T.Y.B.Sc. Computer Science

Semester V - Paper 2 – Software Testing

***Textbook**(s):*

1. Software Engineering for Students, A Programming Approach, Douglas Bell, 4th Edition,, Pearson Education, 2005
2. Software Engineering – A Practitioners Approach, Roger S. Pressman, 5th Edition, Tata McGraw Hill, 2001
3. Quality Management, Donna C. S. Summers, 5th Edition, Prentice-Hall, 2010.
4. Total Quality Management, Dale H. Besterfield, 3rd Edition, Prentice Hall, 2003.

Unit 1 Notes:

Verification is the general term for techniques that aim to produce fault-free software.
Testing is a widely used technique for verification. Software is complex and it is difficult to make it work correctly. Currently the dominant technique used for verification is testing.

Software testing is a process of executing a program or application with the intent of finding the **software bugs**.

- It can also be stated as the **process of validating and verifying** that a software program or application or product:
  - Meets the business and technical requirements that guided it's design and development
  - Works as expected
  - Can be implemented with the same characteristic.

Let's break the definition of **Software testing** into the following parts:

1) Process: Testing is a process rather than a single activity.

2) All Life Cycle Activities: Testing is a process that's take place throughout the **Software Development Life Cycle (SDLC)**.

- The process of designing tests early in the life cycle can help to prevent defects from being introduced in the code. Sometimes it's referred as **"verifying the test basis via the test design"**.
- The **test basis** includes documents such as the requirements and design specifications.
- **3) Static Testing:** It can test and find defects without executing code. Static Testing is done during verification process. This testing includes reviewing of the documents (including source code) and static analysis. This is useful and cost effective way of testing. For example: reviewing, **walkthrough**, **inspection**, etc.
- **4) Dynamic Testing:** In dynamic testing the software code is executed to demonstrate the result of running tests. It's done during validation process. For example: **unit testing**, **integration testing**, **system testing**, etc.
- **5) Planning:** We need to plan as what we want to do. We control the test activities, we report on testing progress and the status of the software under test.
- **6) Preparation:** We need to choose what testing we will do, by selecting test conditions and **designing test cases**.
- **7) Evaluation:** During evaluation we must check the results and evaluate the software under test and the completion criteria, which helps us to decide whether we have finished testing and whether the software product has passed the tests.
- **8) Software products and related work products:** Along with the testing of code the testing of requirement and design specifications and also the related documents like operation, user and training material is equally important.

Nature of Errors :

It would be convenient to know how errors arise, because then we could try to avoid
them during all the stages of development. Similarly, it would be useful to know the
most commonly occurring faults, because then we could look for them during verification.

1. Specifications are a common source of faults. A software system has an overall

specification, derived from requirements analysis. In addition, each component of

the software ideally has an individual specification that is derived from architectural
design. The specification for a component can be:
_ ambiguous (unclear)
_ incomplete
_ faulty.
Any such problems should, of course, be detected and remedied by verification of
the specification prior to development of the component, but, of course, this verification
cannot and will not be totally effective. So there are often problems with a component specification.

2. During programming,

the developer of a component may misunderstand the component specification.
The next type of error is where a component contain faults so that it does not meet
its specification. This may be due to two kinds of problem:
1. errors in the logic of the code – an error of commission
2. code that fails to meet all aspects of the specification – an error of omission.
This second type of error is where the programmer has failed to appreciate and correctly
understand all the detail of the specification and has therefore omitted some necessary code.

3. Finally, the kinds of errors that can arise in the coding of a component are:

_ data not initialized
_ loops repeated an incorrect number of times.
_ boundary value errors.
Boundary values are values of the data at or near critical values. For example, suppose
a component has to decide whether a person can vote or not, depending on their age.
The voting age is 18. Then boundary values, near the critical value, are 17, 18 and 19.

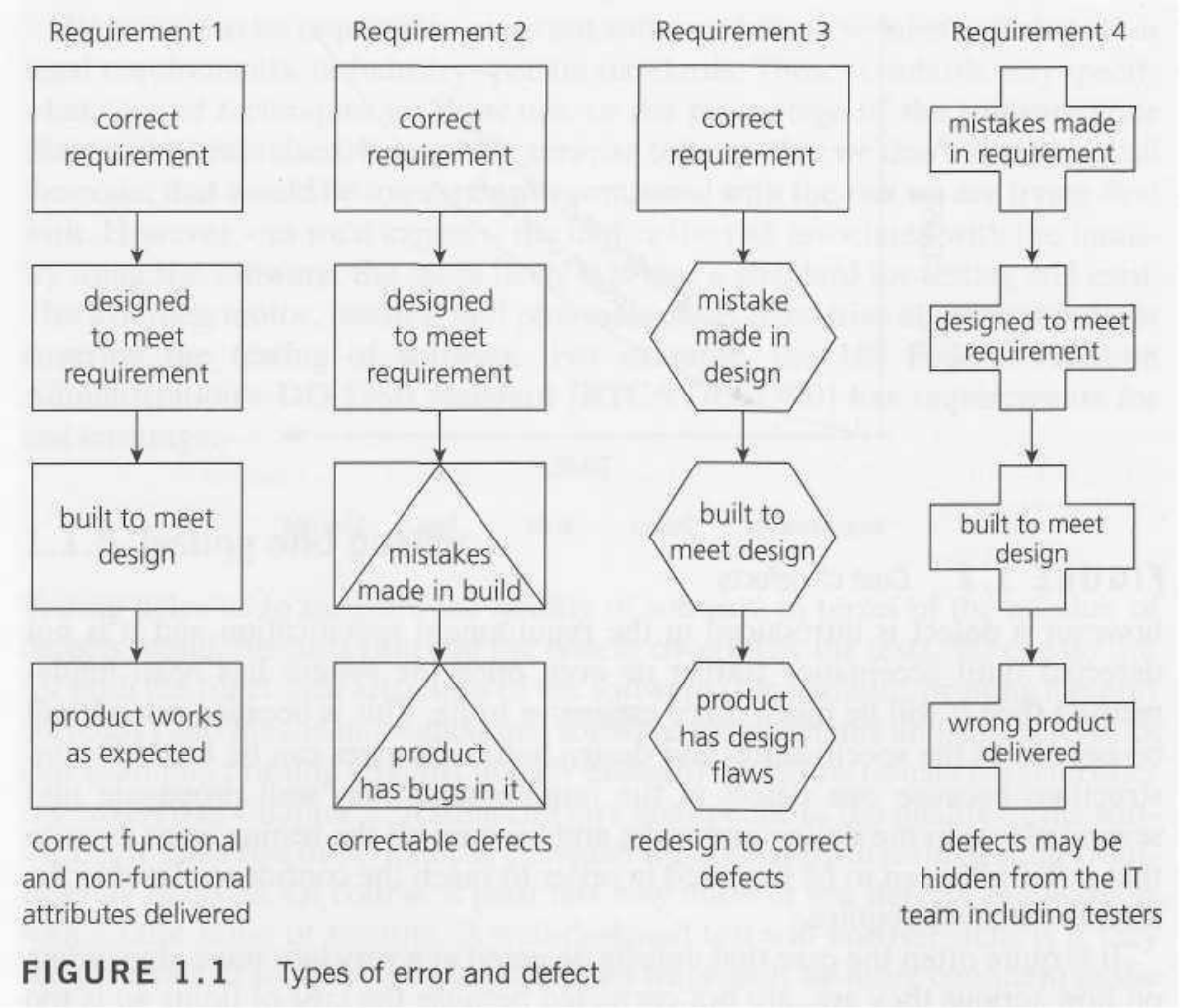


FIGURE 1.1 Types of error and defect

In Figure 1.1 we can see how defects may arise in four requirements for a product.

We can see that requirement 1 is implemented correctly - we understood the customer's requirement, designed correctly to meet that requirement, built correctly to meet the design, and so deliver that requirement with the right attributes: functionally, it does what it is supposed to do and it also has the right non-functional attributes, so it is fast enough, easy to understand and so on.

With the other requirements, errors have been made at different stages. Requirement 2 is fine until the software is coded, when we make some mistakes and introduce defects. Probably, these are easily spotted and corrected during testing, because we can see the product does not meet its design specification. The defects introduced in requirement 3 are harder to deal with; we built exactly what we were told to but unfortunately the designer made some mis-takes so there are defects in the design. Unless we check against the require-ments definition, we will not spot those defects during testing. When we do notice them they will be hard to fix because design changes will be required.

