Unit: Function Oriented Design

Structure

1.0 Introduction

This unit details the concepts of function oriented design and the modeling techniques for the same. Typical procedural software system consists of multiple functions which share the state and interact. We would see methods for designing a modular function, to decompose 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 desired output.

1.1 Basic elements of function oriented design

Following are the key elements of function oriented design:

- **Function:** It is a code unit consisting of sequence of instructions to perform a specific task. It is often referred to as method, routine in different programming languages
- **Function state:** The data on which a function operates. For example a *getEmployeeDetails* function would require the employee state stored in database. In function oriented design the state is centrally stored accessible by all functions.

- **Sub Function:** The main function can be further be broken down into smaller re-usable units of code called sub-function or sub-routine. For instance *getEmployeeDetails* main function can further invoke *getEmployeeContactDetails* and *getEmployeePersonalDetails* sub-routines

1.2 Salient features of function oriented design

Following are the key features of function oriented design:

- **Top down decomposition:** A software system essentially consists of set of functions each of which performs unique functionality. Once the high level functions are identified, each function is further decomposed into modular sub-functions. For instance a high level function like “Update Person” which updates entire person details can further be broken down into:
    - **UpdatePersonPersonalDetails** to update the personal details of person like name, age, height, date of birth etc.
    - **UpdatePersonAddressDetails** to update the address details for person
    - **UpdatePersonContactDetails** to update the contact details like phone, email etc.

  The function decomposition needs to done till we reach an independent and modular function with a single responsibility
- **Function association:** Multiple functions are related for interaction and data sharing. For instance in the above example the main function “Update Person” invokes the sub routines like 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 person details state is shared among all the functions including UpdatePersonPersonalDetails, UpdatePersonAddressDetails and UpdatePersonContactDetails

1.3 Function oriented design vs. Object oriented design

Following table provides differences between the two design approaches:

Concept	Object oriented design	Function oriented design
Modeling	Objects are modeled based on real-world objects. For example in a banking software the objects mimic real-world equivalents like Account, Customer etc.	Functions are modeled on distinct and independent functionality. For example in banking system the key functions include transferFunds, checkBalance etc.
Abstraction	Objects are normally abstracted as nouns like Person, Address etc.	Functions are usually abstracted as verbs like subtractBalance, getAmount etc.
State	The state is usually distributed among various objects. For example each of objects Person, PersonAddress, PersonContact have its own state	The state is normally centralized and functions share and access the state. For example Person details are stored centrally in file or in database and multiple functions like getPersonDetails, UpdatePeronAddress access the centralized state
Hierarchy	Objects are logically grouped based on their interaction	Functions are grouped based on their functionality. Multiple sub functions can be grouped to form a bigger function

2.0 Objectives

After going through this unit, you will be able to:

- Understand general principles of solving a problem using function oriented design
- Identify and apply various decomposition techniques
- Model the function in visual fashion using the commonly employed visual notations
- Learn the best practices of function oriented design

3.0 Constructing solution to a problem

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

3.1 Solution Design Principles

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

- **Top down decomposition:** When we adopt this design principle, we iteratively and sometimes recursively decompose the problem from top. This is often a top-down approach for creation a solution to the problem. In function oriented terminology, we break the main function into smaller sub-functions which are both modular and reusable
- **Divide and conquer:** Using this design, we breakdown the problem functionally into smaller sub-problem. 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 visualized by 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 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 the 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.

3.1.1 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:

3.1.1.1 Structured Analysis (SA) and Structured Design (SD)

- **Structured Analysis:** During Structured analysis stage, the functional decomposition takes place. Each high level function is carefully analyzed and it is broken down into smaller functional modules. In this stage the problem description is also visually depicted in "Data flow diagram". This provides the detailed and visual structure of the system with all components and sub-components. Following are the utilities of SA:
    - Provides visual representation of the problem
    - Helps the users and stake holders to review the design and hierarchy
    - Helps in verifying the completeness of the problem domain elements
- **Structured design is** the next step wherein we map the functional modules identified in the structured analysis phase to map to program specific structures which can be implemented. This helps in:
    - Implementing the functional module in a given programming language

3.1.1.2 Structural decomposition

Structural decomposition aims at designing a function based structure which is function uniqueness (highly cohesive) and function inter-connection (loosely coupled). We will look at cohesiveness and coupling when we look at characteristics of good function oriented design. Data flow diagrams can be converted into structural decomposition.

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

- Start with elements in DFD which are functional processing units. Group all those elements within a single function in structure diagram
- Create input function by grouping all DFD elements related to input like file reading, service invocation etc.
- Create output function by grouping all DFD elements related to output like file writing, page rendering etc.
- Refactor the high level functions into further low-level reusable sub-functions and add the input and outputs accordingly.

3.1.1.3 Detailed design

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

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

3.2 Check your progress 1

1. In function oriented design the state is stored _____
2. _____ design principle involves breaking down single problem into smaller sub-problems
3. Data flow diagram is often designed in _____ stage

4.0 Identifying components and their interaction

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

4.1 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 design module. During this process, the design modules should be designed to have high cohesion and loose inter-module coupling. The modules which 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 like data validation, data conversion, information logging would form good candidates for sub-functions.
4. **Interaction modeling:** The input and output elements from DFD can be used to design the interactions between functions.

4.2 Component and stage mapping

This table identifies the component and association identified in each step 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

5.0 Visualizing the solution

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

5.1 Context Diagram

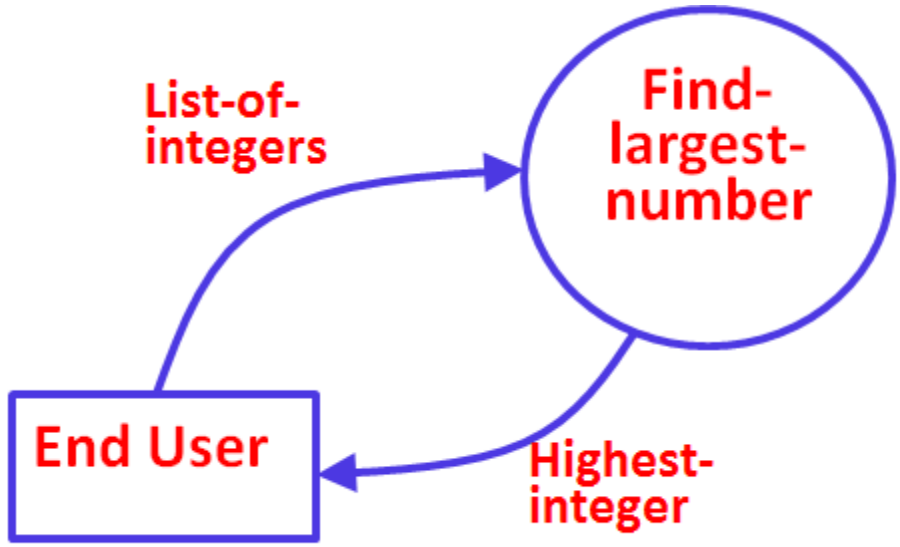
A context diagram is the visual modeling of highest level of abstraction. It provides the high level context of the overall system, the first level input data and the final output data. It serves as the first step for creating a detailed DFD

5.1.1 Development of a context diagram

Let us consider a simple example of a system which calculates 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 can be depicted as the following context diagram:



5.2 Data Flow Diagram (DFD)

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

- Main functions of the system
- Interaction of data within the system

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

5.2.1 Elements of DFD

Following are the key visual elements of DFD

- **Rectangle** to represent physical entities such as Computer, Building etc.
- **Circle** to depict the function like *updateEmployee*
- **Directed line** to show the data flow
- **Horizontal parallel lines** to depict the data structure or data store like File System or Database or ERP system

5.2.2 Process steps for developing DFD

1. Start with system specifications document. Identify each high level function and model it as circle in DFD
2. Identify all input data to this function and depict it in DFD
3. Identify all output data to this function and depict it in DFD

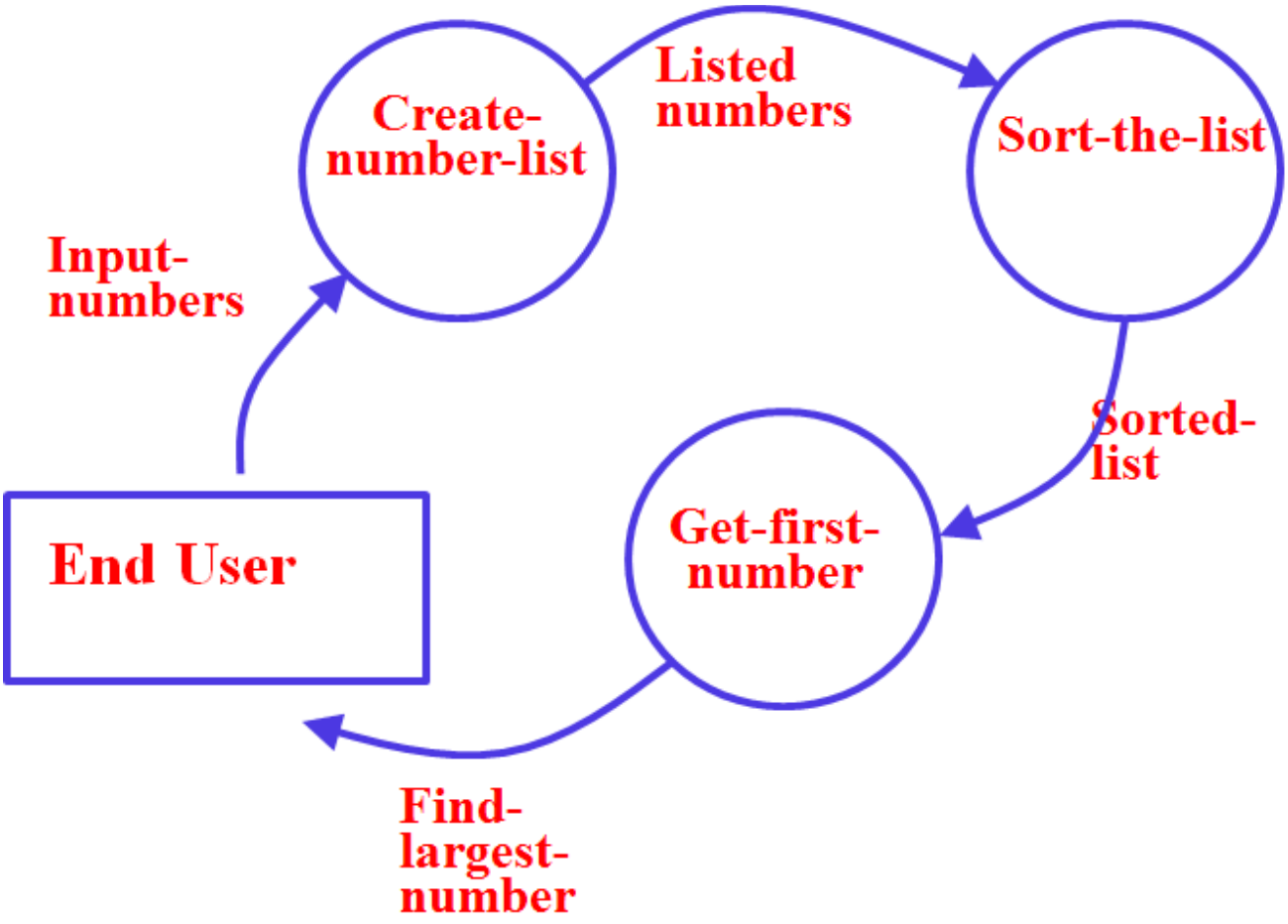
Further refining can be done by structurally decomposing the high-level function circle into smaller modular sub-functions till each sub-function represents a distinct reusable task implementation.

5.2.3 Example of a DFD

Let us extend the scenario given for “calculating largest number” scenario given for context 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. Collect the input into a numerical list
2. Sort the list in ascending order
3. Return the highest number

The DFD for this is given below:



6.0 Characteristics of good function oriented design

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

6.1 High level characteristics

6.1.1 Functional independence

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

- **Reusability**: The function can be made more reusable if it performs single task.
- **Maintainability**: Increased maintainability 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.

6.1.2 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
- **Correctness**: The functions should implement all the requirements as per their specifications.
- **Efficiency**: The functions should efficiently use the costly resources like database connection, file connection etc.
- **Cost**: The functions should aim to reduce the overall cost in terms of maintainability and extensibility

6.2 Module level characteristics

This section details about the main characteristics a module or function needs to possess for a good design.

6.2.1 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 set of tasks. The logic and variables in the function should be present to perform only single activity in the most optimal fashion. This would have a strong internal relationship between internal elements of a function.

Types of cohesion:

- **Coincidental cohesion:** In this type of cohesion the internal functions in a given module are loosely correlated
- **Communication cohesion:** If functions within module update 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 first function is fed as input for 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:

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”);
}
```

Function with strong cohesion

```
Function Connection createConnection () {
        If (con == null)
          con = new Connection ();
        return con;
}
```

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

6.2.2 Coupling

Coupling indicates the dependencies across different modules. If a module is dependent on multiple functions in another module, then it is called 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 much 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 same global data. For instance all modules acting on a common shared persistent store for their functinality
- **Control coupling:** If a function argument passed from first modules controls the logic and order of instructions in another module. For instance 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 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 form a standard data coupling

6.2.3 Connascence

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

6.3 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 Dataflow 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

7.0 Summary

In this unit we started by looking at basic elements and some of the key features of function oriented design. Then we moved on to check various design principles to design a solution for a given problem using techniques such as structured analysis, structured decomposition. We also checked the process to identity 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 saw the good characteristics of good function oriented design including cohesion and coupling.

7.1 Solutions/Answers

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 cohesion and loose coupling

8.0 Further Readings

Reference Books

- Peretz Shoval: Functional and Object Oriented Analysis and Design: An Integrated Methodology;

Reference Web sites

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

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

Unit: Object Oriented Design

Structure

1.0 Introduction

This unit explains the design principles related 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 the real-world objects and design. Many software design principles, design patterns and techniques are often inspired from nature. Software architects find it natural to model the software components based on their real-world counterparts. The role of architects and designers mainly involves in marrying the objects in software world with real-world. This is also done in part to make the end-user accustomed to the software and make their transition from 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 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.

1.1 Motivation for object-oriented design

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 them down to its component objects likes Account, Transfer and its associations.
- **Abstraction:** Object-oriented design provides clear abstraction of things at various levels. For instance the objects in business layer can focus primarily on business logic whereas the objects in data services layer can focus on persistence which provides “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 behavior of a product can be defined in an interface; all product variants which implement the interface would provide the variant specific implementation of the behavior 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 the business logic and provides flexible and extensible design to accommodate future changes

1.2 Key object-oriented concepts

Following are the prominent object-oriented concepts:

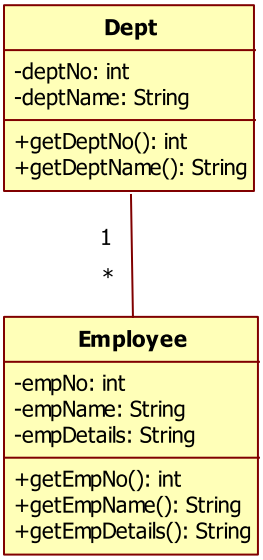
- **Object:** An object is often an instance of a class with its own “state”. It is a logic unit consisting of its own properties and methods to interact with the internal variables.
- **Information abstraction**: Each object has strict policy to hide its internal state for private variables and provide public methods to access the data.
- **Inheritance**: This feature enables the classes/objects to extend their super classes and inheriting the protected variables and methods and also provides ability to override the methods to provide specific implementation wherever required. This would be very useful in modeling the object hierarchy and providing extensible design
- **Polymorphism:** This feature provides an object 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.

1.3 Features of a class

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 the following diagram, deptNo, deptName are the attributes of Dept class and empNo, empName, empDetails are the attributes of Employee class
- **Methods**: Classes provide public accessor methods to act on its variables. In the below example getDepNo(), getDeptName() are public method which provides the value of the deptNo and deptName variable values respectively.
- **Associations**: Associations are depicted by relationship between the classes. In the following diagram “One Dept has many employees” is denoted by “on-to-many” relationship between the classes

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


Dept
-deptNo: int
-deptName: String
+getDeptNo(): int
+getDeptName(): String
1
*
Employee
-empNo: int
-empName: String
-empDetails: String
+getEmpNo(): int
+getEmpName(): String
+getEmpDetails(): String

2.0 Objectives

After going through this unit, you will be able to:

- Understand generic object oriented principles
- Generic guidelines and thumb rules for specifying and identifying problem domain static objects
- Identification and design of application/business logic objects
- Identification and design of utility objects
- Generic guidelines and best practices for identification and design of objects

3.0 Identification of Problem Domain Static Objects

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

In this section we will examine generally followed thumb rules to identify the static objects for a given problem domain

3.1 What are static objects?

Before we start looking at steps for identifying static objects, lets 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 statechart diagram would represent a dynamic view which represents the transition along with time.

3.1.1 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 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 provides his book
4. Librarian makes an entry in his register for the book against student and provides 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.

3.2 Check your progress 1

1. The key features of a class are _____
2. _____ Feature allows the object to be replaced by its sub-objects.
3. Contract and behavior specification is given by _____.

4.0 Specification of Problem Domain Static objects

In this section we will examine how to specify a static object for a problem domain

4.1 What is a 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.

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 nothing but relationship between the class wherein we specify the “is a”, “has” relationships.

Let's look at an example of the specification of the class:

