Government Polytechnic Mumbai

Course: Software Engineering Unit no.: 5

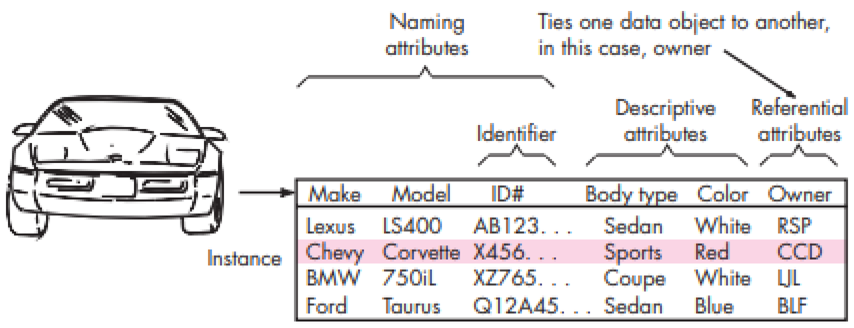
5) Software Modelling and Design

5.1 Translating Requirement Model into Design Model: Data Modelling.

Data Modeling Concept:

1. If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may choose to create a data model as part of overall requirements modeling.
2. A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships.
3. The entity-relationship diagram (ERD) addresses these issues and represents all data objects that are entered, stored, transformed, and produced within an application.

<u>Example</u>



Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

Figure:Tabular representation of data objects

<u>Data Objects</u>

1. <u>A data object is a representation of composite information that must be understood by software.</u> By composite information, I mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.
2. <u>A data object can be an external entity</u> (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a person or a car can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes.
3. <u>A data object encapsulates data only—there is no reference within a data object to operations that act on the data.</u> Therefore, the data object can be represented as a table **as shown in above Figure.** The headings in the table reflect attributes of the object. In this case, a car is defined in terms of make, model, ID number, body type, color, and owner. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object car.

<u>Data Attributes</u>

1. Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to
<u>(1) name an instance of the data object,</u>
<u>(2) describe the instance, or</u>
<u>(3) make reference to another instance in another table.</u>

2. In addition , one or more of the attributes must be defined as an identifier—that is, the identifier attribute becomes a “key” when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object car, a reasonable identifier might be the ID number.
3. From the above figure, The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for car might serve well for an application that would be used by a department of motor vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software. In the latter case, the attributes for car might also include ID number, body type, and color, but many additional attributes (e.g., interior code, drive train type, trim package designator, transmission type) would have to be added to make the car a meaningful object in the manufacturing control context.

Relationships

1. Data objects are connected to one another in different ways. Consider the two data objects, person and car. These objects can be represented using the simple notation **illustrated in Figure a.**
2. A connection is established between person and car because the two objects are related. But what are the relationships? To determine the answer, you should understand the role of people (owners, in this case) and cars within the context of the software to be built. You can establish a set of object/ relationship pairs that define the relevant relationships. For example,
    - A person owns a car.
    - A person is insured to drive a car.

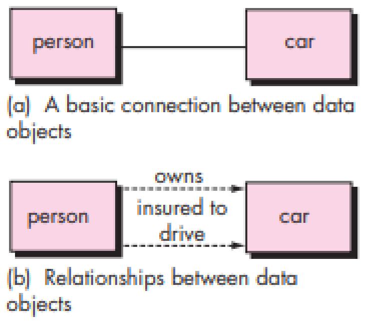

person
car
(a) A basic connection between data objects
owns
person
insured to drive
car
(b) Relationships between data objects

Figure: Relationships between data objects

3. The relationships owns and insured to drive define the relevant connections between person and car. **Figure b** illustrates these object-relationship pairs graphically.
4. The arrows noted in **Figure b** provide important information about the directionality of the relationship and often reduce ambiguity or misinterpretations.

5.2 Analysis Modeling: Elements of Analysis model.

- Analysis Model is a technical representation of the system. It acts as a link between the system description and the design model.
- In Analysis Modelling, information, behavior, and functions of the system are defined and translated into the architecture, component, and interface level design in the design modeling.
- Analysis modeling tools provide the capability to develop scenario-based models, class-based models, and behavioral models using UML notation.
- Requirements modeling (also called analysis modeling) focuses primarily on classes that are extracted directly from the statement of the problem.
- These entity classes typically represent things that are to be stored in a database and persist throughout the duration of the application (unless they are specifically deleted).

Elements of Analysis model

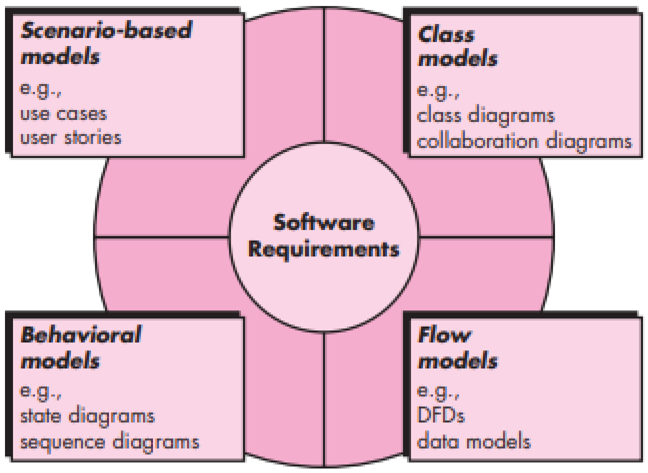

Scenario-based models
e.g., use cases user stories
Class models
e.g., class diagrams collaboration diagrams
Software Requirements
Behavioral models
e.g., state diagrams sequence diagrams
Flow models
e.g., DFDs data models

Figure: Elements of the analysis model

The specific elements of the analysis model are dictated by the analysis modeling method that is to be used. However, a set of generic elements is common to most analysis models.

Scenario-based elements:

1. The system is described from the user's point of view using a scenario-based approach.
2. It is always a good idea to get stakeholders involved. One of the best ways to do this is to have each stakeholder write use-cases that describe how the software engineering models will be used.
3. **<u>Functional</u>**—processing narratives for software functions.
4. **<u>Use-case</u>**- descriptions of the interaction between an "actor" and the system.
5. **<u>User Stories-</u>** descriptions of short views of an actor in the system.

Class-based elements:

1. Each usage scenario implies a set of "objects" that are manipulated as an actor interacts with the system.
2. These objects are categorized into classes- a collection of things that have similar attributes and common behaviors.
3. One way to isolate classes is to look for descriptive nouns in a use-case script. At least some of the nouns will be candidate classes.
4. **<u>Class Diagram provide three sections:</u>** (i)the top section containing the name of the class; (ii) the middle section containing and (iii)class attributes the bottom section representing operations of the class
5. <u>**Class-Responsibility-Collaborator (CRC)**</u> Modeling provides a simple means for identifying and organizing the classes that are relevant to system or product requirements.

Flow-oriented elements:

1. Information is transformed as it flows through a computer-based system.
2. The system accepts input in a variety of forms; applies functions to transform it; and produces output in a variety of forms.
3. Represents how data objects are transformed as they move through the system.
4. **A data flow diagram (DFD)** is the diagrammatic form that is used to complement UML diagrams. Considered by many to be an 'old school' approach, flow-oriented modeling continues to provide a view of the system that is unique. The DFD takes an input-process-output insight into system requirements and flow.
5. **Data objects** are represented by labeled arrows and transformations are represented by circles (called bubbles).

Behavioral elements:

1. The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the analysis model must provide modeling elements that depict behavior.
2. **The state diagram** is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state.
3. A state is any observable mode of behavior. Moreover, the state diagram indicates what actions are taken as a consequence of a particular event.
4. The second type of behavioral representation, called a **sequence diagram** in UML, indicates how events cause transitions from object to object.
5. State diagrams and sequence diagrams are the notations used for behavioral modeling.

5.3 Design Modeling: Fundamental Design Concept (Abstraction, Information hiding, Structure, Modularity, Concurrency, Verification, Aesthetics)

Abstraction

1. When you consider a modular solution to any problem, many levels of abstraction can be posed.
2. **At the highest level of abstraction**, a solution is stated in broad terms using the language of the problem environment.
3. **At lower levels of abstraction,** a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented. As different levels of abstraction are developed, you work to create both procedural and data abstractions. A procedural abstraction refers to a sequence of instructions that have a specific and limited function.
4. **The name of a procedural abstraction** implies these functions, but specific details are suppressed. An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).
5. **A data abstraction is a named collection of data that describes a data object.** In the context of the procedural abstraction open, we can define a data abstraction called door. Like any data object, the data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction door.

Structure/Architecture

1. **Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system"**. In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.
2. In a broader sense, however, components can be generalized to represent major system elements and their interactions. **One goal of software design is to derive an architectural rendering of a system**. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to solve common design problems.
3. **Shaw and Garlan** describe a set of properties that should be specified as part of an architectural design:
    - **Structural properties:** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another.
    - **Extra-functional properties:** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
    - **Families of related systems:** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems.