The defects in requirement 4 were introduced during the definition of the requirements; the product has been designed and built to meet that flawed

requirements definition. If we test the product meets its requirements and design, it will pass its tests but may be rejected by the user or customer. Defects reported by the customer in acceptance test or live use can be very costly.

Some important Terms related to Software Testing

- **Quality management** ensures that an organization, product or service is consistent. It has four main components:
    - **Quality planning**
    - **Quality assurance**
    - **Quality control**
    - **Quality improvement**
- Quality management is focused not only on product and service quality, but also on the means to achieve it. Quality management, therefore, uses quality assurance and control of processes as well as products to achieve more consistent quality.

- **Quality assurance** (**QA**) is a way of preventing mistakes and defects in manufactured products and avoiding problems when delivering solutions or services to customers;
- **Quality control**, or **QC** for short, is a process by which entities review the quality of all factors involved in production. This approach places an emphasis on three aspects
    - **Elements such as controls, job management, defined and well managed processes, performance and identification of records**
    - **Competence, such as knowledge, skills, experience, and qualifications**
    - **Soft elements, such as integrity, confidence, culture, motivation, team spirit, and quality relationships.**
- Software Quality Assurance (SQA)
    - This consists of a means of monitoring the software engineering processes and methods used to ensure quality. The methods by which this is accomplished are many and varied, and may include ensuring conformance to one or more standards
    - SQA encompasses the entire software development process, which includes processes such as requirements definition, software design, coding, source code control, code reviews, software configuration management, testing, release management, and product integration.

- SQA is organized into goals, commitments, abilities, activities, measurements, and verifications.

Role of testing in software development, maintenance and operations

- Human errors can cause a defect or fault to be introduced at any stage within the software development life cycle and, depending upon the consequences of the mistake, the results can be trivial or catastrophic.
- Rigorous testing is necessary during development and maintenance to identify defects, in order to reduce failures in the operational environment and increase the quality of the operational system. This includes looking for places in the user interface where a user might make a mistake in input of data or in the interpretation of the output, and looking for potential weak points for intentional and malicious attack.
- Executing tests helps us move towards improved quality of product and service, but that is just one of the verification and validation methods applied to products. Processes are also checked, for example by audit.
- A variety of methods may be used to check work, some of which are done by the author of the work and some by others to get an independent view. We may also be required to carry out software testing to meet contractual or legal requirements, or industry-specific standards.
- These standards may specify what type of techniques we must use, or the percentage of the software code that must be exercised

Software Quality

A software quality factor is a

- Non-functional requirement, but is a desirable requirement which enhances the quality of the software program.
- None of these factors are binary.

Rather, they are characteristics that one seeks to maximize in one's software to optimize its quality.

Understandability	Clarity of purpose
Completeness	Presence of all constituent parts, with each part fully developed.
Conciseness	Minimization of excessive or redundant information or processing.

Portability	Ability to be run well and easily on multiple computer configurations.
Consistency	Uniformity in notation, symbology, appearance, and terminology within itself.
Maintainability	Propensity to facilitate updates to satisfy new requirements.
Testability	Disposition to support acceptance criteria and evaluation of performance.
Usability	Convenience and practicality of use.
Reliability	Ability to be expected to perform its intended functions satisfactorily.
Efficiency	Fulfillment of purpose without waste of resources
Security	Ability to protect data against unauthorized access

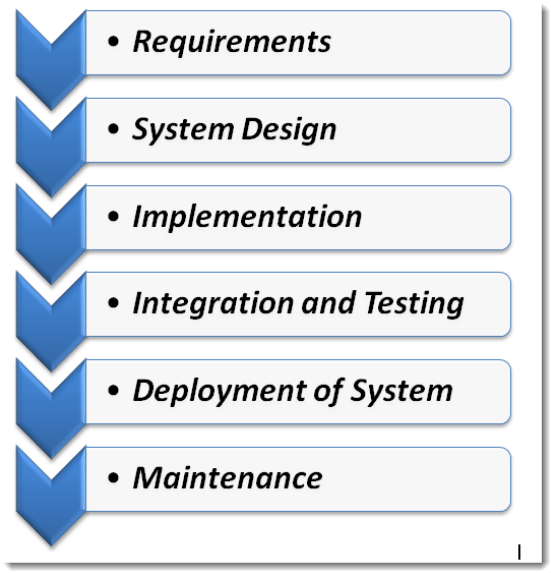
Software Development Models :

Waterfall Model :

The **Waterfall Model** was first Process Model to be introduced. It is very simple to understand and use. In a Waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases. Waterfall model is the earliest **SDLC** approach that was used for software development.

In "**The Waterfall**" approach, the whole process of software development is divided into separate phases. The outcome of one phase acts as the input for the next phase sequentially. This means that any phase in the development process begins only if the previous phase is complete. The waterfall model is a sequential design process in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of **Conception, Initiation, Analysis, Design, Construction, Testing, Production/Implementation and Maintenance.**

As the **Waterfall Model** illustrates the software development process in a linear sequential flow; hence it is also referred to as a **Linear-Sequential Life Cycle Model**.



Sequential Phases in Waterfall Model

- **Requirements:** The first phase involves understanding what need to be design and what is its function, purpose etc. Here, the specifications of the input and output or the final product are studied and marked.
- **System Design:** The requirement specifications from first phase are studied in this phase and system design is prepared. System Design helps in specifying hardware and system requirements and also helps in defining overall system architecture. The software code to be written in the next stage is created now.
- **Implementation:** With inputs from system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality which is referred to as Unit Testing.
- **Integration and Testing:** All the units developed in the implementation phase are integrated into a system after testing of each unit. The software designed, needs to go through constant software testing to find out if there are any flaw or errors. Testing is done so that the client does not face any problem during the installation of the software.
- **Deployment of System:** Once the functional and non-functio
  nal testing is done, the product is deployed in the customer environment or released into the market.
- **Maintenance:** This step occurs after installation, and involves making modifications to the system or an individual component to alter attributes or improve performance.These modifications arise either due to change requests initiated by the customer, or defects uncovered during live use of the system. Client is provided with regular maintenance and support for the developed software.

All these phases are cascaded to each other in which progress is seen as flowing steadily downwards (like a waterfall) through the phases. The next phase is started only after the defined set of goals are achieved for previous phase and it is signed off, so the name “Waterfall Model“.

Advantages of Waterfall Model

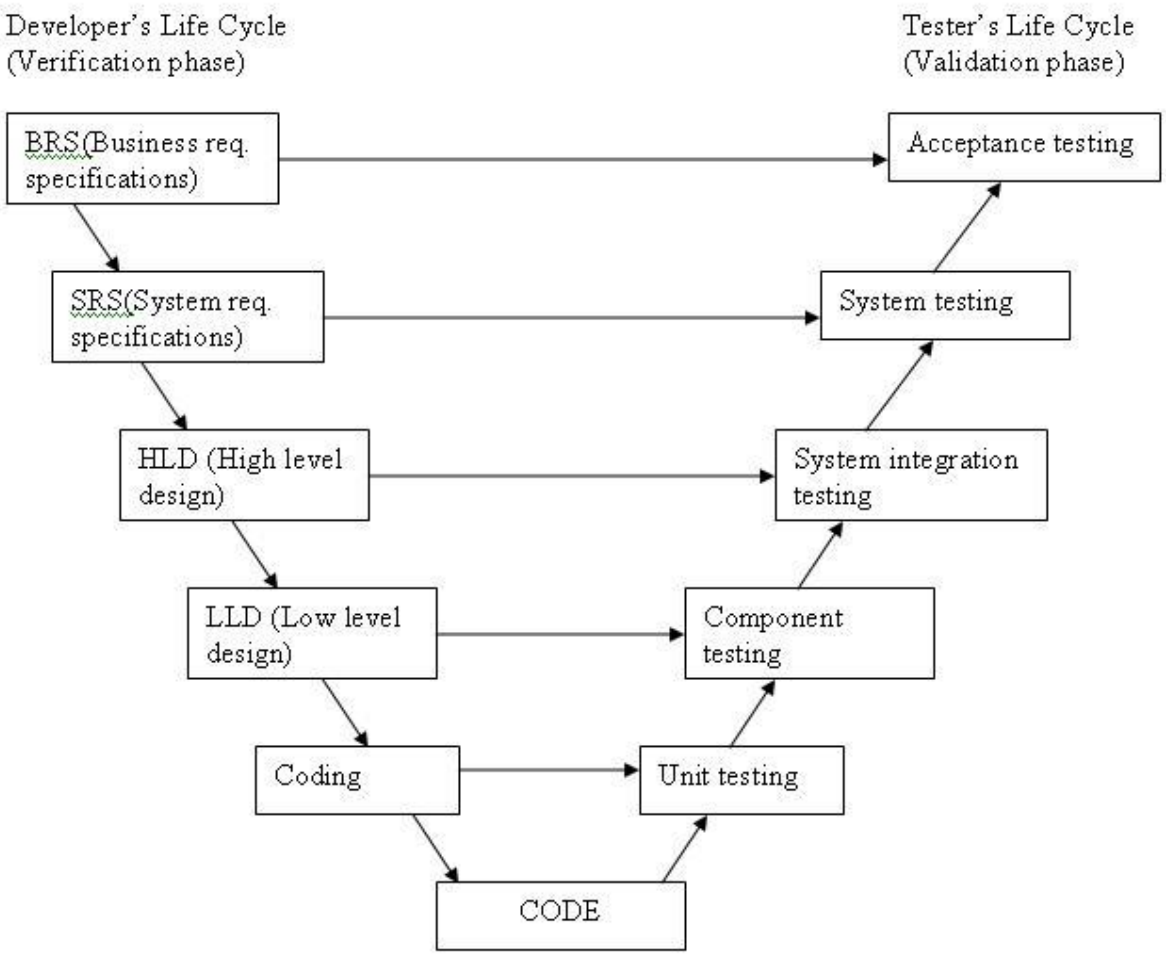
- The advantage of waterfall development is that it allows for departmentalization and control. A schedule can be set with deadlines for each stage of development and a product can proceed through the development process model phases one by one.
- The waterfall model progresses through easily understandable and explainable phases and thus it is easy to use.
- It is easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- In this model, phases are processed and completed one at a time and they do not overlap. Waterfall model works well for smaller projects where requirements are very well understood.

