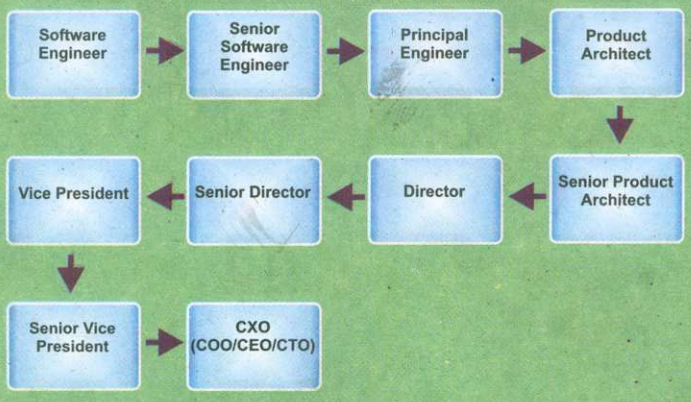
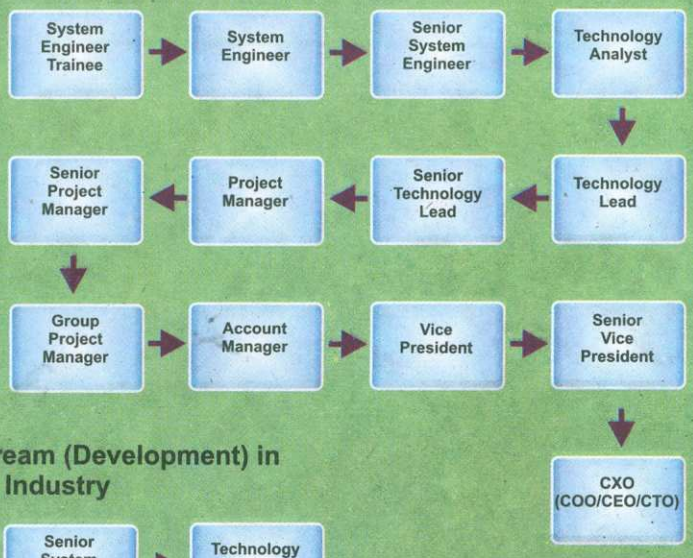
ignou
THE PEOPLE'S
UNIVERSITY
Indira Gandhi National Open University
School of Computer and Information Sciences

BCS-051
Introduction to Software Engineering

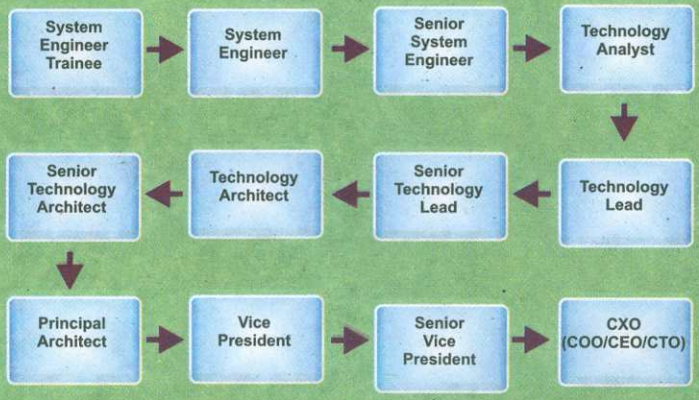
Roles for Technology Stream (Development) in IT Product Development Industry



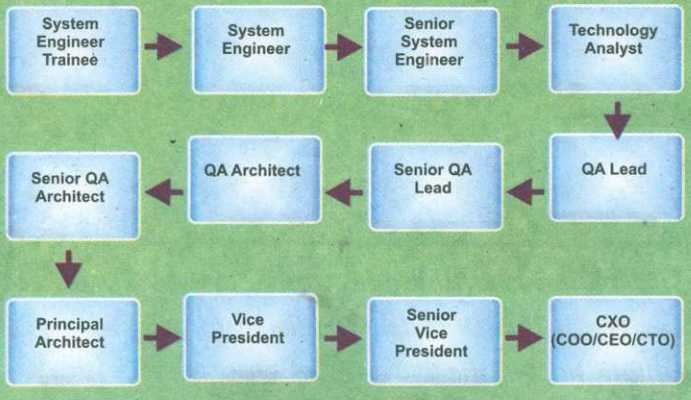
Roles for Project Management Stream (Development) in IT Services Industry



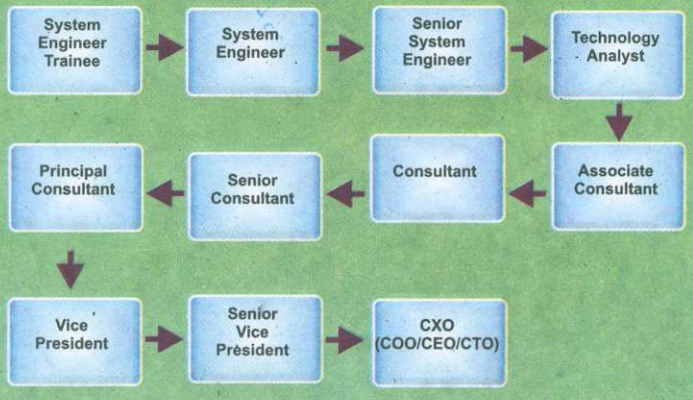
Roles for Technology Stream (Development) in IT Services Industry



Roles for QA Stream (Development) in IT Services Industry



Roles for Consulting Stream in IT Services Industry



Design and Testing

"शिक्षा मानव को बन्धनों से मुक्त करती है और आज के युग में तो यह लोकतंत्र की भावना का आधार भी है। जन्म तथा अन्य कारणों से उत्पन्न जाति एंव वर्तमान विषमताओं को दूर करते हुए मनुष्य को इन सबसे उपर उठाती है।"

– इन्दिरा गांधी

"Education is a liberating force, and in our age it is also a democratising force, cutting across the barriers of caste and class, smoothing out inequalities imposed by birth and other circumstances."

- Indira Gandhi

Indira Gandhi National Open University
School of Computer and Information Sciences

BCS-051
INTRODUCTION TO SOFTWARE ENGINEERING

Block

2

DESIGN AND TESTING

PROGRAMME DESIGN COMMITTEE

Prof. Manohar Lal
SOCIS, IGNOU, New Delhi

Prof. H. M. Gupta
Dept. of Elect. Engg.
IIT, Delhi

Prof. M. N. Doja, Dept. of CE
Jamia Millia, New Delhi

Prof. C. Pandurangan
Dept. of CSE, IIT, Chennai

Prof. I. Ramesh Babu
Dept. of CSE
Acharya Nagarjuna University
Nagarjuna Nagar (AP)

Prof. N. S. Gill
Dept. of CS, MDU, Rohtak

Prof. Arvind Kalia, Dept. of CS
HP University, Shimla

Prof. Anju Sahgal Gupta
SOH, IGNOU, New Delhi

Prof. Sujatha Verma
SOS, IGNOU, New Delhi

Prof. V. Sundarapandian
IITMK, Trivandrum

Prof. Dharamendra Kumar
Dept. of CS, GJU, Hissar

Prof. Vikram Singh
Dept. of CS, CDLU, Sirsa

Sh. Shashi Bhushan
Associate. Prof.
SOCIS, IGNOU, New Delhi

Sh. Akshay Kumar
Associate Prof.
SOCIS, IGNOU, New Delhi

Dr. P. K. Mishra, Associate Prof.
Dept. of CS, BHU, Varanasi

Dr. P. V. Suresh, Associate Prof.
SOCIS, IGNOU, New Delhi

Sh. V. V. Subrahmanyam
Associate Prof.
SOCIS, IGNOU, New Delhi

Sh. M. P. Mishra, Asst. Prof.
SOCIS, IGNOU, New Delhi

Dr. Naveen Kumar, Reader
SOCIS, IGNOU, New Delhi

Dr. Subodh Kesharwani, Asst. Prof.
SOMS, IGNOU, New Delhi

COURSE CURRICULUM DESIGN COMMITTEE

Sh. Shashi Bhushan
SOCIS, IGNOU, New Delhi

Dr. P. V. Suresh
SOCIS, IGNOU, New Delhi

Sh. V. V. Subrahmanyam
SOCIS, IGNOU, New Delhi

Dr. Naveen Kumar
SOCIS, IGNOU, New Delhi

Prof. A. K.Tripathi

Dept. of CSE, IIT(BHU),
Varanasi

Dr. S. R. N. Reddy

Dept. of CSE

IGDTUW

New Delhi

Sh. Akshay Kumar
SOCIS, IGNOU, New Delhi

Prof. Manu Sood,

Dept. of CS

HP University
Shimla

Sh. M.P. Mishra
SOCIS, IGNOU, New Delhi

SOCIS FACULTY

Sh. Shashi Bhushan, Director

Sh. Akshay Kumar
Associate Professor

Sh. M. P. Mishra
Asst. Professor

Prof. Manohar Lal

Sh. V. V. Subrahmanyam
Associate Professor

Dr. Sudhansh Sharma
Asst. Professor

Dr. P. V. Suresh
Associate Professor

Dr. Naveen Kumar
Reader

PREPARATION TEAM

Content Editor:

Dr. S. R. N. Reddy

Dept. of CSE

IGDTUW

New Delhi

Language Editor:

Dr. Pema Eden Samdup
SOH, IGNOU, New Delhi

Block Writers:

Units 1, 2 and 4

Sh. Shailesh Kumar Shiva Kumar, Research Scholar (SOCIS, IGNOU) and Senior Technology Architect (Infosys Technologies)

Unit 3

Ms. Neetu Jain

Research Scholar (SOCIS, IGNOU) and Faculty (Amity University)

Course Coordinator: Dr. P. V. Suresh, SOCIS, IGNOU, New Delhi

Print Production

Mr. S. Burman	Mr. Tilak Raj	Mr. Yash Pal
Dy. Registrar (Publication)	Asstt. Registrar (Publication)	Section Officer (Publication)
MPDD, IGNOU, New Delhi	MPPD, IGNOU, New Delhi	MPDD, IGNOU New Delhi

January 2018 (Reprint)

ISBN-978-81-266-6611-9

Further information about the Indira Gandhi National Open University courses may be obtained from the University's office at Maidan Garhi, New Delhi-110 068.

Printed and published on behalf of the Indira Gandhi National Open University, by Registrar MPDD, IGNOU, New Delhi

Printed at: Ankur Offset Pvt. Ltd., A-54, Sector-63, Noida-201307 (U.P.)

BLOCK INTRODUCTION

This block introduces learner to Function Oriented Design (FOD), Object Oriented Design (OOD), Testing Techniques and Development and Execution of Test Cases.

A typical procedural software system consists of multiple functions that share the same state and interact. We will look at the various methods for designing a modular function, decomposing the function into reusable and loosely coupled sub functions and its visual representation. We will also examine the process and best practices for identification and development of functions. Function oriented design essentially deals with creating functions to convert input into the desired output. Function, Function state and Sub-Function are the key elements of FOD.

Software design often mimics real-world objects and designs. Many software design principles, design patterns and techniques are inspired usually by nature. Software architects find it natural to model software components based on their real-world counterparts. The role of the architects and designers are mainly involved in marrying the objects in the software world with the objects in the real-world. This is also done in part to make the end-user accustomed to the software and make their transition from the real-world to virtual work easier and friendlier. Complex modeling, Abstraction, Hierarchy Modeling, Reusability, Extensibility and Flexibility are some of the motivating factors for having the design using Object-Oriented concepts.

Software testing is a very important phase in Software Development Life Cycle (SDLC). It is a repetitive process that helps identify not only the errors in the developed application but also corrects those errors. One may think that finding errors is a simple task. Then why would we need to pay importance to testing. In this unit, we will discuss the different aspects and importance of testing. Even though testing may seem easy, it is not an easy task and needs a strategic and systemised approach to ensure suitability.

Preparation of test cases is a very important step for effective testing of any software. A testing team which is different from the development team will test the software. One of the base document for preparing test cases is SRS. It can never be claimed that the software is error free as any software can be claimed as error free software only when the software successfully runs until it is withdrawn due to reasons that are not pertaining to itself.

This block consists of four units and is organized as follows:

Unit 1 deals with FOD. It covers concepts as well as other issues associated with FOD such as Cohesion, Coupling, etc.

Unit 2 deals with OOD. It covers concepts as well as other issues associated with problem domain static objects, application logic objects, etc.

Unit 3 deals with Software Testing Techniques. It covers various testing techniques such as White Box Testing, Black Box Testing, etc.

Unit 4 deals with Development and Execution of Test Cases. It covers topics such as test case preparation and their execution.