```
/*
This class checks if the given number is prime or not.
Known constraints: The class has method which would only take 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 check if the number is prime 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);
        }

}
```

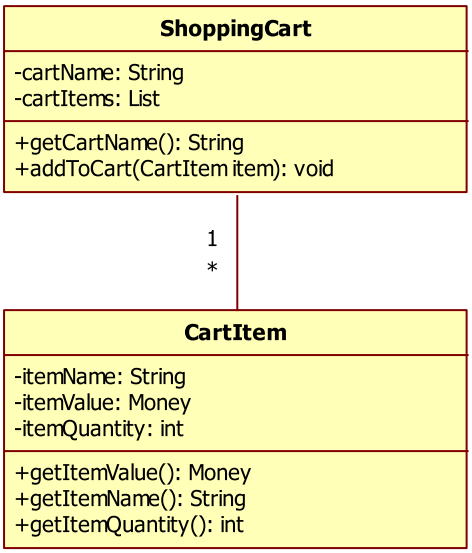
4.2 Specifications for static objects for a sample problem domain

We will see the class specifications for a 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 like name, value and the quantity which user needs.
3. User can select the items and add to his/her shopping cart
4. Once the shopping is complete user can proceed to checkout

Let's consider two classes for modeling: ShoppingCart and CartItem. Following diagram 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:



5.0 Application Logic Objects

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

Business application layer consists of number of objects like Business Objects (BOs), Data Access Objects (DAO), Service façade, delegates, bean objects etc. Normally the application logic will be part of the Business objects.

5.1 Motivation of application logic objects

Following are some of the key motivating factors to create separate application logic objects:

- Centralized access to application logic
- Loose coupling and increased testability
- Layered separation of application logic concern
- Reusability of application logic

5.2 Identification of application logic objects

In a use case all objects which 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
- Used in application transaction

5.3 Sample application logic objects

Let us consider a banking scenario with following use case:

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

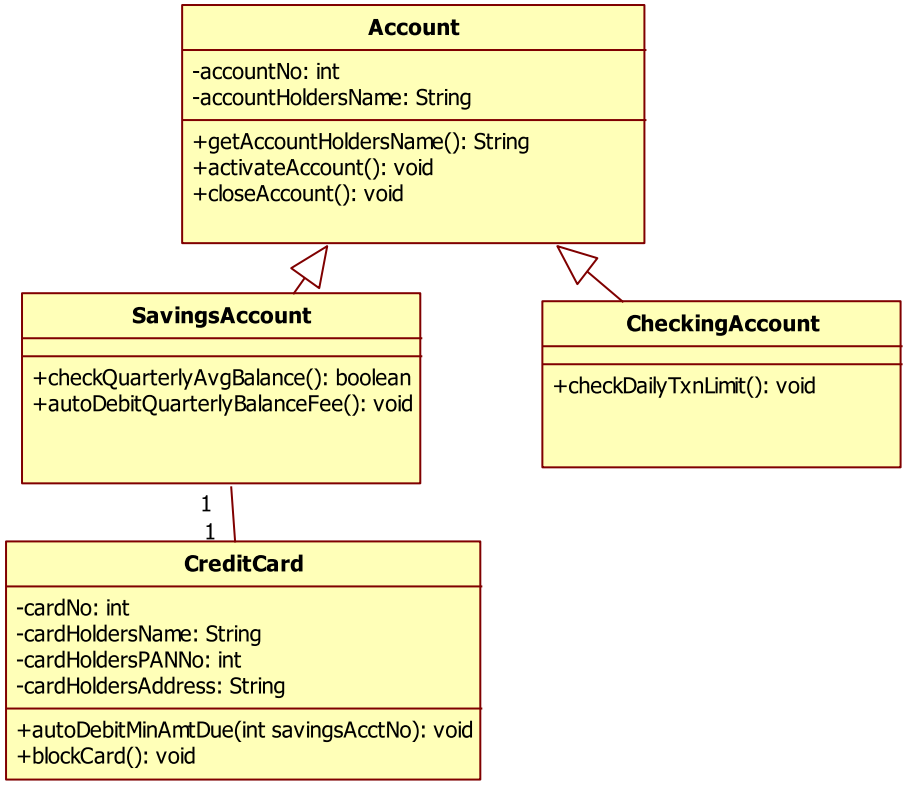
1. User can open an account by providing details required for account
2. An account can be savings account or checking account.
3. Savings account should contain a minimum balance of 10,000 Rupees per quarter. If this is not satisfied a fine of Rs. 500 will be deducted.
4. Each savings account is linked to only one credit card.

5. A credit card's credit limit is based on user's balance in the savings account for an entire year.
6. If user fails to pay the minimum amount due for a credit card, then the credit card is authorized to automatically debit the amount from linked savings account. Credit card will be automatically blocked if minimum amount is not present in savings account

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

1. Account
2. SavingsAccount
3. CurrentAccount
4. CreditCard

Each application logic object has corresponding functions to process the business rules. A sample representation of the class model is given below:



The methods like checkQuarterlyAvgBalance(), autoDebitQuarterlyBalance() contain core business logic which process business rules.

6.0 Identification of necessary utility objects

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

6.1 Criteria for identifying utility objects

Following are the key criteria for identifying utility objects:

- The object should act as 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

6.2 Common utility objects

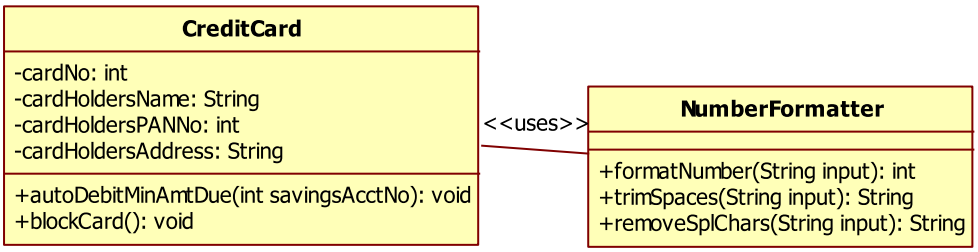
Following are the list of generally used utility objects in a typical enterprise application:

- CachingUtility: To handle the caching requirements
- LoggingHelper: To log the info, debug and error statements
- FileReaderUtility: To read the required files
- ResourceLocator: To provide the URL for a given resource
- BuildUtility: To handle build activities
- EncryptionUtility: To encrypt/decrypt the values as per the enterprise standards
- ExceptionHandlerUtility: To handle the exceptions
- EncoderUtility: To perform HTML encoding/decoding
- StringConverterUtility: To convert string into appropriate display formats
- MultiLangUtility: To get the language specific resource bundle for a given key
- ConfigurationHandler: To handle different configuration files
- ValidationUtility: To perform required field validations

As we can see the above utility objects perform specific re-usable and context independent function which can be used by variety of layers and classes. Due to this context insensitive nature of the utility objects most of the times they are used as static objects and used as singletons.

6.3 Example of utility objects in a problem scenario

Following class diagram indicates how utility classes are used by other domain objects and static objects:



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 like formatNumber(String), trimSpaces(String) are generic functions which act on the input and don't depend on context like user session.

7.0 Methodology of identification of objects

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

7.1 Identification of objects for a given problem domain

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

- **Identify Nouns:** The first step is to look for subjects or Nouns in the given problem domain which has a set of properties. Here we need to identify the real-world subjects and map them to the static objects. This is often called 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 obvious and main static objects in the first step, we will closely look at the problem domain to see if there are any other objects which 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 object inventory. For instance in a banking scenario, a SavingsAccount and CheckingAccount form the special case for Account and hence then can be made as sub-objects (or sub classes) of the Account class
- **Duplicate elimination:** After these two steps we will look for other Nouns and try to eliminate any duplicate, irrelevant or incomplete objects. For instance in a problem statement related to banking scenario there could be description of customer's vehicle. Though vehicle would constitute as a separate object, it might not be relevant for the problem domain at all. We can just add it as another attribute for Customer object instead of creating a separate object.

7.2 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: _____

8.0 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. Next section discussed about 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 few examples. We also saw the criteria for Utility objects, their identification with examples. At the end we check the commonly used methodology of identification of objects.

8.1 Solutions/Answers

Check your progress 1

1. attribute, operation, associations
2. Polymorphism
3. Interface

Check your progress 2

1. Utility objects
2. Application logic objects
3. loose
4. Responsibilities, attribute, operation, constraints

9.0 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 Web sites

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

Block-2 Unit-3 : Testing Techniques

1. Software Testing

In general testing means - checking something for faults, dysfunction, or errors;
Similarly, software testing simply means 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. We can say that any software is designed and implemented because there are set of requirements that this software will meet. Software testing is simply a process of examining if the software meets those stated requirements or not.
Along with meeting the stated requirements, software is also tested for its efficiency and optimization.
That means software testing is process of evaluating its quality in terms of meeting the requirements, accuracy and efficiency.

2. What is Test Case?

3. Types of Testing

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

a. Black Box Testing
b. White Box Testing

3.1 White Box Testing

White Box testing is also called 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 source code. Here the tester has the knowledge of 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 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 can say 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, main focus is to exhaustively execute the program many times with different inputs to ensure that each statement of the code is executed and tested.

Example 1, 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 age is greater than 18*.

And for each test case we must ensure that 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 of testing to ensure that what is implemented in code promises to execute accurately.

An exhaustive white box testing

i. Guarantee that all independent paths have been executed.
ii. Execute all logical decisions on their true and false sides

Example 2 all if – then – else type of decision making code is tested for the true as well as false value.

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

Example 3, 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 t for within values like 1 to 9.

iv. Execute internal data structure to ensure their validity.

3.2 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 testing software is tested to check whether the software is achieving all the specified requirements or not. 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 testing, the software is like a block box to tester where internal details are not known. 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 called as 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 can say that black box testing verifies a software to ensure that it does exactly the same as it is required to do.

To understand further, let us take an example.

Example 4: 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 in 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 is meeting all

the stated five requirements or not. For this a tester will write the test cases for testing each requirement.

With respect to Example 4, the Requirement1 – User should be able to login in the website; test cases would be something like

T1: Check if the login screen is clearly visible to user or not
T2: Check if the user is able to enter username and password in the provided space
T3: Check if the user is properly logged in with a valid username and password
T4: Check whether a proper error message a displayed if user enters the invalid username
T5: Check whether a proper error message a displayed if user enters the invalid password

---

If you notice carefully, 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 deeply check the requirement as per the guidelines.

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

Here, I would like to mention that, black box testing is usually done by a separate dedicated team who are experts of writing the test cases and do the testing. They are those who are not involved in developing the software.

4. Level of Testing

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

1. Unit Testing
2. Module Testing
3. Integration Testing
4. System Testing

4.1 Unit Testing

Unit testing is the first level of testing the software. It is done at the level of individual program unit. Unit testing is done by the developer who develops that unit or program. Hence it is under the category of white box testing.

In unit testing developer tests whether the source code he has written is working and generating the output as it is expected to.

With respect to Example 4, the Requirement1 – User should be able to login in the website;
For Requirement1, developer may have multiple programs or say units like

- A program unit to design screen,
- A program to read and check username,
- A program to read and check password, and so on

Each of the above programs is initially treated as an independent unit of code and implemented separately. Hence each program above is tested independently to find out if the source code implemented is working properly or not

---

4.2 Module Testing

A group of related program units that achieves a single task in an application is usually called as a module. Testing a complete module which is a collection of related program units to check if the module is working properly as a whole and giving the desired output is called as module testing.

This is again performed by developers those are responsible for implementing that module. It could be that a single developer is working on a complete module or there could be multiple

developers simultaneously working 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.

With respect to Example 4,
Requirement1 – User should be able to login in the website;
Requirement1 can be taken as an independent single module – Login module

As stated above, if this module is divided into three program units, then in module testing all the units together will be tested as a whole to check if the functionality of the login module is working properly or not.

Similarly, all other requirements can be mapped as other different modules like Requirement2 –User should be able to see books catalogue will make an independent catalogue module and will tested independently as a module

---

So in the module testing, not the independent program unit but all the related program units for a module will be tested.

4.3 Integration testing

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

In unit testing or module testing, we only check independent programs mainly for finding errors. Here no attention is given to test the interconnection or interfaces between the various modules. In integration testing our focus is mainly on testing the interfaces between the modules.

In an application different modules interact with each other. Usually output of one module is passed as input to other module. Different modules communicate with each other using interfaces. In integration testing, firstly main purpose is to find out problems in interfaces between modules. We test whether the parameters passed by one module are exactly the input other module is expecting. We 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 code.

Strategy for integration Testing

A normal thought may come here that how to perform integration testing? What modules should be grouped together and so on? Usually basic two strategies are talked about

a. A simple approach is to group together all the modules or say the modules those make most of the system and do the testing. This is called **Big Bang** approach. In this all the modules combined together and system is tested usually as whole. This approach is not the preferred one as it results in chaos. Many errors are identified and it becomes difficult to solve and isolate the problem and the module causing the problem.