Disadvantages of Waterfall Model

- It is difficult to estimate time and cost for each phase of the development process.
- Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage.
- Not a good model for complex and object-oriented projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing.

V-Model :

The **V-Model** is **SDLC** model where execution of processes happens in a sequential manner in V-shape. It is also known as **Verification and Validation Model**. V-Model is an extension of the **Waterfall Model** and is based on association of a testing phase for each corresponding development stage. This means that for every single phase in the development cycle there is a directly associated testing phase. This is a highly disciplined model and next phase starts only after completion of the previous phase.



Phases of V-model

Business Requirement Analysis : In this first phase, the product requirements are understood from the customer perspective. This phase involves detailed communication with the customer to understand their expectations and exact requirement. The users are interviewed and a document called the **User Requirements Document** is generated. The user requirements document will typically describe the system's functional, interface, performance, data, security and other requirements as expected by the user. The user's carefully review this document as this document would serve as the guideline for the system designers in the system design phase.

System Design: In the phase, system developers analyze and understand the business of the proposed system by studying the user requirements document. They figure out possibilities and techniques by which the user requirements can be implemented. System design comprises of understanding and detailing the

complete hardware and communication setup for the product under development. System test plan is developed based on the system design.

Architecture Design: This is also referred to as **High Level Design (HLD)**. This phase focuses on system architecture and design. It provides overview of solution, platform, system, product and service/process. Usually more than one technical approach is proposed and based on the technical and financial feasibility the final decision is taken. An integration test plan is created in order to test the pieces of the software systems ability to work together.

Module Design: The module design phase can also be referred to as low-level design. In this phase the actual software components are designed. It defines the actual logic for each and every component of the system. The designed system is broken up into smaller units or modules and each of them is explained so that the programmer can start coding directly. It is important that the design is compatible with the other modules in the system architecture and the other external systems.

Validation Phases

In the V-Model, each stage of verification phase has a corresponding stage in the validation phase. The following are the typical phases of validation in the V-Model.

Unit Testing: Unit tests designed in the module design phase are executed on the code during this validation phase. Unit testing is the testing at code level and helps eliminate bugs at an early stage. A unit is the smallest entity which can independently exist, e.g. a program module. Unit testing verifies that the smallest entity can function correctly when isolated from the rest of the codes/units.

Component testing: searches for defects in and verifies the functioning of software components (e.g. modules, programs, objects, classes etc.) that are separately testable;

The **Component Testing** is like **Unit Testing** with the difference that all Stubs and Simulators are replaced with the real objects.

Integration Testing: Integration testing is associated with the architectural design phase. These tests verify that units created and tested independently can coexist and communicate among themselves within the system. integration testing: tests interfaces between components, interactions to dif

ferent parts of a system such as an operating system, file system and hard ware or interfaces between systems;

System Testing: System Tests Plans are developed during System Design Phase. System Test Plans are composed by client's business team. System Test ensures that expectations from application developed are met. The whole application is tested for its functionality, interdependency and communication. User acceptance testing: Acceptance testing is associated with the business requirement analysis phase and involves testing the product in user environment. UAT verifies that delivered system meets user's requirement and system is ready for use in real time.

Advantages and Disadvantages of V-Model

Advantages

- This is a highly disciplined model and Phases are completed one at a time.
- Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
- Simple and easy to understand and use.
- Testing activities like planning, test designing happens well before coding. This saves a lot of time. Hence higher chance of success over the waterfall model.
- The implementation of testing starts right from the requirement phase, defects are found at early stage.
- Works well for small projects where requirements are easily understood.

Disadvantages

- Very rigid and least flexible so adjusting scope is difficult and expensive.
- Software is developed during the implementation phase, so no early prototypes of the software are produced.
- Not suitable for the projects where requirements are at a moderate to high risk of changing.

Verification and Validation

Verification is The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Verification is a static practice of verifying documents, design, code and program. It includes all the activities associated with producing high quality software: inspection, design analysis and specification analysis. It is a relatively objective process.

Verification will help to determine whether the software is of high quality, but it will not ensure that the system is useful. Verification is concerned with whether the system is well-engineered and error-free.

Methods of Verification : <u>Static Testing</u>

- Walkthrough
- Inspection
- Review

Validation is The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements. Validation is the process of evaluating the final product to check whether the software meets the customer expectations and requirements. It is a dynamic mechanism of validating and testing the actual product.

Methods of Validation : <u>Dynamic Testing</u>

- Testing
- End Users

Verification	**Validation**
1. **Verification** is a static practice of verifying documents, design, code and program.	1. **Validation** is a dynamic mechanism of validating and testing the actual product.

2. It does not involve executing the code.	2. It always involves executing the code.
3. It is human based checking of documents and files.	3. It is computer based execution of program.
4. **Verification** uses methods like inspections, reviews, walkthroughs, and Desk-checking etc.	4. **Validation** uses methods like black box (functional) testing and white box (structural) testing etc.
5. **Verification** is to check whether the software conforms to specifications.	5. **Validation** is to check whether software meets the customer expectations and requirements.
6. It can catch errors that validation cannot catch. It is low level exercise.	6. It can catch errors that verification cannot catch. It is High Level Exercise.
7. Target is requirements specification, application and software architecture, high level, complete design, and database design etc.	7. Target is actual product-a unit, a module, a bent of integrated modules, and effective final product.
8. **Verification** is done by QA team to ensure that the software is as per the specifications in the SRS document.	8. **Validation** is carried out with the involvement of testing team.
9. It generally comes first-done before validation.	9. It generally follows after **verification**.

Static Testing Techniques provide a powerful way to improve the quality and productivity of software development by assisting engineers to recognize and fix their own defects early in the software development process. In this software is tested without executing the code by doing **Review, Walk Through, Inspection or Analysis** etc.
Static Testing may be conducted manually or through the use of various software testing tools. It starts early in the **Software Development Life Cycle** and so it is done during the **Verification Process**.

Review :

Informal Review :

Informal reviews are applied at various times during the early stages in the life cycle of a document. A two-person team can conduct an informal review, as the author can ask a col-league to review a document or code.

Phases of a formal review

In contrast to informal reviews, formal reviews follow a formal process. A typical formal review process consists of six main steps:

1 Planning - The review process for a particular review begins with a 'request for review' by the author to the **moderator** (or inspection leader). A moderator is often assigned to take care of the scheduling (dates, time, place and invitation) of the review.

2 Kick-off - An optional step in a review procedure is a kick-off meeting. The goal of this meeting is to get everybody on the same wavelength regarding the document under review and to commit to the time that will be spent on checking.