Modularity

1. **Modularity** is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.
2. It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78].
3. **Monolithic software** (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
4. **Module & Software Cost** -

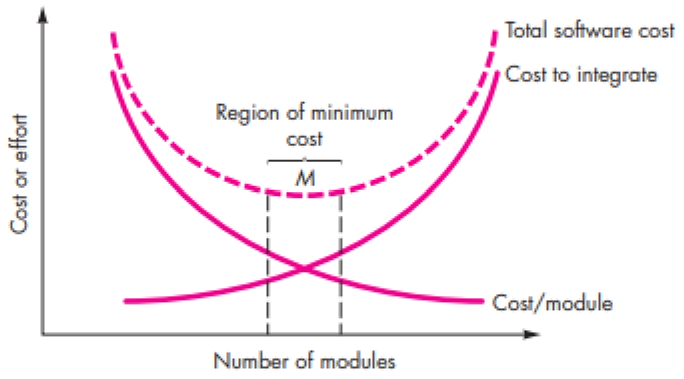


Figure: Module & Software Cost

5. **The curves shown in Figure** do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of M. **Undermodularity or overmodularity** should be avoided. But how do you know the vicinity of M? How modular should you make software?
6. **These characteristics lead to a total cost or effort curve shown in the figure.** There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

Information hiding

1. The concept of modularity leads you to a fundamental question: "How do I decompose a software solution to obtain the best set of modules?" **The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others."**
2. In other words, modules should be specified and designed so that **information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.**
3. **Hiding** implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.
4. **Abstraction helps to define the procedural (or informational) entities that make up the software.** Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.
5. **The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance.** Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Concurrency

1. Computer has limited resources and they must be utilized efficiently as much as possible. To utilize these resources efficiently, multiple tasks must be executed concurrently.
2. This requirement makes concurrency one of the major concepts of software design. Every system must be designed to allow multiple processes to execute concurrently, whenever possible.

3. For example, if the current process is waiting for some event to occur, the system must execute some other process in the mean time.

Verification

1. Verification refers to the set of tasks that ensure that software correctly implements a specific function. Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
2. Boehm states this another way: Verification: “Are we building the product right?”
3. A verification mechanism that validates the identity of all clients and servers, allowing communication to occur only when both sides are verified.

Aesthetics

1. Aesthetics should never supersede functionality. For example, a simple button might be a better navigation option than an aesthetically pleasing, but vague image or icon whose intent is unclear.
2. There’s no question that each of us has a different and very subjective vision of what is aesthetic.
3. And yet, most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and an obvious “presence” that are hard to quantify but are evident nonetheless. Aesthetic software has these characteristics.
4. Do layout, color, typeface, and related characteristics lead to ease of use? Do users “feel comfortable” with the look and feel of the WebApp?

5.4 Design Notations: Data Flow Diagram (DFD), Structured Flowcharts and Decision Tables

Data Flow Diagram (DFD):

1. The DFD takes an input-process-output view of a system. That is, data objects flow into the software, are transformed by processing elements, and resultant data objects flow out of the software.
2. Data objects are represented by labeled arrows, and transformations are represented by circles (also called bubbles).
3. The DFD is presented in a hierarchical fashion. That is, the first data flow model (sometimes called a level 0 DFD or context diagram) represents the system as a whole. Subsequent data flow diagrams refine the context diagram, providing increasing detail with each subsequent level.

Creating a Data Flow Mode

1. The data flow diagram enables you to develop models of the information domain and functional domain. As the DFD is refined into greater levels of detail, you perform an implicit functional decomposition of the system. At the same time, the DFD refinement results in a corresponding refinement of data as it moves through the processes that embody the application.
2. A few simple guidelines can aid immeasurably during the derivation of a data flow diagram:
   (1) The level 0 data flow diagram should depict the software/system as a single bubble;
   (2) **primary input and output should be carefully noted;**
   (3) **refinement should begin by isolating candidate processes, data objects, and data stores to be represented at the next level;**
   (4) **all arrows and bubbles should be labeled with meaningful names;**
   (5) **information flow continuity must be maintained from level to level,2** and
   (6) **one bubble at a time should be refined.** There is a natural tendency to overcomplicate the data flow diagram. This occurs when you attempt to show too much detail too early or represent procedural aspects of the software in lieu of information flow.

DFD Level 0

1. **It** is also called a Context Diagram. It’s a basic overview of the whole system or process being analyzed or modeled.

2. It’s designed to be an at-a-glance view, showing the system as a single high-level process, with its relationship to external entities. It should be easily understood by a wide audience, including stakeholders, business analysts, data analysts and developers.

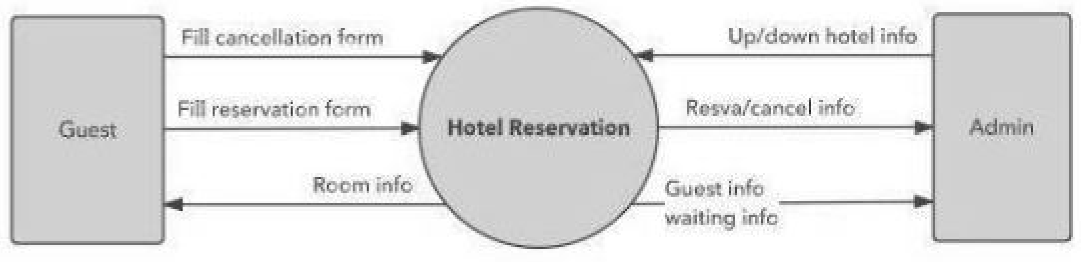


Figure: DFD Level 0

DFD **Level 1**

1. The Level 0 DFD is broken down into more specific, Level 1 DFD.
2. Level 1 DFD depicts basic modules in the system and flow of data among various modules.
3. Level 1 DFD also mentions basic processes and sources of information.
   - It provides a more detailed view of the Context Level Diagram.
   - Here, the main functions carried out by the system are highlighted as we break into its sub-processes.

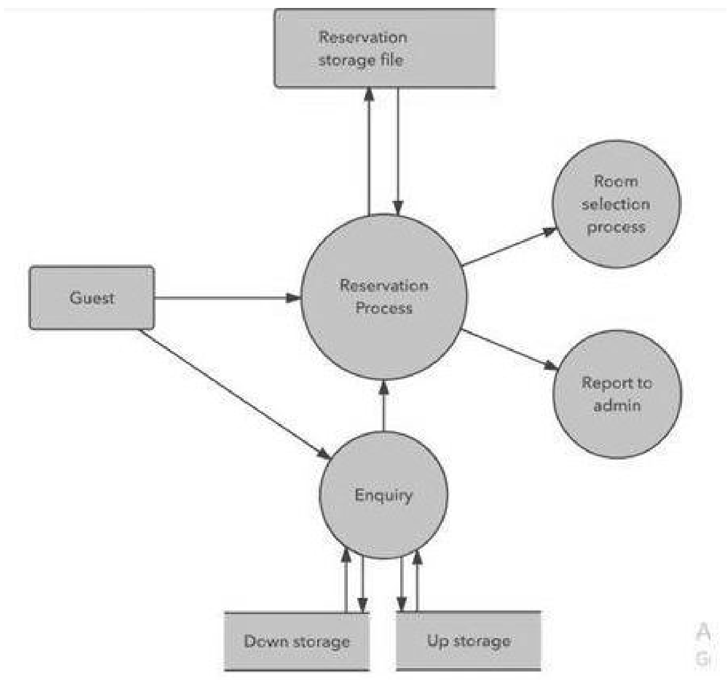


Figure: DFD Level 1

Self Practice Example:
1) Design DFD level 0, Level1 & Level 2 for Railway Registration In IRCTC.
2) Design DFD Level 0, Level 1 & Level 2 for Course Registration in MIS of GPM.

Advantages of using Data Flow Diagrams (DFD) include:

1. Easy to understand: DFDs are graphical representations that are easy to understand and communicate, making them useful for non-technical stakeholders and team members.

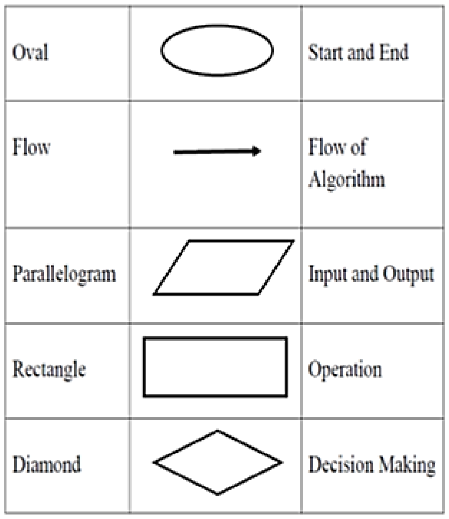
2. Improves system analysis: DFDs are useful for analyzing a system’s processes and data flow, which can help identify inefficiencies, redundancies, and other problems that may exist in the system.
3. Supports system design: DFDs can be used to design a system’s architecture and structure, which can help ensure that the system is designed to meet the requirements of the stakeholders.
4. Enables testing and verification: DFDs can be used to identify the inputs and outputs of a system, which can help in the testing and verification of the system’s functionality.
5. Facilitates documentation: DFDs provide a visual representation of a system, making it easier to document and maintain the system over time.