UNIT 1 FUNCTION ORIENTED DESIGN

Structure

1.0 INTRODUCTION

This unit details the concepts of function oriented design and the modeling techniques for the same. A typical procedural software system consists of multiple functions that share the same state and interact. We will look at the various methods for designing a modular function, decomposing the function into reusable and loosely coupled sub functions and its visual representation. We will also examine the process and best practices for identification and development of functions

Function oriented design essentially deals with creating functions to convert input into the desired output.

The following are the key elements of function oriented design:

- **Function:** It is a unit of code consisting of a sequence of instructions required to perform a specific task. It is often referred to as method or routine in different programming languages.
- **Function state:** It's the data on which a function operates. For example a *getEmployeeDetails* function would require the employee state stored in the database. In a function oriented design, the state is centrally stored and is accessible to all functions.
- **Sub Function:** The main function can be further broken down into smaller re-usable units of code or sub-functions or sub-routines. For instance *getEmployeeDetails* main function can further invoke *getEmployeeContactDetails* and *getEmployeePersonalDetails* sub-routines.

The following are the key features of the function oriented design:

- **Top down decomposition:** A software system essentially consists of a set of functions, each of which performs a unique functionality. Once the high level functions are identified, each function is further decomposed into

modular sub-functions. For instance, a high level function such as, "Update Person" updates the persons details entirely, can further be broken down into:

- **UpdatePersonPersonalDetails** to update the personal details of a person such as, name, age, height, date of birth, etc.
- **UpdatePersonAddressDetails** to update the address details of the person.
- **UpdatePersonContactDetails** to update the contact details of the person such as, phone, email, etc.

Function decomposition needs to be performed till we reach an independent and modular function with a single responsibility.

- **Function association:** Multiple functions are often related for interaction and data sharing. For instance, in the above example, the main function "Update Person" invokes sub routines such as, UpdatePersonPersonalDetails with person Id which can uniquely identify the person.
- **Function state sharing:** The state of the system is shared among various functions. For instance, in the above example, the state of person's details is shared among all the functions including UpdatePersonPersonalDetails, UpdatePersonAddressDetails and UpdatePersonContactDetails.

The following table explains the concepts associated with Function Oriented Design:

Concept	Function oriented design
Modeling	Functions are modeled on distinct and independent functionality. For example, in the banking system the key functions include transferFunds, checkBalance, etc.
Abstraction	Functions are usually abstracted as verbs such as, subtractBalance, getAmount, etc.
State	The state is normally centralised and functions share and access the state. For example a Persons details are stored centrally in a file or in database and multiple functions such as, getPersonDetails, UpdatePeronAddress access the centralised state
Hierarchy	Functions are grouped based on their functionality. Multiple sub functions can be grouped to form a bigger function

1.1 OBJECTIVES

After going through this unit, you should be able to

- understand the general principles of solving a problem using a function oriented design,
- identify and apply various decomposition techniques,
- model the function in visual fashion using commonly employed visual notations, and
- learn the best practices of function oriented design.

1.2 CONSTRUCTING A SOLUTION TO A PROBLEM

Let us examine a few of the commonly used methods in function oriented design for designing a solution to a problem.

1.2.1 Solution Design Principles

Before looking at the actual methods, we will first look at the driving principles on which these methods are based. This will help us to appreciate the principles of design better:

- **Top down decomposition:** When we adopt this design principle, we iteratively and sometimes recursively decompose the problem from the top. This is often a top-down approach for creation of a solution to a problem. In function oriented design, we break the main function into smaller sub-functions which are both modular and reusable.
- **Divide and conquer:** Using this design, we break down the problem functionally into smaller sub-problems. Each sub-problem is then independently solved. For instance, a simple binary search function would first break the problem space into two equal sub-problems and then solve each of those sub-problems individually.
- **Hierarchical modeling:** The problem is visualised through the hierarchy of its composite elements. After this, a function will be designed for each of the elements in the hierarchy.
- **Modular:** A modular system can be constructed by ensuring that each of its component modules is loosely coupled with minimal dependencies so, as to allow, the flexibility to independently change the components.
- **Abstraction modeling:** The design states that each functional module should provide a complete abstraction of the service it performs. The functional module should provide a structured interface for its consumers. For instance, a checking account functional module should perform all functions related to checking a given account.

1.2.2 Solution Design Methods in Function Oriented Design

Let us look at a sample problem domain and apply the steps mentioned above. Here is the sample scenario of a library system:

Structured Analysis (SA) and Structured Design (SD)

- **Structured Analysis:** During the Structured analysis stage, the functional decomposition takes place. Each high level function is carefully analysed and it is broken down into smaller functional modules. In this stage, the problem description is also visually depicted in a "Data flow diagram". This provides the detailed and visual structure of the system with all the components and the sub-components.

  The following are the utilities of SA:

  - Provides a visual representation of the problem.
  - Helps the users and stakeholders to review the design and the hierarchy.
  - Helps in verifying the completeness of the problem domain elements.

- **Structured design is** the next step, wherein, we take the functional modules identified in the structured analysis phase to map to program specific structures that can be implemented.

  The following is the utility of SD:

  - This helps in the implementation of the functional module in a given programming language.

Structural Decomposition

Structural decomposition aims at designing a function based structure that is highly cohesive and loosely coupled. We will look at cohesiveness and coupling when we look at the characteristics of a good function oriented design.

The process of converting DFD elements into structure chart involves three broad steps:

- Start with elements in DFD that are functional processing units. Group all those elements within a single function in a structure diagram.
- Create input function by grouping all DFD elements related to input such as reading, service invocation, etc.
- Create output function by grouping all DFD elements related to output such as file writing, page rendering, etc.

Refactor the high level functions into further low-level reusable sub-functions and add the inputs and outputs accordingly.

Detailed Design

In this design methodology, detailed design specifications would be developed for each of the functions to come up with the architecture of the system. These specifications can later be used for the actual implementation in a given programming language.

The following are the main elements of a specification document:

- Detailed problem specification
- Key data types
- Specification for all functional modules
- Known constraints
- Significant architecture and design decisions

☞ **Check Your Progress 1**

(1) In function oriented design the state is stored .....

(2) ..... design principle involves breaking down a single problem into smaller sub-problems.

(3) Data flow diagram is often designed in the ..... stage.

1.3 IDENTIFYING COMPONENTS AND THEIR INTERACTION

In this section, we will look at the process of component identification and their interaction.

1.3.1 Identification Process

1. **Functionality modeling:** Component identification always starts by decomposing the business domain problem structure into a well-defined visual model. Data flow diagram is one of the most popular visual models to depict function oriented design.
2. **Design module development:** The next step is to convert individual elements in the DFD into a design module. During this process, the design modules need to be designed to ensure high cohesion and loose inter-module coupling. The modules that perform similar kind of functionalities and processes qualify for the main design module. Design modules would then become key function component.
3. **Sub function development:** The main functions need to be broken down into sub functions. Utilities such as, data validation, data conversion, information logging would be good candidates for sub-functions.
4. **Interaction modeling:** The input and output elements from DFD can be used to design the interactions between functions.

1.3.2 Component and Stage Mapping

The following table indicates the component/association identified at each level of the process:

Stage	Component/Association
Functional modeling	Data flow diagram
Design module development	High level function
Sub function development	Sub functions
Interaction modeling	Input and output data

1.4 VISUALIZING THE SOLUTION

In this section, we will examine the visual modeling of function oriented design.

1.4.1 Context Diagram

A context diagram is the visual modeling of the highest level of abstraction. It provides a high level context of the overall system, the first level input data and the final output data. It serves as the first step to creating a detailed DFD.

Development of a Context Diagram

Let us consider a simple example of a system that finds the largest number in a given set of data elements:

- User provides the list of numerical values.
- Sorting system processes the list of elements provided.
- The sorting system provides the highest numerical value from the provided list.

The above scenario is depicted in Figure 1.1:

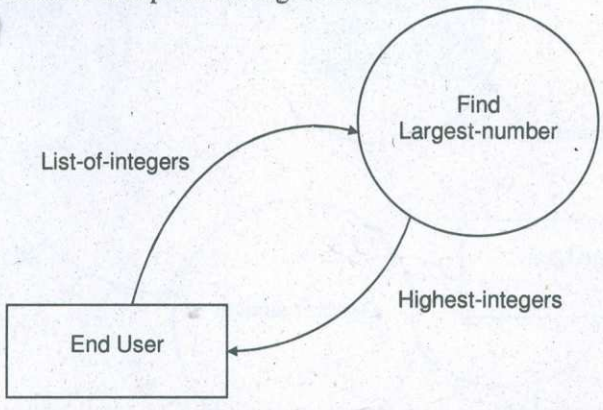


Figure 1.1 : Context Level DFD (Find Largest Number)

1.4.2 Data Flow Diagram (DFD)

DFD is created at the end of the structural analysis stage. It mainly depicts:

- The main functions of the system.
- The interaction of data within the system.

The diagram would depict the input and output data of a function.

Elements of DFD

The following are the key visual elements of the DFD:

- **Rectangle** to represent physical entities such as, Computer, Building, etc.

- **Circle** to depict function such as, *updateEmployee*.
- **Directed line** to show data flow.
- **Horizontal parallel lines** to depict the data structure or data store such as, File System or Database or ERP system.

Process Steps for Developing the DFD

(1) Start with the system specifications document. Identify each high level function and model it as a circle in DFD.

(2) Identify all the input data to this function and depict it in the DFD.

(3) Identify all the output data to this function and depict it in the DFD.

Further, refining can be done by structurally decomposing the high-level function circles into smaller modular sub-functions till each sub-function represents a distinct reusable task implementation.

Example of a DFD

Let us take an example for "finding largest number" scenario given for the contextual diagram. In the context diagram "Find largest number" is a high level abstract for the function which calculates the largest number. However, internally this function performs two more tasks:

(1) It collects the input and places it in a numerical list.

(2) It sorts the list in ascending order.

(3) It returns the highest number.

The DFD for this is given in Figure 1.2:

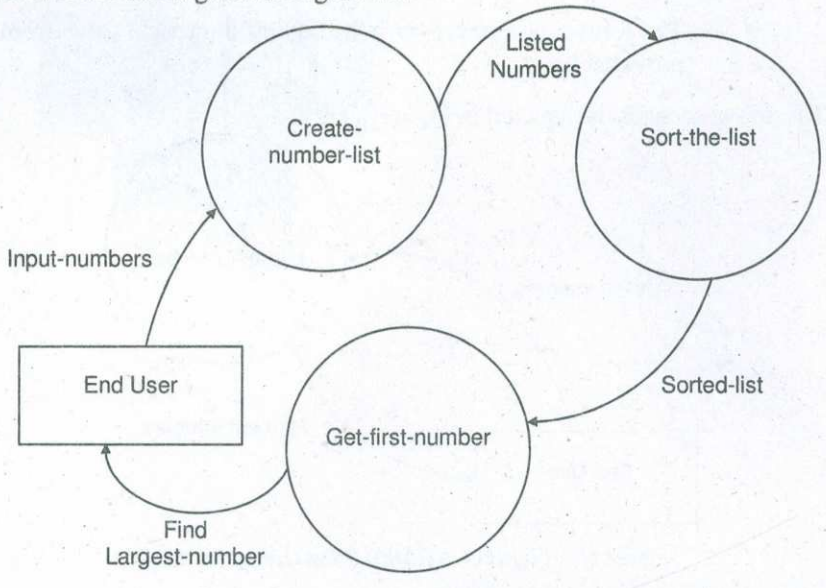


Figure 1.2 : Level-1 DFD (Find Largest Number)

1.5 CHARACTERISTICS OF A GOOD, FUNCTION ORIENTED DESIGN

This section describes some of the characteristics of a good, function oriented design.

1.5.1 High Level Characteristics

The following are the high level characteristics of a good function oriented design:

Functional Independence

The modeled functions should be independent in terms of the tasks they perform. This would help in

- **Reusability**: The function can be more reusable if it performs a single task.
- **Maintainability**: Maintainability is increased due to single task execution and loose coupling.
- **Troubleshooting**: Helps in quicker and easier debugging in case of exception scenarios.
- **Understandability**: The function is more easily understood.

Adherence to Key Design Criteria

The functions should adhere to other important design criteria:

- **Completeness**: The functions should implement all the requirements specified in the requirements document completely.
- **Correctness**: The functions should implement all the requirements as per their specifications correctly.
- **Efficiency**: The functions should efficiently use resources such as, database connection, file connection, etc.
- **Cost**: The functions should aim to reduce the overall cost in terms of maintainability and extensibility.

1.5.2 Module Level Characteristics

In this section, we shall look at the main characteristics that are to be possessed by for a good design.

Cohesion

Cohesion is the measure of how well the internal elements of a module or a function are connected to each other. In other words, the function should perform only a single tasks. The logic and variables present in the function should be to perform a single activity only, in the most optimal fashion. This enables a strong internal relationship between the internal elements of a function.