3 Preparation - The participants work individually on the document under review using the related documents, procedures, rules and checklists provided. The individual participants identify defects, questions and comments, according to their understanding of the document and role. All issues are recorded, preferably using a logging form.

4 Review meeting - The meeting typically consists of the following elements (partly depending on the review type): logging phase, discussion phase and decision phase. During the logging phase the issues, e.g. defects, that have been identified during the preparation are mentioned page by page, reviewer by reviewer and are logged either by the author or by a scribe. For a more formal review, the issues classified as discussion items will be handled during this meeting phase. Informal reviews will often not have a sep-arate logging phase and will start immediately with discussion. At the end of the meeting, a decision on the document under review has to be made by the participants. The most important exit criterion is the average number of critical and/or major defects found per page (e.g. no more than three critical/major defects per page).

5 Rework - Based on the defects detected, the author will improve the document under review step by step.

6 Follow-up - The moderator is responsible for ensuring that satisfactory actions have been taken on all (logged) defects, process improvement suggestions and change requests.

Walkthrough

A walkthrough is characterized by the author of the document under review guiding the participants through the document and his or her thought processes, to achieve a common understanding and to gather feedback. This is especially useful if people from outside the software discipline are present, who are not used to, or cannot easily understand software development documents. The content of the document is explained step by step by the author, to reach consensus on changes or to gather information.

Within a walkthrough the author does most of the preparation. The participants, who are selected from different departments and backgrounds, are not required to do a detailed study of the documents in advance. Because of the way the meeting is structured, a large number of people can participate and this larger audience can bring a great number of diverse viewpoints regarding the contents of the document being reviewed as well as serving an educational purpose. If the audience represents a broad cross-section of skills and disciplines, it can give assurance that no major defects are 'missed' in the walkthrough. A walkthrough is especially useful for higher-level documents, such as requirement specifications and architectural documents.

The specific goals of a walkthrough depend on its role in the creation of the document. In general the following goals can be applicable:

- to present the document to stakeholders both within and outside the soft ware discipline, in order to gather information regarding the topic under documentation;
- to explain (knowledge transfer) and evaluate the contents of the docu ment;

- to establish a common understanding of the document;
- to examine and discuss the validity of proposed solutions and the viability of alternatives, establishing consensus.

Key characteristics of walkthroughs are:

- The meeting is led by the authors; often a separate scribe is present.
- Scenarios and dry runs may be used to validate the content.
- Separate pre-meeting preparation for reviewers is optional.

Technical review

A **technical review** is a discussion meeting that focuses on achieving consensus about the technical content of a document. Compared to inspections, technical reviews are less formal. During technical reviews defects are found by experts, who focus on the content of the document. The experts that are needed for a technical review are, for example, architects, chief designers and key users. In practice, technical reviews vary from quite informal to very formal.

The goals of a technical review are to:

- assess the value of technical concepts and alternatives in the product and project environment;
- establish consistency in the use and representation of technical concepts;
- ensure, at an early stage, that technical concepts are used correctly;
- inform participants of the technical content of the document.

Key characteristics of a technical review are:

- It is a documented defect-detection process that involves peers and technical experts.
- It is often performed as a peer review without management participation.
- Ideally it is led by a trained moderator, but possibly also by a technical expert.
- A separate preparation is carried out during which the product is examined and the defects are found.
- More formal characteristics such as the use of checklists and a logging list or issue log are optional.

Inspection

Inspection is the most formal review type. The document under inspection is prepared and checked thoroughly by the reviewers before the meeting, comparing the work product with its sources and other referenced documents, and using rules and checklists. In the inspection meeting the defects found are logged and any discussion is postponed until the discussion phase. This makes the inspection meeting a very efficient meeting.

Depending on the organization and the objectives of a project, inspections can be balanced to serve a number of goals. For example, if the time to market is extremely important, the emphasis in inspections will be on efficiency. In a safety-critical market, the focus will be on effectiveness.

The generally accepted goals of inspection are to:

- help the author to improve the quality of the document under inspection;
- remove defects efficiently, as early as possible;
- improve product quality, by producing documents with a higher level of quality;
- create a common understanding by exchanging information among the inspection participants;
- train new employees in the organization's development process;
- learn from defects found and improve processes in order to prevent recur rence of similar defects;
- sample a few pages or sections from a larger document in order to measure the typical quality of the document, leading to improved work by individuals in the future, and to process improvements.

Key characteristics of an inspection are:

- It is usually led by a trained moderator (certainly not by the author).
- It uses defined roles during the process.
- It involves peers to examine the product.
- Rules and checklists are used during the preparation phase.
- A separate preparation is carried out during which the product is examined and the defects are found.
- The defects found are documented in a logging list or issue log.
- A formal follow-up is carried out by the moderator applying exit criteria.
- Optionally, a causal analysis step is introduced to address process improve ment issues and learn from the defects found.
- Metrics are gathered and analyzed to optimize the process.

Test Case Design Techniques :

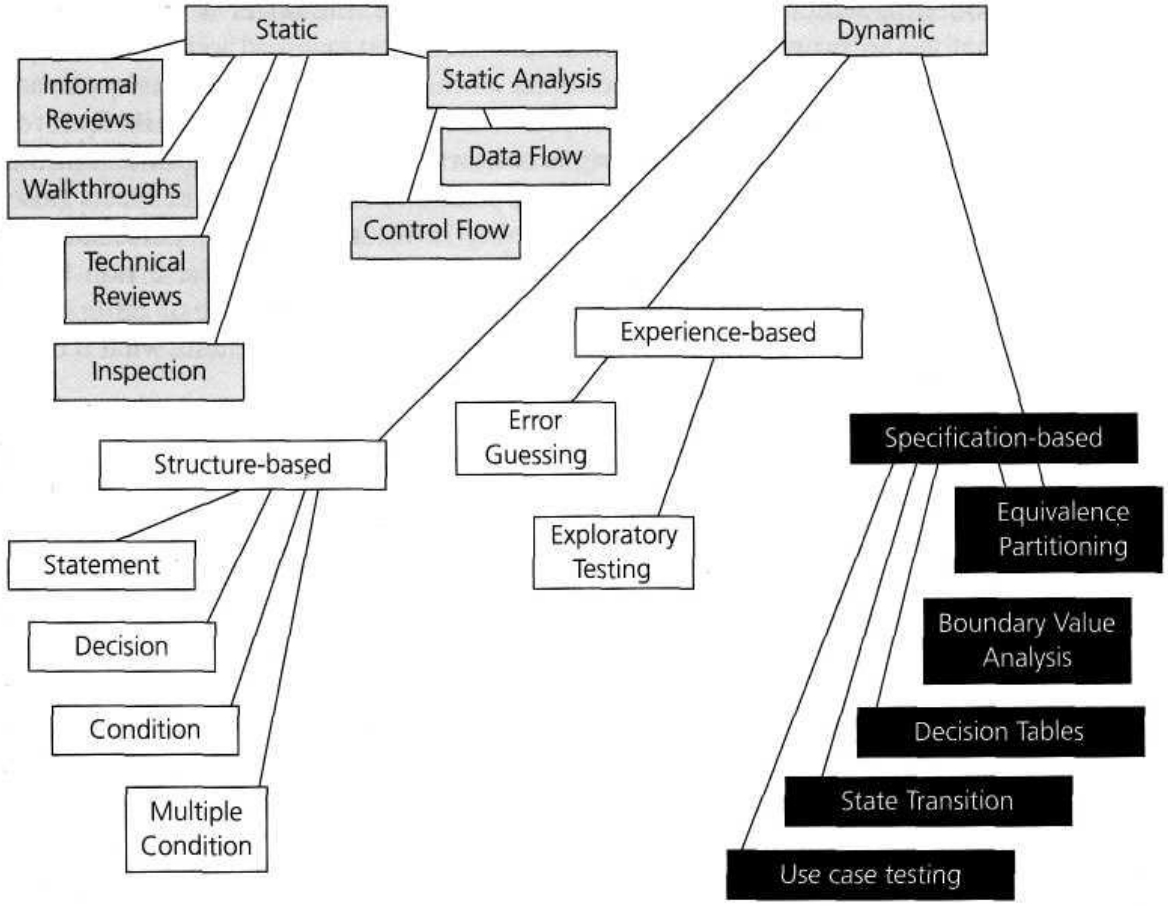


FIGURE 4.1 Testing techniques

There are many different types of software testing technique, each with its own strengths and weaknesses. Each individual technique is good at finding particular types of defect and relatively poor at finding other types. For example, a technique that explores the upper and lower limits of a single input range is more likely to find boundary value defects than defects associated with combinations of inputs. Similarly, testing performed at different stages in the software development life cycle will find different types of defects; component testing is more likely to find coding logic defects than system design defects.