Disadvantages of using DFDs include:

1. Can be time-consuming: Creating DFDs can be a time-consuming process, especially for complex systems.
2. Limited focus: DFDs focus primarily on the flow of data in a system, and may not capture other important aspects of the system, such as user interface design, system security, or system performance.
3. Can be difficult to keep up-to-date: DFDs may become out-of-date over time as the system evolves and changes.
4. Requires technical expertise: While DFDs are easy to understand, creating them requires a certain level of technical expertise and familiarity with the system being analyzed.
5. Overall, the benefits of using DFDs outweigh the disadvantages. However, it is important to recognize the limitations of DFDs and use them in conjunction with other tools and techniques to analyze and design complex software systems.

Structured Flowcharts

1. A structured flowchart appears to have emerged in response to demands of easing the impact of complexity. In this context, it emerges as an illustration in which “all of the processes and decisions must fit into one of a few basic structured elements.”
2. For example, a flowchart emerges when we sketch an illustration that details the execution of a series of actions predicated on a range of primary
3. A flowchart is a graphical representation of an algorithm. It gives flow of the algorithm.
4. The table1 shows the various symbols for drawing the flowchart.



Oval	⬭	Start and End
Flow	→	Flow of Algorithm
Parallelogram	▱	Input and Output
Rectangle	▭	Operation
Diamond	◇	Decision Making

Figure: Symbol

Guidelines for Developing Flowcharts
These are some points to keep in mind while developing a flowchart −

1. Flowchart can have only one start and one stop symbol
2. On-page connectors are referenced using numbers
3. Off-page connectors are referenced using alphabets
4. General flow of processes is top to bottom or left to right
5. Arrows should not cross each other

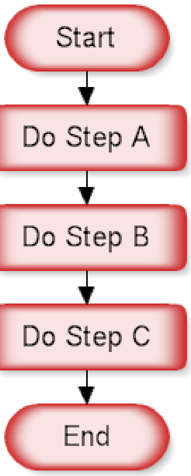
Drawing a Structured Flowchart

You can make your flowcharts easier to understand and less subject to errors by using only a fixed set of structures. These structures include:

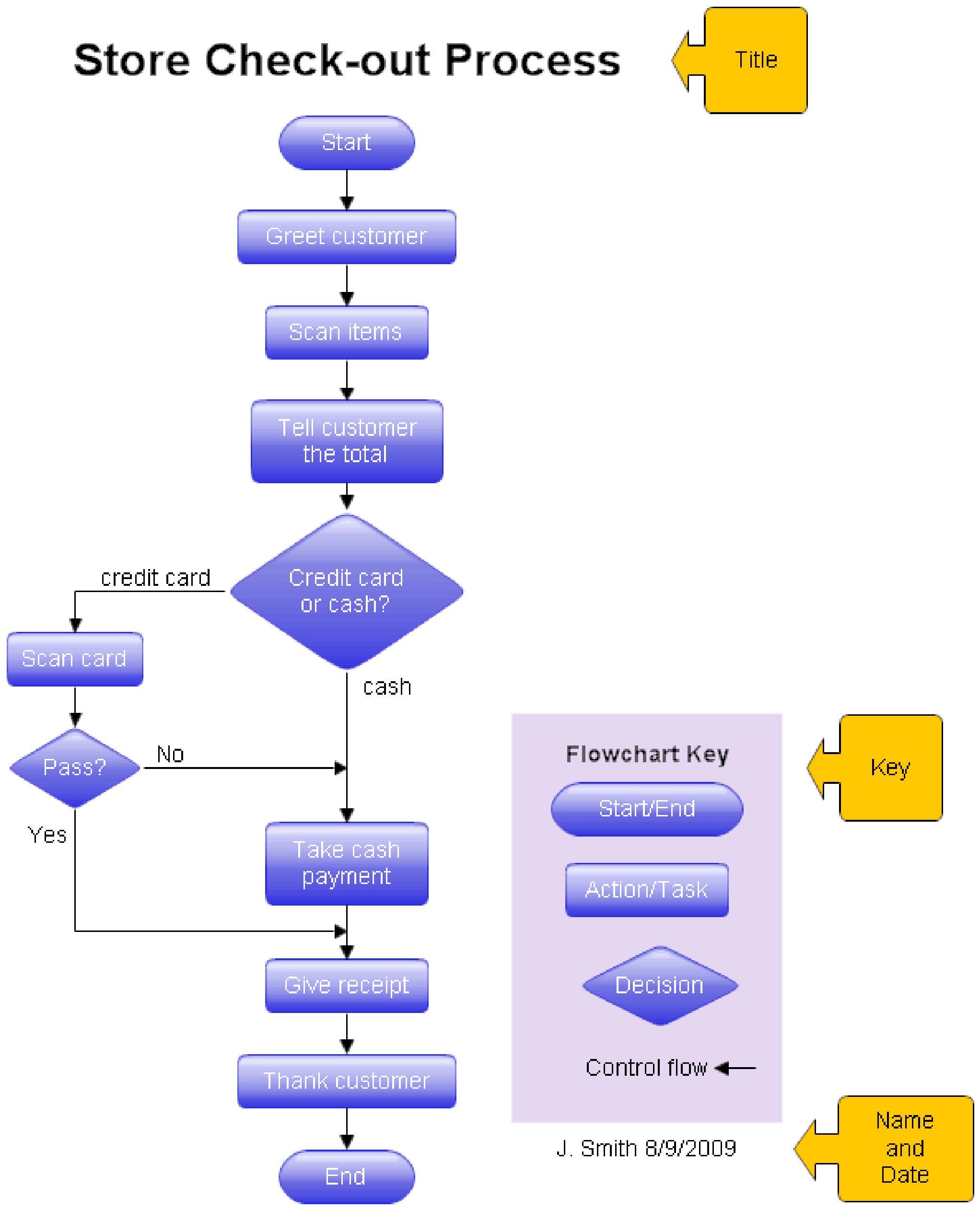
- Sequence
- Decision
- Loop
- Case

Whether you are flowcharting software programs or business processes, using only these structures will make it easier to find and correct errors in your charts. Each structure has a simple flow of control with one input and one output. These structures can then be nested within each other. Any chart can be drawn using only these structures. You do not have to use GOTO or draw spaghetti diagrams just because you are drawing a flowchart. You can draw structured flowcharts.

Sample



Example



Decision Tables

What are Decision Tables?

The act of making a choice may be made much easier with the help of a decision table. It is a collection of rules laid down in rows and columns. Conditions or circumstances that influence the best course of action are represented in the columns, and their permutations are shown in the rows.

1. Purpose of Decision Tables

Decision tables are meant to aid programmers in making difficult choices in a systematic and precise manner. Decision tables give a systematic framework for examining and optimising the decision-making process by decomposing a choice into its component pieces.

2. Types of Decision Tables

Decision tables might have a small number of entries, a large number of entries, or both. The simplest kind of table, limited-entry tables only provide the data that is absolutely essential to make a choice. Extended-entry tables provide extra columns for unusual circumstances and exceptions, whereas full-entry tables have all conceivable conditions.

3. Use Cases for Decision Tables

Business rule management, risk management, quality assurance, and project management are just few of the many software development activities that benefit greatly from the usage of decision tables. They are also applicable in the realms of finance and medical, where nuanced judgements are often called for.

Elements of a Decision Table

1. Conditions

What should be done depends on the current circumstances. In the decision table, each column corresponds to a yes or false answer.

2. Actions

The results of a particular set of circumstances are actions. In the decision table, they are also shown as columns.

3. Rules

Rules are sets of criteria that, when met, dictate what must be done. They are generated by examining the many permutations of the circumstances and are shown as rows in the decision table.

4. Rows and columns

A decision table consists of columns and rows. The circumstances and responses are shown in columns, while the alternative outcomes are shown in rows.

How to Create a Decision Table

1. Identify the Problem

Defining the issue at hand is the first step in developing a decision table. The first step is to identify the relevant factors and decide what has to be done next.

2. Define the Conditions and Actions

When the issue has been recognised, it is necessary to specify the circumstances and measures to be taken. Dissecting the issue and identifying its constituent elements is a necessary step in this process.

3. Create the Rules

After the circumstances and responses have been established, the rules may be written. To do this, you must consider all the many permutations of the criteria and decide what to do in each one.