Types of Cohesion

The following are the different types of Cohesion:

- **Coincidental cohesion:** In this type of cohesion, the internal functions in a given module are loosely correlated.
- **Communication cohesion:** In this type of cohesion, the functions within the module update the same data type.
- **Sequential cohesion:** The functions of a module are said to have sequential cohesion if individual functions of a module form a sequence for executing a given functionality. For instance, the output of the first function is fed as input for the second function.
- **Functional cohesion:** This is the strongest type of cohesion wherein, all functions within a module are designed to achieve a single functionality.

Examples of Cohesion

The following is a function with weak cohesion.

```
Function updateEmployee (int empId, String newname) {

    Logger log = new Logger();

    Connection con = new Connection ();

    Statement stmt = new Statement();

    Stmt.executeQuery ("Update employee set name = "||newname||" where
    empno = "||empId);

    Log.log("Statement executed successfully"); }
```

The following is a function with strong cohesion:

```
Function Connection createConnection () {

    If (con == null)

     con = new Connection ();

    return con;}
```

The above function performs a single task of creating a connection. The internal variables and logic is completely concerned with opening a connection. Strong internal cohesion is always considered a best practice as it helps in re-usability and maintainability.

Coupling

Coupling indicates the dependencies across different modules. If, a module is dependent on multiple functions in another module, then it is known as strong coupling; lesser number of dependencies indicate loose coupling. A module with loose coupling on another module also has strong cohesion among its internal functions wherein, internal functions co-ordinate to implement the module behavior.

The design thumb rule is, to have loose coupling so that, each of the individual modules and its internal structure can be changed with greater impact on other modules.

Types of Coupling

- **Content coupling:** In this type of coupling, one module is dependent and updates the internal state of another module. This is a very tight form of coupling.
- **Common coupling:** If modules share the same global data, it is known as common coupling. For instance, all modules acting on a common shared persistent store for their functionality, it is an example of common coupling.
- **Control coupling:** If a function argument passed from the first module controls the logic and order of instructions in another module, for instance, it is an illustration of a control coupling if we pass control flags and switches from one module to function in another module through which the sequence of steps and branching can be varied, it forms control coupling.

- **Data coupling:** If two modules are coupled by a function parameter it is said to be data coupling. This is an example of loose coupling. For instance, a function call through the specified argument forms a standard data coupling.

Connascence

Two modules are said to be in connascence if a change in the first module requires a mandatory change in the second module for ensuring overall functionality. This is also a form of a very tightly coupled modules.

☞ **Check Your Progress 2**

(1) ..... is used to depict external system in a data flow diagram.

(2) ..... is the main difference between a Context diagram and a Data flow diagram.

(3) ..... characteristic determines the degree of strong connectivity internally among functions within a module.

(4) A good design characteristic is to have ..... cohesion and ..... coupling.

1.6 SUMMARY

In this unit, we started by looking at the basic elements and some of the key features of a function oriented design. Then, we moved on to check various design principles to designing a solution for a given problem using techniques such as structured analysis, and structured decomposition. We also checked the process to identify various components and its associations. In the next section, we looked at various visual elements such as context diagram and data flow diagram for visually depicting the function oriented design. Finally, we talked about the characteristics of a good, function oriented design including cohesion and coupling.

1.7 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

(1) Centrally

(2) Divide and conquer

(3) Structured Analysis

Check Your Progress 2

(1) Rectangle

(2) Level of abstraction

(3) Cohesion

(4) high , loose

1.8 FURTHER READINGS

Reference Books

Peretz Shoval: *Functional and Object Oriented Analysis and Design: An Integrated Methodology.*

Reference Websites

http://en.wikipedia.org/wiki/Coupling_(computer_programming)

http://en.wikipedia.org/wiki/Cohesion_(computer_science)

UNIT 2 OBJECT ORIENTED DESIGN

Structure

2.0 INTRODUCTION

This Unit explains the design principles related to object oriented design. We will look at some of the key object oriented design concepts and later examine the detailed process of specifying and identifying problem domain static objects.

Software design often mimics real-world objects and designs. Many software design principles, design patterns and techniques are inspired usually by nature. Software architects find it natural to model software components based on their real-world counterparts. The role of the architects and designers are mainly involved in marrying the objects in the software world with the objects in the real-world. This is also done in part to make the end-user accustomed to the software and make their transition from the real-world to virtual work easier and friendlier. For instance, a typical grocery shopping involves grocery items, shopping cart and purchase. If this process is modelled in the software world, it would involve almost similar objects including ShoppingCart object and checkout process.

The process of modeling also involves identification of various properties of the objects, their association, data flow, etc.

The following are the key motivating factors for having the design using object-oriented concepts:

- **Complexity modeling:** Object-oriented design is best suited to model the complex real-world systems and scenarios along with its properties,

interactions and associations. For instance, a complex banking system can be best modeled by breaking it down to its component objects such as, Account, Transfer and its associations.

- **Abstraction:** Object-oriented design provides clear abstraction of things at various levels. For instance, the objects in the business layer can focus primarily on business logic whereas, the objects in the data services layer can focus on persistence which provide "separation of concern".
- **Contract specification:** Object-oriented design also allows us to specify the clear contract for an object via interfaces. All implementations of this interface must adhere to the contract enabling controlled extension. For instance, the behaviour of a product can be defined in an interface. All product variants that implement the interface would provide the variant specific implementation of the behaviour methods.
- **Hierarchy modeling:** The hierarchical relationship between real-world objects can be modeled using relationships between objects.
- **Reusability, extensibility and flexibility:** Object-oriented design helps us in re-using business logic and provides flexible and extensible design to accommodate future changes.

The following are the prominent object-oriented concepts:

- **Object:** An object is often an instance of a class with its own "state". It is a logic unit consisting of its own properties and methods that interact with the internal variables.
- **Information abstraction**: Each object has a strict policy to hide its internal state for private variables and provides public methods for accessing data.
- **Inheritance**: This feature enables the classes/objects to extend their super classes and inherit protected variables and methods and also provide the ability to override the methods to provide specific implementation wherever required. This would be very useful in modelling the object hierarchy and provide extensible design.
- **Polymorphism:** This feature provides an object that is to be replaced by its sub-objects thereby altering the behavior suitable for the context.
- **Interface specification:** Interface specifies the behavior signature and defers the implementation. The implementers should provide specific implementation for the object.

Let's look at some of the core features of the class which are instantiated by objects:

- **Attributes**: These are the properties of the class. During instantiation of the class, an object provides specific values for attributes and this is often referred to as "state" of the object. In Figure 2.1, deptNo, deptName are the attributes of Dept class and empNo, empName, and empDetails are the attributes of Employee class.
- **Methods**: Classes permit public accessor methods to act on its variables. In the example given below getDepNo(), getDeptName() are public methods that provide the value of the deptNo and deptName variable values respectively.
- **Associations**: Associations are depicted through relationships between the classes. In the following diagram, "One Dept has many employees" is denoted by "one-to-many" relationship between the classes.

When a class is instantiated by objects, each object has its own value for all the variables.

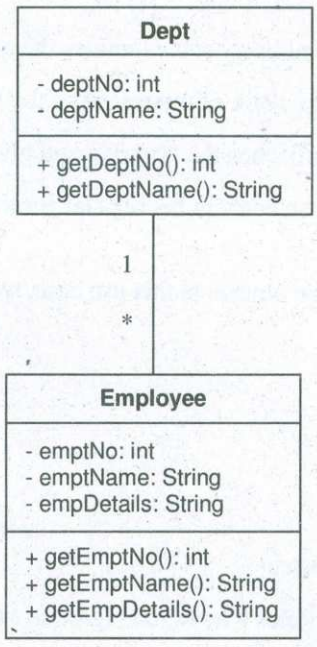


Figure 2.1 : Department and Employee Classes

2.1 OBJECTIVES

After going through this unit, you should be able to

- understand generic object oriented principles,
- know generic guidelines and thumb rules for specifying and identifying problem domain static objects,
- identify and design of application/business logic objects,
- identify and design of utility objects, and
- know generic guidelines and best practices for identification and design of objects.

2.2 IDENTIFICATION OF PROBLEM DOMAIN STATIC OBJECTS

Identification of objects constitutes the first step in object-oriented design. Modelling a real world problem into objects is one of the key activities in the design phase.

2.2.1 Static Objects

Before we start looking at steps to identifying static objects, let's understand the meaning of static objects. In object-oriented design, classes and their instances, objects are categorized as "static" because they represent time-invariant view of the system. For instance, let's consider an object Person with personId and personName as its attributes. The values of these two attributes would not change with time once it is instantiated. In other words, the object does not undergo a time-dependent change. However, a timing diagram or a state chart diagram would represent a dynamic view representing the transition along with time.

2.2.1 Problem Domain – Static Object Mapping Deep Dive

Let us look at a sample problem domain and apply the steps mentioned above. Here is the sample scenario of a library management system:

Book checkout Use case

1. Student enters the library and examines the available books.
2. Student selects the book of interest from the library rack.
3. Student then approaches the librarian and gives the book.
4. Librarian makes an entry in his register for the book against student and issues the book.

If we were to identify the static objects in this use case, we can identify these objects:

- Student
- Library
- Book
- Librarian
- TransactionRegistry

Though "Library rack" constitutes a noun, it is ignored as it is irrelevant for the core use case.

☞ **Check Your Progress 1**

(1) The key features of a class are .....

(2) Contract and behavior specification are given by .....

2.3 SPECIFICATION OF PROBLEM DOMAIN STATIC OBJECTS

In this section, we will see how we can specify a static object for a problem domain.

2.3.1 Specification of a Static Object

A specification essentially provides a complete definition for an object including its purpose, field/operation descriptions so that the user can understand and use the object as expected.

The following are the specifications for a class:

- **Purpose/Responsibilities:** This would specify the main purpose or concern addressed by the class. This is often provided as class-level and method-level documentation.
- **Attributes:** This provides details of attributes including their name, data type and optional initial value.
- **Operations:** This provides details of functions including the return type, arguments.
- **Constraints:** This provides list of constraints of the class.

When we have multiple classes involved, we will also specify the association between them. Association is the relationship between the class wherein, we specify the "is a", "has" relationships.

The following is an example of the specification of the class:

```
/*
This class checks if the given number is a prime number or not.
Known constraints: The class has method which would only accept numerical values
```

```
*/
Public class PrimeNumberChecker {
//The class level variable to hold the input value for processing
Private int inputNumber;

/*
The method takes an integer value and checks if the number is a prime number or not
Returns true if number is prime else returns false
*/
Public Boolean  isPrime (int no) {
If  (no==null || !isNumber(no)) return false;
Return checkPrime(no);
}

}
```

2.3.2 Specifications of Static Objects for a Sample Problem Domain

Now we will see the class specifications for an e-commerce use case.

Let us consider this primary use case for an e-commerce application:

1. The application provides a list of items to the user after search.
2. Each item has details such as, name, value and the quantity which user needs.
3. The user can select the items and add it to his/her shopping cart.
4. Once the shopping is complete, the user can proceed to the checkout

Let's consider two classes for modelling: ShoppingCart and CartItem. Figure 2.2 provides specification for both the classes including the attribute and operation definition and depicts the relationship between them. The relationship specified is "one-to-many" to indicate that one shopping cart can contain multiple cart items.

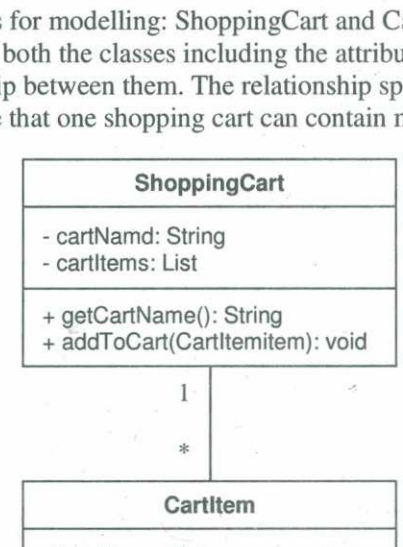


Figure 2.2 : Shopping Cart and Cart Item Classes

2.4 APPLICATION LOGIC OBJECTS

In a typical multi-tiered architecture, the application logic or the business logic will be in the business application layer. The layered architecture provides "separation of concerns" so that objects in individual layers can be independently modified and extended due to loose coupling.

The business application layer consists of a number of objects such as, Business Objects (BOs), Data Access Objects (DAOs), Service façade, delegates, bean objects, etc. Normally, the application logic would be part of business objects.

2.4.1 Motivation of Application Logic Objects

The following are some of the key motivating factors for creating separate application logic objects:

- Centralised access to application logic,
- Loose coupling and increased testability,
- Layered separation of application logic concern, and
- Reusability of application logic.

2.4.2 Identification of Application Logic Objects

In a use case, all objects that consist of core application logic are potential candidates for application logic objects. We can summarize the basic qualities for application logic objects as follows:

- Owner of core business logic,
- Interacts with other application logic objects, and
- Used in application transaction.