Each testing technique falls into one of a number of different categories. Broadly speaking there are two main categories, static and dynamic. Static tech-niques were discussed in Chapter 3. Dynamic techniques are subdivided into three more categories: specification-based (black-box, also known as behavioural techniques), structure-based (white-box or structural techniques) and experience-based. Specification-based techniques include both functional and non-functional techniques (i.e. quality characteristics).

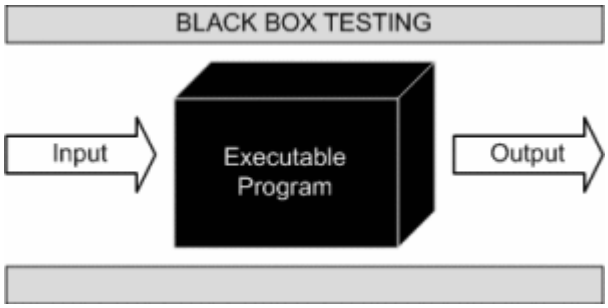
Specification-based (black-box) testing techniques

The first of the dynamic testing techniques we will look at are the specification-based testing techniques. These are also known as **'black-box'** or input/output-driven testing techniques because they view the software as a black-box with inputs and outputs, but they have no knowledge of how the system or testing, but the structures are different. For example, the coverage of menu options or major business transactions could be the structural element in system or acceptance testing.

Experience-based techniques are used to complement specification-based and structure-based techniques, and are also used when there is no specification, or if the specification is inadequate or out of date. This may be the only type of technique used for low-risk systems, but this approach may be particularly useful under extreme time pressure - in fact this is one of the factors leading to exploratory testing.

SPECIFICATION-BASED OR BLACK-BOX TECHNIQUES

BLACK BOX TESTING, also known as Behavioral Testing, is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester. These tests can be functional or non-functional, though usually functional.



This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see. This method attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavior or performance errors
- Initialization and termination errors

Example

A tester, without knowledge of the internal structures of a website, tests the web pages by using a browser; providing inputs (clicks, keystrokes) and verifying the outputs against the expected outcome.

Levels Applicable To

Black Box Testing method is applicable to the following levels of software testing:

- Integration Testing
- System Testing
- Acceptance Testing

The higher the level, and hence the bigger and more complex the box, the more black-box testing method comes into use.

Techniques

Following are some techniques that can be used for designing black box tests.

- *Equivalence Partitioning:* It is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data.
- *Boundary Value Analysis:* It is a software test design technique that involves the determination of boundaries for input values and selecting values that are at the boundaries and just inside/ outside of the boundaries as test data.
- *Cause-Effect Graphing:* It is a software test design technique that involves identifying the cases (input conditions) and effects (output conditions), producing a Cause-Effect Graph, and generating test cases accordingly.

Advantages

- Tests are done from a user's point of view and will help in exposing discrepancies in the specifications.
- Tester need not know programming languages or how the software has been implemented.
- Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias.
- Test cases can be designed as soon as the specifications are complete.

Disadvantages

- Only a small number of possible inputs can be tested and many program paths will be left untested.
- Without clear specifications, which is the situation in many projects, test cases will be difficult to design.
- Tests can be redundant if the software designer/developer has already run a test case.
- Ever wondered why a soothsayer closes the eyes when foretelling events? So is almost the case in Black Box Testing.

Equivalence partitioning

Equivalence partitioning (EP) is a good all-round specification-based black-box technique. It can be applied at any level of testing and is often a good technique to use first. The idea behind the technique is to divide (i.e. to partition) a set of test conditions into groups or sets that can be considered the same (i.e. the system should handle them equivalently), hence 'equivalence partitioning'. **Equivalence partitions** are also known as equivalence classes - the two terms mean exactly the same thing.

The equivalence-partitioning technique then requires that we need test only one condition from each partition. This is because we are assuming that all the conditions in one partition will be treated in the same way by the software. If one condition in a partition works, we assume all of the conditions in that partition will work, and so there is little point in testing any of these others. Conversely, if one of the conditions in a partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition. Of course these are simplifying assumptions that may not always be right but if we write them down, at least it gives other people the chance to challenge the assumptions we have made and hopefully help to identify better partitions. If you have time, you may want to try more than one value from a partition, especially if you want to confirm a selection of typical user inputs.

Example of Equivalence Partitioning Technique
Test cases for input box accepting numbers between 1 and 1000 using Equivalence Partitioning:

1. *One input data class with all valid inputs. Pick a single value from range 1 to 1000 as a valid test case. If you select other values between 1 and 1000 then result is going to be same. So one test case for valid input data should be sufficient.*
2. *Input data class with all values below lower limit. I.e. any value below 1, as a invalid input data test case.*
3. *Input data with any value greater than 1000 to represent third invalid input class.*

So using *equivalence partitioning* you have categorized all possible test cases into three classes and that can be (-10, 100 & 1010). Test cases with other values from any class should give you the same result.

Example on Equivalence Partitioning Test Case Design Technique:

Example 1:

Assume, we have to test a field which accepts Age 18 – 56

EQUIVALENCE PARTITIONING		
Invalid	**Valid**	**Invalid**
<=17	18-56	>=57

Valid Input: 18 – 56

Invalid Input:less than or equal to 17 (<=17), greater than or equal to 57 (>=57)

Valid Class: 18 – 56 = Pick any one input test data from 18 – 56

Invalid Class 1: <=17 = Pick any one input test data less than or equal to 17

Invalid Class 2: >=57 = Pick any one input test data greater than or equal to 57

We have one valid and two invalid conditions here.

Example 2:

Assume, we have to test a filed which accepts a Mobile Number of ten digits.

EQUIVALENCE PARTITIONING		
Invalid	**Valid**	**Invalid**
987654321	9876543210	98765432109

Valid input: 10 digits

Invalid Input: 9 digits, 11 digits

Valid Class: Enter 10 digit mobile number = 9876543210

Invalid Class Enter mobile number which has less than 10 digits = 987654321

Invalid Class Enter mobile number which has more than 11 digits = 98765432109

Example 2:

Grocery Store Example

Consider a software module that is intended to accept the name of a grocery item and a list of the different sizes the item comes in, specified in ounces. The specifications state that the item name is to be alphabetic characters 2 to 15 characters in length. Each size may be a value in the range of 1 to 48, whole numbers only. The sizes are to be entered in ascending order (smaller sizes first). A maximum of five sizes may be entered for each item. The item name is to be entered first, followed by a comma, then followed by a list of sizes. A comma will be used to separate each size. Spaces (blanks) are to be ignored anywhere in the input.

Derived Equivalence Classes

1. Item name is alphabetic (valid)
2. Item name is not alphabetic (invalid)
3. Item name is less than 2 characters in length (invalid)

4. Item name is 2 to 15 characters in length (valid)
5. Item name is greater than 15 characters in length (invalid)
6. Size value is less than 1 (invalid)
7. Size value is in the range 1 to 48 (valid)
8. Size value is greater than 48 (invalid)
9. Size value is a whole number (valid)
10. Size value is a decimal (invalid)
11. Size value is numeric (valid)
12. Size value includes nonnumeric characters (invalid)
13. Size values entered in ascending order (valid)
14. Size values entered in nonascending order (invalid)
15. No size values entered (invalid)
16. One to five size values entered (valid)
17. More than five sizes entered (invalid)
18. Item name is first (valid)
19. Item name is not first (invalid)
20. A single comma separates each entry in list (valid)
21. A comma does not separate two or more entries in the list (invalid)
22. The entry contains no blanks (???)
23. The entry contains blanks (????)

Black Box Test Cases for the Grocery Item Example based on the Equivalence Classes Above.