b. Incremental integration
   In incremental integration testing, step by step modules are grouped together and tested. Modules are combined together in segments incrementally and independent groups are tested. Then the groups are combined together to make bigger groups and tested. In the incremental approach it is easier to identify errors and the responsible module and solve it. Two ways to perform incremental testing:
   i. Top Down Approach
   ii. Bottom Up approach

Regression Testing

An important part of integration testing is called Regression testing. During integration testing modules are added incrementally. As a result the software changes each time as new functionality is added. Now the added module may changes or cause behavior of already working modules. So in practice, whenever a new module is added we also test all the functionality or test cases of the previously working modules. This is done to ensure that any changes cased due to new addition have not affected the previous ones.

This may be understood simply sayin

With respect to Example 4, we can have following modules

Module1 – Login module
Module2 – Catalogue module
Module3 – Oder Module
Module4 – Payment Module
Module5 – Logout module

Now suppose initially we test Module1 and Module2 together as a group. 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 here first we will do regression testing to test all the test cases of Module1 and Module2 again to make sure that the functionality working previously is not changed or affected due to addition of Module3. This

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

4.5 System Testing

Simply system testing is testing the complete system as a whole for functionality, compatibility with hardware, compatibility with different operating system and performance.

We first must understand that any application we develop ultimately becomes the part of a computer based system. Software is incorporated with hardware or other software to make it work. For example a website based application depends highly on a web browser. Here idea to mention this that we must first understand that any software does not work in isolation. It has many dependencies in terms of other software, operating system or hardware.

System testing is the phase in testing process, where in an application is first tested for all its functionality against the complete requirements and then dependencies are tested and problems are identified. For example for a website based application after testing the available functionality, we must also test the application on all popular and available browsers.

System testing is a black box testing. We do not intend to find defects in code, rather on functionality.

Depending of what is tested we categorize system testing into following categories:

a. Performance Testing

This testing is done to determine performance of system in terms of scalability, reliability and how efficiently resources are used

This tests the stability of system when the system is highly loaded. It also tests the time of responses in idle time or during highly loaded time.

b. Load testing:

This is a performance testing wherein we try to test how system behaves when it is highly. For example, In online booking system, a load testing is done to determine how system behaves when maximum number of systems are logged in and making booking. This helps in identifying the response time of the system during the time when it is highly used.

c. Stress Testing:

In stress testing, we try to test the application to find out the maximum load that a system can handle and work. This is typically to identify the upper limit of the capacity.

d. Compatibility testing

Compatibility testing focus on testing the compatibility of the system with the environment in which it will operate. In compatibility testing, we can test hardware based comapatibility as well as software based

- Compatibility with Operating Systems is tested
- Compatibility with Peripherals is tested like with printer or DVD drive etc. if required
- Compatibility with Database is tested like with Oracle, MySQL etc.
- Compatibility with network
- Compatibility with Browser

e. Regression testing

As explained before regression testing is done to retest all the test cases again and again to make sure that changes made in software have not affected the previously tested functionality. For details refer in integration testing.

f. Recovery testing

A recovery testing tests how effectively an application recovers itself in possible failures like crash, network failure or hardware failure etc. We can have recovery testing performing tests like:

- If the application is on network and receiving or sending data over network, then try to unplug the network cable and plug the cable again. And then test if the data connection is suitably restored or not.
- If software is communicating to a web server or database, try to test the application that what happens if the web server is shut down suddenly or database is switched down.

In all such abrupt failures, application must respond properly and system integrity must be recovered.

5. Other types

5.1 Alpha Testing

5.2 Beta testing

Unit: Development and Execution of Test Cases

Structure

1.0 Introduction

This unit explains the basic principles involved in software testing including various kinds of testing and philosophy in development and execution of various test cases. Software testing is essentially a validation process which 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 analyzing 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:

- Detect the defects early and contribute to the increased quality of the project
- Identify 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.
- Minimize the overall cost of the software product
- Check if the system meets the non-functional requirements (NFR) such as scalability, performance, availability.
- Analyze the software adherence to the given specifications and overall correctness
- Adhere to the organizational quality process and other legal regulations related to quality
- Provide critical inputs to the production deployment of the product

Overall a comprehensive testing strategy is required for developing a robust software product which is bullet-proof to handle all 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 with the advance of each phase in the software lifecycle. The cost of fixing the bug in requirements phase is 1 unit whereas it is 100 times more than this if it is uncovered after the product is launched.

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

1.1 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/modification 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.

1.2 Scope of software testing

Broadly the scope of software testing activities can be categorized 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 satisfies all the involved business rules. Here various formal methods like structured testing, manual inspection will be adopted
- **Validation**: This process ensures that 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 process and documents

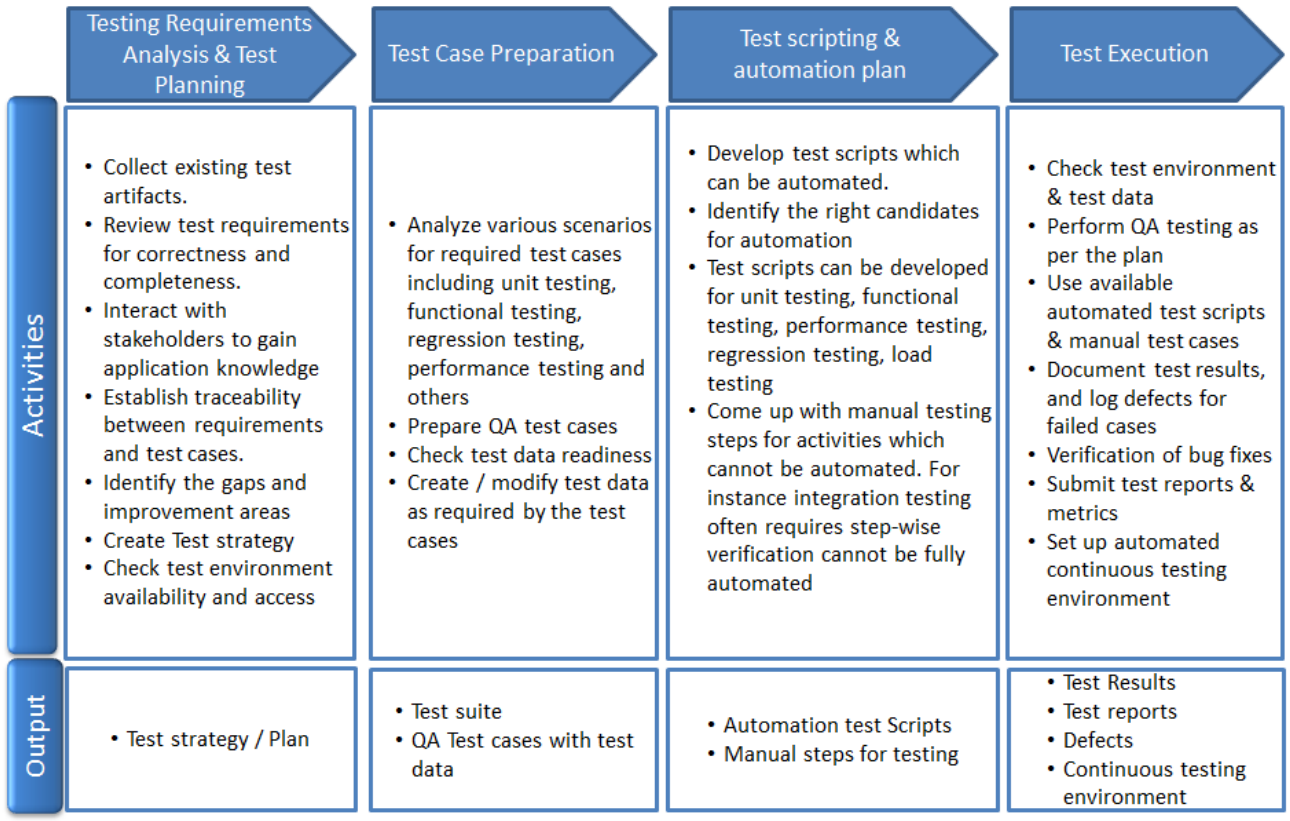
1.3 Principles of software testing

IEEE literature identifies 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

1.4 Testing Lifecycle

Following are the high level testing activities in testing lifecycle in a software project:



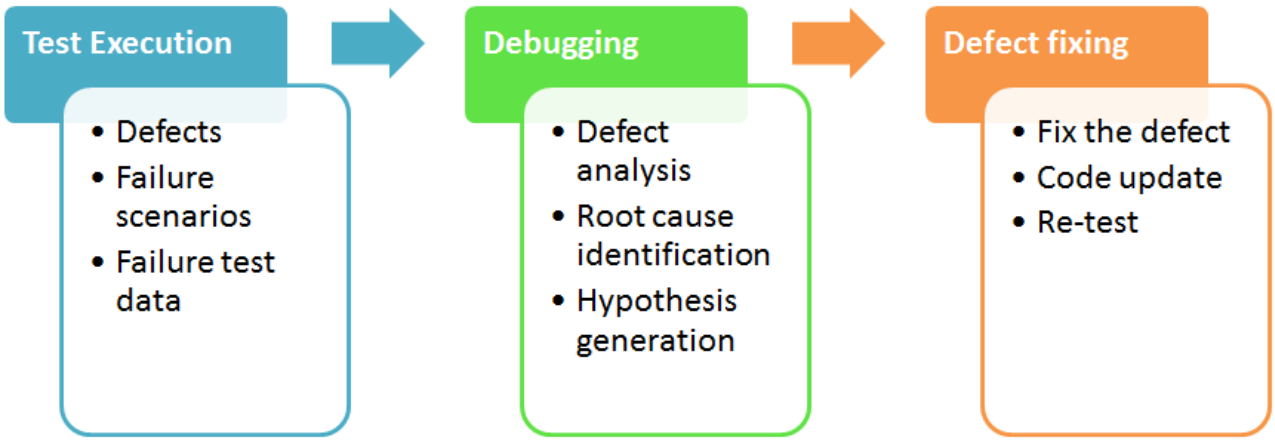
2.0 Objectives

After going through this unit, you will be able to:

- Understand various test scenarios;
- Know different debugging techniques;
- Understand different kinds of testing
- Knowledge of various kinds of testing tools, environment and
- Understand devising test cases and plans for various kinds of testing.

3.0 Debugging

Debugging is the act of analyzing and locating the root cause of defects/errors identified during 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 testing cycle is shown below along with activities in each step:



3.1 Debugging challenges

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

- **Triggering condition/variable isolation:** Few defects only get triggered only under specific conditions. Identifying the exact variable or condition would be complex as there is large number of variables. For instance a function call may fail only 10% of the times. There are number of variables like 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 to variety of reasons like intermittent network issues, dependency on another condition, hardware failure etc.
- **Hidden under multiple complex steps:** Often few defects only 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.

3.2 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 which involves different layers and components. Here we systematically eliminate one layer or component at a time starting with the layer/component which has most probability of the defect. For instance if there is an issue 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 definitely identify the exact set of variables which 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 is causing the issue.
- **Memory profiling:** This strategy is normally followed for memory related issues. A profiler tool is used to analyze the 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 analyzing 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 when the defect happens.
- **Call tracing:** This is a top-down approach wherein we trace the call from top-most component to the root data source. We examine the return value at each level to isolate the code which is causing the issue.

3.3 Scenarios for applying debugging strategies

Following table provides the scenarios which 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 which lead to the defect	• Use this technique when we know the data set and steps which lead to the defect
Memory profiling	Analyze and profile the memory during program execution	• Apply this technique when the defect is related to memory like 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 which 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 which his causing the defect

3.4 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.

4.0 Testing Tools & Environments

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

4.1 Testing tools

Testing tools help testing in varying ways: automatic creation of test cases, execution of test cases, debugging defects. Following table indicates various popular testing tools used. The tools are categorized 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 •

4.2 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.

Following table provides list of various testing environments, its purpose and the testing team responsible

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

5.0 Types of Test Cases

In this section we will look at various kinds of testing and test cases that will be designed for each kind of testing.

5.1 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

Table listing various approaches for testing

Testing Approach	Testing type
Box approach	• Black box testing: Tests the system without the knowledge of internal structure of the system • White box testing: Tests inner structure of program or component
Execution based approach	• Static testing: Involves code walkthroughs and other static methods like code inspection • Dynamic testing: Test 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 which impact release schedule
Manual/automated approach	• Manual testing: Testers manually execute test cases • Automated testing: Test case execution is automated

5.2 Types of testing methods

Following are main types of testing which 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 testing to verify if the code works as expected at low level • Requires knowledge and flow of the code structure • For instance, testing a method which does an addition of two integers by passing two positive values and asserting their sum • Usually done in development environment	• Developers
Integration Testing	• The testing involves various software components and interfacing systems to validate the interaction between them • Contains wide range of scope items like data flow testing, services testing, performance testing, exception handling testing etc. • Done in SIT environment	• QA team • Integration testing team
Functional testing	• The testing involves high-level design of functional modules/components to verify it is working as expected for the design specifications • Done in QA environment	• Quality Analysis (QA) Team • Functional testing team
System Testing	• This involves testing the entire testing in fully-integrated environment which is very similar to that of customer and testing the customer requirements • Testing is done to ensure compliance with design specifications, legal regulations • Non-functional requirements like scalability, usability will be tested	• Quality Analysis (QA) Team • Integration testing team • Security team
Performance testing	• Testing is done to check the compliance of system with specified performance guidelines. • Key performance metrics like page load time, process	• Performan ce testing team

	completion time will be tested	
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 the specified acceptance criteria like defect percentage, performance etc.	• QA Team
Regression testing	• This testing is done for incremental updates to software to ensure if the updates have not accidentally broken earlier functionality causing unintended errors. • Checks if the system still satisfies the required functionality	• QA team

5.3 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 explores further into the development of test cases for some of the prominent test types.

5.3.1 Test case for unit testing

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

- Positive flow for expected behavior
- 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.

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 which takes two integers and return 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) == 984822052691088389952);
   }
}
```

