/10	1/10	4	4	4	
14	1.004	Tables	Warn	B	708	15	IHB	1/8	1/9	4	5	5	
15	1.005	Printing	Skip					1/10		4			Out of runway
16		**Suite Summary**						**1/10**	**1/10**	**20**	**23**	**23**	
17													
18	2.000	*Performance/Stress*											
19	2.001	Solaris Server	Warn	A,B,C	701	1	EM	1/10	1/13	4	8	24	Replan 1/11
20	2.002	NT Server	Fail	A,B,C	724	2	EM	1/11	1/14	4	4	24	Replan 1/12
21					713	2							
22					725	1							
23	2.003	Linux Server	Skip					1/12		4			Out of runway
24		**Suite Summary**						**1/12**	**1/14**	**12**	**12**	**48**	
25													
26	3.000	*Error Handling/Recovery*											
27	3.001	Corrupt File	Fail	A	701	1	LTW	1/8	1/9	4	8	8	
28					706	2							
29					707	4							
30					709	1							
31					710	5							
32					713	2							
33	3.002	Server Crash	Fail	A	712	6	LTW	1/9	1/10	4	6	6	
34					713	2							
35					717	1							
36		**Suite Summary**						**1/9**	**1/10**	**8**	**14**	**14**	
37													
38	4.000	*Localization*											
39	4.001	Spanish	Skip										
40	4.002	French	Skip										
41	4.003	Japanese	Skip										
42	4.004	Chinese	Skip										
43		**Suite Summary**						**1/0**	**1/0**	**0**	**0**	**0**	

FIGURE 5.1 Test case summary worksheet

Example

Application Entities Interconnectivity

Level_4 Test Report Example

Test Log, Example System 2

i) Purpose:

To provide a chronological record of relevant details about the execution of interconnectivity tests between a Picker PQ2000 CT scanner and a General Electric Advantage Review Workstation. This log represents the results of testing of General Electric Advantage Review Workstation.

ii. Outline

A test log shall have the following structure:

(1) Test-log identifier

(2) Description
(3)) Activity and event entries

1. Test-Log Identifier

The Test Log Identifier shall be a unique identifier for both suppliers of the Application Entities tested. This Test Log Example has the ID: 1.2.840.113702.1.4.3.1.

2. Description

The items to be tested for interconnectivity are the Local DICOM Import Application Entity as implemented in Software Revision 1.03 of the General Electric Advantage Review Workstation and the HANI Application Entity as implemented in Software Revisions 4.2 and 4.3A of the Picker PQ 2000 CT Scanner.

A GE Advantage Review Workstation which is in routine clinical use was utilized for testing after normal working hours. Similarly, the Picker PQ2000 CT scanner was temporarily taken out of service and dedicated for testing. The Advantage Review workstation was connected via level 5 Unshielded Twisted Pair to a Lantronix 10BaseT wiring concentrator/repeater. The Picker CT scanner was connected via 10Base5 coaxial cable to a Cabletron Mini-MMAC configured as a bridge. Both the Lantronix and Cabletron units were connected via multi-mode optical fiber to two different ports on a Cisco 2500 series enterprise router. This router is part of the campus backbone of the Milton S. Hershey Medical Center and Penn State College of Medicine. The network connecting the two devices under test experienced representative (albeit off hours) medical center network traffic during the testing period.

3. Activity and Event Entries

3.1 Execution Description

The Joint Test Schema (ID: 1.2.840.113702.1.4.1.1) was executed on November 3-4, 1995 by Fred Prior, Ph.D. and Franklyn Bradshaw, Ph.D.