#	Test Data	Expected Outcome	Classes Covered
1	xy,1	T	1,4,7,9,11,13,16,18,20,22
2	AbcDefghijklmno,1,2,3 ,4,48	T	1,4,7,9,11,13,16,18,20,23
3	a2x,1	F	2
4	A,1	F	3
5	abcdefghijklmnop	F	5
6	Xy,0	F	6
7	XY,49	F	8
8	Xy,2.5	F	10
9	xy,2,1,3,4,5	F	14
10	Xy	F	15
11	XY,1,2,3,4,5,6	F	17

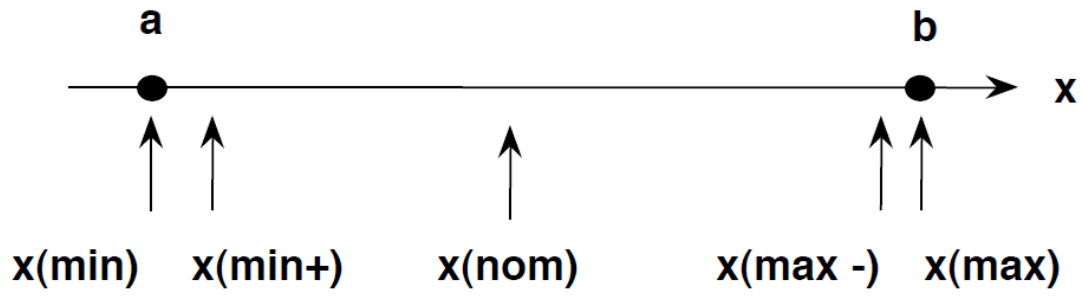
12	1,Xy,2,3,4,5	F	19
13	XY2,3,4,5,6	F	21
14	AB,2#7	F	12

Equivalence partitioning can be applied to different types of input as well. Our examples have concentrated on inputs that would be typed in by a (human) user when using the system. However, systems receive input data from other sources as well, such as from other systems via some interface - this is also a good place to look for partitions (and boundaries). For example, the value of an interface parameter may fall into valid and invalid equivalence partitions. This type of defect is often difficult to find in testing once the interfaces have been joined together, so is particularly useful to apply in integration testing (either component integration or system integration).

Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values.

- So these extreme ends like Start- End, Lower- Upper, Maximum-Minimum, Just Inside-Just Outside values are called boundary values and the testing is called "boundary testing".
- The basic idea in boundary value testing is to select input variable values at their:

1. Minimum
2. Just above the minimum
3. A nominal value
4. Just below the maximum
5. Maximum

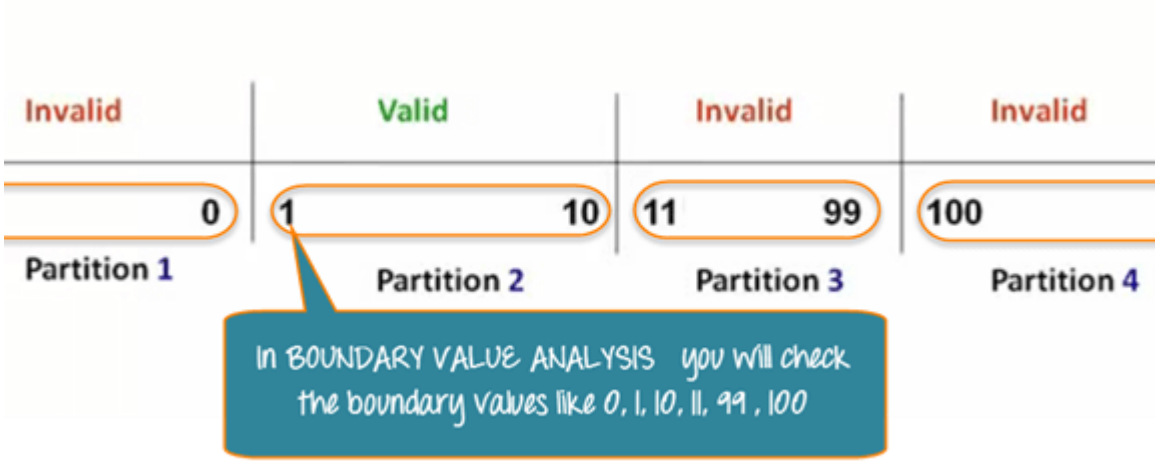


Example 1: Equivalence and Boundary Value

- Let's consider the behavior of tickets in the Flight reservation application, while booking a new flight.
- Ticket values 1 to 10 are considered valid & ticket is booked. While value 11 to 99 are considered invalid for reservation and error message will appear, **"Only ten tickets may be ordered at one time."**

Here is the test condition

1. Any Number greater than 10 entered in the reservation column (let say 11) is considered invalid.
2. Any Number less than 1 that is 0 or below, then it is considered invalid.
3. Numbers 1 to 10 are considered valid
4. Any 3 Digit Number say -100 is invalid.



As you may observe, you test values at **both valid and invalid boundaries**. Boundary Value Analysis is also called **range checking**.

Example 2: Equivalence and Boundary Value

Suppose a password field accepts minimum 6 characters and maximum 10 characters

That means results for values in partitions 0-5, 6-10, 11-14 should be equivalent

Test #	Test Scenario Description	Expected Outcome

1	Enter 0 to 5 characters in password field	System should not accept
2	Enter 6 to 10 characters in password field	System should accept
3	Enter 11 to 14 character in password field	System should not accept

Examples 3: Input Box should accept the Number 1 to 10

Here we will see the Boundary Value Test Cases

Test Scenario Description	**Expected Outcome**
Boundary Value = 0	System should NOT accept
Boundary Value = 1	System should accept
Boundary Value = 2	System should accept
Boundary Value = 9	System should accept
Boundary Value = 10	System should accept
Boundary Value = 11	System should NOT accept

Why Equivalence & Boundary Analysis Testing

1. This testing is used to reduce very large number of test cases to manageable chunks.
2. Very clear guidelines on determining test cases without compromising on the effectiveness of testing.

3. Appropriate for calculation-intensive applications with large number of variables/inputs

Summary:

- Boundary Analysis testing is used when practically it is impossible to test large pool of test cases individually
- Two techniques - Equivalence Partitioning & Boundary Value Analysis testing techniques is used
- In Equivalence Partitioning, first you divide a set of test condition into a partition that can be considered.
- In Boundary Value Analysis you then test boundaries between equivalence partitions

Decision table testing

The techniques of equivalence partitioning and boundary value analysis are often applied to specific situations or inputs. However, if different combinations of inputs result in different actions being taken, this can be more difficult to show using equivalence partitioning and boundary value analysis, which tend to be more focused on the user interface. The other two specification-based techniques, decision tables and state transition testing are more focused on business logic or business rules.

A **decision table** is a good way to deal with combinations of things (e.g. inputs). This technique is sometimes also referred to as a 'cause-effect' table. The reason for this is that there is an associated logic diagramming technique called 'cause-effect graphing' which was sometimes used to help derive the decision table

Decision tables provide a systematic way of stating complex business rules, which is useful for developers as well as for testers. Decision tables can be used in test design whether or not they are used in specifications, as they help testers explore the effects of combinations of different inputs and other software states that must correctly implement business rules.

If you are a new customer opening a credit card account, you will get a 15% discount on all your purchases today. If you are an existing customer and you hold a loyalty card, you get a 10% discount. If you have a coupon, you can get 20% off today (but it can't be used with the 'new customer' discount). Discount amounts are added, if applicable.

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
New customer (15%)	T	T	T	T	F	F	F	F
Loyalty card (10%)	T	T	F	F	T	T	F	F
Coupon (20%)	T	F	T	F	T	F	T	F
Actions								
Discount (%)	X	X	20	15	30	10	20	0

Note that we have put X for the discount for two of the columns (Rules 1 and 2) - this means that this combination should not occur. You cannot be both a new customer and already hold a loyalty card! There should be an error message stating this, but even if we don't know what that message should be, it will still make a good test.

We have made an assumption in Rule 3. Since the coupon has a greater discount than the new customer discount, we assume that the customer will choose 20% rather than 15%. We cannot add them, since the coupon cannot be used with the 'new customer' discount. The 20% action is an assumption on our part, and we should check that this assumption (and any other assumptions that we make) is correct, by asking the person who wrote the specification or the users.

For Rule 5, however, we can add the discounts, since both the coupon and the loyalty card discount should apply (at least that's our assumption).

Rules 4, 6 and 7 have only one type of discount and Rule 8 has no discount. so 0%.

If we are applying this technique thoroughly, we would have one test for each column or rule of our decision table. The advantage of doing this is that we may test a combination of things that otherwise we might not have tested and that could find a defect.