5.3.2 Test case for functional testing

Functional test cases should test the functionality of the component if they match the expectations of the designed specifications.

Usually functional test cases are based on the use cases and 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:

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

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	Fail
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	Fail
Login use case	Login_004	1. Click on “Forgot password” 2. Enter non-registered user id	Non-registered user should not receive email	Email was received	Fail

5.3.3 Test case 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 following steps:

1. Perform setup. This step includes establishing connection with the interface and initializing all necessary variables
2. Populate dummy data
3. Test the scenarios

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

   //Perform setup and initialization 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);
 }
}
```

5.4 Check your progress 2

1. The environment used for testing integration test cases is _____
2. The testing phase which covers security testing is _____
3. The testing that can be applied for sub-systems like ERP or database is called _____
4. The testing that is done normally on incremental software updates is called _____
5. The _____ testing approach assumes no knowledge of inner structure of the component.

6.0 Test Plans

Test plan is essentially a comprehensive planning artifact which covers 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

6.1 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

7.0 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 further looked 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 looked at the sample test plan with various sections.

7.1 Solutions/Answers

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. System testing
3. Integration testing
4. Regression testing
5. Black-box testing

8.0 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

Block-3 Unit-1: Software Development Models

Structure **Page no.**

1. Introduction

In the current scenario, information systems are important part of any organization. As compared to 1970's and 1980's, information systems are becoming more and more complex. In the early years before 1940 software was not an independent discipline. It was the extension of hardware. Programs were written in assembly language and were not large

in size due to complexity of the language. The responsibilities of the programmer were coding of the product, execution of code, testing of code and fix the errors if occurred in the software.

In the current scenario information systems become more and more complex and organization becomes more dependent on software. So it generates a need to develop the software in an organized manner. A survey was conducted by researchers in seventies and it was found that most of the software used by company was of poor quality. If the software developer leave the project in between than the new programmer start it from the beginning due to lack of documentation and it lead high cost of the project. Researchers found that software product was not same as hardware product as it was to be engineered or developed and it did not wear out. Most of the people consider the program and software to be same but software not only consists of program but also the supporting manuals. Some important characteristics of software are:-

- Software does not wear out.
- Software is not manufactured. It is developed or engineered.
- Software can be assembled from existing components called as custom built software.
- It is flexible and hence can easily accommodate changes.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- Difference between software and program.
- What is software engineering?
- Understand the evolution of software engineering;
- Understand the characteristics of software;
- Learn about different phases of software development life cycle, and
- Selecting a SDLC for a project.

1.2 Program vs. Software:-

Program is subset of software. We can define software as a collection of programs, related documents, and procedures to operate the software. For example GUI (Graphical User Interface) gives us the way to operate the software, documents are help files, from where we can learn to

operate the software, programs are installed in the PC such as software setup and it gives an icon on the screen of our PC's monitor.

1. Software is broad term that covers computer programs as well as the components that it needs to run while program is a term used to describe any code that is used to run a device.
2. Program exists before software.
3. Software typically consists of files while programs can be files.

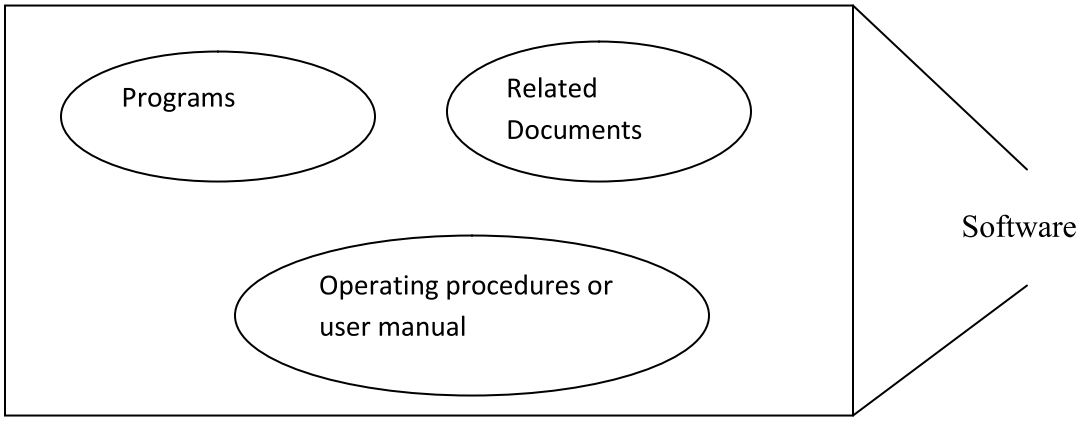


Fig1:- Relation between software & Program

1.3 What is software engineering?

Definition of software engineering:-

A number of definitions of software engineering are as follows:-

At the first conference on software engineering in 1968, Fritz Bauer defined software engineering as "The establishment and use of sound engineering principles in order to obtain economically developed software that is reliable and works efficiently on real machines".

Stephen Schach defined the same as "A discipline whose aim is the production of quality software, software that is delivered on time, within budget, and that satisfies its requirements".

According to (pfleeger 87) "Software engineering is a strategy for producing quality software."

According to IEEE standard 610.12-1990 software engineering is defined as

1. The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software i.e., the application of engineering to software.
2. The study of approach as in (1).

In short we can say if we want to develop software in an organized manner so that it can produce good quality product within time, within budget and without failure of the project then it is called as software engineering. It means we have divided our software development process in well defined phases. These phases are Requirement analysis, feasibility analysis, designing, analysis, coding, testing and operation & maintenance. The advantages of using software engineering for developing software are:-

- ❖ Improved requirement specification
- ❖ Improved quality
- ❖ Improved budget and schedule estimates.
- ❖ Use of automated tools and techniques.
- ❖ Less defects in final product
- ❖ Better post and pre maintenance of software.
- ❖ Well defined process
- ❖ Improved productivity
- ❖ Improved reliability

1.4 Software development life cycle (SDLC) models

1.4.1 Why do we need a model

The duration of time that begins with concepts and analysis of software being developed and ends after system is discarded after its usage, is denoted by software development life cycle i.e. SDLC. We have many different kinds of SDLC models. Each one is categorized based on its

- Requirements
- Feasibility analysis
- Time needed for completion of project
- Size of team (number of team members) in the project

1.4.2 Software Development Life Cycle Models

Depending upon the scope, complexity and magnitude of the project a particular life cycle model is selected and this selection of life cycle model significantly contributes towards the successful completion of the project. SDLS models are as follows:-

1.4.3. Build and fix model: - Techniques used in the initial years of software development resulted into the term Build and Fix. In this we build the project and check it. If there is any error then we have to fix it. This process will run repeatedly until we get the desired product or project.

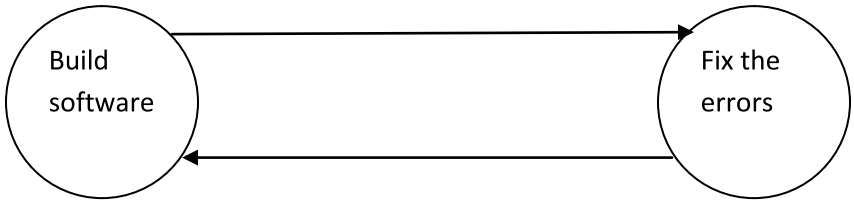


Fig. 2:- Build & fix Model

Advantages:-

- The model is useful only for small size projects.
- This is used for 100 to 150 lines project.

Disadvantages:-

- Models are not used for large projects.
- As requirements are not defined so product result will be full of errors.

- Repeating the process to build the project increases the cost.
- Maintenance of the product is difficult due to non-documentation.

1.4.4. Waterfall Model:-

It was proposed by Royce in 1970 as an alternative to **Build and fix model.** The main principal of this model is **"define before designing" and "design before coding".** This model requires complete and exact requirements before starting the project and this may not be possible in most of the cases. This model is mainly used to automate banking systems, student management systems etc., we know all the requirements of these kinds of systems because we are doing work manually for these systems from many years.

Phases of this model are as follows:-

a) Feasibility study
b) Requirement analysis and specification
c) System design
d) Implementation & Unit testing
e) Integration and system testing
f) Operation & Maintenance

a) Feasibility study: - It explores system requirements to determine project feasibility. All projects are feasible given unlimited resources and infinite time (Pressman 92). A feasibility study decides whether or not the proposed system is worthwhile. It can be categorized as follows:-

i. **Economic feasibility:-** If the system can be engineered using current technology and within budget
ii. **Technical Feasibility:-** If the system can be integrated with other systems that are used
iii. **Operational feasibility: -** If the system contributes to organisational objectives. In this we also see that current employees of the organization are able to learn new system or not, they may require some training on new system etc.

Some other feasibility studies are schedule feasibility, legal and contractual feasibility and political feasibility.

b) Requirement analysis and specification:-

This phase focuses on understanding the problem the problem domain and representing the requirements in a form which are understandable by all the stakeholders of the project i.e. analyst, user, programmer, tester etc. the output of this stage is document called Requirements Specification Document (RSD) or Software requirements Specification (SRS). All the successive stages of software life cycle are dependent on this stage as SRS produced here is used in all the other stages of the software lifecycle.

c) System design: - This phase translates the SRS into the design document which shows the overall structure of the program and the interaction between modules. The main focus of this phase is on the high level and low level design of the software. High level design describes the main components of software and their internals. Low level design focuses on the transforming the high level design into a more detailed level in terms of a algorithm used, data structure used etc.

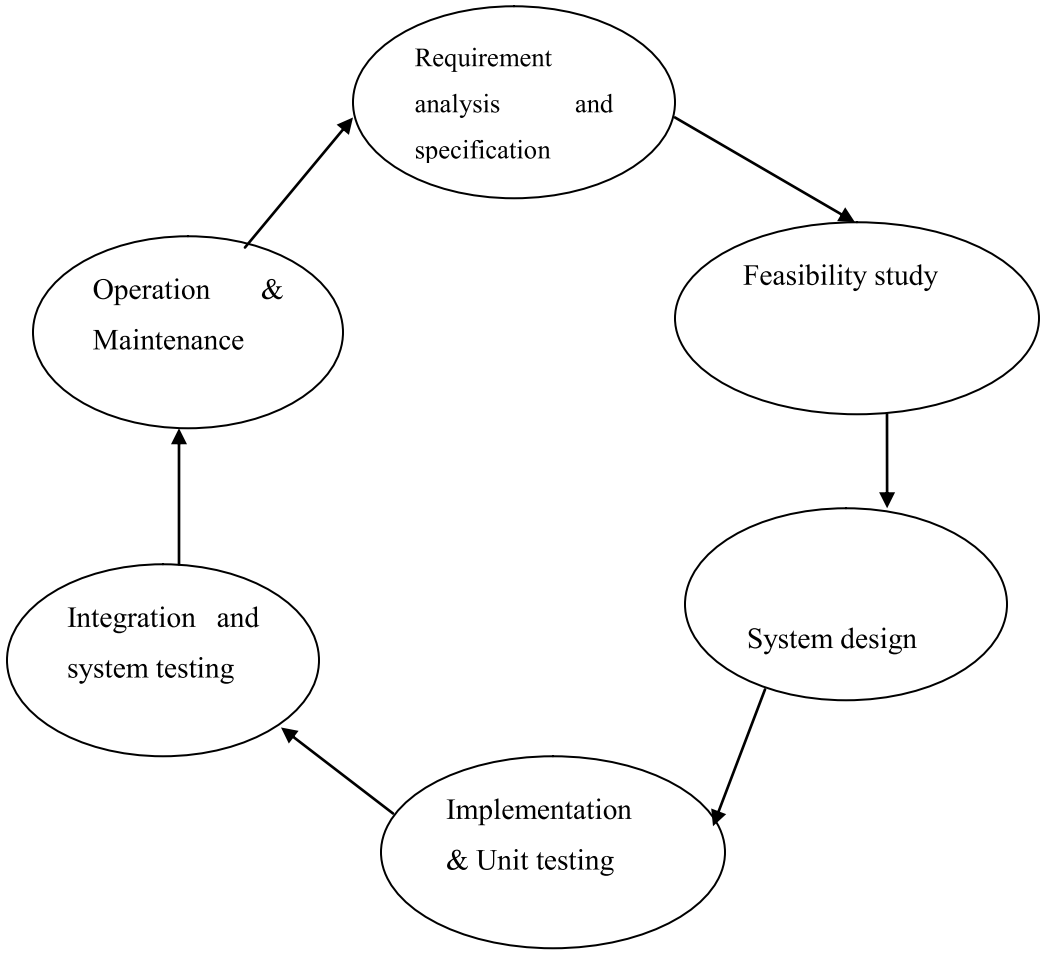


Fig.3:- Waterfall Model

d) **Implementation & Unit Testing: -** In this phase we have to transform the design into coding. For this we require programmer in the desired language. To do this task we can hire the programmer or we can recruit the programmer depend upon the size and requirement of the software development organization. After coding all the modules we have to test all the modules individually, this is called as unit testing.

e) **Integration & System testing: -** In this phase we integrate at least two modules at a time and test them for any error in terms of input error or output error or both. If there is any error then we have to fix them. After this process we integrate all the modules in form of a complete system and test the system for any discrepancy. We divide the whole project in modules so that we programmer can code individual module parallel. This process speedups the coding phase.

f) **Operation and maintenance:** - This phase resolves the software errors, failures etc. it enhances the requirements if required and modifies the functionality to meet the consumer needs. This is something which continues throughout the use of product by consumer.

Advantages:-

- Easy to understand
- Each phase has well defined requirement on terms of input and output.
- Easy to use as software development proceeds.
- Each phase has well defined milestones.

Disadvantages:-

- It is difficult to define all requirements at the beginning of a project.
- This model is not suitable for accommodating any change
- A working version of the system is not seen until late in the project's life
- It does not scale up well to large projects.
- Real projects are rarely sequential.
- The biggest drawback of this model is that it does not support iteration.

1.4.5. Prototype Model:-

This model is used when the developers do not have the clear requirement of customer or customers do not what will be the actual input or output of the project. According to Robert T. Futrell "Prototyping as a process of developing working replica of a system". This model is used to develop a dummy project and get the feedback from customer on the developed dummy project and modifications take place based on the feedback. This process will continue repeatedly until users accept the product. Two approaches of prototyping can be used:-

a) Throwaway prototyping: - This approach is used for developing the systems or part of the systems where the project team does not have the understanding of the system. The prototypes are built quickly, verified with the customer and if there is any change then throw it away and develop the modified product. In this we throw all the intermediary products.

b) Evolutionary prototyping: - This approach is used when development knows some requirements in advance. In this we maintain all the intermediary products. The prototypes are not throw away but evolved with time.

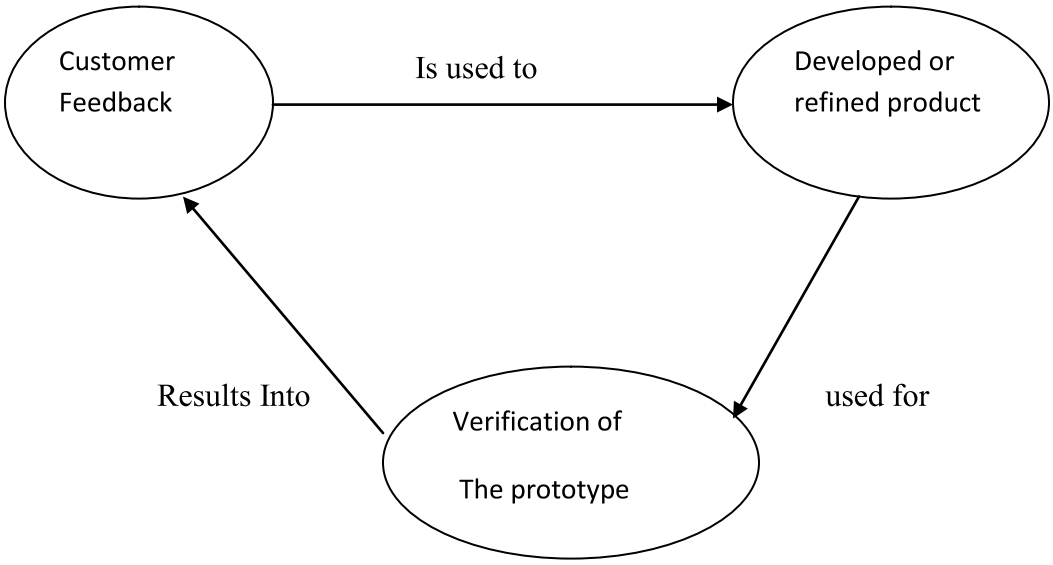


Fig.4. - Prototype Model

Advantages:-

- New requirements can be easily taken due to refinement of the product.
- A partial product is built in the initial stage that's why customer can see the product in early life cycle.
- Requirements become clearer resulting into n accurate product.
- A user from the customer is involved in testing all the time to give feedback.

- Flexibility in design and development is also supported by the model.

Disadvantages:-

- After seeing an early prototype users demand the actual system to be delivered soon.
- Developers in a hurry to build prototypes may end up with suboptimal solutions.
- If user is not satisfied with initial prototype than user may lose interest in the project.
- Poor documentation.

1.4.6. The Incremental SDLC Model:-

They are effective in the situations where requirements are defined precisely and there is no confusion about the functionality of the final product. After every cycle a useable product is given to the customer. Model is popular particularly when we have to quickly deliver a limited functionality system. In this we divide all the requirements in small sets. Each and every set is developed as waterfall model. The only difference between waterfall and incremental model is that in waterfall we have only one set of requirements while in incremental we have many set of requirements.