4. Organize the Decision Table

Finally, the decision table has to be laid up in a way that is easy to understand and use. To do this, we must organise the rules in a logical order, classifying similar circumstances and actions together.

Benefits of Using Decision Tables

1. Increased Lucidity

The decision-making process may be represented in a clear and succinct visual format using decision tables. Decision tables help developers comprehend and improve their code by partitioning large, complicated choices into smaller, more digestible pieces.

2. Productivity Boost

The decision-making process may be simplified with the use of decision tables, which also contribute to increased productivity. Developers may benefit from using decision tables because they provide a transparent framework for assessing and improving the decision-making process.

3. Enhanced Precision

By eliminating room for mistake, decision tables further improve decision quality. Decision tables let developers catch and fix any issues before they become serious by breaking down difficult choices into smaller, more manageable portions.

4. Greater Accuracy

The decision-making process may also be documented in a clear and succinct format with the help of decision tables. Developers may guarantee their code is well-documented and simple to comprehend by generating a visual depiction of the decision-making process.

5. Enhanced Communication

Developers and stakeholders may better communicate with one another using decision tables. Decision tables help developers better convey their ideas and solutions to others by giving a visual picture of the decision-making process

Tools for Creating Decision Tables

1. Spreadsheet Software

Simple decision tables may be made in spreadsheet programmes like Microsoft Excel or Google Sheets. For those comfortable with spreadsheet programmes, this is an easy and convenient method for creating decision tables.

2. Decision Table Software

Software like RuleDesigner and DecisionTools Suite are dedicated to decision tables. These programmes are superior to others in that they provide superior functionality and are tailored to the task of making decision tables.

3. Code Generators

Code generators exist in languages like Java and Python that may be used to make decision tables. Code is generated automatically by these generators according to certain specifications.

4. Online Tools

Decision tables may be made with the help of several internet tools that don't need any additional software installation. These programmes often don't cost anything and provide enough capabilities to make basic decision tables.

Best Practices for Creating Decision Tables

1. Keep it Simple

Keeping the decision table you're making as straightforward as feasible is a must. To do so, one must use simple language to convey the circumstances and the steps to be taken, and one must break down difficult judgements into smaller, more manageable components.

2. Always Use the Same Terminology And Notation

While deciding table, it's crucial to utilise consistent terminology and notation. This ensures that the table is straightforward, and that all parties engaged are using the same language.

3. Test and Validate the Decision Table

It is crucial to do extensive testing and validation on a decision table before putting it into commercial use. This requires verifying that the table is correct and generates the desired results under a variety of conditions.

4. Update the Decision Table as Needed

It is important to keep decision tables up to date as circumstances change. When the situation changes, new rules may need to be added, and existing ones may need changing.

5. Incorporate Feedback from Stakeholders

Finally, decision tables should factor in input from relevant parties. This ensures that the requirements of all parties are represented fairly at the table and that everyone benefits as much as possible from the decision-making process.

Examples of Decision Tables in Action

1. Software Design

In order to aid programmers in making complicated judgements in a timely and correct manner, decision tables are often employed in software development. To decide what action to take based on the user's input, a decision table, for instance, may be employed.

2. Business Rules

Business rule management, in which businesses utilise decision tables to better manage their many decision-making processes, is another popular use of decision tables. A decision table might be used to decide the best course of action for handling a customer complaint depending on the nature of the complaint and the customer's previous interactions with the business.

3. Quality Assurance

In software quality assurance, decision tables help find and fix bugs before they are released to the public. A decision table, for instance, may be used to ascertain the proper action to take when a certain fault is encountered in the code.

4. Risk Management

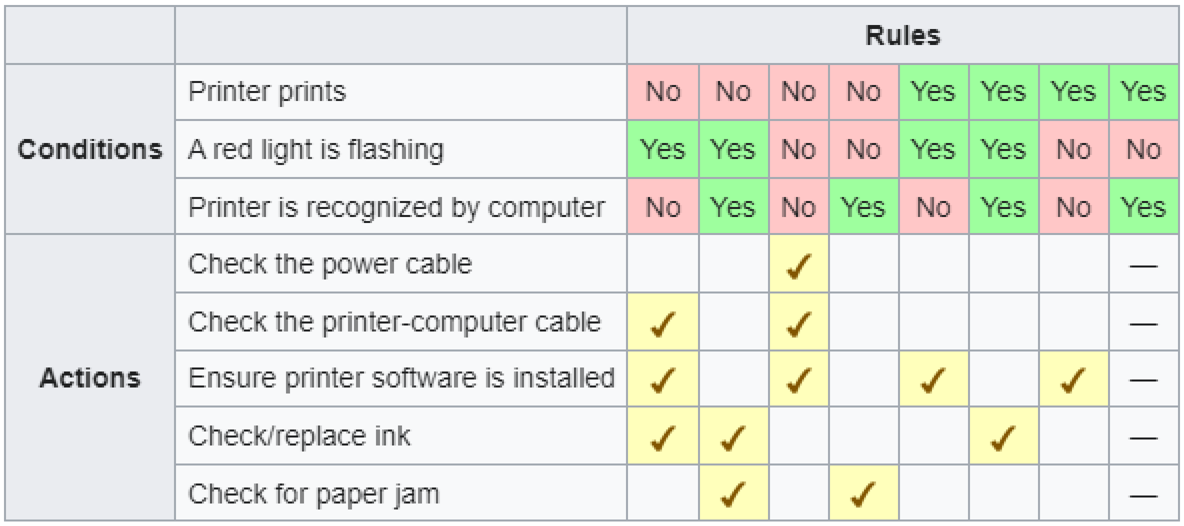
Organizations may use decision tables to assist manage risks and decide how to respond to them. If there has been a security breach, for instance, the proper reaction may be determined using a decision table that takes into account the severity of the breach and the possible effect on the business.

5. Project Management

Project management is another field that may benefit from the usage of decision tables. A decision table might be used, for instance, to evaluate the effect of a modification request on the project's schedule and budget and choose the most suitable answer.

Example

Printer troubleshooter



		Rules							
Conditions	Printer prints	No	No	No	No	Yes	Yes	Yes	Yes
	A red light is flashing	Yes	Yes	No	No	Yes	Yes	No	No
	Printer is recognized by computer	No	Yes	No	Yes	No	Yes	No	Yes
Actions	Check the power cable			✓					—
	Check the printer-computer cable	✓		✓					—
	Ensure printer software is installed	✓		✓		✓		✓	—
	Check/replace ink	✓	✓				✓		—
	Check for paper jam		✓		✓				—

Limitations of Decision Tables

1. Complexity

When dealing with many circumstances and rules, decision tables may become complicated, which is one of its primary drawbacks. Because of this, decision tables may be complicated to use and operate, and their upkeep may demand a lot of time and energy.

2. Limited Scope

However, decision tables have certain restrictions and may not work for all kinds of decision-making scenarios. They work best when there are explicit rules and criteria regulating the decision-making process and when choices can be broken down into smaller, more manageable portions.

3. Flexibility Issues

When decision tables are built to handle just one set of circumstances or one set of decisions, they may be rather rigid. This might make it tough to modify the decision table in light of fresh knowledge or shifting conditions.

4. Difficulties in Maintenance

Finally, decision tables may be cumbersome to update over time, particularly if the issue being solved changes or if more rules and circumstances are introduced. It might be time-consuming and resource-intensive to continually update and check the decision table

5.5 Testing- Meaning and purpose, Testing methods-Black-box and White-box, Level of Testing-Unit Testing, Integration Testing ,User Acceptance Testing
Testing- Meaning and purpose,

1. Software is tested to uncover errors that were made inadvertently as it was designed and constructed.
2. Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which you can place specific test case design techniques and testing methods—should be defined for the software process.
3. Testing begins at the component level and works “outward” toward the integration of the entire computer-based system.
4. Software testing is the process of evaluating a software item to detect differences between given input and expected output. Also to assess the feature of A software item.
5. Testing assesses the quality of the product. Software testing is a process that should be done during the development process.
6. In other words software testing is a verification and validation process. Basics of software testing. There are two basics of software testing: black box testing and whitebox testing.

Blackbox Testing Black box testing is a testing technique that ignores the internal mechanism of the system and focuses on the output generated against any input and execution of the system. It is also called functional testing.

Whitebox Testing White box testing is a testing technique that takes into account the internal mechanism of a system. It is also called structural testing and glass box testing. Black box testing is often used for validation and white box testing is often used for verification.