2.4.3 Sample Application Logic Objects

Let us consider a banking scenario with the following use case:

Given below is a simple example of a Use case for opening an account:

(1) The user can open an account by providing the details required for opening an account.

(2) An account could be a savings account or a checking account.

(3) A savings account should contain a minimum balance of 10,000 Rupees per quarter. If this is not satisfied then a fine of Rs. 500 will be deducted.

(4) Each savings account is linked to only one credit card.

(5) A credit card's credit limit is based on the user's balance in the savings account for an entire year.

(6) If the user fails to pay the minimum amount due on the credit card, then the credit card is authorised to automatically debit the amount from the linked savings account. The credit card will be automatically blocked if the minimum amount is not present in the savings account.

In the above use case, there are multiple business rules that help us identify the application logic objects. Applying the thumb rules mentioned in the previous section, we can identify the following application logic objects:

(1) Account

(2) SavingsAccount

(3) CurrentAccount

(4) CreditCard

Each application logic object has corresponding functions for processing business rules. A sample representation of the class diagram is given in Figure 2.3.

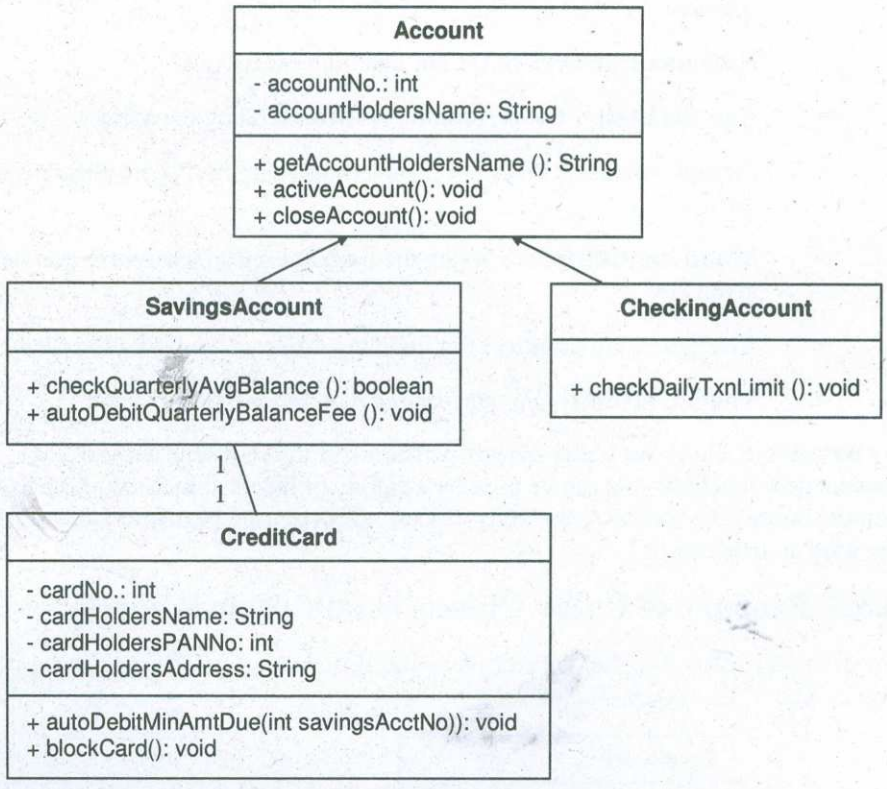


Figure 2.3 : Class Diagram

Methods such as, checkQuarterlyAvgBalance(), autoDebitQuarterlyBalance() contain core business logic that processes business rules.

2.5 IDENTIFICATION OF NECESSARY UTILITY OBJECTS

Utility objects are often provided in a framework to address specific utility functions. They are often used as "helpers" by the rest of the objects.

2.5.1 Criteria for Identifying Utility Objects

The following are the key criteria for identifying utility objects:

- The object should act as a helper to other framework classes.
- The object should provide a re-usable context-independent utility throughout the system.
- The utility provided by the object should be used across various layers.

2.5.2 Common Utility Objects

The following are the list of utility objects generally used in a typical enterprise application:

- **CachingUtility:** For handling the caching requirements
- **LoggingHelper:** For logging info, debug and error statements
- **FileReaderUtility:** For reading the required files
- **ResourceLocator:** For providing the URL for a given resource

- **BuildUtility:** For handling build activities
- **EncryptionUtility:** For encrypting/decrypting the values as per enterprise standards
- **ExceptionHandlerUtility:** For handling exceptions
- **EncoderUtility:** For performing HTML encoding/decoding
- **StringConverterUtility:** For converting string into appropriate display formats
- **MultiLangUtility:** For geting the language specific resource bundle for a given key
- **ConfigurationHandler:** For handling different configuration files
- **ValidationUtility:** For performing required field validations

As we can see, the above utility objects perform specific re-usable and context independent functions that can be used by a variety of layers and classes. Due to this context insensitive nature of the utility objects, they are used mostly as static objects and are used as singletons.

2.5.3 Example of Utility Objects in a Problem Scenario

The following class diagram indicates how utility classes are used by other domain objects and static objects (Figure 2.4):

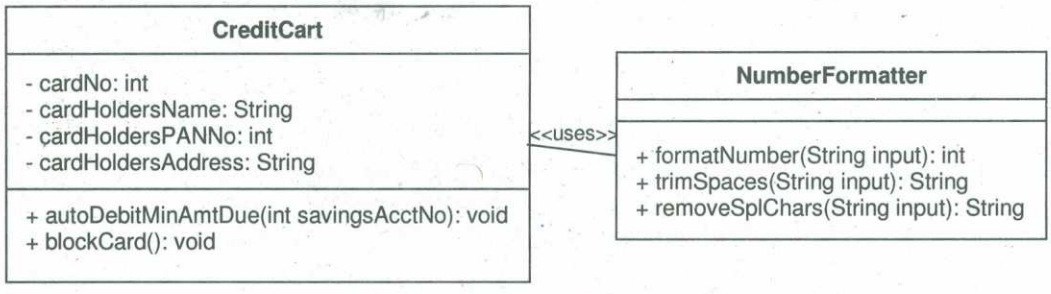


Figure 2.4 : Class Diagram

In the above example, CreditCard is the application logic object handling the core business rules. Internally it uses a utility object NumberFormatter to convert the user input into a properly formatted 16 digit credit number. The methods provided in NumberFormatter such as, formatNumber(String), trimSpaces(String) are generic functions which act on the input and do not depend on context such as user session.

2.6 METHODOLOGY IDENTIFICATION OF OBJECTS

In this section, we will see the generally employed thumb rules for the identification of objects

Once the problem domain is given, the next step is to start identifying the candidate objects in that domain. The following are some of the commonly used ways to start identifying static objects:

- **Identify nouns:** The first step is to look for subjects or nouns in the given problem domain that has a set of properties. Here, we need to identify real-world subjects and map them to the static objects. This is often known as "nounification" and it should represent the proper abstraction of the real-world objects. For instance, in a bank transaction scenario, we can easily identify these static objects: Account, Customer, Deposit, etc.

Similarly, in an e-commerce scenario, we can identify these objects: Item, ShoppingCart, Order, Payment, etc.

- **Grouping similar objects:** Once we identify the obvious and main static objects in the first step, we will look closely at the problem domain to see if there are any other objects that are similar to the identified objects. Similarity can either be in terms of similar attributes or similar category or any relationship to the identified objects. These objects can then be added to the object inventory. For instance, in a banking scenario, a SavingsAccount and CheckingAccount form the special case for Account and hence then, can be made of sub classes of the Account class.

  Duplicate elimination: After these two steps we would need to look for other Nouns and try to eliminate any duplicate, irrelevant or incomplete objects. For instance, in a problem statement related to the banking scenario, there could be description of the customer's vehicle. Though vehicle would constitute a separate object, it might not be relevant for the problem domain at all.

☞ **Check Your Progress 2**

(1) The objects which act as helpers are called .....

(2) The objects which handle core application logic are called .....

(3) ..... coupling is one of the key motivating factors for application logic objects.

(4) The four key specification attributes for a class are: ....., ..., ..., and .....

2.7 SUMMARY

In this unit, we started by looking at the core object oriented concepts and features of a class. We then moved on to map real world objects of a problem domain to objects with a sample use case. In the next section, we discussed the specification of static objects with a sample use case. We then checked the motivating factors and identification of application logic objects with the help of a few examples. We also saw the criteria for Utility objects and their identification (with examples). We concluded by checking commonly used methodology for identification of objects.

2.8 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

(1) attribute, operation, associations

(2) Interface

Check Your Progress 2

(1) Utility objects

(2) Application logic objects

(3) Loose

(4) Responsibilities, attributes, operations and constraints.

2.9 FURTHER READINGS

Reference Books

Grady Booch: '*Object-oriented Analysis and Design*'; Addison-Wesley.

Cox, B.J.; '*Object Oriented Programming – An Evolutionary Approach*'; Addison-Wesley, Mass.; 1986.

Reference Websites

http://en.wikipedia.org/wiki/Object-oriented_analysis_and_design

UNIT 3 SOFTWARE TESTING TECHNIQUES

Structure

3.0 INTRODUCTION

Software testing is a very important phase in software development. It is a repetitive process that helps identify not only the errors in the developed application but also corrects those errors. One may think that finding errors is a simple task. Then why would we need to pay importance to testing. In this unit, we will discuss the different aspects and importance of testing. Even though testing may seem easy, it is not an easy task and needs a strategic and systemised approach to ensure suitability.

In this unit, we will begin by defining software testing followed by why there is a requirement for software testing.

Further, in this unit, we will discuss two basic categories of testing i.e. *Black Box* testing and *White Box* testing. Any type of test we perform will fall under either of these categories. During the development of the software, continuous testing is performed at each level. We will discuss the four levels of testing i.e., unit testing, module testing, integration testing and system testing.

Unit testing is performed to test a single program whereas module testing is required to test a complete module. The purpose of integration testing is to test that communication between different modules is error free and the modules are compatible with each other in terms of interfaces and the data that they share. System testing is meant for testing software in the configuration environment in which it would be made functional. We will also discuss the basic types of testing that can be performed in system testing such as, stress testing, recovery testing, security testing and performance testing.

Finally, when the software is ready to be delivered, it is tested by the end users who will be the actual users of the application. Under the category of end user testing, we would discuss acceptance testing, alpha testing and beta testing.

3.1 OBJECTIVES

After going through this unit, you should be able to

- explain the meaning of software testing,
- explain what is the need and requirement for software testing,
- examine the personnel involved in testing,
- write and design test cases for a particular requirement,
- explain the concepts of two basic types of testing – black box testing and white box testing,
- explain the various levels in software testing i.e. unit testing, module testing, integration testing, and system testing,
- explain and design test cases required for unit testing for any requirement,
- explain the importance and requirement of Integration testing,
- explain the approaches to be adopted while conducting integration testing,
- explain the various types of testing, those that are part of system testing such as, stress testing, security testing, recovery testing, etc.,
- explain the concept and importance of Acceptance testing, and
- explain the concept and importance of Alpha testing and Beta Testing.

3.2 SOFTWARE TESTING

In general, testing means "checking something for faults, dysfunction, or errors".

Similarly, software testing simply means the process of checking software for any errors in it. This can also be defined as

"Software testing is a process of executing it (software or program) again and again with the purpose of finding errors in it."

Any software system is implemented with a purpose or objective or requirement. We may say that software is designed and implemented because there are a set of requirements that this software has to meet.

"Software testing is also a process of examining if the software meets those stated requirements or not."

The purpose of software is not only to meet the requirements, but to give a good performance in terms of time and resource utilisation. Hence, along with meeting the stated requirements, software is also tested for its efficiency and optimisation. Hence we can state that

"*Software testing is the process of evaluating software for its quality in terms of meeting the requirements, accuracy and efficiency."*

3.3 NEED FOR SOFTWARE TESTING

When software is implemented, it has errors. The errors in software are usually known as *bugs*. We must understand that not even a perfect and experienced developer is capable of writing a bug free code. The main and initial purpose of software testing is to find out the errors or bugs in a developing software. A program is run again and again with a purpose to find errors in it. A program or software is run and each of its functionality is tested to verify accuracy. While testing bugs are found and reported. The developer

corrects the code that is causing the bug and it is tested again. Hence, and testing is a repetitive exercise to identify the bugs and correcting them, identifying and correcting them again.

In this process of finding and correcting errors, software testing hence, assures the reliability of software in terms of its functionality with respect to its requirement specifications.

Software testing is also required to test the performance of the system. Software of a good quality not only works correctly but efficiently as well. Hence, the important purpose of software testing is also to check the performance of the software. If the software is not able to achieve its functionality efficiently (say if it is taking a lot of time) then also it needs to be corrected. We must remember that along with functionality, time constraint to complete a task by the software is equally important.