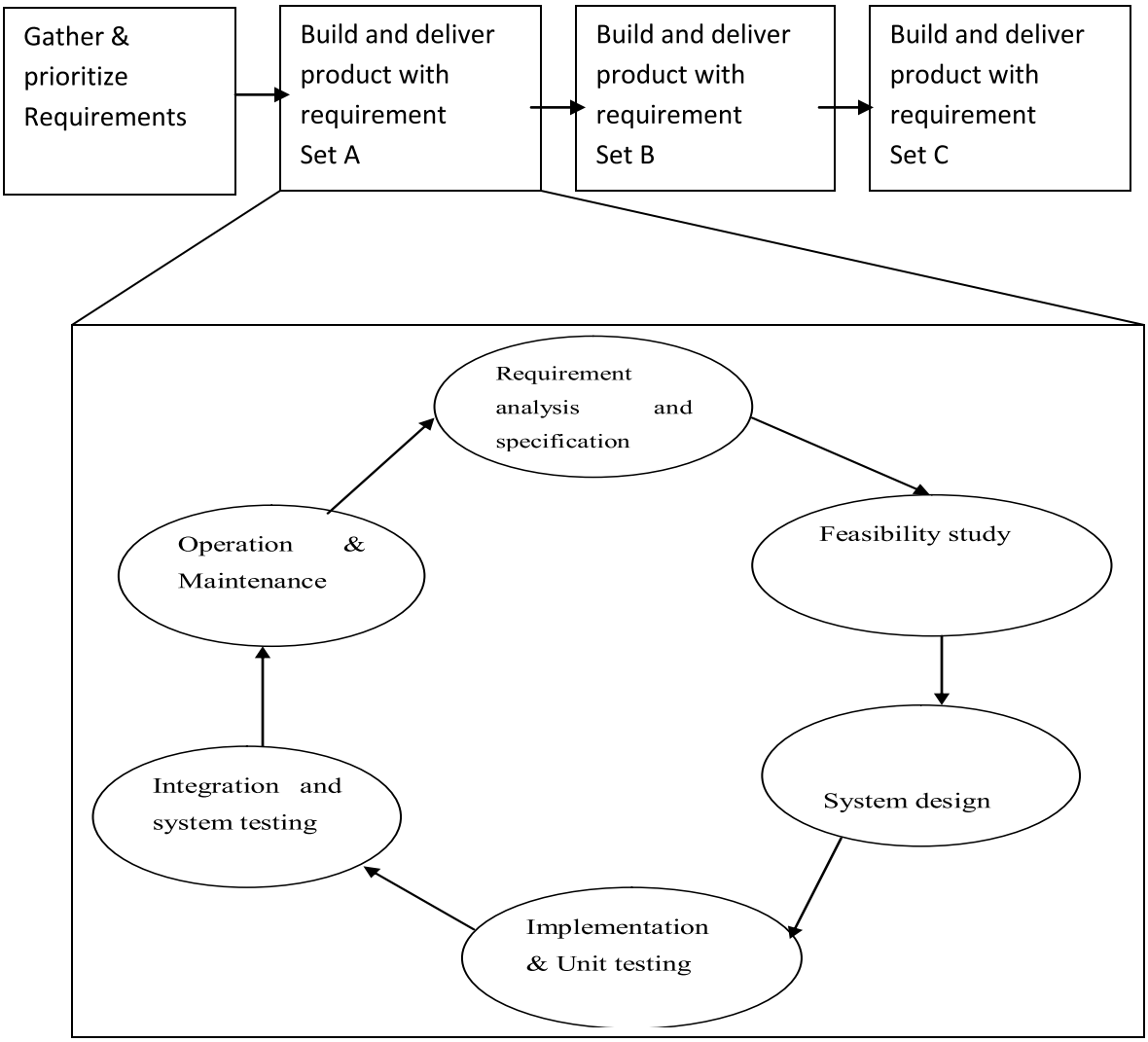


Fig. 5:- Incremental Model

Advantages:-

- As product is to be delivered in parts, total cost of the project is distributed.
- Limited number of staff is required due small set of requirements.
- As development activities for next release and use of early version of product is done simultaneously, if found errors can be corrected.
- Testing becomes easy due to small modules.
- Risk of failure is decreased because users can give feedback since first delivery of product.

Disadvantages:-

- As product is delivered in parts, total development cost will be high.
- Users can change their requirements at any instance of time because they know that complete product is not delivered till now.
- Well defined interfaces are required to connect modules.
- Testing will also increase the cost because we test all modules again and again until we delivered complete product.

1.4.7. The spiral Model:-

It is an evolutionary process model. This model was proposed by Bohem in 1988. This is basically used for large size of project due to larger risk of failure. This model use prototyping to minimize the risk.

The radial coordinate in the diagram represents the total cost involved till date. Each loop of the spiral represents phases of the development.

In the first quadrant, objectives, alternatives means to develop product and constraints imposed on the product are identified.

The next quadrant identifies the risk and resolves the risks.

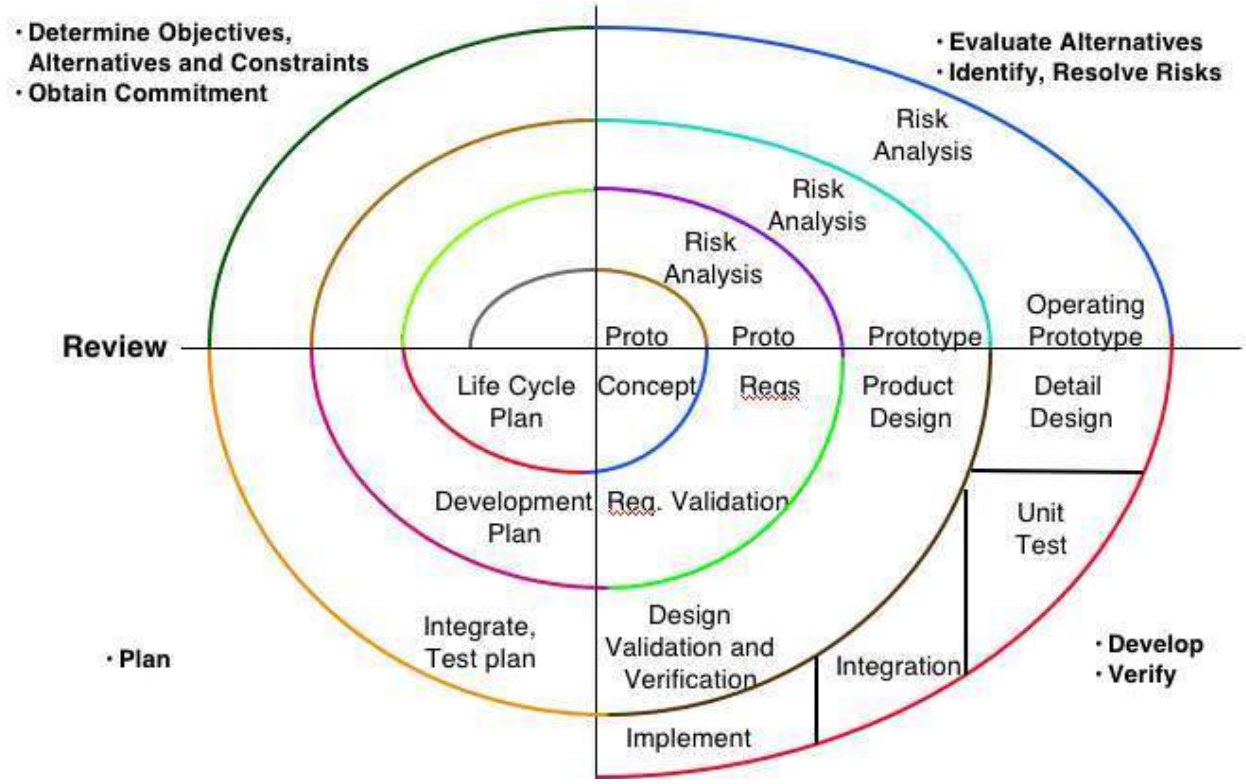


Fig 6:- Spiral Model (From software engineering by K.K. Agarwal & Yogesh Singh)

Third quadrant represents the waterfall model consisting of activities like design, testing etc.

Forth quadrant shows that after every phase customer evaluates the product, requirements are further refined and so is the product.

Each and every spiral is also analyzed and verified by the experts i.e. software analyst.

This model is mainly used for general product i.e. not related for any specific organization.

Advantages:-

- The model tries to resolve all possible risks involved in the project starting with the highest risk.

- Users are involved in every phase of model.
- With each phase as product is refined after user's feedback, the model ensures a good quality product.
- The model makes use of techniques like reuse, prototype and component based design.

Disadvantages:-

- This model requires more experienced developers so the total cost of the project is very high.
- The model requires expertise in risk management and excellent management skills.
- Risk analyses also increase the cost of project.
- Different persons involved in the project may find it complex to use.

1.4.8 The Rapid Application Development (RAD) Model:-

This model was proposed by IBM in 1980s and later introduced to software community by James Martin.

The important characteristic of RAD model is increased involvement of the user at all phases of SDLC through the use of dominant development gizmo. In this model software development company require a person who can identify the requirements and group them based on its attributes. Then the individual sets of requirements are developed in the form software by different programmers or by the team of programmers depend upon the size of project. This model comes in action when a user want product within two or three months. Due to time constraints this model requires different teams of programmers.

The RAD model consists of following four phases:-

a) **Requirements Planning: -** This focus on collecting requirements using brainstorming, interview, questionnaires and by observation methods.
b) **User Description: -** Requirements are detailed by taking users feedback by building prototype using development tools.
c) **Construction: -** The prototype is refined to build the product and released to the customer.
d) **Cutover: -** It involves acceptance testing by the user and their training.

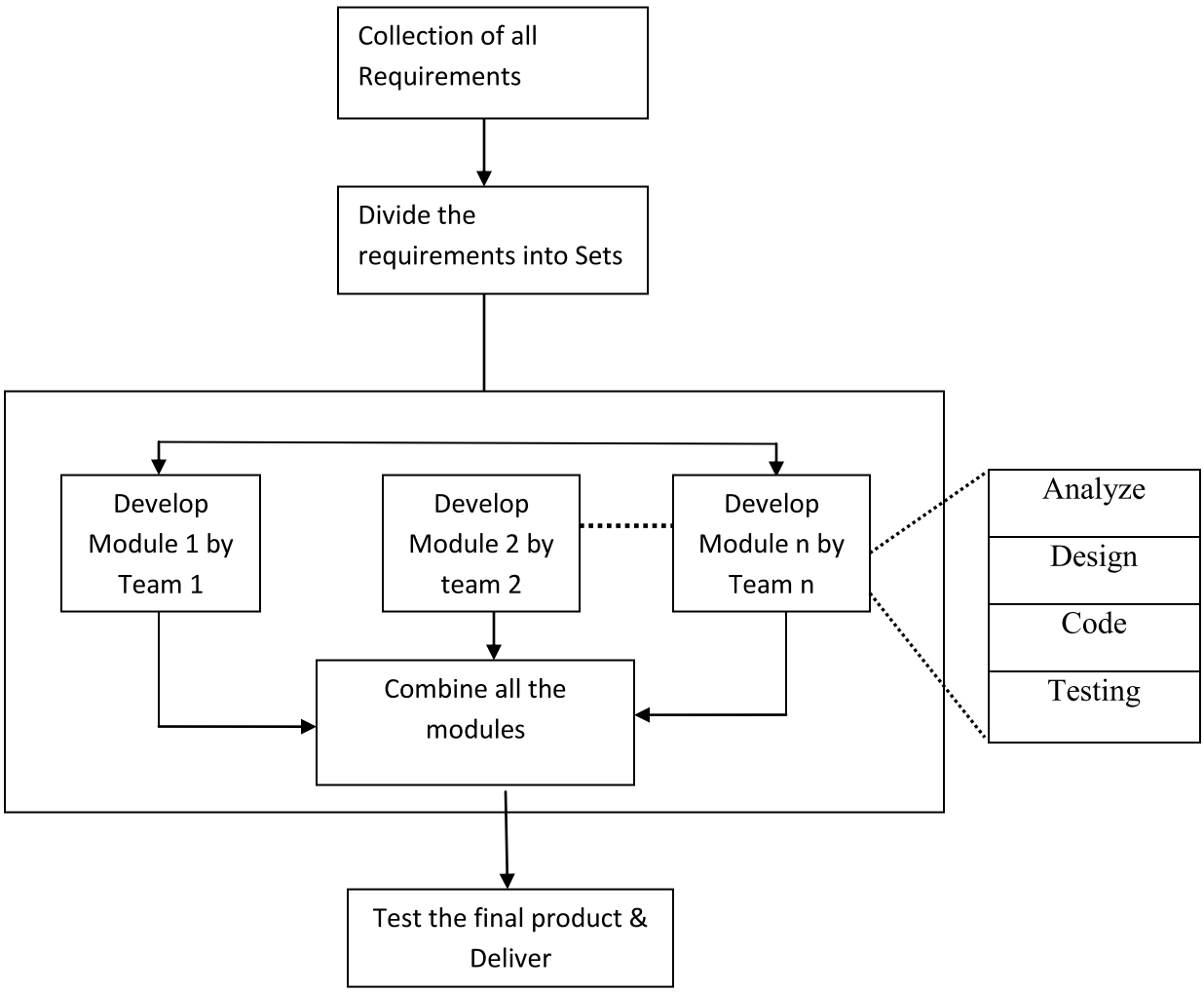


Fig. 7:- RAD Model

Advantages:-

- As customer is involved at all stages of development so it leads to a product achieving customer satisfaction.
- Usage of powerful development tools results into reduced SDLC time.
- Feedback from customer is available at all the stages without any delay due to customer involvement in the project.

Disadvantages:-

- The model uses the powerful development tools so it requires experts to run the tools.
- Team leader must work closely with developers and customers/users to complete the project in time.
- Absence of reusable components can lead to failure of the project.

1.5. Comparisons of software process models:-

Characteristics	Waterfall Model	Incremental Model	Spiral Model	Prototype Model	RAD Model
1. Well defined requirements	Yes	NO	NO	NO	Yes
2. Domain Knowledge of team members	Adequate	Adequate	Very Less	Very Less	Adequate
3. Expertise of users in problem Domain	Very Less	Adequate	Very Less	Adequate	Adequate
4. Availability of reusable components	No	No	Yes	Yes	Yes
5. Users involvement in all phases of SDLC	No	No	No	Yes	Yes
6. Complexity of system	Simple	Complex	Complex	Complex	Medium

1.6 Summary:-

- Softwar e worth billion and trillions of dollars has gone waste in the past due to lack of proper techniques used for developing software resulting into software crisis.
- Softwar e engineering is application of engineering principles to develop quality software.
- The duration of time that begins with conceptualization of software being developed and ends after system is discarded after its usage is called SDLC.
- A number of software process models are proposed to organize software engineering approaches into phases.
- Popular models are e waterfall, incremental, spiral etc.

1.7 Exercise

1) Indicate various problems related with software development.
2) Give a comparative analysis of various types of software process models.
3) What are various phases of software development life cycle?
4) Define software engineering.
5) List the impact of software engineering on developing software.

1.8 Further Readings:-

1) Software Engineering: A Practitioner's Approach by Roger S. Pressman from Addison-Wesley Longman Publishing Co.
2) Software engineering 3rd edition by K.K. Agarwal and Yogesh singh form new age international.

UNIT 2 SOFTWARE PROJECT MANAGEMENT

Structure **Page Nos.**

2.0 Introduction

In today's world, Software Project Management is an Art. It is "the process of planning, organizing, staffing, monitoring, controlling and leading a software project". Another definition for SPM (Software Project Management) is "the application of knowledge, skills, tools, and techniques to project activities to meet project requirements". Every software project must have a manager who leads the development team and is the interface with the initiators, suppliers and senior management. Software development is a complex process involving such activities as domain analysis, requirements specification, communication with the customers and end-users, designing and producing different artifacts, adopting new paradigms and technologies, evaluating and testing software products, installing and maintaining the application at the end-user's site, providing customer support, organizing end-user's training, visualizing potential upgrades and negotiating about them with the customers, and many more.

In order to keep everything under control, eliminate delays, always stay within the budget, and prevent project runaways, i.e. situations in which cost and time exceed what was planned, software project manager must exercise control and guidance over the development team throughout the project's lifecycle. In doing so, they apply a number of tools of both economic and managerial nature. The first category of tools includes budgeting, periodic budget monitoring, user chargeback mechanism, continuous cost/benefit analysis, and budget deviation analysis. The managerial toolbox includes both long-range and short-term planning, schedule monitoring, feasibility analysis, software quality assurance, organizing project steering committees, and the like. Software Project Management has to address many challenges such as resources, security, defects etc as given in Figure1.

FIGURE 1: The set of problems to be addressed by software project management

The skillful integration of software technology, economics and human relations in the specific context of a software project is not an easy task. The software project is a highly people-intensive effort that spans a very lengthy period, with fundamental implications on the work and performance of many different classes of people. The problems related to software project, if not handled properly, will result into poor quality, loss of reputation, failure to deliver etc as shown in Figure 2.

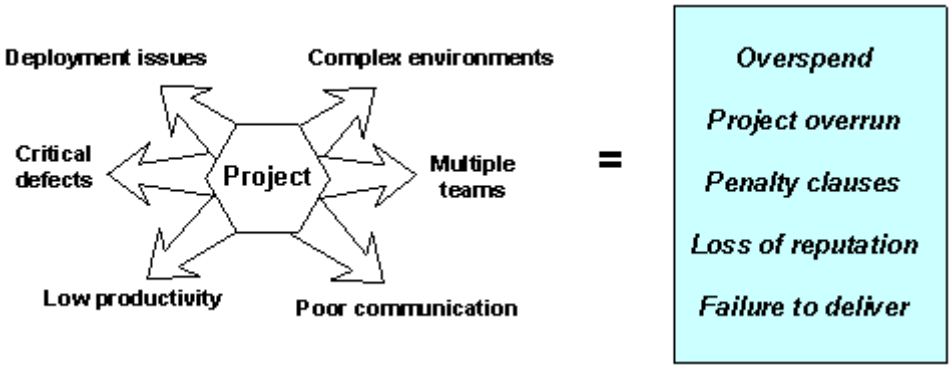


FIGURE 2: Problems of Software project

The project triangle (Figure 3) is also known as the "iron triangle" and, less poetically, the "triple constraints." Whatever you call it, it amounts to the same thing: You can't change a project's budget, schedule, or scope without affecting at least one of the other two parts. Balancing the top three project management areas of project scope, project time and project cost, is one of the biggest tasks of the project manager. Basically, if you pull one end of the triangle, one of the other ends needs to be "pulled"/extended to restore balance. On the other hand, if you "push" on one end, another end will need to be "pushed"/reduced OR "pulled"/extended in order to bring the triangle back into balance. Decisions affecting time, scope and/or cost must be carefully considered as they are likely to have implications on overall project quality, and customer satisfaction.

Figure 3: The Project Triangle

2.1 Objective

After going through this unit, you will be able to:

- Understand the various aspects of project management like planning, monitoring, controlling and execution.
- Understand the reasons for major software project failure.
- Understand the importance of Software Metrics and also various metrics for project monitoring and control.
- Finally, the use of PERT and Gantt charts in project management.

2.2 Planning of Software Projects

A project plan should be based on the project requirements and developed estimates, and is a formal, consistent and approved document that provides for the essential project guidelines. Generally, the purpose of the document is to support management and control of the project execution. The plan should cover all phases of the project and should ensure that all involved plans are consistent with each other.

There is no single reason for a software engineering project to have a project plan. A project plan usually contains several parts produced in order to help the project team with their project. The main objectives with a project plan are the following:

- Guide project execution
- Document project planning assumption
- Document project planning decisions regarding alternatives chosen
- Facilitate communication among stakeholders
- Define key management reviews as to content, extent, and timing
- Provide a baseline for progress measurement and project control

The project plan is generally developed in the initial phase of the project and needs to be reviewed and agreed upon by all concerned persons. However, the plan is expected to change

over time and is updated each time the actual progress differs from the plan or when project conditions changes, which require new approaches. A carefully prepared project plan if properly followed and committed to, should lead to a successful project and eliminate many of the pitfalls inherent in the project management process. It provides leadership vision and facilitates for management to utilize available resources efficiently.

2.2.1 Different types of project plans

In addition to the main software project plan, different types of specific plans may be developed to support the main plan in different areas. Example of such plans may be the following:

1) Quality plan - Includes the quality procedures and standards that concern the project.
2) Validation plan - Covers approaches, resources and schedule involved in the system validation.
3) Configuration management plan – Consists of the configuration management procedures and structures to support the project.
4) Maintenance plan - Predicts the maintenance requirements, maintenance costs and the effort required.
5) Staff development plan - Includes how available skills and experience will be developed.

2.2.2 An overview of the 'Step Wise' framework

Step 0: Select project
Step 1: Establish project scope and objectives

- Identify objectives and measures of effectiveness in meeting them.
  What are we trying to achieve? How do we know if we have succeeded?
- Establish of a project authority.
  Where is the overall authority for the project?'
- Identify all stakeholders in the project and their interests.
  Who will be affected by the project and who will need to be involved?'
- Modify objectives in the light of stakeholder analysis.
  Do we need to do things to win the commitment of stakeholders?
- Establish of methods of communications with all parties.
  How do we keep in contact?'

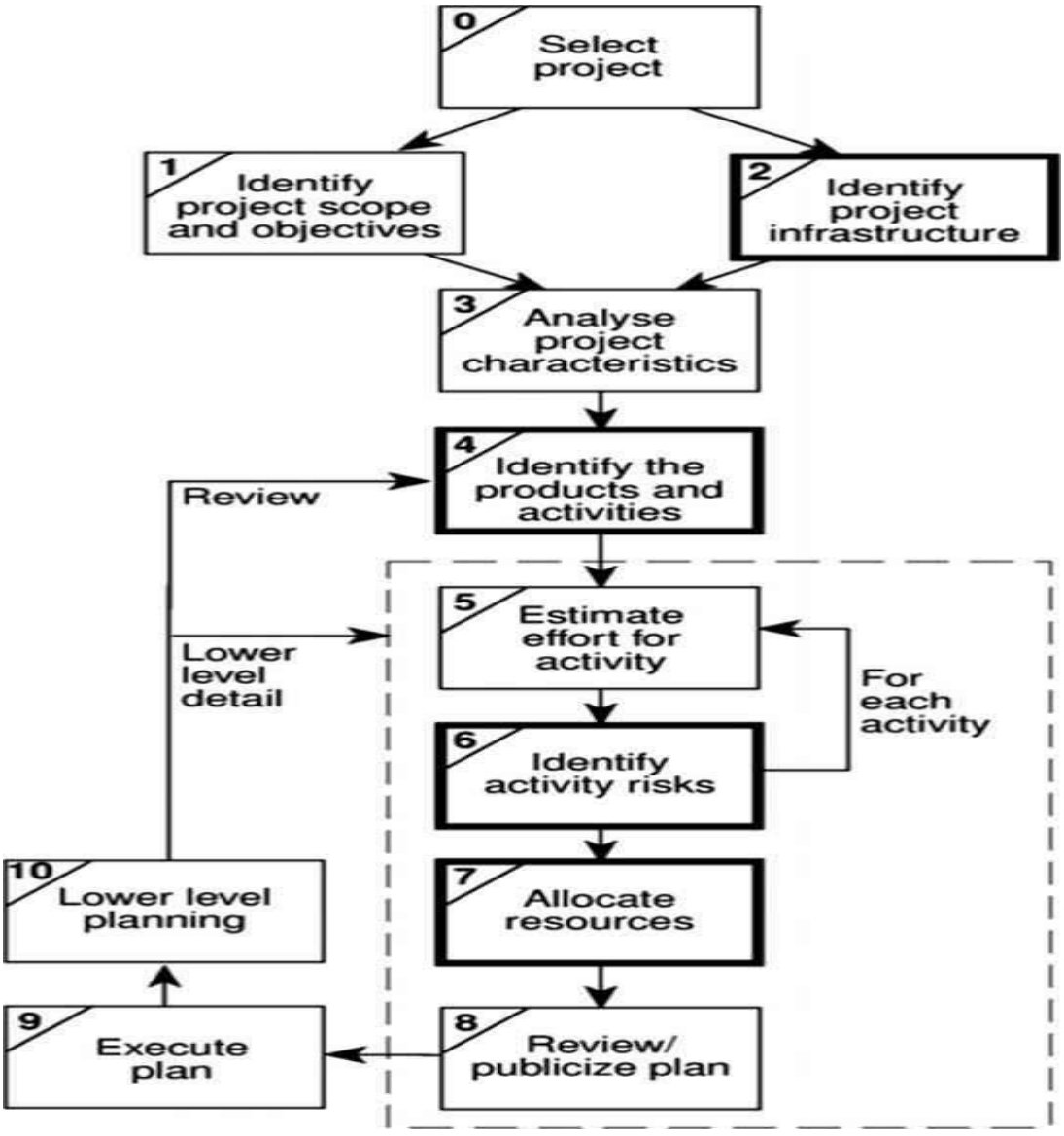


Figure 4: The Stepwise Framework

Step 2: Establish project infrastructure

- Establish relationship between project and strategic planning.

Why do they want this project?'

- Identify Installation standards and procedures.
  What standards do we have to follow?
- Identify project team organization
  What is the organizational framework?

Step 3: Analysis of project characteristics

- Distinguish the project as either objective or product based.
  Is there more than one way of achieving success?
- Analyze other project characteristics (including quality based ones).
  What is different about this project?
- Identify high level project risks.
  What could go wrong?
- Take into account user requirements concerning implementation.
- Select general life cycle approach.
  What is the best approach? Increments? Prototypes? Waterfall?
- Review overall resource estimates.
  Does all this make us change our mind about probable costs?

Step 4: Identify project products and activities

- Identify and describe project products (including quality criteria).
  What sort of things do we have to create?
- Document generic product flows.
  In what order do we produce them?
- Recognize product instances.
  Can we identify particular individual cases of each type of product? What modules do we have to code?