State transition testing

State transition testing is used where some aspect of the system can be described in what is called a 'finite state machine'. This simply means that the system can be in a (finite) number of different states, and the transitions from one state to another are determined by the rules of the 'machine'. This is the model on which the system and the tests are based. Any system where you get a different output for the same input, depending on what has happened before, is a finite state system. A finite state system is often shown as a **state diagram**

For example, if you request to withdraw $100 from a bank ATM, you may be given cash. Later you may make exactly the same request but be refused the money (because your balance is insufficient). This later refusal is because the state of your bank account has changed from having sufficient funds to cover the withdrawal to having insufficient funds. The transaction that caused your account to change its state was probably the earlier withdrawal. A state diagram can represent a model from the point of view of the system, the account or the customer.

Another example is a word processor. If a document is open, you are able to close it. If no document is open, then 'Close' is not available. After you choose 'Close' once, you cannot choose it again for the same document unless you open that document. A document thus has two states: open and closed.

A state transition model has four basic parts:

- the states that the software may occupy (open/closed or funded/insufficient funds);
- the transitions from one state to another (not all transitions are allowed);
- the events that cause a transition (closing a file or withdrawing money);
- the actions that result from a transition (an error message or being given your cash).

Note that in any given state, one event can cause only one action, but that the same event - from a different state - may cause a different action and a dif-ferent end state.

We will look first at test cases that execute valid state transitions.

Figure 4.2 shows an example of entering a Personal Identity Number (PIN) to a bank account. The states are shown as circles, the transitions as lines with arrows and the events as the text near the transitions. (We have not shown the actions explicitly on this diagram, but they would be a message to the customer saying things such as 'Please enter your PIN'.)

The state diagram shows seven states but only four possible events (Card inserted, Enter PIN, PIN OK and PIN not OK). We have not specified all of the possible transitions here - there would also be a time-out from 'wait for PIN' and from the three tries which would go back to the start state after the time had elapsed and would probably eject the card. There would also be a transi-tion

from the 'eat card' state back to the start state. We have not specified all the possible events either - there would be a 'cancel' option from 'wait for PIN' the card. The 'access account' state would be the beginning of another state diagram showing the valid transactions that could now be performed on the account.

However this state diagram, even though it is incomplete, still gives us information on which to design some useful tests and to explain the state transition technique.

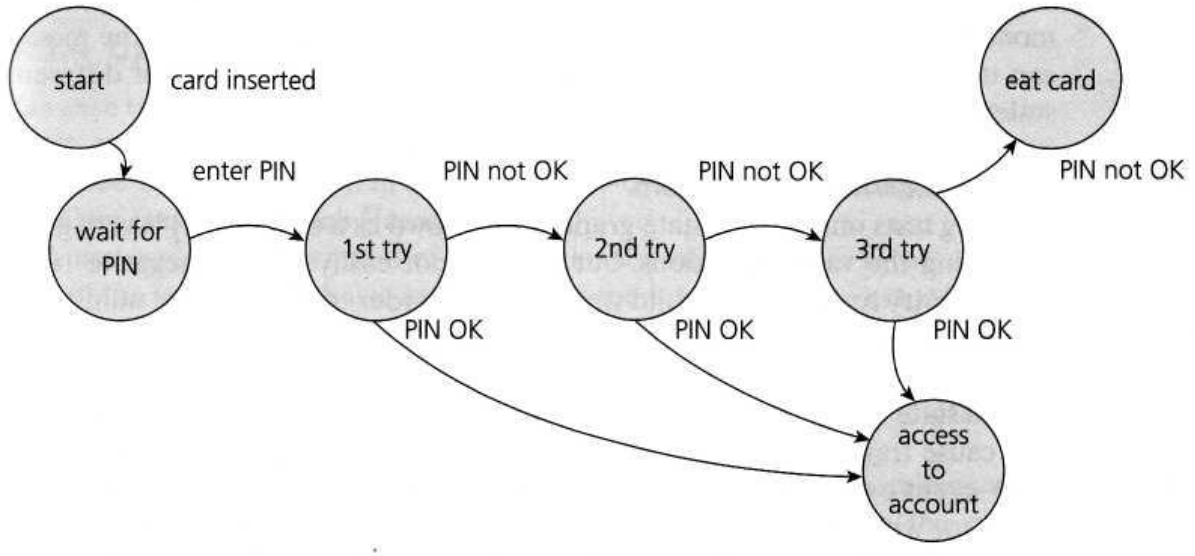


FIGURE 4.2 State diagram for PIN entry

Use case testing

Use case testing is a technique that helps us identify test cases that exercise the whole system on a transaction by transaction basis from start to finish. A use case is a description of a particular use of the system by an actor (a user of the system). Each use case describes the interactions the actor has with the system in order to achieve a specific task (or, at least, produce something of value to the user). Actors are generally people but they may also be other systems. Use cases are a sequence of steps that describe the interactions between the actor and the system.

Use cases are defined in terms of the actor, not the system, describing what the actor does and what the actor sees rather than what inputs the system expects and what the system'outputs. They often use the language and terms of the business rather than technical terms, especially when the actor is a

business user. They serve as the foundation for developing test cases mostly at the system and acceptance testing levels.

Use cases can uncover integration defects, that is, defects caused by the incorrect interaction between different components.

	Step	Description
Main Success Scenario **A: Actor** **S: System**	1	A: Inserts card
	2	S: Validates card and asks for PIN
	3	A: Enters PIN
	4	S: Validates PIN
	5	S: Allows access to account
Extensions	2a	Card not valid S: Display message and reject card
	4a	PIN not valid S: Display message and ask for re-try (twice)
	4b	PIN invalid 3 times S: Eat card and exit

FIGURE 4.3 Partial use case for PIN entry

WHITE BOX TESTING (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester. The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential.

Example

A tester, usually a developer as well, studies the implementation code of a certain field on a webpage, determines all legal (valid and invalid) AND illegal inputs and verifies the outputs against the expected outcomes, which is also determined by studying the implementation code.

White Box Testing is like the work of a mechanic who examines the engine to see why the car is not moving.

Levels Applicable To

White Box Testing method is applicable to the following levels of software testing:

- Unit Testing: For testing paths within a unit.
- Integration Testing: For testing paths between units.

However, it is mainly applied to Unit Testing.

Advantages

- Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.
- Testing is more thorough, with the possibility of covering most paths.

Disadvantages

- Since tests can be very complex, highly skilled resources are required, with a thorough knowledge of programming and implementation.
- Test script maintenance can be a burden if the implementation changes too frequently.
- Since this method of testing is closely tied to the application being tested, tools to cater to every kind of implementation/platform may not be readily available.

3 Main White Box Testing Techniques:

1. Statement Coverage
2. Branch Coverage
3. Path Coverage

Note that the statement, branch or path coverage does not identify any bug or defect that needs to be fixed. It only identifies those lines of code which are either never executed or remains untouched. Based on this further testing can be focused on.

Let's understand these techniques one by one with a simple example.

#1) Statement coverage:

In a programming language, a statement is nothing but the line of code or instruction for the computer to understand and act accordingly. A statement becomes an executable statement when it gets compiled and converted into the object code and performs the action when the program is in a running mode.

Hence *"Statement Coverage"*, as the name itself suggests, it is the method of validating whether each and every line of the code is executed at least once.

#2) Branch Coverage:

"Branch" in a programming language is like the "IF statements". An IF statement has two branches: T**rue and False**.

So in Branch coverage (also called Decision coverage), we validate whether each branch is executed at least once.

In case of an "IF statement", there will be two test conditions:

- One to validate the true branch and,
- Other to validate the false branch.

Hence, in theory, Branch Coverage is a testing method which is when executed ensures that each and every branch from each decision point is executed.

#3) Path Coverage

Path coverage tests all the paths of the program. This is a comprehensive technique which ensures that all the paths of the program are traversed at least once. Path Coverage is even more powerful than Branch coverage. This technique is useful for testing the complex programs.

White Box Testing Example

Consider the below simple pseudocode:

```
INPUT A & B

C = A + B

IF C>100

PRINT "ITS DONE"
```

For ***Statement Coverage*** – we would only need one test case to check all the lines of the code.

That means:

If I consider *TestCase_01 to be (A=40 and B=70),* then all the lines of code will be executed.

Now the question arises:

1. Is that sufficient?
2. What if I consider my Test case as A=33 and B=45?

Because Statement coverage will only cover the true side, for the pseudo code, only one test case would NOT be sufficient to test it. As a tester, we have to consider the negative cases as well.