Software testing is a mandatory and required phase in software development where the sole purpose is to deliver accurate, reliable and efficient software.

3.4 WHO DOES THE TESTING?

Here, we need to be clear that testing starts as soon as the very first program is written by the developer. A programmer is responsible for testing the code she/he writes. The first testing of any code is usually done or should be done by the programmer who develops it. The programmer checks the program he/she has written for functionality, accuracy and efficiency as well.

Then, from a complete software perspective, there is usually a dedicated team of engineers who are responsible for thorough testing of the software. This team is usually known as the testing team. The prime purpose of this team is to test the software from all perspectives and to report it accordingly, in case, there is some scope for improvement. The software is not delivered or considered to be complete until this team signs off.

Depending upon the nature of the software, the testing team carries out various types of tests. Each testing type has a nature and purpose. We will discuss all these types of testing in the following sections.

3.5 TEST CASES

We could begin by saying that a test scenario is a test case. Software is developed for a specific and a fixed set of requirements. We usually give a serial number or unique number to each requirement such as, Req1, Req2, and so on. During testing we test software with respect to each requirement and make sure that each functionality is implemented. In order to test a single requirement we may have multiple scenarios. Each scenario will test an aspect of the requirement. These possible scenarios for each requirement are well designed, listed and documented. These are known as test cases.

Example 1

Suppose we have two requirements as
Req1: User should be able to copy a source file file1 into destination file file2.

For this requirement a basic set of test cases would be something like this:

Requirement Id	Test Case Id	Description
Req1	T1Req1	Check if source file is successfully copied into destination file
Req1	T2Req1	Check that proper error is shown if the source file doesn't exist
Req1	T3Req1	Check that proper message is displayed if the destination file file2 already exists

3.6 TYPES OF TESTING

Depending upon what is tested and who the tester is, there are broadly two types of testing:

(i) White Box Testing

(ii) Black Box Testing

3.6.1 White Box Testing

White Box testing is also known as structural testing or glass box testing. Its goal is to test the internal code of the software. It tests the program at the level of the source code. Here, the tester has the knowledge of the actual source code of the software and what is tested, is the inner structure of the program. Test cases are written with the knowledge of the logic of the program. We are only concerned with the testing of accuracy of the logic of the program. We do not focus upon the requirements of the software.

We may state that white box testing is the deep and detailed inspection of the logic and structure of the source code of an application or program. Here, the main focus is to exhaustively execute the program several times, with different inputs to ensure that each statement of the code is executed and tested.

Example 2

If we have a line of code as below:

```
If (age >= 18) {}
```

For testing this code, we must run the program to test three different test cases,

T1: when *the value of age is less than 18*,

T2: *when the value of age is equal to 18*

T3: *when the value of is greater than 18.*

And for each test case we must ensure that the right code is executed.

In white box testing, all the test cases are written with the knowledge of the internal structure and logic of the code to make maximum test coverage of the code. This is primarily done by the programmer or developer who develops the code. It is a first step to testing and to ensure that what is implemented in the code promises to execute accurately.

An exhaustive white box testing:

i. Guarantees that all independent paths have been executed.

ii. Executes all logical decisions on their true and false sides.

Example 3

if – then – else type of decision making code is tested for the true as well as false value.

```
If (choice == 1) {
}
else If (choice == 2) {
}
else If (choice == 3) {
}
```

In the above case, we must test it for all the cases for the choice value i.e. 1, 2, and 3.

iii. Executes all loops at their boundary values and within values.

Example 4

For a loop structure like

for (counter = 0; counter <= 10; counter++) {}

We must test it separately for boundary values of loop variable counter i.e. 0 and 10. We must test for within values like 1 to 9.

iv. Executes internal data structure to ensure their validity

3.6.2 Black Box Testing

Black box testing is also known as functional testing. The sole purpose of black box testing is to test the application or software from its functionality point of view. In this types of testing, the software is tested to check whether the software fullfills all the specified requirements. In black box testing, a tester is not concerned about testing the logic of the program. The internal details of the program are not known to the tester. In this types of testing, the software is like a black box to the tester where internal details are undisclosed. The tester only tests the functionality of the program by supplying an input and observing the output.

As already stated, an application or software is developed to fulfill certain objectives or requirements. Black box testing is a detailed inspection of the software functionality against the already specified requirements for which it is developed. The test cases are carefully written for each and every requirement specified. We may agree that black box testing verifies a software to ensure that it does exactly that, which it is required to do.

To understand further, let us take an example.

Example 5

Suppose we are required to build software for purchasing books online. The simpler requirements can be stated as

Requirement1 – User should be able to login to the website

Requirement2 – User should be able to see books catalogue

Requirement3 – User should be able to place an order

Requirement4 – User should be able to make the payment

Requirement5 – User should be able to logout

Now, developers will write the complete code for implementing all the above stated five requirements. A tester will then test the software to see if the developed software meets all the stated five requirements. For this, a tester will write the test cases for testing each requirement.

Example 6

With respect to Example 5

Requirement1 – User should be able to login to the website

For this requirement, a basic set of test cases would be something such as:

Test Case Id	Test Case Description
Test Case 1:	Check if the login screen is clearly visible to user or not
Test Case 2:	Check if the user is able to enter username and password in the provided space
Test Case 3:	Check if the user is properly logged in with a valid username and password
Test Case 4:	Check whether a proper error message is displayed if user enters an invalid username
Test Case 5:	Check whether a proper error message is displayed if user enters an invalid password

If you look carefully, you will realize that, for a single requirement, we have written five test cases to check all the possible inputs and expected output. Further, we can write more detailed test cases to check the requirement as per the guidelines more thoroughly.

On a similar note, test cases are written for all the specified requirements to test the functionality of the developed software. If you notice, no internal details are used or required by the tester. Tester only needs to understand the requirements specified in the requirement specification document and write all possible test cases for each requirement.

Here, it would be pertinent to mention that, black box testing is usually done by a separate dedicated team who are experts in writing the test cases and carrying out the testing. They are not those who were involved in developing the software

3.7 LEVELS OF TESTING

During the complete software development life cycle, software is tested at different phases. We can list various levels of testing as below:

- Unit Testing
- Module Testing
- Integration Testing
- System Testing

In Figure 3.1, we show different levels of testing in a hierarchical way. Here P1, P2, … till P7 are independent programs. M1, M2, … , M9 represent modules.

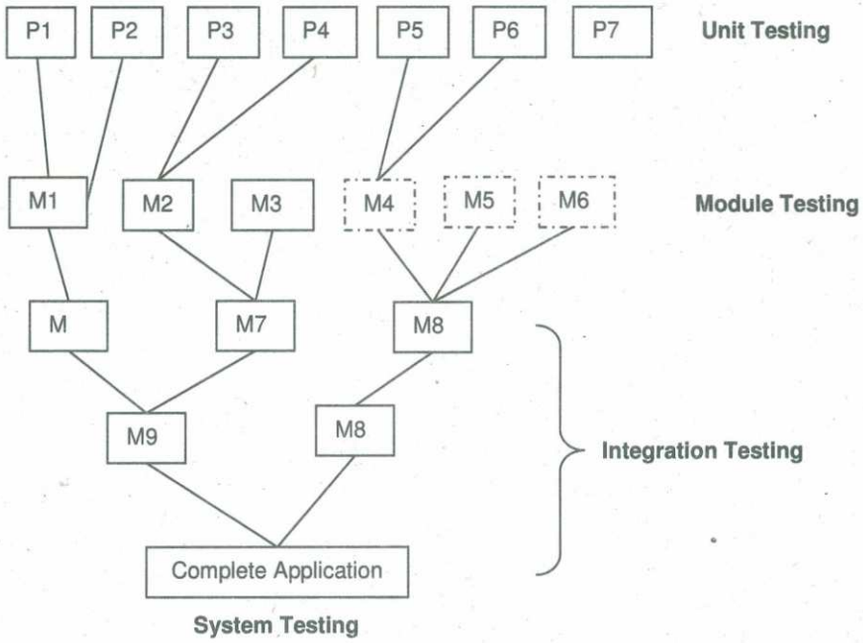


Figure 3.1 : Levels of Testing

3.7.1 Unit Testing

Unit testing is the first level of testing of the software. It is done at the level of each individual program. Unit testing is done by the developer who develops that unit or program. Hence, it is under the category of white box testing. We will be using the term developer or programmer hence forth. Both the terms mean the person – who writes the code for the software.

In unit testing, the tester is the developer who has written that program. The Developer tests the source code for its accuracy from different perspectives. The Programmer tests whether the correctness or accuracy of the source code works, to ensure that it works properly. Generates the output as it is expected to in all possible cases.

The main point here is to note that, unit testing is the responsibility of the programmer who writes the programs. The Programmer is required to test the program keeping in view of the code implemented. For instance,

- If the for loop is implemented in the logic of the program, the programmer must test the program for boundary values as well as for some inner values and ensure that it gives the correct output in all possible cases.
- If a program has file handling, then the programmer must include testing depending on files features such as, different file attributes, end of file cases, file exist or not etc.
- If a program has mathematical computations, then the programmer must ensure that the operation is understood and implemented, integer and floating point arithmetic is taken care of, and precision errors are handled properly. For each case the program must be executed and tested.

Discussed above are just a few particular cases. The Programmer must thoroughly test the program, carefully looking at the code he/she has written. The Programmer must ensure that all the paths she/he has implemented are executed and tested.

Example 7

With respect to Example 5, Requirement1 – User should be able to login to the website:

For this Requirement1, software may have multiple programs or say units such as:

- A program or unit to design the screen,
- A program to read and check username,
- A program to read and check password,
- .
- .
- .

Each of the above programs is initially treated as an independent unit of the code and implemented separately. Each program may be developed by the same programmer or by separate programmes. This is not our concern here. We must understand that each program given above is considered as a unit and first tested independently to ensure that the implemented source code is working properly.

3.7.2 Module Testing

Let us first understand the meaning of a module. Logically, a group of related programs that achieve a single task in an application is usually known as a module. Testing a complete module, a collection of related program units to check if the module is working properly as a whole and producing the desired output is known as module testing.

Module testing may fall under the category of white box testing as well as black box testing.

In the first step, the testing is performed by developers responsible for implementing that module. It could be that a single developer has worked on a complete module or there could be multiple developers working simultaneously on the same module. All those who are involved in implementing the module are responsible for testing the working of the module as a whole. Again, we must understand that developers test the complete module

more for checking the accuracy of the code. The developer conducts the white box testing of the module. In unit testing, the focus was only on testing a single program at once. Here, multiple programs constituting that module are tested as a whole for complete accuracy.

Once the developer is convinced that the program fulfils the requirements, the module is delivered to the testing team or say, "the QA team". The module can be given to the QA team for thorough functionality testing of that module. The QA team again tests the module from different perspective. The module is first tested for its basic functionality. Once the QA team is assured that the module works as it is expected to work and all features are fully implemented, it is tested for other parameters such as efficiency, load, and abnormal cases. This phase of module testing falls under the black box testing category.

Example 8

With respect to Example 5,

Requirement1 – User should be able to login to the website:

Requirement1 can be taken as an independent single module – ***Login module***

As stated above, the Login module has three program units

- A program or unit to design the screen
- A program to read and check the username
- A program to read and check the password

Now, in module testing we can assume that each program has been tested individually and undergone unit testing already. Here, in module testing all the three programs are tested as a whole to check that the functionality of the login module is working properly.

This task is done by developers first, and the module is then handed over to the QA team that tests it not only for functionality but for its performance as well. Performance, in this case, should include the following:

- Time taken to login
- Number of users that may login at a point of time
- The behavior of the module in the case of maximum users logging in and so on

Similarly, all other requirements can be mapped as different modules.

Requirement2 – User should be able to view books catalogue will make an independent ***catalogue module*** and will be tested independently as a module.

Requirement3 – User should be able to place an order will be implemented as an ***order module***.

Requirement4 – User should be able to make the payment as **a *Payment module.***

Requirement5 – User should be able to logout will be **a *logout module.***

3.7.3 Integration Testing

After the different modules are implemented and tested independently, the next level of the testing process is integration testing. Integration testing is the step of software testing in which different modules are integrated and tested together.

Usually one assumes that there is no requirement for the integration testing when thorough unit testing and module testing has been done. But, here, we need to understand that in unit testing or module testing, we only check independent programs mainly for

finding errors. Here, no attention is paid to test the interconnection or interfaces between the various modules. Software, as we know, is the integrated collection of all the modules. And these modules communicate with each other. In integration testing, our focus is mainly on testing the interfaces between the modules.

In software, different modules interact with each other. Different modules communicate with each other using interfaces or data. Usually the output of one module is passed as input to another module.

In integration testing, the main purpose is to detect problems in interfaces between modules. We test whether the arguments passed by the calling module are exactly the parameters expected by the called module. We must test for any parameter mismatch on both sides.