- Produce ideal Activity Network.
- Modify ideal to take into account need for stages and checkpoints.

Step 5: Estimate effort for each activity

- Estimate effort for activity.
  How long will this activity take?
- Revise plan to create controllable activities.
  Do we need to combine activities or spiit them to get to a convenient size?

Step 6: Identify activity risks

- Identify and quantify activity-based risks.
  What could go wrong with this activity?
- Plan risk reduction and contingency measures where appropriate.
  How can we stop it going wrong? What do we do if it goes wrong anyway?
- Adjust plans and estimates to take into account risks.

Step 7: Allocate resources

- Identify and allocate resources.
  What resources do we need for this activity?
- Revise plans and estimates to take to account resource constraints.
  When will the resources be available?

Step 8: Review/publicize plan

- Review quality aspects of project plan.
- Complete/review documentation of plan.
- Publicize plan and obtain agreement of parties to project.

Step 9/10: Execute plan/ lower levels of planning

This may quire the reiteration of the planning process at a lower level.

2.3 Execution of Software Projects

A project development methodology is more or less the same for every software development project. Each part or point of the methodology is a unique approach to the overall development. The technique comprises of a few vital steps which are defined and detailed below:

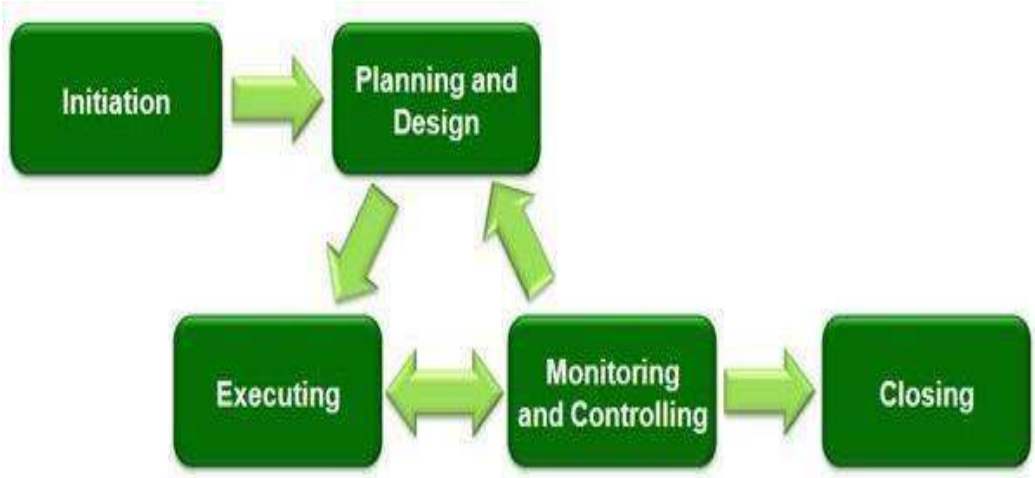


Figure 5: Software Project Management life cycle

2.3.1 Initiation

Project analysis is a most critical phase of web or software development project execution. The better a project is analyzed, the better it is executed. The analysis report serves as the base of the entire execution. The client's requirements, the project goals, the future scope of the project, the available resources, the required skills, the required tools, the available technology, the investment, the pricing and the time span are the subjects of analysis at this phase. Project analysts in cooperation with project managers perform this task ensuring attention to details. Risks and challenges, associated with the project, are analyzed too.

2.3.2 Planning

Once the delivery deadline is fixed up based on the project analysis report, the project plan is chalked out. Planning is a critical requirement to decide on the mode and methodology of executing the project. The better is the planning; the more efficient is the project execution. Of the several web or software development models such as Agile Model, Spiral Model, Iterative Model and Waterfall Model, the most suitable one is selected in keeping with the complexity level of the project. According to the plan, the tasks at different phases of the execution are allocated to the development team, divided into different units. If needed, a special team is built of professionals having the technological skills required for the project.

2.3.3 Execution

Project execution is the most crucial phase which demands for attention and dedication on part of developers. The selected project development model is deployed in this process. With continuous flow of the project execution phases, the methodology is followed to the point. Once a unit of the team is done with the development part assigned to it, the project is passed to the next unit for the next part of the development. The project manager leading the team acts a project coordinator too. He ensures precise coordination among the units of the team to get the project executed timely as it as planned. This phase is also referred to as construction. Quality project management plays instrumental in it.

2.3.4 Testing

This is the pre-delivery phase of the project execution cycle. The developed project is tested against a set of quality parameters before delivery. Testing is conducted through the application of a cutting-edge technology. The functionality of each component of the developed project is tested on multiple various levels. Then, the inter-functionality of the integrated components of the software is checked through integration testing. If any functional or technical issue is noticed, it is resolved to ensure the precision and accuracy of the software.

2.3.5 Acceptance and delivery

If the outcome is positive on completion of the testing process, the developed software is deployed as a beta version to be used by key users. The client is the chief user. Based on the user experience, it is determined if the software needs further development. Then, the user documentation is prepared defining the functionality of each of the components integrated to the framework. The documentation helps with the seamless installation, use and maintenance of the software. The project cycle comes to an end, and the end-product is delivered finally.

2.4 Monitoring of Software Projects

Monitoring means–collecting, recording, and reporting information concerning project performance that project manager and others wish to know.

Controlling means –uses data from monitor activity to bring actual performance to planned performance.

We monitor the project simply because we know that things don't always go according to plan (no matter how much we prepare) and to detect and react appropriately to deviations and changes to plans. We monitor the following:

Men (human resources), Machines, Materials, Money, Space, Time, Tasks and Quality/Technical Performance.

When do we monitor? The answer is "AT End of the project or continuously or logically or While there is still time to react or As soon as possible or At task completion or At pre-planned decision points (milestones)".

How do we monitor? The answer to this question is

- Through meetings with clients, parties involved in project (Contractor, supplier,etc.)
- For schedule –Update CPA, PERT Charts, Update Gantt Charts
- Using Earned Value Analysis
- Calculate Critical Ratios
- Milestones
- Reports
- Tests and inspections
- Delivery or staggered delivery
- PMIS (Project Management Info Sys) Updating

The following issues related to monitoring are discussed in the meetings.

- What problems do you have and what is being done to correct them?
- What problems do you anticipate in the future?
- Do you need any resources you do not yet have?
- Do you need information you do not have yet?

- Do you know anything that will give you schedule difficulties?
- Any possibility your task will finish early/late?
- Will your task be completed under/over/on budget?

2.5 Controlling of software projects

Projects are very dynamic – new requirements are discovered leading to changes to project scope, deliverables fail to meet quality standards, funding for the project gets cut, staff turnover, and other changes will necessitate a way of controlling all of these dynamics. Change control is one of the biggest challenges confronting a project since change often increases the risk, costs, and duration of the project. A centralized control process will be needed to manage changes to scope, schedule and budgets. Depending upon how significant the change is, there will be different levels of authority for approving the change.

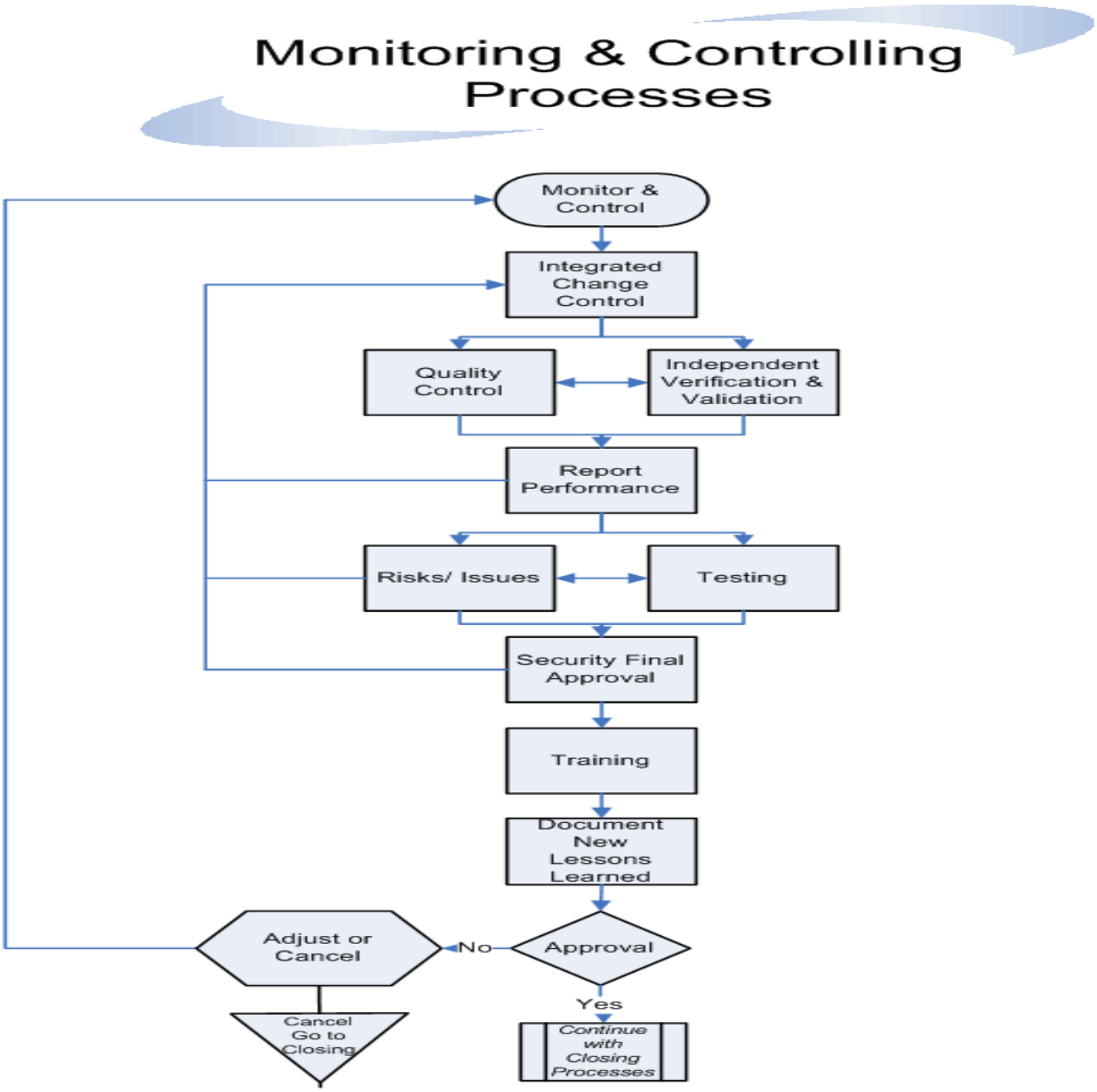


Figure 6: Project Control Cycle

In addition to internal project changes (scope, schedule and budget), projects usually are confronted with external change. External changes are much more difficult to control since the Project Team has little control over the process. Regardless, you will need to address both internal and external changes:

1. Recognize different changes that require control.
2. Establish control forms, communication channels, and other controls to capture and review the change.
3. Determine the impact of the change and communicate the change to those affected.
4. Where appropriate, have approvals and documentation procedures for managing the change.

5. Process and integrate the change into the project, such as changes to the Performance Measurement Baseline.
6. Verify that the change was in fact processed.

Projects often attempt to centralize the change control process by using a Change Control Log to document the date of the change, who originated the change, status (approved, open, canceled, etc.), amount of the change if applicable, highest WBS levels impacted, type of change and other important elements. In the case of technology related projects, this may take the form of Configuration Management. Configuration Management is a process for managing all of the components and work products that go into building a system. This includes the life cycle of managing the system, from determining the requirements to maintaining the system. Additionally, most systems tend to "evolve" over time and you need a formal model to manage these changes, such as:

1. Submit – Changes are submitted and processed through a standard form or screen.

2. Evaluate – Analyze the change and work back upstream in the life cycle to determine how requirements must change. Decide how the change should be handled.

3. Approve – Authority for changes decisions often follow threshold impacts and the ultimate authority for a change usually resides with the customer.

4. Implement – Develop a plan to implement the change, test the change, and report the successful implementation of the change.

5. Test – Run tests to make sure the change worked as expected and no new changes pop-up as a result of this change.

6. Update – Once you know the change is correct, you need to update your production related elements. This might include technical and user training documents. It might also involve software applications. Version control numbers are used for hard documents and release control numbers are used for software programs.

Projects will also impose enter and exit criteria during the project life cycle. These tollgates keep things on track and put discipline behind overall project management.

2.6 Software Projects

A failure is defined as any software project with severe cost or schedule overruns, quality problems, or that suffers outright cancellation. It has been suggested that there is more than one reason for a software development project to fail. However, most of the literature that discusses project failure tends to be rather general, supplying us with lists of risk and failure factors, and focusing on the negative business effects of the failure. Very little research has attempted an in-

depth investigation of a number of failed projects to identify exactly what are the factors behind the failure. While there does not appear to be any overarching set of failure factors we discovered that all of the projects suffered from poor project management. Most projects additionally suffered from organizational factors outside the project manager's control.

The following are the list of the major software project failure factors:

• Organizational structure,

• Unrealistic or unarticulated goals,

• Software that fails to meet the real business needs,

• Badly defined system requirements, user requirements and requirements specification,

• The project management process, poor project management,

• Software development methodologies, sloppy development practices,

• Scheduling and project budget,

• Inaccurate estimates of needed resources,

• Poor reporting of the project status,

• Inability to handle project complexity,

• Unmanaged risks,

• Poor communication among customers, developers and users,

• Use of immature technology,

• Stakeholder politics,

• Commercial pressures,

• Customer satisfaction,

• Product quality,

• Leadership, upper management support,

• Personality conflicts,

• Business processes and resources,

• Poor, or no tracking tools.

❖ Check Your Progress 1

1) Good software planning can eliminate all interactions. True/False
2) _____ are used to stop and measure the status of software projects.
3) A fault occurs when a program misbehaves**.** True/False

2.7 Software Metrics

Software process and project metrics are quantitative measures that enable software engineers to gain insight into the efficiency of the software process and the projects conducted using the process framework. In software project management, we are primarily concerned with productivity and quality metrics. There are four reasons for measuring software processes, products, and resources (to characterize, to evaluate, to predict, and to improve).

Measures and Metrics

• Measure - provides a quantitative indication of the size of some product or process attribute.
• Measurement - is the act of obtaining a measure.
• Metric - is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.

Process Indicators

• Metrics should be collected so that process and product indicators can be ascertained
• Process indicators enable software project managers to: assess project status, track potential risks, detect problem area early, adjust workflow or tasks, and evaluate team ability to control product quality.

Process Metrics

• Private process metrics (e.g., defect rates by individual or module) are only known to by the individual or team concerned.
• Public process metrics enable organizations to make strategic changes to improve the software process.
• Metrics should not be used to evaluate the performance of individuals.
• Statistical software process improvement helps and organization to discover where they are strong and where they are weak.

Statistical Process Control

1. Errors are categorized by their origin.
2. Record cost to correct each error and defect.
3. Count number of errors and defects in each category.
4. Overall cost of errors and defects computed for each category.
5. Identify category with greatest cost to organization.

6. Develop plans to eliminate the most costly class of errors and defects or at least reduce their frequency.

Project Metrics

• A software team can use software project metrics to adapt project workflow and technical activities.
• Project metrics are used to avoid development schedule delays, to mitigate potential risks, and to assess product quality on an on-going basis.
• Every project should measure its inputs (resources), outputs (deliverables), and results (effectiveness of deliverables).

Software Measurement

• Direct measures of a software engineering process include cost and effort.
• Direct measures of the product include lines of code (LOC), execution speed, memory size, defects reported over some time period.
• Indirect product measures examine the quality of the software product itself (e.g., functionality, complexity, efficiency, reliability, maintainability).

Size-Oriented Metrics

• Derived by normalizing (dividing) any direct measure (e.g., defects or human effort) associated with the product or project by LOC.