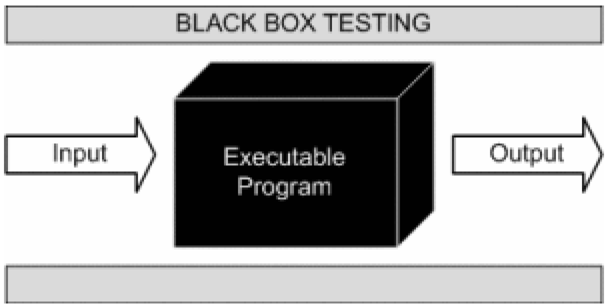
Types of testing

There are many types of testing like

- Unit Testing
- Integration Testing
- Functional Testing
- System Testing
- Stress Testing
- Performance Testing
- Usability Testing
- Acceptance Testing
- Regression Testing
- Beta Testing

Testing methods-
Black-box Testing

1. **Black-box testing** is a method of software testing that examines the functionality of an application based on the specifications.
2. It is also known as Specifications based testing. Independent Testing Team usually performs this type of testing during the software testing life cycle.
3. This method of test can be applied to each and every level of software testing such as unit, integration, system and acceptance testing.



Black Box Testing - Steps

Here are the generic steps followed to carry out any type of Black Box Testing.

• Initially requirements and specifications of the system are examined.

• Tester chooses valid inputs (positive test scenario) to check whether SUT processes them correctly . Also some invalid inputs (negative test scenario) are chosen to verify that the SUT is able to detect them.

• Tester determines expected outputs for all those inputs.

• Software tester constructs test cases with the selected inputs.

• The test cases are executed.

• Software tester compares the actual outputs with the expected outputs.

• Defects if any are fixed and re-tested.

Types of Black Box Testing:-

There are many types of Black Box Testing but following are the prominent ones -

Types of Black Box Testing Techniques: Following black box testing techniques are used for testing the software application.

• Boundary Value Analysis (BVA)

• Equivalence Class Partitioning

• Functional Testing

• Non-Functional Testing

• Regression Testing

1) Boundary Value Analysis (BVA):

Boundary Value Analysis is the most commonly used test case design method forblack box testing. As all we know the most of errors occurs at boundary of the input values. This is one of the techniques used to find the error in the boundaries of input values rather than the center of the input value range.

Boundary Value Analysis is the next step of the Equivalence class in which all test cases are design at the boundary of the Equivalence class.

2) Equivalence Class Partitioning:-

The equivalence class partition is the black box test case design technique used for writing test cases. This approach is use to reduce huge set of possible inputs to small but equally effective inputs. This is done by dividing inputs into the classes and gets one value from each class. Such method is used when exhaustive testing is most wanted & to avoid the redundancy of inputs.

In the equivalence partitioning input are divided based on the input values:

• If input value is Range, then we one valid equivalence class & two invalid equivalence classes.

• If input value is specific set, then we one valid equivalence class & one invalid equivalence classes.

• If input value is number, then we one valid equivalence class & two invalid equivalence classes.

• If input value is Boolean, then we one valid equivalence class & one invalid equivalence classes.

•

• **Functional testing** - This black box testing type is related to functional requirements of a system; it is done by software testers.

• **Non-functional testing** - This type of black box testing is not related to testing of a specific functionality, but non-functional requirements such as performance, scalability, usability.

• **Regression testing** - Regression Testing is done after code fixes, upgrades or any other system maintenance to check the new code has not affected the existing code.

Advantages of Black Box Testing:-

• Efficient when used on large systems.

• Since the tester and developer are independent of each other, testing is balanced and unprejudiced.

• Tester can be non-technical.

• There is no need for the tester to have detailed functional knowledge of system.

Disadvantages of Black Box Testing:-

• Test cases are challenging to design without having clear functional specifications.

• It is difficult to identify tricky inputs if the test cases are not developed based on specifications.

• It is difficult to identify all possible inputs in limited testing time. As a result, writing test cases may be slow and difficult.

White-box

1. **White Box Testing** is the testing of a software solution's internal coding and infrastructure. It focuses primarily on strengthening security, the flow of inputs and outputs through the application, and improving design and usability.
2. White box testing is also known as *Clear Box testing*, Open Box testing, Structural testing, Transparent Box testing, Code-Based testing, and Glass Box testing.
3. It is one of two parts of the "box testing" approach of software testing. Its counter-part, blackbox testing, involves testing from an external or end-user type perspective.
4. On the other hand, Whitebox testing is based on the inner workings of an application and revolves around internal testing.

What do you verify in White Box Testing?

White box testing involves the testing of the software code for the following:

• Internal security holes

• Broken or poorly structured paths in the coding processes

• The flow of specific inputs through the code

• Expected output

• The functionality of conditional loops

• How do you perform White Box Testing?

• To give you a simplified explanation of white box testing, we have divided it into two basic steps. This is what testers do when testing an application using the white box testing technique:

. STEP 1) UNDERSTAND THE SOURCE CODE

The first thing a tester will often do is learn and understand the source code of the application. Since white box testing involves the testing of the inner workings of an application, the tester must be very knowledgeable in the programming languages used in the applications they are testing. Also, the testing person must be highly aware of secure coding practices.

```
    Security is often one of the primary objectives of testing software. The tester
should be able to find security issues and prevent attacks from hackers and naive users
who might inject malicious code into the application either knowingly or unknowingly.
```

• Step 2) CREATE TEST CASES AND EXECUTE

The second basic step to white box testing involves testing the application's source code for proper flow and structure. One way is by writing more code to test the application's source code. The tester will develop little tests for each process or series of processes in the application.

```
      This method requires that the tester must have intimate knowledge of the code and
is often done by the developer. Other methods include Manual Testing, trial and error
testing and the use of testing tools as we will explain further on in this article.
```

White Box Testing Techniques:-

A major White box testing technique is Code Coverage analysis. Code Coverage analysis, eliminates gaps in a Test Case suite. It identifies areas of a program that are not exercised by a set of test cases. Once gaps are identified, you create test cases to verify untested parts of code, thereby increase the quality of the software product.

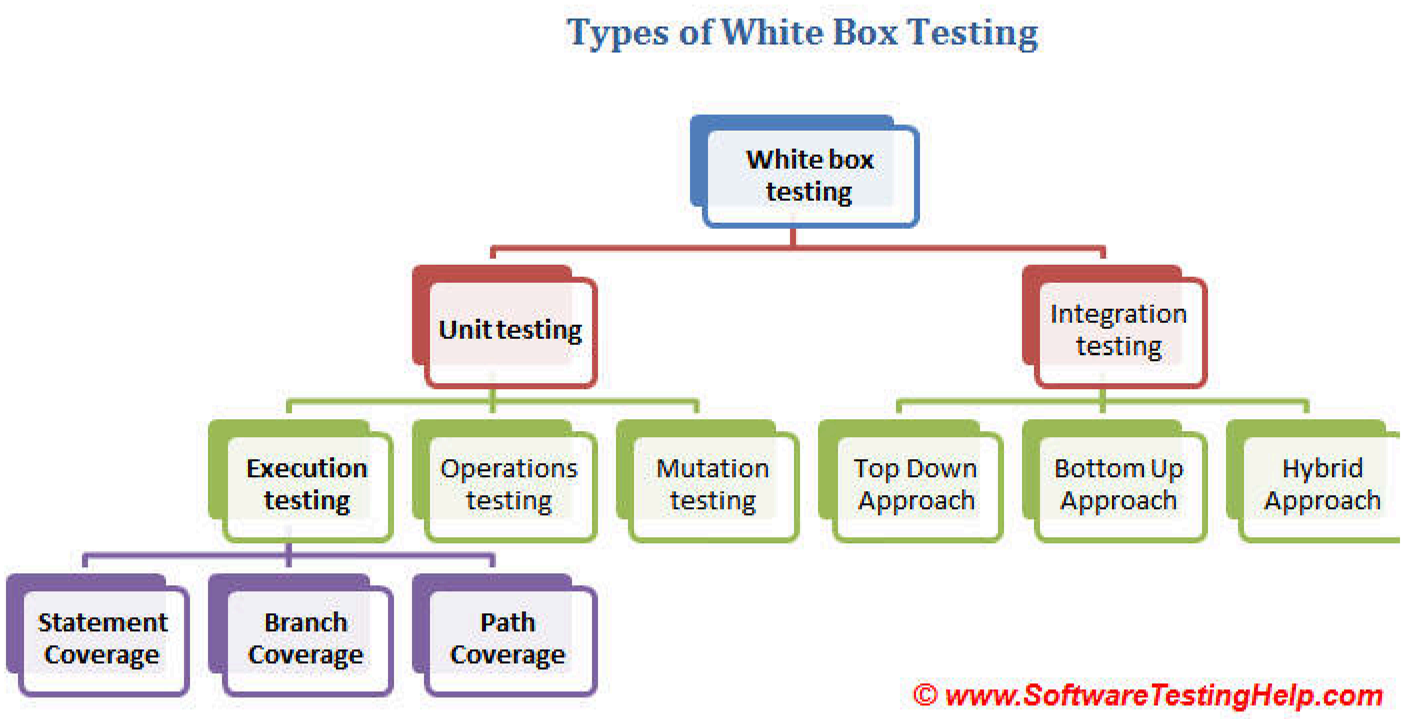
There are automated tools available to perform Code coverage analysis. Below are a few coverage analysis techniques:-

Statement Coverage - This technique requires every possible statement in the code to be tested at least once during the testing process.