Another main focus in integration testing is to move towards testing functionality of modules as well. In this phase, different modules grouped together are checked for functions they are required to perform. At this step, we start taking the application as black box and test for functionality rather than for finding bugs in the code.

Strategy for Integration Testing

A normal problem may arise here – the problem of how to go about performing integration testing? What modules should be grouped together and how?

When we talk about performing integration testing, we basically talk about two strategies:

Non-Incremental Integration

A simple approach is to group together all the modules or the modules that make most of the system and conduct the testing. This is known as the **Big Bang** approach. In this, all the modules are combined together and the system is tested as a whole. This approach is not the preferred approach. Once all the modules are integrated and tested it may result in chaos. Many errors are identified. It becomes difficult to isolate the problem and the module causing the problem. A lot of time is wasted in narrowing down the code that is causing the error. In practice, it has been experienced that an error initially seeming to be in one module is further narrowed down to some other unrelated module. Hence this practice is time consuming and inefficient.

Incremental Integration

In incremental integration testing, step-by-step modules are grouped together and tested. Modules are combined together in steps incrementally to make subgroups. Each subgroup is then independently tested. Then the subgroups are combined together to make larger groups and these groups are then tested again. Incrementally, subgroups are combined and larger groups are created and tested. This is repeated till all the modules are grouped as a single unit and have been tested for flows.

If we perform the integration testing in an incremental way carefully, it will be easier to identify errors and the module where the error lies.

Two ways to perform incremental testing are the following:

(i) Top Down Approach

(ii) Bottom Up approach.

Example 9

Refer to Example 5 and Example 8,

In Example 8, we mentioned that for each functionality, we will test different modules. For Example 5, we can have Login module, Catalogue module, Order module and Payment module.

- We can first integrate Login and catalogue module as module M7 and test both together.
- Similarly, we can integrate order module and payment module and can test them as a single module M8.
- In next step module M7 and M8 can be integrated and tested together.

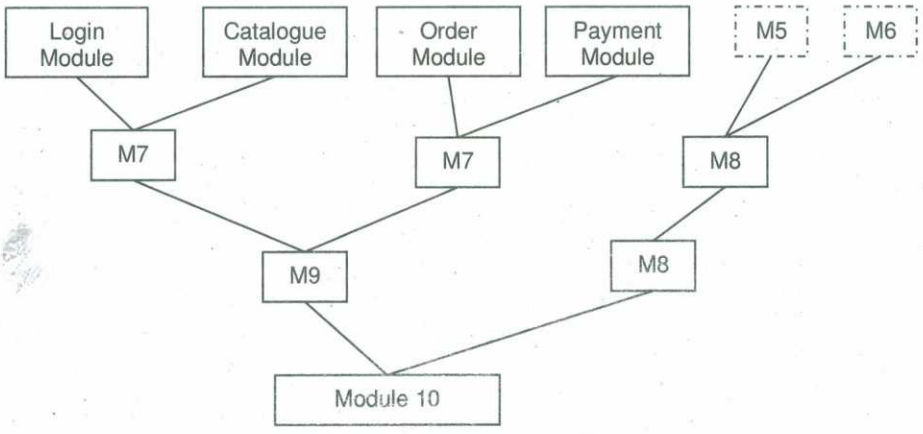


Figure 3.2: Incremental Module Testing

Figure 3.2 shows how modules are grouped together step-by-step and tested. This is an example of incremental testing.

In the next section, we shall discuss an important type of testing that must be performed during Integration testing. It is known as Regression testing.

3.7.4 Regression Testing

An important part of Integration testing is Regression testing.

As discussed in the previous section during integration testing modules are added incrementally and tested. The process is repeated several times. As a result the software changes each time as a new functionality or module is added. The added module changes the behavior of the software i.e. it impacts the behavior of the modules already working.

In practice, whenever a new module is added we first test the functionality that may be added by the new module. Then we must also test all the functionality or test cases of the modules that were previously working well. This is done to ensure that any changes caused due to addition of new module does not affect the ones that were working prior to the changed/added modules.

This is what is known as regression testing. This may be defined as, whenever any change is made to the code or a new module is implemented and integrated, all the possible test cases those that have been executed before are also executed again to ensure that no error entered into the already working software.

Regression testing is very important and performed by the QA team. A new module is implemented and integrated into the software; it is handed over to the QA team. It is the responsibility of the QA team to run all the possible test cases as part of Regression Testing. If it is observed that a newly added module is affecting the behavior of modules that were working previously, then it is reported as a major bug as part of the failure of integration testing.

Example 10

With respect to Example 5, we can have the following modules

Module1 – Login module

Module2 – Catalogue module

Module3 – Order Module

Module4 – Payment Module

Module5 – Logout module

- Now suppose we test Module1 and Module2 together as a group initially. In this, we will test all the functionality and test cases of Login Module and Catalogue module.
- Then, a step further, we will add Oder Module to the previous group. Now, first, we will perform regression testing to test all the test cases of Module1 and Module2 again to ensure that the functionality working previously is not changed or affected due to the addition of Module3.

Thus, regression testing is done at each step when we integrate a new module or change any previous code.

3.7.5 System Testing

As the name suggests, System testing is the level of testing wherein a complete system is tested as a whole. We must first understand what a system is. You might have read in some fundamental course earlier, that, a system is not the only software we develop. Software is usually incorporated with other softwares or installed in an environment or within a hardware. A software with its complete environment where it becomes operational is what we call a system.

In system testing, we test the system as a whole with all parts integrated together in an operational environment. This falls in the area of black box testing. No logic or code is visible at this stage when the system testing is performed. Testing is performed totally from an operational point of view of the system.

To test various parts and aspects of the complete system, system testing comprises of different types of tests. Each test type has a defined scope and purpose.

In the next few sections, we shall discuss the following types of testing performed as part of system testing:

- Stress Testing
- Security Testing
- Load Testing
- Performance Testing
- Recovery Testing

Stress Testing

Stress testing is performed to test the stability of a system in abnormal conditions and unotable environments.

It is performed by executing the system in such a manner that the system demands resources in enormous quantities beyond normal capacity till the breaking point. Then the results are observed. A stress testing of a software system is performed to measure its robustness and to identify the breaking point or usage limit of the system.

The goal of stress testing is to identify how the system behaves in abnormal conditions. The goal is to make sure that the system does not stop abruptly in case of heavy demands in terms of resources such as, memory or disk space, etc.

Security Testing

Security testing is crucial for software systems that deals with sensitive data and information. When a system uses sensitive data and information, it is very important to perform security testing. Such systems that deals with important data are usually the target of improper use or say penetration to steal the information.

Some of the common examples of such systems could be the Banking System, the Stock management systems, systems involving online payment, or systems dealing with individual personal information or government data. Such systems are always under the threat of improper use by attackers who would be interested in stealing information such as passwords, codes, etc. Hence, for such systems, security testing is crucial.

Security testing tests the protection mechanism implemented in the system. The main purpose is to identify (if any) improper penetration of the information from the system.

Security testing is performed by experienced testers usually hired and trained for this purpose. They are asked to execute the system as a person who wishes to penetrate the information. They try to retrieve the password, generate system error or browse insecure data to find access to the complete system and so on.

Recovery Testing

Recovery testing is also termed as failure testing.

During the execution of the system, the system could fail. System failure means that a system may hang, or it may crash. It may also happen that during normal execution, network connectivity may be lost, hardware may fail, database may not respond and so on. There could be different types of system failure due to various reasons.

A well developed and robust system must be either able to recover from failure by itself and resume functioning normally or must stop working without causing any harm (also known as ***graceful exit*** of) to the system. In a nutshell, a system should be fault tolerant.

Recovery testing is system testing that is performed to test the recovery mechanisms of the system. Recovery testing is performed to test whether the system is fault tolerant or otherwise. In recovery testing, the tester knowingly and forcefully causes the failure of the system and observes if the recovery takes place as it should.

Recovery testing also tests the time taken by a system to recover in case of abnormal crash or any other failure.

Some common failure test cases for study can be:

- Close the application with the help of the task manger and check if the application exits gracefully.
- Restart the computer and verify that data integrity is maintained even after the abnormal exit of the application.
- If the system is communicating with the database on a network, unplug the network cable. Then, plug the network cable after some time and observe if the communication starts properly from the point where it was disconnected.

Similarly, there could be various failure cases depending upon the nature of the software. The purpose of recovery testing is to identify and test all such cases. And then to ensure that software recovers itself from any failure.

Performance Testing

The software must meet all the requirements from the functionality point of view. But, it is equally important that software response time should be within the acceptable ranges to the end user. Time taken to perform a task by a software is of utmost concern to the end user.

Specifically, the time taken by a software to do a task is of high importance in case of real time applications. In real time applications, time is the primary factor for consideration and performance is of highest priority.

Performance testing is system testing wherein the system is tested for its performance mainly in terms of time taken to respond. Though it is of high importance in real time applications, all software undergoes performance testing.

In performance testing, for example, we test the length/duration of time taken to login on a server. If a search is performed, the time taken to show results? If more time is taken in showing the results, the user may not like to use the application. Hence, along with the functionality for each task, time performance and memory usage is also tested.

3.8 ACCEPTANCE, ALPHA AND BETA TESTING

These types of testing are performed by the end users who would be actually using the software. The end user always executes the software from a different perspective. Developers are always careful and cautious while testing the application. An end user, not aware of the course of development of the application, would execute the software without any prior assumptions.

To understand these types of testing, we must also have clear understanding about the nature of the software developed. In general, software may fall in either of these two categories – project for a specific customer or a product.

Project for a specific customer is developed only for that client. In this case, only those requirements provided by the customer are implemented. The end user is the one customer or say client. In this particular case, acceptance testing is performed.

The product is the software which is developed for general use for various customers. A product software example could be Microsoft Office, Adobe Photoshop etc. These are developed for general usage where the customer may vary from an enterprise to a student. The types of testing performed by end users are – Alpha Testing and Beta Testing.

3.8.1 Acceptance Testing

Software for a specific customer is built for one client with specific requirements. In this case, only that customer is the end user of the test. Once the software is ready, it is given to the customer. The customer who is the end user of this application tests the software against all specified requirements. This type of testing is known as *acceptance testing*. This is not performed by the developer but rather by the client himself/herself.

The purpose of acceptance testing is to find only if, the actual user is ready to accept the software that has been developed. It depends upon the nature of the application and on the customer and would want to perform the acceptance testing. Acceptance testing can be informal and may take a very short time. For some applications and clients, it could be a systematic and time consuming process.

Briefly, we may define acceptance testing as being performed by the customer for whom the software is developed. It is performed by the customer to validate his/her requirements as specified earlier. Only after the acceptance testing, is the software accepted by customer.

Failing the acceptance test means, there is a gap between the software implemented and the requirements specified by the customers. Or it could mean that though the functionality is in place, the user's requirements are not met as desired, by him/her. So, in this case, again, the requirements are analysed and implemented as per customer's input. After the changes, all types of testing will be performed by the developer and the QA team again. Then, the software will be handed over to the customer for the acceptance testing.

This cycle may go on till the customer accepts the software.

Usually, to avoid repeated acceptance testing, the customer is consulted from time to time during the development phase itself. When a module is implemented and tested by the developers and the QA team, it may be shown to the customer for his/her feedback on

that module. This is done to plug any gaps in the requirements during the early stages itself. Identifying the gaps in the early stages requires less work on the part of the developer.

3.8.2 Alpha and Beta Testing

If the software developed is a product, then the testing performed by the end users will fall into two categories – Alpha testing and Beta testing.

In alpha testing, a proposed customer (end user) who is ready to use the software is identified. End user performs the testing at the developer's site itself. The software is used by the end user, while the developer observes the process. The end user uses the software in its natural way, and the developer focuses on noticing the way the user takes and uses the software. Developers focus on noticing the usability problems faced by the end user as well as the errors countered.

In beta testing, an end user is identified who is ready to perform the testing at his site. Beta testing is performed at the site of the end user and the developer is not present. The Software product is treated as "live" application by the end user and used in its natural way. Note that the developer is not present at the moment. Hence, in beta testing, the environment in which testing is performed is actually the simulation of the environment where the product will be actually used. The end user tests the software and records all the errors encountered. The recorded errors and usability problems are conveyed to the developer. Developers takes their time to understand the errors and make necessary changes in the software.

☞ **Check Your Progress 1**

(1) ..... is a process of executing it (software or program) again and again with the purpose of finding errors in it.

(2) ..... is also known as structural testing or glass box testing.

(3) ..... is also known as functional testing.

3.9 SUMMARY

Testing is a very important phase in the development cycle of software. No software can be used without proper systemised testing.

Software testing can be defined as a process of executing it (software or program) again and again with the purpose of finding errors in it. Other than finding the errors in software, the important task of testing is to ensure that requirements specified are implemented as desired by the software. Software testing also focuses on analysing the software from its performance point of view. Good software is not only supposed to meet the requirements but to perform better in terms of time and resource utilisation. If the software performs a task in time that is not acceptable to the user, then such software is of no use in practically.