Hence for maximum coverage, we need to consider **"*Branch Coverage*"**, which will evaluate the "FALSE" conditions.
In the real world, you may add appropriate statements when the condition fails.

So now the pseudocode becomes:

```
INPUT A & B




C = A + B

IF C>100

PRINT "ITS DONE"

ELSE



PRINT "ITS PENDING"
```

Since Statement coverage is not sufficient to test the entire pseudo code, we would require Branch coverage to ensure maximum coverage**.**
So for Branch coverage, we would require two test cases to complete the testing of this pseudo code.

TestCase_01: A=33, B=45
TestCase_02: A=25, B=30
With this, we can see that each and every line of the code is executed at least once.

Here are the Conclusions that are derived so far:

- Branch Coverage ensures more coverage than Statement coverage.
- Branch coverage is more powerful than Statement coverage.
- 100% Branch coverage itself means 100% statement coverage.
- But 100 % statement coverage does not guarantee 100% branch coverage.

Now let's move on to ***Path Coverage:***

As said earlier, Path coverage is used to test the complex code snippets, which basically involve loop statements or combination of loops and decision statements.

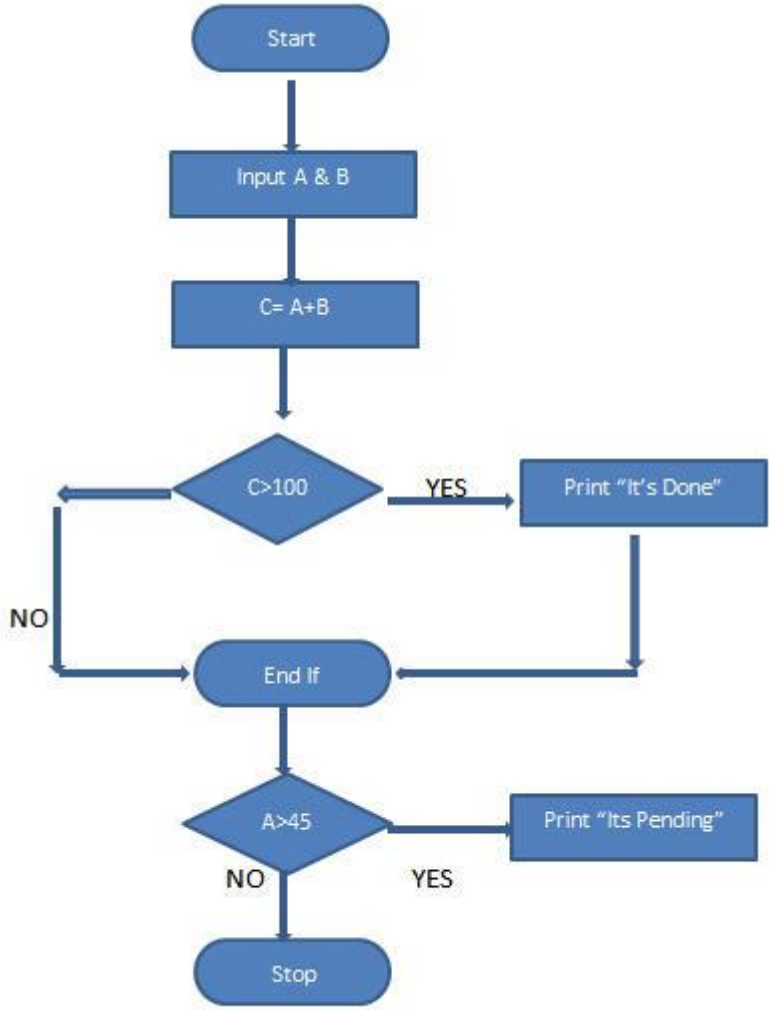
Consider this pseudocode:

```
INPUT A & B

C = A + B

IF C>100

PRINT "ITS DONE"



END IF

IF A>50

PRINT "ITS PENDING"

END IF
```

Now to ensure maximum coverage, we would require 4 test cases.

How? Simply – there are 2 decision statements, so for each decision statement, we would need two branches to test. One for true and the other for the false condition. So for 2 decision statements, we would require 2 test cases to test the true side and 2 test cases to test the false side, which makes a total of 4 test cases.

To simplify these let's consider below flowchart of the pseudo code we have:



Path Coverage

In order to have the full coverage, we would need following test cases:

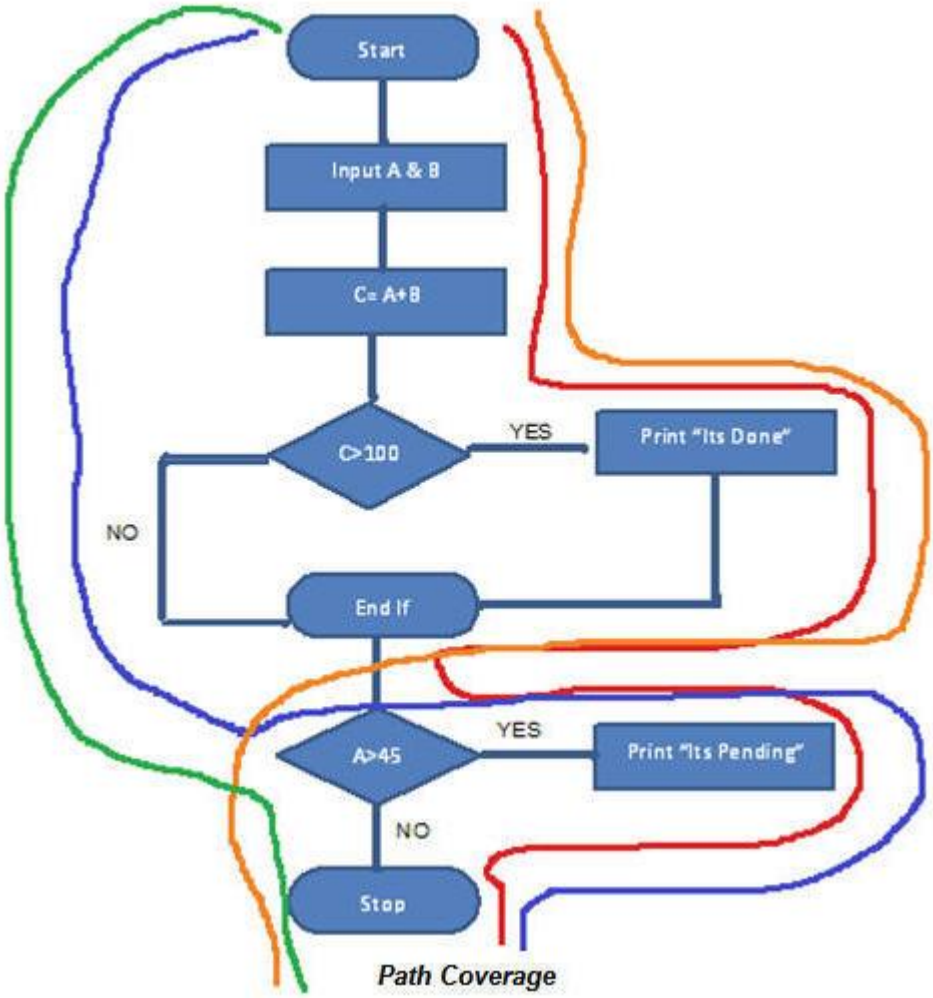
TestCase_01: A=50, B=60

TestCase_02: A=55, B=40

TestCase_03: A=40, B=65

TestCase_04: A=30, B=30

So the path covered will be:



Red Line – TestCase_01 = (A=50, B=60)

Blue Line = TestCase_02 = (A=55, B=40)

Orange Line = TestCase_03 = (A=40, B=65)

Green Line = TestCase_04 = (A=30, B=30)

The Differences Between Black Box Testing and White Box Testing are listed below.

Criteria	**Black Box Testing**	**White Box Testing**
Definition	Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the item	White Box Testing is a software testing method in which the internal structure/ design/ implementation of

	being tested is NOT known to the tester	the item being tested is known to the tester.
Levels Applicable To	Mainly applicable to higher levels of testing:Acceptance Testing System Testing	Mainly applicable to lower levels of testing:Unit Testing Integration Testing
Responsibility	Generally, independent Software Testers	Generally, Software Developers
Programming Knowledge	Not Required	Required
Implementation Knowledge	Not Required	Required
Basis for Test Cases	Requirement Specifications	Detail Design