3.2 Procedure Results

Test Case	Test Case ID	Results	P/F
Basic TCP/IP connectivity	1.2.840.113702.1.3.14.1	pqeast is alive	Pass
Basic Association Establishment and image	1.2.840.113702.1.3.15.1	Data Set 1: 26 images; "DICOM Import Received : 26 Images	Pass

transfer (SCU and SCP roles)		from HANI"	
Association Establishment - Single Association Mode	1.2.840.113702.1.3.16.1	Data Sets 2-5: 55, 49, 45, 48 images; "DICOM Import Received : {55, 49, 45, 48} Images from HANI"	Pass
Repeatability of Transfers	1.2.840.113702.1.3.17.1	Data Set 6: 9 images; "DICOM Import Received : 9 Images from HANI"; study appeared twice in the local directory.	Pass ?
Association Abort	1.2.840.113702.1.3.18.1	Data Set 7: 28 images; "DICOM Import Received : 15 Images from HANI"	Pass ?
Association Establishment - Multiple Association Mode	1.2.840.113702.1.3.15.1	Data Set 8: 44 images; "DICOM Import Received : 44 Images from HANI"	Pass
Association Establishment - Multiple Association Mode	1.2.840.113702.1.3.16.1	Data Sets 9-12: 61, 58, 70, 65 images; "DICOM Import Received : {61, 58, 70, 65} Images from HANI"	Pass
Boundary Behavior - Multiple Simultaneous Associations	1.2.840.113702.1.3.19.1	Data Sets 13-20: approx. 60 images each; "DICOM Import Received : {n} Images from HANI"	Pass
Boundary Behavior - Resource boundary	1.2.840.113702.1.3.20.1	Data Set 21: 44 images; "DICOM Import Received : 44 Images from HANI"; study did not appear in the local directory."	?

3.3 Environmental Information

3.4 Anomalous Events

It was anticipated that the Repeatability of Transfers Test Case (ID: 1.2.840.113702.1.3.6.2) would generate an error (warning) condition. It did not. The same case was transmitted twice

and accepted by the SCP both times.

The Association Abort test yielded an anomalous result. The SCU aborted after 2 images were successfully transferred, however the SCP received 15 images. Twenty eight images were originally queued for transfer.

The Disk Full boundary test yielded an anomalous result. It was assumed that an error status (warning) would be returned indicating the disk full condition. A success status was returned but the images were not inserted into the local directory.

3.5 Incident-Report Identifiers

A joint Incident Report covering both SCU and SCP behavior was generated (ID: 1.2.840.113702.1.4.5.1).

2.2.2 TEST INCIDENT REPORT

After logging the incidents that occur in the field or after deployment of the system we also need some way of reporting, tracking, and managing them. It is most common to find defects reported against the code or the system itself. However, there are cases where defects are reported against requirements and design specifications, user and operator guides and tests also.

Why to report the incidents?

There are many benefits of reporting the incidents as given below:

- In some projects, a very large number of defects are found. Even on smaller projects where 100 or fewer defects are found, it is very difficult to keep track of all of them unless you have a process for reporting, classifying, assigning and managing the defects from discovery to final resolution.
- An incident report contains a description of the misbehaviour that was observed and classification of that misbehaviour.
- As with any written communication, it helps to have clear goals in mind when writing. One common goal for such reports is to provide programmers, managers and others with detailed information about the behaviour observed and the defect.
- Another is to support the analysis of trends in aggregate defect data, either for understanding more about a particular set of problems or tests or for understanding and reporting the overall level of system quality. Finally, defect reports, when

analyzed over a project and even across projects, give information that can lead to development and test process improvements.

- The programmers need the information in the report to find and fix the defects. Before that happens, though, managers should review and prioritize the defects so that scarce testing and developer resources are spent fixing and confirmation testing the most important defects.

While many of these incidents will be user error or some other behavior not related to a defect, some percentage of defects gets escaped from quality assurance and testing activities.

The **defect detection percentage,** which compares field defects with test defects, is an important metric of the effectiveness of the test process.

Here is an example of a DDP formula that would apply for calculating DDP for the last level of testing prior to release to the field:

$$DDP = \frac{\text{defects (testers)}}{\text{defects (testers)} + \text{defects (field)}}$$

Often, it aids the effectiveness and efficiency of reporting, tracking and managing defects when the defect-tracking tool provides an ability to vary some of the information captured depending on what the defect was reported against.

What is a test incident report?

The Test Incident Report documents all issues (bugs) found during different test phases. It's the detail of the actual versus expected results of a test, when a test has failed, and anything indicating why the test failed.