In brief, we may say that Software testing is a process for evaluating the software for its quality in terms of meeting requirements, accuracy and efficiency.

Generally, in all types of applications developed, the developers and the QA team are responsible for this task. The developer is the person or team that actually develops the code for the application. Developers test the application keeping in mind the code implemented. The QA team is a team of engineers who are experts in testing and they test the software from the point of view of functionality and performance.

There are basically two categories of testing – White Box testing and Black Box Testing.

White Box testing is also known as *structural testing* or *glass box* testing. Its goal is to test the internal code of the software. It tests the program at the level of the source code. Here, the tester has the knowledge of the actual source code of the software and what is tested is the inner structure of the program. The testing performed by the developer is white box testing.

Black box testing is also known as *functional testing*. Here, the internal code is not visible to the tester. The purpose of black box testing is to test the application or software from its functionality point of view. The tester tests the application without knowing its implementation details. Here, while testing, we are not concerned with testing the logic of the program, rather, we are concerned with testing the functionality against the requirements.

Depending upon the nature of the software developed, testing is performed at various levels – Unit testing, Module testing, Integration testing and System testing.

Unit testing is under the category of white box testing. In unit testing, a single program or unit is tested for its implemented code. It is performed by the developer who has developed the program. The developer focuses on testing only one program at a time from the perspective of the code that was actually written.

Module testing is used for testing a complete module. A module is logically defined as a group of programs that achieves a complete functional unit. In module testing, we focus on testing the complete group as a single module to check the functionality provided by that module. It may be done by the developers. Hence, falls under the white box category. If module testing is performed by the QA team members, then, it may fall under black box testing.

Integration testing is performed to test and make sure that everything functions correctly and runs smoothly when all modules are grouped together. Software is the integrated collection of all the modules. Modules interact with each other by calling each other's functions using interfaces. A module can use data generated by another module. A complete application works only when all the modules integrated together work properly. The purpose of integration testing is to test the interface compatibility between the various modules in the software.

Regression testing is also performed during integration testing. In regression testing, whenever, a module is integrated, then, all the previous functionality is tested first. This is done to make sure that the module integrated later has not affected the working functionality of the software. Every time, we integrate a module, the regression test has to be performed.

The next and the final level of testing is *System testing*. Software does not work alone. Software collaborates with other hardware, database, network etc. to work as a complete system. In system testing, we test the software in the configuration environment in which it is supposed to run. Different types of testing is done during system testing – Security testing, Stress Testing, Performance Testing, and Recovery Testing.

Finally, software must be tested by the actual end user who will be using it. Different types of end user testing is performed depending on the nature of the software.

If the software is a customized project that is developed for a particular customer, then acceptance testing is performed by the end user. In *acceptance testing*, the end user is actually the customer for whom software has been developed. Here, the end user (or customer) tests the software strictly against the requirements specified by him/her. He/she tests to find if the developed software is acceptable or not. Once the customer is satisfied and has done thorough acceptance testing, then only will the software be delivered to him

In alpha testing, certain end users are identified and invited to the developer's site. These end users test the system while the developers observe them. The developers observe and record the ways software is being used and all the errors that have occurred. The developer also records the difficulty faced by the end user in using the software. Accordingly, developers make changes in the software to remove recorded errors and difficulties. Alpha testing is performed by the end users but in a controlled environment.

In Beta testing, a few real end user sites are identified and the software is deployed there. In beta testing, the software is tested in a "live" environment in which it will be used further. The errors are recorded by the users and given to the developer's team.

3.10 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

(1) Software Testing

(2) White Box Testing

(3) Black Box Testing.

3.11 FURTHER READINGS

Reference Books

Roger S. Pressman, *Software Engineering A Practitioner's Approach* (McGRAW-HILL).

K. K. Aggarwal, Yogesh Singh, *Software Engineering* (NEW AGE INTERNATIONAL PUBLISHERS).

Reference Websites

http://en.wikipedia.org/wiki/Software_testing.

UNIT 4 DEVELOPMENT AND EXECUTION OF TEST CASES

Structure

4.0 INTRODUCTION

This unit will examine the basic principles involved in software testing including various kinds of testing and the philosophy in the development and execution of various test cases. Software testing is essentially a validation process involved that analyses the software from various aspects to verify if it satisfies the expected functionality. IEEE defines software testing as *"Software testing is the process of analysing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item"*. Testing is an essential part of the software development lifecycle (SDLC) for the following obvious reasons:

- It detects the defects early and contributes to the increased quality of the project.
- It identifies hidden erroneous code in the software such as, failure to handle exceptions, erroneous handling of various forms of input data and in boundary conditions, failure to encode/decode the input/output and others.
- It minimises the overall cost of the software product.
- It checks if the system meets the non-functional requirements (NFR) such as scalability, performance, and availability.
- It analyses the software adherence to the given specifications and overall correctness.
- It adheres to the organisational quality process and other legal regulations related to quality, and
- It provides critical inputs to the production deployment of the product.

Overall, a comprehensive testing strategy is required for developing a robust software product which is suited to bullet-handling to handle proof all types of scenarios while satisfying the stated requirements.

One key point worth mentioning here is the impact of cost of an undetected defect which highlights the importance of early detection of the bug. The cost of fixing the defect grows exponentially as each phase in the software lifecycle progresses/advances. The cost of fixing the bug in the requirements phase is 1 unit whereas, it is 100 times more than this if it is uncovered after the product is launched.

Overall, the quality of the software is one of the four key parameters for the success of the software and, the other parameters being cost, time and budget. Software testing provides the crucial metrics that helps us to measure and eventually control the quality of the product.

Definition of Key Terms

- **Test case:** It is an executable artifact with a specific input data covering a scenario and has a deterministic pass/fail termination.
- **Test case specification:** It is the set of requirements that need to be satisfied by the test cases.
- **Test suite:** It is a set of test cases.
- **Regression testing:** This is a type of testing which is normally conducted during software updates/modifications to uncover any new software bugs. This is usually done during incremental software releases, software updates/patches.
- **Defect/Bug/Fault:** It is an incorrect or unexpected behaviour caused by software system differing from the specified or expected results in a given scenario.

Scope of Software Testing

Broadly, the scope of software testing activities can be categorised into two groups:

- **Verification**: This is the process to ensure that software is coded as per the given specifications. For instance, banking software should correctly implement a "Fund transfer" transaction and satisfy all involved business rules. Here, various formal methods such as structured testing, manual inspection will be adopted.
- **Validation**: This process ensures whether the software meets the end-user requirements. For example, a validation process would check if the customer can run the banking software on a mobile device. This would involve verification of various testing processes and documents.

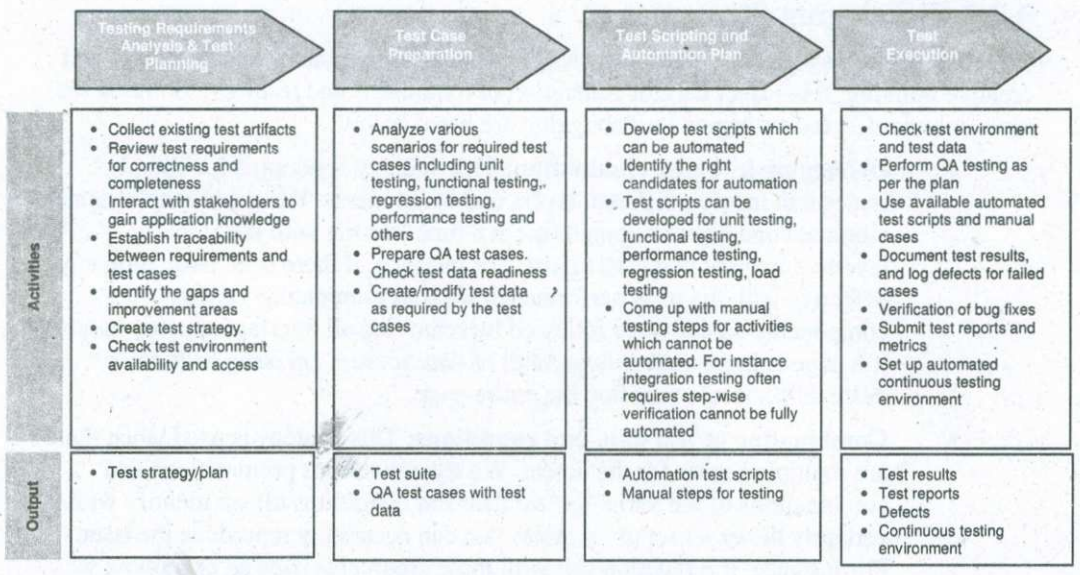
Principles of Software Testing

IEEE literature identifies the following seven key principles of software testing:

- Testing shows presence of errors.
- Exhaustive testing is not possible.
- Test early and regularly.
- Accumulation of errors.
- Fading effectiveness.
- Testing depends on context.
- False conclusion: no errors equals usable system.

Testing Lifecycle

The following are the high level testing activities involved in testing the lifecycle of a software project.



4.1 OBJECTIVES

After going through this unit, you should be able to

- understand various test scenarios,
- learn about different debugging techniques,
- understand different kinds of testing,
- gain knowledge about various kinds of testing tools, environment, and
- understand devising test cases and plans for various kinds of testing.

4.2 DEBUGGING

Debugging is the act of analysing and locating the root cause of defects/errors identified during the testing phase. This is often a creative process involving root cause analysis (RCA) and formulating a hypothesis about the probable cause of the defect. The role of debugging during the testing cycle is shown below along with activities in each step:

4.2.1 Debugging Challenges

Debugging is often a complex process due to the following challenges:

- **Triggering condition/variable isolation:** Only a few defects triggered under specific conditions. Identifying the exact variable or condition would be complex as there are a large number of variables. For instance, a function call may fail only 10% of the times. There are a number of variables such as, total load, network bandwidth available, data set used, timing, etc. Identifying an exact cause or combination of variables would be a challenge.
- **Intermittency factor:** Sometimes, the defect only occurs intermittently and there would not be any deterministic steps to reproduce the issue all the time. This temporary issue could be happening due to a variety of reasons such as, intermittent network issues, dependency on another condition, hardware failure, etc.
- **Hidden under multiple complex steps:** Often only a few defects occur after executing a huge number of steps under certain conditions. Reproducing and root causing those defects would be challenging.
- **Performance and memory issues:** Normally performance and memory leak issues are difficult to debug as it involves multiple layers, systems and components. Network bandwidth and user load would further add to the complexity.

4.2.2 Debugging Strategies

Effective debugging involves deep knowledge of underlying systems and processes and creative thinking; it involves careful generation of hypothesis and testing it. Some of the most commonly used strategies for debugging are listed below:

- **Debugging by cause elimination:** This strategy is adopted for complex issues that involve different layers and components. Here, we systematically eliminate one layer or component at a time starting with the layer/component that is has most. For instance, if there is an issue probably defective with the page performance we start eliminating various components on that page followed by removing all interfacing components. If a page is integrated with number of data sources, an issue with one of those data sources may hog the entire page.
- **Combination of test data and conditions:** This strategy is used when there are multiple causes for the defect. We try out various permutations and combinations of the variables/test data and conditions till we identify with certainly the exact set of variables that can accurately reproduce the issue. For instance, if a function call with three arguments rises an exception, we try combinations of three arguments to understand the exact argument that is causing the issue.
- **Memory profiling:** This strategy is normally followed for memory related issues. A profiler tool is used to analyse heap memory during program execution. This would give insights into components consuming more memory and the ones which are orphan/not garbage-collected. With this information we can then deduce the cause for memory leak issues. This technique also involves analysing memory/thread dumps.
- **Step-wise debugging:** This is another technique which is often carried out with the help of Integrated development environment (IDE) tools. In this technique, we start the program in "debug mode" wherein, we can control each step of program execution using IDE. We can then "step-in" and "step-out" each line of code inspecting values of various variables. This helps us to get the exact method, line and variable value where the defect has accused/occurs.
- **Call tracing:** This is a top-down approach wherein, we trace the call from the top-most component to the root data source. We examine the return value at each level to isolate the code that is the cause for the issue.

4.2.3 Scenarios for Applying Debugging Strategies

The following table provides scenarios that warrant a specific debugging strategy:

Debugging Strategy	Brief Detail	Suitable Scenario
Debugging by cause elimination	Eliminate various participating layers and components step-wise	• Apply this technique when the defect occurs in complex multi-layered systems involving numerous components.
Combination of test data and conditions	Try out combination of test data and steps that lead to the defect	• Use this technique when we know the data set and steps that lead to the defect
Memory profiling	Analyse and profile the memory during program execution	• Apply this technique when the defect is related to memory such as, abrupt program termination, out-of-memory issues, and memory dump issues.
Step-wise debugging	Execute the program one step at a time to root cause the issue	• Use this technique when we want to take the snapshot of variable values at each step and to isolate the exact step that is causing the issue
Call tracing	Trace the function call starting from the top-most component till the component causing error is identified	• Use this technique when we know the sequence of calls that is causing the defect

☞ **Check Your Progress 1**

1. The strategy that is best suited for debugging memory dump issues is .....
2. ..... debugging strategy is used to check the values of variables at various points in program execution.
3. Removing one cause at a time is called ..... debugging strategy.