Branch Coverage - This technique checks every possible path (if-else and other conditional loops) of a software application. Tools: An example of a tool that handles branch coverage testing for C, C++ and Java .

Apart from above, there are numerous coverage types such as Condition Coverage, Multiple Condition Coverage, Path Coverage, Function Coverage etc. Each technique has its own merits and attempts to test (cover) all parts of software code.

Using Statement and Branch coverage you generally attain 80-90% code coverage which is sufficient.



Types of White Box Testing:-

The 3 main White Box Testing Techniques are:

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

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: True 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.

Advantages of White Box Testing:-

• Code optimization by finding hidden errors.

• White box tests cases can be easily automated.

• Testing is more thorough as all code paths are usually covered.

Disadvantages of White Box Testing:-

• White box testing can be quite complex and expensive.

• Developers who usually execute white box test cases detest it. The white box testing by developers is not detailed can lead to production errors.

• White box testing requires professional resources, with a detailed understanding of programming and implementation.

Comparison Table

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 being tested is NOT known to the testerAlso called behavioural testing**	**White Box Testing is a software testing method in which the internal structure/ design/implementation of the item being tested is known to the tester.Also called glass box testing**
Levels Applicable To	**Mainly applicable to higher levels of testing:**	**Mainly applicable to lower levels of testing:**

	- Acceptance Testing - System 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
Type	Black box testing means functional test or external testing	White box testing means structural test or interior testing
Aim	check on what functionality is performing by the system	check on how System is performing
Suitable for	This type of project suitable for large projects	This type of project suitable for small projects

Level of Testing-
Unit Testing

1. Unit testing focuses verification effort on the smallest unit of software design—the software component or module.
2. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module.
3. The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing.
4. The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

Unit-test considerations

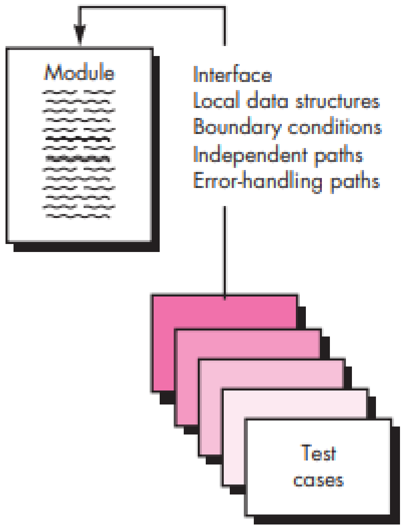


Figure: *Unit-test considerations*

Unit tests are illustrated schematically in Figure.

1. The module interface is tested to ensure that information properly flows into and out of the program unit under test.
2. **Local data structures** are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
3. **All independent paths** through the control structure are exercised to ensure that all statements in a module have been executed at least once.
4. **Boundary conditions** are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
5. And finally, all **error-handling paths** are tested. Data flow across a component interface is tested before any other testing is initiated. If data do not enter and exit properly, all other tests are moot.
6. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing. Selective testing of execution paths is an essential task during the unit test.
7. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.
8. Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the nth element of an n-dimensional array is processed, when the ith repetition of a loop with i passes is invoked, when the maximum or minimum allowable value is encountered.
9. Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

Unit-test procedures

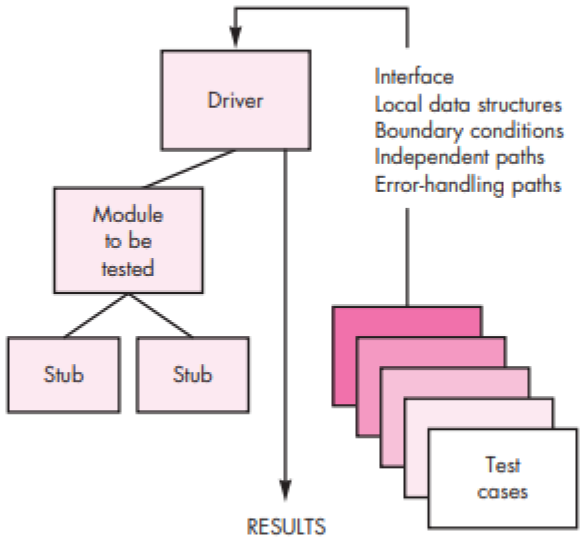


Figure: Unit-test environment

1. **Unit testing is normally considered as an adjunct to the coding step.** The design of unit tests can occur before coding begins or after source code has been generated.
2. **A review of design information provides guidance for establishing test cases that are likely to uncover errors**. Each test case should be coupled with a set of expected results. Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test.
3. The unit test environment is illustrated in Figure. In most applications a driver is nothing more than a **"main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results.**
4. **Stubs serve to replace modules that are subordinate (invoked by) the component to be tested.** A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing. Drivers and stubs represent testing "overhead."
5. That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. **If drivers and stubs are kept simple, actual overhead is relatively low.**
6. **Unfortunately, many components cannot be adequately unit tested with "simple" overhead software.** In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).
7. **Unit testing is simplified when a component with high cohesion is designed.** When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

Integration Testing

1. Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.
2. The objective is to take unit-tested components and build a program structure that has been dictated by design. There is often a tendency to attempt non incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance.
3. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.
4. Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. In the paragraphs that follow, a number of different incremental integration strategies are discussed

Top-down integration

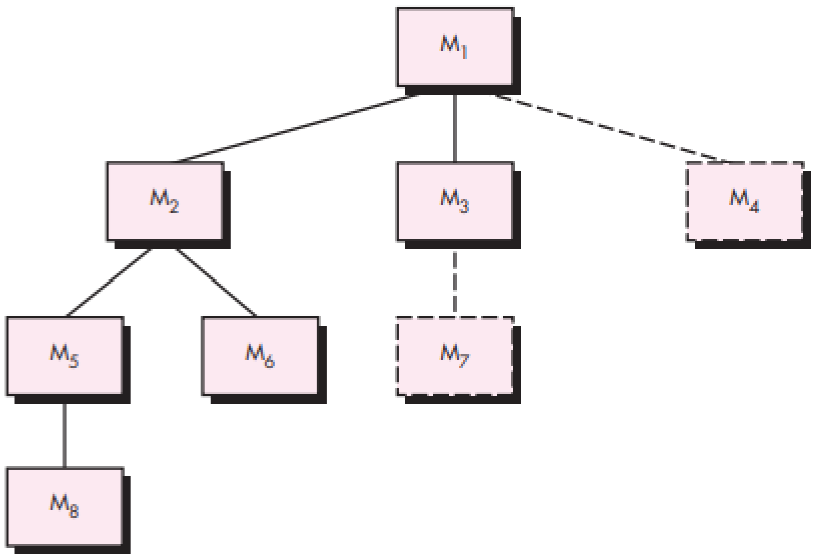


Figure: Top-down integration

1. Top-down integration testing is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either **a depth-first or breadth-first manner**.
2. **Referring to Figure,** depth-first integration integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics.
3. **For example,** selecting the left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built. Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows.
4. **The integration process is performed in a series of five steps:**
   1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
   2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
   3. Tests are conducted as each component is integrated.
   4. On completion of each set of tests, another stub is replaced with the real component.
   5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.

Bottom-up integration.

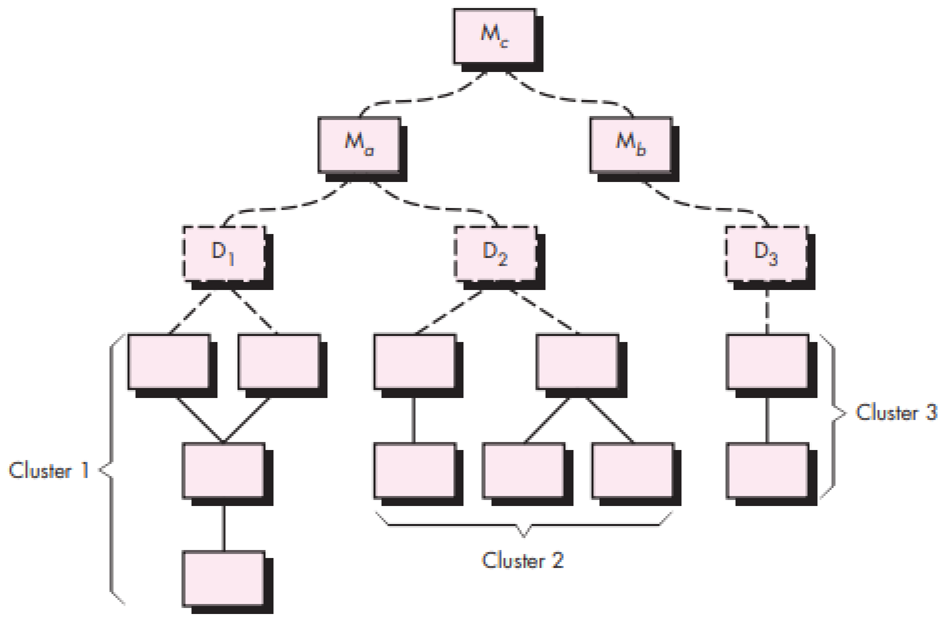


Figure: Bottom-up integration

1. Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules **(i.e., components at the lowest levels in the program structure).**
2. Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated.
3. **A bottom-up integration strategy may be implemented with the following steps:**
    - Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.
    - A driver (a control program for testing) is written to coordinate test case input and output.
    - The cluster is tested.
    - Drivers are removed and clusters are combined moving upward in the program structure.
4. **Integration follows the pattern illustrated in Figure.** Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to Ma.
5. **Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma.** Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both Ma and Mb will ultimately be integrated with component Mc, and so forth.