• Size-oriented metrics are widely used but their validity and applicability is a matter of some debate.

Function-Oriented Metrics

• Function points are computed from direct measures of the information domain of a business software application and assessment of its complexity.
• Once computed function points are used like LOC to normalize measures for software productivity, quality, and other attributes.
• The relationship of LOC and function points depends on the language used to implement the software.

Object-Oriented Metrics

• Number of scenario scripts (NSS).
• Number of key classes (NKC).

• Number of support classes (e.g., UI classes, database access classes, computations classes, etc.).
• Average number of support classes per key class.
• Number of subsystems (NSUB).

Web Engineering Project Metrics

• Number of static Web pages (Nsp).
• Number of dynamic Web pages (Ndp) • Customization index: C = Nsp / (Ndp + Nsp).
• Number of internal page links.
• Number of persistent data objects.
• Number of external systems interfaced.
• Number of static content objects.
• Number of dynamic content objects.
• Number of executable functions.

Software Quality Metrics

• Factors assessing software quality come from three distinct points of view
(Product operation, product revision, product modification).
• Software quality factors requiring measures include
o correctness (defects per KLOC)
o maintainability (mean time to change)
o integrity (threat and security)
o usability (easy to learn, easy to use, productivity increase, user attitude)
• Defect removal efficiency (DRE) is a measure of the filtering ability of the quality assurance and control activities as they are applied throughout the process framework
DRE = E / (E + D)
E = number of errors found before delivery of work product.
D = number of defects found after work product delivery.

Integrating Metrics with Software Process

• Many software developers do not collect measures.
• Without measurement it is impossible to determine whether a process is improving or not.
• Baseline metrics data should be collected from a large, representative sampling of past software projects.
• Getting this historic project data is very difficult, if the previous developers did not collect data in an on-going manner.

Metrics for Small Organizations

• Most software organizations have fewer than 20 software engineers.
• Best advice is to choose simple metrics that provide value to the organization and don't require a lot of effort to collect.
• Even small groups can expect a significant return on the investment required to collect metrics, if this activity leads to process improvement.

Establishing a Software Metrics Program

1. Identify business goal.
2. Identify what you want to know.
3. Identify subgoals.
4. Identify subgoal entities and attributes.
5. Formalize measurement goals.
6. Identify quantifiable questions and indicators related to subgoals.
7. Identify data elements needed to be collected to construct the indicators.
8. Define measures to be used and create operational definitions for them.
9. Identify actions needed to implement the measures.
10. Prepare a plan to implement the measures.

2.8 Application of PERT chart

Program Evaluation and Review Technique (PERT) is a scheduling method originally designed to plan a manufacturing project by employing a network of interrelated activities, coordinating optimum cost and time criteria. PERT emphasizes the relationship between the time each activity takes, the costs associated with each phase, and the resulting time and cost for the anticipated completion of the entire project.

PERT is an integrated project management system. These systems were designed to manage the complexities of major manufacturing projects, the extensive data necessary for such industrial efforts, and the time deadlines created by defense industry projects. Most of these management systems developed following World War II, and each has its advantages.

PERT was first developed in 1958 by the U.S. Navy Special Projects Office on the Polaris missile system. Existing integrated planning on such a large scale was deemed inadequate, so the Navy pulled in the Lockheed Aircraft Corporation and the management consulting firm of Booz, Allen, and Hamilton. Traditional techniques such as line of balance, Gantt charts, and other systems were eliminated, and PERT evolved as a means to deal with the varied time periods it takes to finish the critical activities of an overall project.

The line of balance (LOB) management control technique collected, measured, and analyzed data to show the progress, status, and timing of production projects. It was introduced at

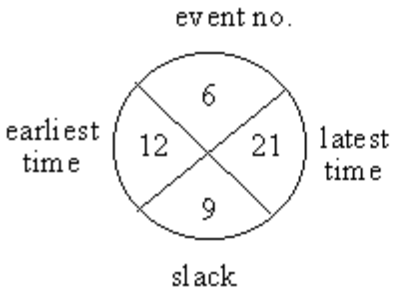
Goodyear Tire and Rubber Company in 1941 and fully utilized during World War II in the defense industry. Even older is the Gantt chart, developed during World War I by Harvey Gantt, a pioneer in the field of scientific management. It is a visual management system, on which future time is plotted horizontally and work to be completed is indicated in a vertical line. The critical path method (CPM) evolved parallel to PERT. CPM is a mathematically ordered network of planning and scheduling project management; it was first used in 1957 by E.I. du Pont de Nemours & Co. PERT borrows some CPM applications. PERT proved to be an ideal technique for one-of-a-kind projects, using a time network analysis to manage personnel, material resources, and financial requirements. The growth of PERT paralleled the rapid expansion in the defense industry and meteoric developments in the space race. After 1960, all defense contractors adopted PERT to manage the massive one-time projects associated with the industry. Smaller businesses, awarded defense related government contracts, found it necessary to use PERT. At the same time, du Pont developed CPM, which was particularly applied in the construction industry. In the last 30 years, PERT has spread, as has CPM, as a major technique of integrated project management.

PERT centers on the concept of time and allows flexible scheduling due to variations in the amount of time it takes to complete one specific part of the project. A typical PERT network consists of activities and events. An event is the completion of one program component at a particular time. An activity is defined as the time and resources required to move from one event to another. Therefore, when events and activities are clearly defined, progress of a program is easily monitored, and the path of the project proceeds toward termination. PERT mandates that each preceding event be completed before succeeding events, and thus the final project, can be considered complete.

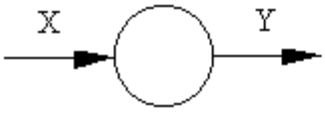
One key element to PERT's application is that three estimates are required because of the element of uncertainty and to provide time frames for the PERT network. These three estimates are classed as optimistic, most likely, and pessimistic, and are made for each activity of the overall project. Generally, the optimistic time estimate is the minimum time the activity will take—considering that all goes right the first time and luck holds for the project. The reverse is the pessimistic estimate, or maximum time estimate for completing the activity. This estimate takes into account Murphy's law—whatever can go wrong will—and all possible negative factors are considered when computing the estimate. The third is the most likely estimate, or the normal or realistic time an activity requires. Two other elements comprise the PERT network: the path, or critical path, and slack time. The critical path is a combination of events and activities that will necessitate the greatest expected completion time. Slack time is defined as the difference between the total expected activity time for the project and the actual time for the entire project. Slack time is the spare time experienced in the PERT network.

2.8.1 PERT Diagram Symbols

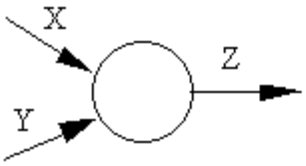
activity (task to be carried out in the project)



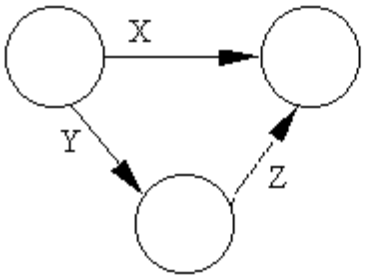
event (situation which occurs on completion of a job or jobs)



Job X must be completed before Job Y can start



What does this mean ?
If you can not work it out, mail your tutor for the answer.



Z is a dummy activity, it is represented by a dotted line. It is required to avoid activities having the same head and tail events.

2.8.2 How to Produce the PERT Diagram

1. There is a single start and end event.
2. Time flows from left to right (so does the numbering sequence).
3. Events are given a unique number (activities then have a unique label i.e. head & tail event numbers).
4. The network can then be drawn taking into account the dependencies identified.
5. Working from the start event forward, calculate the earliest times, setting the earliest time of the first event to zero. Add the job duration time to the earliest event time to arrive at

the earliest time for the successor event. Where the successor has more than one activity dependent on to the latest time is entered.

6. Workings from the finish event backwards, calculate the latest times. Set the latest time to the earliest time for the finish event. Subtract job duration from the latest time to obtain predecessor latest event times. Where the predecessor event has more than one arrow emanating from it enter the earliest time.
7. Event slack is calculated by subtracting the earliest event time from the latest event time.
8. Critical path(s) are obtained by joining the events with zero event slack.

2.8.3 PERT Example

List of activities for the network:

Task	Dependent On	Duration
A	-	3
B	-	6
C	-	3
D	A	5
E	C	2
F	B, D, E	6
G	A	9

Calculate Earliest Time and Latest Time.

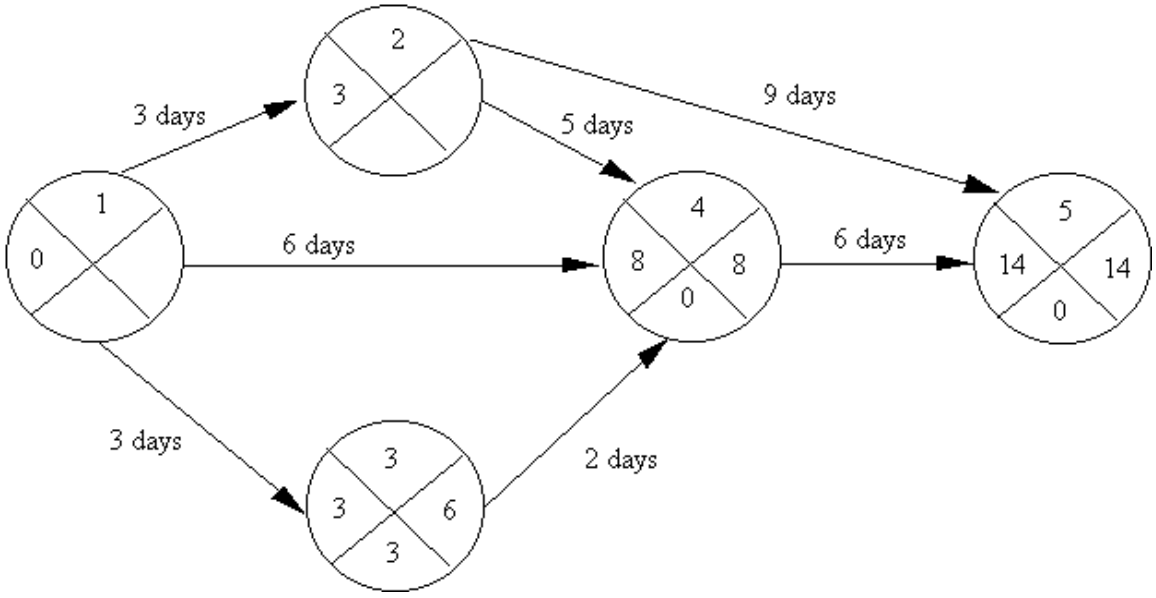


Figure 8: PERT Diagram

2.9 Gantt Charts

Henry Laurence Gantt (1861-1919) was a mechanical engineer, management consultant and industry advisor. Henry Laurence Gantt developed Gantt charts in the second decade of the 20th century. Gantt charts were used as a visual tool to show scheduled and actual progress of projects. Accepted as a commonplace project management tool today, it was an innovation of world-wide importance in the 1920s. Gantt charts were used on large construction projects like the Hoover Dam started in 1931 and the interstate highway network started in 1956.

2.9.1 Gantt Chart Basics

Gantt charts are a project planning tool that can be used to represent the timing of tasks required to complete a project. Gantt charts are simple to understand and easy to construct. Hence, they are used by most project managers for all but the most complex projects.

In a Gantt chart, each task takes up one row. Dates run along the top in increments of days, weeks or months, depending on the total length of the project. The expected time for each task is represented by a horizontal bar whose left end marks the expected beginning of the task and whose right end marks the expected completion date. Tasks may run sequentially, in parallel or overlapping.

As the project progresses, the chart is updated by filling in the bars to a length proportional to the fraction of work that has been accomplished on the task. This way, one can get a quick reading of project progress by drawing a vertical line through the chart at the current date. Completed tasks lie to the left of the line and are completely filled in. Current tasks cross the line and are

behind schedule if their filled-in section is to the left of the line and ahead of schedule if the filled-in section stops to the right of the line. Future tasks lie completely to the right of the line.
In constructing a Gantt chart, keep the tasks to a manageable number (no more than 15 or 20) so that the chart fits on a single page. More complex projects may require subordinate charts which detail the timing of all the subtasks which make up one of the main tasks. For team projects, it often helps to have an additional column containing numbers or initials which identify who on the team is responsible for the task.

Often the project has important events which you would like to appear on the project timeline, but which are not tasks. For example, you may wish to highlight when a prototype is complete or the date of a design review. You enter these on a Gantt chart as "milestone" events and mark them with a special symbol, often an upside-down triangle.

2.9.2 Features of Gantt chart

A Gantt chart is a horizontal bar or line chart which will commonly include the following features:
activities identified on the left hand side;
time scale is drawn on the top (or bottom) of the chart;
a horizontal open oblong or a line is drawn against each activity indicating estimated duration;
dependencies between activities are shown;
at a review point the oblongs are shaded to represent the actual time spent (an alternative is to represent actual and estimated by 2 separate lines);
a vertical cursor (such as a transparent ruler) placed at the review point makes it possible to establish activities which are behind or ahead of schedule

2.9.3 Gantt Chart Example

Develop the network activity chart and identify the critical path for a project based on the following information. Draw the activity network as a Gantt chart using MSProject. What is the expected duration of the project?

Activity	Expected Duration	Predecessors
A	5 days	--
B	10 days	A
C	8 days	A
D	1 day	A
E	5 days	B, C
F	10 days	D, E
G	14 days	F

H	3 days	G
I	12 days	F
J	6 days	H, I

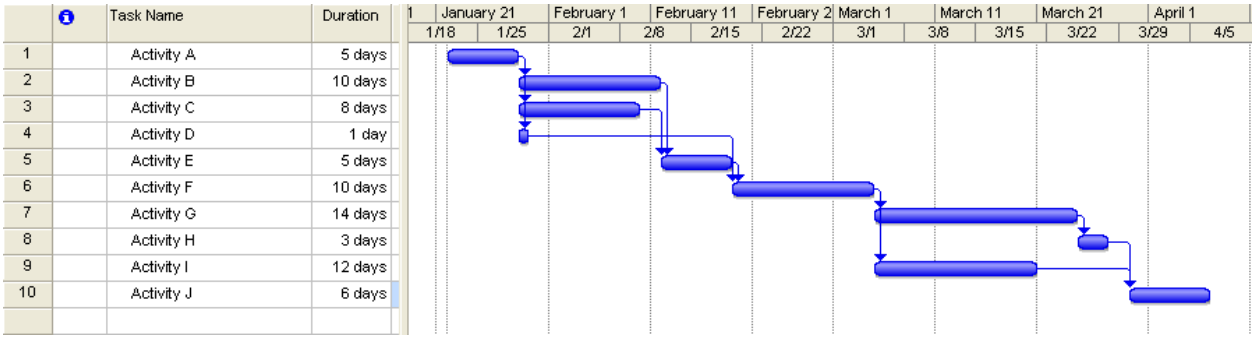


Figure 9: Gantt chart for activity network

Expected duration of the project can be found by adding the lengths of the linked paths. In this case, it is 53 days.

❖ Check Your Progress 2

1) PERT stands for _____.

2) Performance is done after implementation True/False.

3) The technique used for scheduling the tasks and tracking of the progress of energy management projects is called _____.

2.10 Summary

Software project management is perhaps the most important factor in the outcome of a project. Without proper project management, a project will almost certainly fail. Many organizations have evolved effective project management processes. At the top level, the project management process consists of three phases: planning, execution, and closure. When creating a project schedule, managers will find both Program Evaluation and Review Technique (PERT) and Gantt charts to be essential tools for successfully

completing the project at hand. Both types of charts provide tools for managers to analyze projects through visualization, helping divide tasks into manageable parts. As a part of it, reasons for software project failures are discussed.

2.11 Solutions / Answers

Check Your Progress 1

1) False

2) Check Points

3) True

Check Your Progress 2