- It's uniquely numbered and tracked to resolution.
- Defect tracking tools are preferred.

ELEMENTS OF A TEST INCIDENT REPORT

The following are elements of a test incident report.

1. Test Incident Report Identifier. Some type of unique company generated number to identify this incident report, its level and the level of software that it is related to. The number may also identify what level of testing the incident occurred at. This is to assist in coordinating software and testware versions within configuration management and to assist in the elimination of incidents through process improvement.

2. Summary This is a summation/description of the actual incident. Provide enough details to enable others to understand how the incident was discovered and any relevant supporting information such as:
    - References to: ·
    - Test Procedure used to discover the incident ·
    - Test Case Specifications that will provide the information to repeat the incident ·
    - Test logs showing the actual execution of the test cases and procedures · Any other supporting materials, trace logs, memory dumps/maps etc.

3. Incident Description Provide as much details on the incident as possible. Especially if there are no other references to describe the incident. Include all relevant information that has not already been included in the incident summary information or any additional supporting information including: ·
    - Inputs ·
    - Expected Results ·
    - Actual Results ·
    - Anomalies ·
    - Date and Time ·
    - Procedure Step ·
    - Attempts to Repeat ·
    - Testers ·
    - Observers

4. Impact Describe the actual/potential damage caused by the incident. This can include either the Severity of the incident and the Priority to fix the incident or both. Severity and Priority need to be defined in the standards documents so as to ensure consistent use and interpretation, for example:

    Severity – The potential impact to the system
    - · Mission Critical - Application will not function or system fails
    - · Major - Severe problems but possible to work around
    - · Minor – Does not impact the functionality or usability of the process but is not according to requirements/design specifications

Priority – The order in which the incidents are to be addressed · Immediate – Must be fixed as soon as possible

- Delayed – System is usable but incident must be fixed prior to next level of test or shipment
- Deferred – Defect can be left in if necessary doe to time or costs

SESSION 3: REVIEW THE TESTING PROCESS FOR A COMPUTER PROGRAM

On completion of this section you will be able to Review the testing process for a computer program against organisation policy and procedures.

1. The review allows improvements to be made to the application testing process.
2. The review follows organisation policy and procedures.

3.1 REVIEWING PROGRAM TESTING PROCESS

A review is a process or meeting conducted to find the potential issues in the testing process of any program. Another significance of review is that all the team members get to know about the progress of the test and sometimes the diversity of thoughts may result in excellent suggestions. Documents are directly examined by people and discrepancies are sorted out.

Reasons for reviews

The following are some of the reasons for reviews;

- Early defect detection and correction
- Reduced development timescales
- Reduced cost and time
- For improvement of development productivity
- To get fewer defect at later stage of testing

Review Process

Types of Reviews vary from informal (no written instructions for the reviewer) to systematic (includes team participation, documented results of the review and documented procedures for conducting the review) and formality of a review process depends on factors such as maturity of the development process, any legal or regulatory requirements or need for an audit trail.

The way a review is carried out depends on the agreed objectives of the review like find defects, gain understanding, educate testers and new team members.

Activities of a Formal Review

1. Planning
    - Defining the review criteria
    - Selecting the personnel
    - Assigning roles
2. Defining the entry and exit criteria for more formal reviews (eg inspections)
   Selecting which part of the document to review.
3. Kick-off
   Distributing documents
   Explaining the objectives and process to the participants.
4. Checking the entry criteria
5. Individual preparation
   Preparing for the review meeting by reviewing the documents

6. Noting down the questions and comments
7. Examination/Evaluation/Recording of results (review meeting)
   Discussing or logging the documented results or minutes
   Noting, making recommendation and decision regarding handling defects
8. Rework
9. Fixing defects found (by author)
   Recording updated status of defects
10. Follow up
    Checking that defects have been addressed
    Gathering metrics
11. Checking on exit criteria
    Checking that defects have been addressed
    Gathering metrics

Elements for a review

The review must focus on the following;

- Test deliverables,
- Review of test incident report,
- Pass/Fail criteria,
- Contingencies,
- QA Approval.