User acceptance testing (UAT)

- The UAT is conducted by the customer to ensure that the system satisfies the contractual acceptance criteria before being signed off as meeting user needs.
- Actual planning and execution of the acceptance tests do not have to be undertaken directly by the customer.
- Often third party consulting firms offer their services to do this task. However, the customer must specify the acceptance criteria for the third party to seek in the product.
- It concentrates on meeting the customer's functional/software/hardware requirements.
- User Acceptance Testing is also known as Beta Testing, application testing or end user testing.
- When performing UAT , there are seven basic steps to ensure the system is tested thoroughly and meets the business needs :
  1.Analyse Business Requirements
  2.Identify UAT scenarios
  3.Define the UAT Test Plan
  4.Create UAT Test Cases
  5.Run the Tests

6.Record the Results
7.Confirm Business Objectives are met.

- UAT is one of the final and critical software project procedures that must occur before newly developed software is rolled out to the market.
- A common UAT is a factory acceptance test (FAT) in the industrial sector which takes place before the installation of the concerned equipment.

5.6 Test Documentation- Test Case Template, Test plan, Introduction to defect report, Test Summary Report.

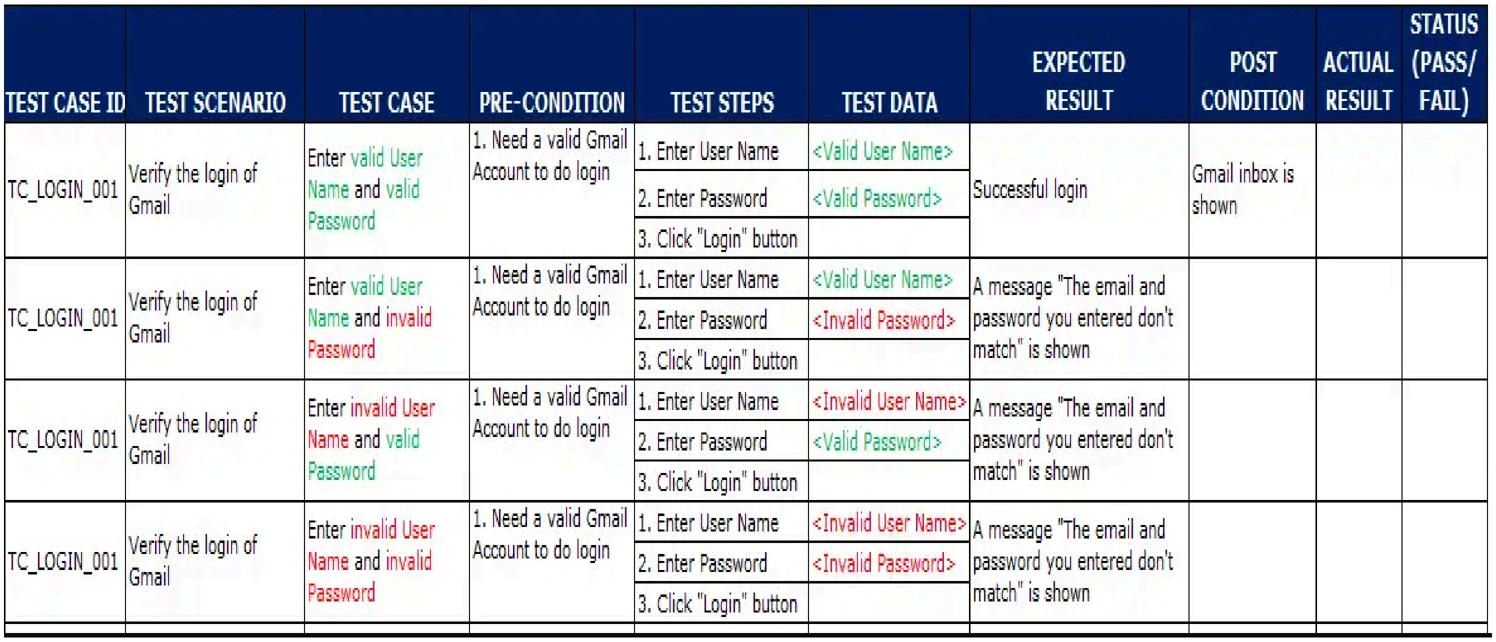
Test Case Template

- A test case is a document, which has a set of test data, preconditions, expected results and post conditions, developed for a particular test scenario in order to verify compliance against a specific requirement.
- Test Case acts as the starting point for the test execution, and after applying a set of input values, the application has a definitive outcome and leaves the system at some end point or also known as execution post condition.

Typical Test Case Parameters:

1. Test Case ID
2. Test Scenario
3. Test Case Description
4. Test Steps
5. Prerequisite
6. Test Data
7. Expected Result
8. Test Parameters
9. Actual Result
10. Environment Information
11. Comments

Example:



TEST CASE ID	TEST SCENARIO	TEST CASE	PRE-CONDITION	TEST STEPS	TEST DATA	EXPECTED RESULT	POST CONDITION	ACTUAL RESULT	STATUS (PASS/ FAIL)
TC_LOGIN_001	Verify the login of Gmail	Enter valid User Name and valid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Valid User Name>	Successful login	Gmail inbox is shown		
				2. Enter Password	<Valid Password>				
				3. Click "Login" button					
TC_LOGIN_001	Verify the login of Gmail	Enter valid User Name and invalid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Valid User Name>	A message "The email and password you entered don't match" is shown			
				2. Enter Password	<Invalid Password>				
				3. Click "Login" button					
TC_LOGIN_001	Verify the login of Gmail	Enter invalid User Name and valid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Invalid User Name>	A message "The email and password you entered don't match" is shown			
				2. Enter Password	<Valid Password>				
				3. Click "Login" button					
TC_LOGIN_001	Verify the login of Gmail	Enter invalid User Name and invalid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Invalid User Name>	A message "The email and password you entered don't match" is shown			
				2. Enter Password	<Invalid Password>				
				3. Click "Login" button					

Test plan
IEEE Std 829-1983 has described the test plan components. These components are listed below,

1. Test Plan Identifier
2. Introduction
3. Test item to be tested
4. Features to be tested
5. Features not to be tested
6. Approach
7. Item Pass/Fail Criteria
8. Suspension Criteria and Resumption require
9. Test deliverable
10. Testing tasks
11. Environmental needs
12. Responsibilities
13. Staffing and training needs
14. Scheduling
15. Risks and contingencies
16. Testing costs
17. Approvals

Test Plan Identifier

Each test plan is tagged with a unique identifier so that it is associated with a project.

Introduction

The test planner gives an overall description of the project with:

- Summary of the items and features to be tested
- Requirement and history of each item (optional)
- High-level description of testing goals
- References to related documents, such as project authorization, project plan, QA plan, configuration management plan, relevant policies, and relevant standards

Test item to be Tested

- Name, identifier, and version of test items
- Characteristics of their transmitting media where the items are stored, for example, disk, CD, etc.
- References to related documents, such as requirements specification, design specification, users guide, operations guide, installation guide
- References to bug reports related to test items
- Items which are specifically not going to be tested (optional)

Features to be Tested

This is a list of what needs to be tested from the user's viewpoint. The features may be interpreted in terms of functional and quality requirements.

- All software features and combinations of features are to be tested.
- References to test-design specifications associated with each feature and combination of features.

Features Not to be Tested

This is a list of what should 'not' be tested from both the user's viewpoint and the configuration management /version control view:

- All the features and the significant combinations of features, which will not be tested.
- Identify why the feature is not to be tested. There can be many reasons:

(i) Not to be included in this release of the software

(ii) Low-risk has been used before, and was considered stable

(iii) Will be released but not tested or documented as a functional part of the release of this version of the software

Approach

Let us discuss the overall approach to testing.