4.3 TESTING TOOLS AND ENVIRONMENTS

This section provides a list of the various testing tools that are widely used for carrying out testing activity.

4.3.1 Testing Tools

Testing tools help testing in varying ways: automatic creation of test cases, execution of test cases, debugging defects. The following table indicates various popular testing tools used. The tools are categorised based on the type of testing where the tool is used.

Tool	Category	Testing phase
Junit	Unit testing tool/Open source	• Unit testing • Code coverage report
SOAPUI	Integration Testing tool	• Service testing • JDBC testing
HP LoadRunner	Performance testing tool/Commercial	• Load testing • Stress testing • Peak load testing • Endurance testing
Selenium	Web testing/Open source	• Cross browser testing • Automatic web testing
IBM Rational functional tester	Functional testing tool/commercial	• Functional testing
Jenkins/Hudson	Continuous integration/open source	• Continuous integration testing • Code coverage reports
HP Quality Center	Test case management/commercial	• Management of test cases • Scheduling of test cases • Requirement traceability
PMD/Checkstyle	Automated Code review/Open source	• Static code analysis • Cyclomatic complexity
JMeter	Load testing/Open source	• Load testing • Web service testing • Database testing

4.3.2 Testing Environment

Enterprises prefer to conduct testing in various environments as part of code promotion activity. The testing team and scope of testing would vary across environments.

The table given below, provides a list of various testing environments, its purpose and the testing team responsible for it.

Testing Environment	Purpose	Testing team
Development Environment	• Performs unit testing • Verifies bug fixes	• Developers
QA Environment	• Performs functional testing	• Quality Analysis (QA) Team • Functional testing team
System Integration Testing (SIT) Environment	• Performs integration testing • Conducts performance testing • Carries out Load/Stress testing • Performs Cross-browser testing • Carries out Security testing	• QA Team • Integration testing team • Security team
User Acceptance Testing (UAT) Environment	• Performs end-user testing • Performs business testing	• Business users
Pre-production environment	• Performs Beta testing • Carries out Regression testing • Conducts Production fixes	• Sample set of end users • QA Team

4.4 TYPES OF TEST CASES

In this section, we will look at various types of testing and test cases that will be designed for each type of testing.

4.4.1 Types of Testing Approaches

Before we understand the types of test cases, let's look at some of the most widely used types of testing in large software projects

The following table lists various approaches to testing:

Testing Approach	Testing type
Box approach	• Black box testing: Tests the system without the knowledge of the internal structure of the system • White box testing: Tests inner structure of a program or a component
Execution based approach	• Static testing: Involves code walkthroughs and other static methods such as, code inspection • Dynamic testing: Tests the actual code by executing it
A/B approach	• Alpha testing: End-user testing at developer's site • Beta testing: End-user testing at their local sites
Release based approach	• Smoke testing: High level testing to uncover "show-stopper" defects that could impact release schedule
Manual/automated approach	• Manual testing: Testers manually execute test cases • Automated testing: Test case execution is automated

4.4.2 Types of Testing Methods

Given below in a tabular form, are the main types of testing that falls into one of the above categories of testing. Test cases will be designed for each of these testing types.

Testing Type	Brief description	Test Owner
Unit testing	• White box test to verify if code works as-expected at low level • Requires knowledge and flow of the code structure • For instance, testing a method that adds two integers by passing two positive values and asserting their sum • Usually done in a development environment	• Developers
Integration Testing	• Testing involves various software components and interfacing systems to validate the interaction between them • Contains wide range of scope items such as, data flow testing, services testing, performance testing, exception handling testing etc. • Done in a SIT environment	• QA team • Integration testing team
Functional testing	• The testing involves high-level design of functional modules/components to verify its working as expected by the design specifications • Done in a QA environment	• Quality Analysis (QA) Team • Functional testing team
System Testing	• This involves carrying out the entire testing in a fully-integrated environment that is very similar to that of the customer and also testing the customer requirements • Testing is done to ensure compliance with design specifications, legal regulations • Non-functional requirements such as, scalability, usability will be tested	• Quality Analysis (QA) Team • Integration testing team • Security team
Performance testing	• Testing is done to check the compliance of the system with specified performance guidelines. • Key performance metrics such as, page load time, process completion time will be tested	• Performance testing team
Stress testing	• This testing is done to test the system beyond the limits of specified requirements. • Include applying more user load, request for high volume of data	• QA Team
Usability testing	• Testing is done to check how friendly the system is for end user interface in terms of data entry, result interpretation, navigation, information discovery, etc.	• QA Team
Acceptance Testing	• Test whether the system satisfies specified acceptance criteria such as, defect percentage, performance etc.	• QA Team
Regression testing	• This testing is done on incremental updates to software to ensure that the updates have not accidentally broken down earlier functionality causing unintended errors. • Checks if the system still satisfies the required functionality	• QA team

4.4.3 Deep Dive into Development of Test Cases

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

This section, delves deeper into the development of test cases for some of the prominent test types.

4.4.4 Test Cases for Unit Testing

The unit test case should test the expected behaviour of low-level code structures such as, functions, classes and components. A good unit test case should check the following on a given code structure:

- Positive flow for expected behaviour
- Boundary conditions including null inputs, empty inputs, large data sets, reserved characters as inputs, multi-language inputs
- Check all the branches of the flow. If the code fragment contains if-else conditions or loops then the input data should be designed to cover all branches and looping criteria to ensure the code is fully covered.

The following is a code example for a simple unit testing test case:

Let's consider the following code fragment. The class IntegerMultiplication has a single method multiply that takes two integers and returns the product of two input values.

```
//Class for multiplication
class IntegerMultiplication {
    long multiply(int a, int b) {
        if (a == null || b == null) return null;
        return (a * b);
    }
}
```

The unit test case for the above class is given below which covers all the scenarios mentioned:

```
//Unit test case for the TestIntegerMultiplication class
public class TestIntegerMultiplication {
    //This test case tests the positive flow with two integers
     public void testmultiply_for_normal_flow() {
        IntegerMultiplication intmul = new IntegerMultiplication();
        assertTrue(intmul.multiply(50, 40) == 200);
            assertTrue(intmul.multiply(20, 10) == 200);
    }

//This test case tests for negative input values
 public void testmultiply_for_negative_values() {
    IntegerMultiplication intmul = new IntegerMultiplication();
    assertTrue(intmul.multiply(-50, 40) == -200);
        assertTrue(intmul.multiply(-20, -10) == 200);
}

//This test case tests the method for various boundary conditions
 public void testmultiply_for_boundary_conditions() {
    IntegerMultiplication intmul = new IntegerMultiplication();
        //Check if the method returns null
```

```
    assertNull(intmul.multiply(null, 40));
        //Check if the method accepts a varied number of arguments
    assertFalse(intmul.multiply(10));
        //Check if the method handles very large values
    assertTrue(intmul.multiply(38292817182, 25718192736) == 98482205269108838952);
}
}
```

4.4.5 Test Cases for Functional Testing

Functional test cases should test the functionality of the component to check whether they match the expectations of the design specifications.

Usually functional test cases are based on the use cases that they are tested for:

1. Various data inputs
2. Exception/boundary scenarios
3. Test cases intended to break the system

Let's consider a simple use case and a functional test case for the same (refer to the following table):

Use Case Name	Login use case
Pre-conditions	• User should contain valid user id and password • User should has access to the login page
Post condition	• User should be successfully logged in • User session should be created
Normal Flow	• User accesses the login.html page • User enters valid user id and password • User submits "login" button
Other Flows	Forgot Password flow • User clicks on "Forgot password" button • User enters the valid user id Register Flow • First time user accesses login.html • User clicks on "Register" button • User is taken to register.html page
Exception Flow	• User enters null values for user id and password • User enters invalid user id and password combination

The following are the sample functional test cases for the login use case:

Use case	Test Case id	Test steps	Expected Output	Actual Output	Result
Login use case	Login_001	1. Enter valid user id and password	User should be successfully logged in and a user session should be created	User was logged in with a valid session	Pass
Login use case	Login_002	1. Enter invalid user id and password	System should throw error "Invalid user id or password"	There was no error message	Failed
Login use case	Login_003	1. Enter user id as &name==name	System should throw error "Invalid user id or password"	User was logged in with a valid session	Failed
Login use case	Login_004	1. Click on "Forgot password" 2. Enter non-registered user id	A Non-registered user should not receive email	Email was received	Failed

4.4.6 Test Cases for Integration Testing

Integration testing will be done to test the following key aspects of integration:

(1) Handling of data flow across interfaces

(2) Handling of exception scenarios and boundary conditions across interfaces

(3) Performance of upstream systems

(4) Check sub-systems

(5) Error handling (time outs, network outage, transaction rollback) across interfaces.

A typical integration test case consists of the following steps:

(1) Perform setup. This step includes establishing connection with the interface and initialising all necessary variables.

(2) Populate dummy data.

(3) Test scenarios.

```
//Integration test case for a DBConnection object
public class testDBConnection {
Connection con;
Context ctx;

    //Perform setup and initialisation activities
    protected void setUp() throws Exception {
        con = getConnection();
        ctx = initContext()
        populateDummyTableData();
    }

    public testCRUD() throws Exception {
        Statement stmt = con.createStatement();
        String sql = "SELECT bizname from bizTab";
        ResultSet rs = stmt.executeQuery(sql);
        assertTrue(rs.getFetchSize() == 5);
    }
}
```

☞ **Check Your Progress 2**

(1) The environment used for testing integration test cases is .....

(2) The testing that can be applied for sub-systems like ERP or database is called .....

(3) The testing that is done normally on incremental software updates is called .....

(4) The ..... testing approach assumes no knowledge of inner structure of the component

4.5 TEST PLANS

Test plan is essentially a comprehensive planning artifact which covers the following aspects:

- Document containing details about scope, schedule and resources for testing throughout the lifecycle of the project. Test plan usually takes inputs from the project plan and system specification document

- It should cover all kinds of testing including unit, functional, integration and regression testing that will be carried out in the project.
- Testing approach: agile or traditional approach, manual or automated approach.
- Should contain specifications of test units, key features that needs to be tested and the expected outcome.
- It should contain the list of test deliverables.

Sample Test Plan

A sample template for the test plan containing the "table of content" items is given below:

Sample Test Plan : Test Plan ABC Project

TABLE OF CONTENTS

1. Introduction
2. Objectives
3. Reference Documents
   - 3.1. Project Plan
   - 3.2. Requirements specifications
   - 3.3. High level design
   - 3.4. Low level design document
   - 3.5. Enterprise standard guidelines
4. Scope
5. Testing Strategy
   - 5.1. Unit Testing
   - 5.2. System and Integration Testing
   - 5.3. Integration testing
   - 5.4. Service testing
   - 5.5. Performance and Stress Testing
   - 5.6. User Acceptance Testing
   - 5.7. Regression Testing
6. Hardware Requirements
7. Test Schedule
8. Features to Be Tested
9. Features Not to Be Tested
10. Roles & Responsibilities
11. Schedules
12. Dependencies
13. Risks and Assumptions
14. Test phase entry and exit criteria
15. Test Deliverables
    - 15.1. Test estimates
    - 15.2. Test schedules
    - 15.3. Test specification
    - 15.4. Test script
    - 15.5. Test data
    - 15.6. Test reports
16. Tools
17. Approvals

4.6 SUMMARY

In this unit, we started by laying out the key design principles and looked at various stages of a testing life cycle. We then examined various challenges in debugging a defect discovered during testing and commonly employed strategies to debug the defect. We looked further at various testing tools that are used in testing and different testing environment. We also discussed different kinds of testing and designing test cases for some of the prominent ones. We finally examined at the sample test plan with various sections.

4.7 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

(1) Memory profiling

(2) Step-wise debugging

(3) Debugging by cause elimination

Check Your Progress 2

(1) System Integrated Testing environment

(2) Integration testing

(3) Regression testing

(4) Black-box testing

4.8 FURTHER READINGS

Reference Books

Cockburn, Alistair (2000) *Writing Effective Use Cases*, Addison-Wesley.

Collard, R. (1999). "Developing test cases from use cases," *Software Testing & Quality Engineering*.

Reference Web sites

http://en.wikipedia.org/wiki/Software_testing

http://www.ieee.org

MPDD-IGNOU/P.O. 2.0 K/Sep. 2018 (Reprint)

ISBN : 978-81-266-6611-9