- For each major group of features or combinations of features, specify the approach.
- Specify major activities, techniques, and tools, which are to be used to test the groups.
- Specify the metrics to be collected.
- Specify the number of configurations to be tested.
- Specify a minimum degree of comprehensiveness required.
- Identify the techniques, which will be used to judge comprehensiveness.
- Specify any additional completion criteria.
- Specify techniques which are to be used to trace requirements.
- Identify significant constraints on testing, such as test-item availability, testing-resource availability, and deadline.

Item Pass/Fail Criteria

This component defines a set of criteria based on which a test case is passed or failed. The failure criteria is based on the severity levels of the defect. Thus, an acceptable severity level for the failures revealed by each test case is specified and used by the tester. If the severity level is beyond an acceptable limit, the software fails.

Suspension Criteria and Resumption Requirements

Suspension criteria specify the criteria to be used to suspend all or a portion of the testing activities, whereas resumption criteria specify when the testing can resume after it has been suspended.

For example, system integration testing in the integration environment can be suspended in the following circumstances:

- Unavailability of external dependent systems during execution.
- When a tester submits a 'critical' or 'major' defect, the testing team will call for a break in testing while an impact assessment is done.

System integration testing in the integration environment may be resumed under the following circumstances:

- When the "critical' or 'major' defect is resolved.
- When a fix is successfully implemented and the testing team is notified to continue testing.

Test Deliverables

- Identify deliverable documents: test plan, test design specifications, test case specifications, test item transmittal reports, test logs, test incident reports, test summary reports, and test harness (stubs and drivers).
- Identify test input and output data.

Testing Tasks

- Identify the tasks necessary to prepare for and perform testing.
- Identify all the task interdependencies. .- Identify any special skills required.

All testing-related tasks and their interdependencies can be shown through a work breakdown structure (WBS). WBS is a hierarchical or tree-like representation of all testing tasks that need to be completed in a project.

Environmental Needs

- Specify necessary and desired properties of the test environment: physical characteristics of the facilities including hardware, communications and system software, the mode of usage (i.e., stand-alone), and any other software or supplies needed.
- Specify the level of security required.
- Identify any special test tools needed.
- Identify any other testing need.
- Identify the source for all needs, which are currently not available.

Responsibilities

- Identify the groups responsible for managing, designing, preparing, executing, checking, and resolving.
- Identify the groups responsible for providing the test items identified in the test items section.
- Identify the groups responsible for providing the environmental needs identified in the environmental needs section.

Stating and Training Needs

- Specify staffing needs by skill level.
- Identify training options for providing necessary skills.

Scheduling

- Specify test milestones.
- Specify all item transmittal events.
- Estimate the time required to perform each testing task.
- Schedule all testing tasks and test milestones.
- For each testing resource, specify a period of use.

Risks and Contingencies

Specify the following overall risks to the project with an emphasis on the testing process:

- Lack of personnel when testing is to begin
- Lack of availability of required hardware, software, data, or tools.
- Late delivery of the software, hardware, or tools
- Delays in training on the application and/or tools
- Changes to the original requirements or designs

- Complexities involved in testing the applications

Specify the actions to be taken for various events. An example is given below.

Requirements definition will be complete by January 1, 20XX and, if the requirements change after that date, the following actions will be taken:

- The test schedule and the development schedule will move out an appropriate number of days.This rarely occurs, as most projects tend to have fixed delivery dates.
- The number of tests performed will be reduced.
- The number of acceptable defects will increase.
- These two items may lower the overall quality of the delivered product,
- Resources will be added to the team.
- The test team will work overtime.
- The scope of the plan may be changed.
- There may be some optimization of resources. This should be avoided, if possible, for obvious reasons.

Testing Costs

The IEEE standard has not included this component in its specification. However, it is a usual component of any test plan, as test costs are allocated in the total project plan. To estimate the costs testers will need tools and techniques. The following is a list of costs to be included:

- Cost of planning and designing the tests
- Cost of acquiring the hardware and software required for the tests
- Cost to support the environment
- Cost of executing the tests
- Cost of recording and analysing the test results
- Cost of training the testers, if any
- Cost of maintaining the test database

Approvals

- Specify the names and titles of all the people who must approve the plan.
- Provide space for signatures and dates.

Test Plan Format :

Sr.No.	Components	Value
1	Test Plan Identifier	
2	Introduction	
3	Test item to be tested	
4	Features to be tested	
5	Features not to be tested	
6	Approach	
7	Item Pass/Fail Criteria	
8	Suspension Criteria and Resumption require	

9	Test deliverable Testing tasks	
10	Environmental needs	
11	Responsibilities	
12	Staffing and training needs	
13	Scheduling	
14	Risks and contingencies	
15	Testing costs	
16	Approvals	

Introduction to defect report

- A defect report is a document that has concise details about what defects are identified, what action steps make the defects show up, and what are the expected results instead of the application showing error (defect) while taking particular step by step actions.
- Defect reports are usually created by the Quality Assurance team and also by the end-users (customers). Often customers detect more.

Defect has following attributes:

1) Defect ID: Identifies defect as there are many defects might identified in system. a. i.e. D1, D2, etc.

2) Defect Name: Name of defect which explains the defect in brief. a. It must be short but descriptive. i.e. Login error**.**

3) Project Name: Indicates project name in which defect is found

4) Module /Sub-module name: for which the defect is found.

5) Phase introduced: Phase of life cycle to which the defect belongs to.

6) Phase found: Phase of project when the defect is found is added here. It is used to find defect leakage or stage**.**

7) Defect type: Defines defect type. i.e. security defect, functional defect, GUI defect etc.

8) Severity: Declared in test plan, i.e. high medium or low.

9) Priority: defines on the basis of how the project decides a schedule to take the defects for fixing.

10) Summary: Describes short about the defect.

11) Description: Describes it in detail.

12) Status: dynamic field, open, assigned, resolved, closed, hold, deferred, or reopened, etc.

13) Reported by/ Reported on: Who found defect, and on what date.

14) Assigned to: The tester is being assigned to some testing team member.

DEFECT TEMPLATE: In most companies, a defect reporting tool is used and the elements of a report can vary. However, in general, a defect report can consist of the following elements.

i). Reporting a bug/defect properly is as important as finding a defect.

ii). If the defect found is not logged/reported correctly and clearly in bug tracking tools (like Bugzilla, Clear Quest etc.) then it won‘t be addressed properly by the developers, so it is very important to fill as much information as possible in the defect template so that it is very easy to understand the actual issue with the software.

Sample defect template

Attribute	**Value**
Abstract : **Platform :** **Test case Name :** **Release :** **Build Level :** **Client Machine IP/Hostname :** **Client OS :** **Server Machine IP/Hostname :** **Server OS :** **Defect Type :** **Priority :** **Severity :** **Developer Contacted :** **Test Contact Person :** **Attachments :** **Any Workaround :** **Steps to Reproduce 1. 2. 3.** **Expected Result:** **Actual Result:**	

Test Summary Report

- A report that summarizes the result of a test cycle is the test summary report.
- There are two types of test summary report:
    1. Phase wise test summary ,which is produced at the end of every phase
    2. Final test summary report. A Summary report should content:
    3. Test Summary report Identifier
    4. Description : Identify the test items being reported in this report with test id
    5. Variances: Mention any deviation from test plans, test procedures, if any.
    6. Summary of results: All the results are mentioned here with the resolved incidents and their solutions.
    7. Comprehensive assessment and recommendation for release should include Fit for release assessment and recommendation of release
- A test report is any descriptions, explanation or justification the status of a test project.
- A comprehensive test report is all of those things together.
- Test reporting is a means of achieving this communication. Types of reports that might be generated for a classical waterfall based project include:
    1. summary report for the phase;
    2. technical review report;
    3. walkthrough report;
    4. audit report.
    5. test report

Summary Report:

- The phase may be between five and ten pages in length. It would cover the major points drawn out from the testing.
- It is suggested that the report be structured with a single highlight sheet at the front based on a small commentary and a table that will show key areas where successful testing has been completed and areas where concern must be documented.
- The report should also address any issues that arose from external dependencies that may have affected the schedule.

Agile Test Summary Report Format

Test objective:	Explain what type of testing you executed and why.
Test cases/ execution details/ test coverage:	Explain the test suite and include when the test was executed, what type of test was executed, and where it is stored.
Defect status:	This section should include the following information about any bugs or defects discovered during testing: ● Total number of bugs discovered during testing ○ This data can help determine the

	overall quality of the product being tested. ● Status of bugs discovered during testing and links to issues or bug reports ○ This reports whether the bugs are still open, closed, or being addressed by the development and testing team. ● Breakdown by severity and priority ○ This information can help you better understand the significance of each bug and the level of attention it needs.
Platform and test environment configuration details:	Include information about the testing environment. If you share any test environment details regarding an application’s code, ensure you consider security and compliance.