1) Program Evaluation and Review Technique

2) True

3) Gantt chart

2.12 Further Readings

Reference Books

1) Software Project Management (5th Edition) 2011 Hughes, Bob and Cotterell, Mike Mc Graw Hill Higher Education

2) IT Project Management: On Track from Start to Finish, Third Edition Joseph Phillips Mc Graw Hill

Reference Websites

http://www.epmbook.com/

Block-3 Unit-3 : SOFTWARE ENGINEERING FUNDAMENTALS

Structure:

3.0. INTRODUCTION

Software Configuration Management is an umbrella activity that is applied throughout the software process. Because change can occur at any time, SCM activities are developed to (1) identify change (2) control change, (3) ensure that change is being properly implemented and (4) report change to others who may have an interest. It is important to make a clear distinction between software maintenance and software configuration management. Maintenance is a set of software engineering activities that occur after software has been delivered to the customer and put into operation. Software configuration management is a set of tracking and control activities that begin when a software project

begins and terminate only when the software is taken out of operation. A primary goal of software engineering is to improve the ease with which changes can be accommodated and reduce the amount of effort expended when changes must be made. Quality assurance consists of the auditing and reporting functions of management. The goal of quality assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals.

3.1. OBJECTIVES

After going through this unit you will be able to:

- Understand the importance, need and activities of Software Configuration Management
- Understand Software Maintenance and its types
- Study the various models of Software Maintenance and estimate Maintenance cost
- Understand the concept of Software Quality
- Study the Software Quality Models
- Understand the SQA activities

3.2. SOFTWARE CONFIGURATION MANAGEMENT

Changes are inevitable when software is built. A primary goal of software engineering is to improve the ease with which changes can be made to software. Configuration management (CM) is the discipline of controlling the evolution of complex systems; software configuration management (SCM) is its specialization for computer programs and associated documents i.e. **"Software configuration management (SCM) is the discipline of controlling the evolution of complex software systems".**
Every software engineer has to be concerned with how changes made to work products are tracked and propagated throughout a project. To ensure that quality is maintained the change process must be audited. A Software Configuration Management (SCM) Plan defines the strategy to be used for change management. A slightly more formal definition of software configuration management can be: **"SCM is a software-engineering discipline comprising the tools and techniques (processes or methodology) that a company uses to manage change to its software assets"**.

Differences between SCM and CM
SCM and CM differ from each other in the following two ways:

1) Software is easier to change than Hardware, and it therefore changes faster. Even relatively small software systems, developed by a single team, can experience a significant rate of change, and in large systems, such as telecommunications systems, the update activities can totally overwhelm manual configuration management procedures.
2) SCM is potentially more automatable since all components of a software system are easily stored on-line. CM for physical systems is hampered by having to handle objects that are not within reach of programmable controls. As CAD/CAM and robotics bring manufacturing processes more and more under computer control

physical configuration management will undoubtedly adopt some of the approaches used for software. VLSI has already achieved this by managing Circuit design and circuit processing in a manner similar to software design and compilation.

3.2.1. Basic Concepts and Definitions for SCM

The following are some basic concepts and terminologies used in software configuration management:

Revision management

This is the core function of SCM and the foundation upon which other functions have been built. Revision management is the storage of multiple .images of the development files in an application, which can be shared by multiple developers. With revision management, one can get an image, a snapshot in time of the development process and can re-create any file to the way it was at any point in time.

Version management

While revision management shows development files in progress, a version is a particular instance of an entire development p view checkpoints and approvals before content is posted. A good SCM solution must manage the progress roject. A version is usually thought of as all the functions, features and complete builds you can use right now. Files managed, version-labeled and promoted by the revision management system are used to build the version.

Workflow and process management

As software or web content is developed, it is said to go through a lifecycle. Whether formal or informal, there are usually milestones in this lifecycle, such as development-test-alpha-beta-release for applications, test and production for packaged applications or, for websites, staging servers, re of code and content through the lifecycle. This is many levels above the simple source code protection built into some development platforms.

Build management

When the code files, compilers, development tools and other components needed to create an application are ready, developers make an application build. Build management imposes automated, repeatable procedures that speed the build process, improve build accuracy and make it easier to create and maintain multiple versions of an application. An SCM toolset can become a vital integrative resource for building an application across multiple platforms. A strong cross-platform build management capability also eliminates the need to redefine the build process (the script) for each platform one is targeting.

Change requests

The days of creating software, shrink-wrapping it, selling it and walking away are long gone. Today the software undergoes a constant process of revision. Development teams can be overwhelmed with the many software change requests (SCR) they receive, everything from users' wish lists, to new business requirements that must be accommodated. SCM enables teams to streamline the SCR process by automatically tracking updates, change requests, problems and issues; assigning ownership and hand-offs within the development team; and communicating the process through to resolution.

Code and project auditing

Cross-platform development introduces special problems because the data to be audited may reside on multiple platforms. The best SCM systems provide a single point of access for auditing all development information. They also work across all kinds of development

platforms, operating systems, network protocols and hardware platforms, and provide tailored clients within the developer's preferred development interface.

Security

In a mainframe or UNIX-only environment, it is possible to define an executable as having a user ID. You then create permissions on shared development directories to limit user access. On Windows/PC platforms, this security is absent. The SCM system itself must provide the ability to control access, including secure, protected access via the Web.

3.2.2 Software Configuration Items (SCI)

The software items to be configured through SCM include:

- Computer programs (both source and executable)
- Documentation (both technical and user)
- Data (contained within the program or external to it)

3.2.3 Necessity of Software Configuration Management

There are several reasons for putting a SCI under configuration management. But possibly the most important reason is to control the access to the different deliverables. Unless strict discipline is enforced regarding updation and storage of different SCIs, several problems appear. The following are some of the important problems that appear if configuration management is not used.

a. ***Inconsistency problem:*** The problem of inconsistency arises when the SCIs are replicated. Consider a situation where every software engineer has a personal copy of the SCI (e.g. source code). As each engineer makes changes to his local copy, he is expected to intimate them to other engineers, so that the changes in interfaces are uniformly changed across all modules. However, most of the times the changes are not communicated to the team thus making the different copies of the SCI inconsistent, leading to the failure of the integrated product.
b. ***Concurrent Access:*** Simultaneous access and modifications of the different parts of an SCI may lead to unintentional overwriting of the changes made by the other teammates. Thus the final SCI may be deemed unusable for all.
c. ***Stable Development Environment:*** When a project is underway, the team members need a stable environment to make progress. For example if one engineer is trying to integrate module A, with the modules B and C, he cannot proceed if developer of module C keeps changing C; this can be especially frustrating if a change to module C forces him to recompile A. With an effective SCM in place, the project leader freezes the SCIs to form a base line. Anyone needing an SCI under configuration control, he is provided with a copy of the base line item. The requester makes changes to his private copy. Only after the requester is through with all modifications to his private copy, the configuration is updated and a new base line gets formed instantly. This establishes a baseline for others to use and depend on. Also, configuration may be frozen periodically. Freezing a configuration may involve archiving everything needed to rebuild it.
d. ***System accounting and maintaining status information***: System accounting keeps track of who made a particular change and when the change was made.
e. ***Handling variants:*** Existence of variants of a software product causes some peculiar problems. Suppose somebody has several variants of the same module, and finds a bug

in one of them. Then, it has to be fixed in all versions and revisions. To do it efficiently, he should not have to fix it in each and every version and revision of the software separately.

3.2.4. Software Configuration Management Activities

Normally, a project manager performs the configuration management activity by using an automated configuration management tool. A configuration management tool provides automated support for overcoming the problems mentioned above. In addition, a configuration management tool helps to keep track of various deliverable objects, so that the project manager can quickly and unambiguously determine the current state of the project. The configuration management tool enables the engineers to change the various components in a controlled manner. Configuration management is carried out through two principal activities:

3.2.4.1. Configuration Identification: Configuration identification involves deciding which parts of the system should be kept track of. The project manager classifies the SCIs associated with a software development effort into two main categories:

- Controlled: Controlled objects are those which are already put under configuration control. One must follow some formal procedures to change them. Requirements specification document, Design documents, Tools used to build the system, such as compilers, linkers, lexical analyzers, parsers, etc., Source code for each module, Test cases, Problem reports are objects in this category
- Uncontrolled: Uncontrolled objects are not and will not be subjected to configuration control.

The configuration management plan is written during the project planning phase and it lists all controlled objects. The managers who develop the plan must strike a balance between controlling too much, and controlling too little. If too much is controlled, overheads due to configuration management increase to unreasonably high levels. On the other hand, controlling too little might lead to confusion when something changes.

3.2.4.2. Configuration control: Configuration control ensures that changes to a system happen smoothly. It is the process of managing changes to controlled objects. Configuration control is the part of a configuration management system that most directly affects the day-to-day operations of developers. The configuration control system has following functions:

a. prevents unauthorized changes to any controlled objects. In order to change a controlled object such as a module, a developer can get a private copy of the module by a reserve operation
b. allow only one person to reserve a module at a time. Once an object is reserved, it does not allow any one else to reserve this module until the reserved module is restored thereby solving the problems associated with concurrent access.

***Baselines*:** A work product becomes a baseline only after it is reviewed and approved. A baseline is a milestone in software development that is marked by the delivery of one or more configuration items. Once a baseline is established each change request must be

evaluated and verified by a formal procedure before it is processed. Baseline work products are place in a project database or repository

Steps to modify any object under configuration control:

1. The engineer needing to change a module first obtains a private copy of the module through a reserve operation.
2. He carries out all necessary changes on this private copy.
3. Restoring the changed module to the system configuration requires the permission of a change control board (CCB).
4. The CCB (comprising the project manager and some senior members of the development team) reviews the changes made to the controlled object and certifies the following about the change:
    a) Change is well-motivated.
    b) Developer has considered and documented the effects of the change.
    c) Changes interact well with the changes made by other developers.
    d) The change has been validated is consistent with the requirement.
5. The project manager updates the old base line through a restore operation

3.2.5. Configuration management tools

SCCS and RCS are two popular configuration management tools available on most UNIX systems. SCCS or RCS can be used for controlling and managing different versions of text files. They provide an efficient way of storing versions that minimizes the amount of occupied disk space. Suppose, a module SWM is present in three versions SWM1.1, SWM1.2, and SWM1.3. Then, SCCS and RCS stores the original module SWM1.1 together with changes needed to transform SWM1.1 into SWM1.2 and SWM1.2 to SWM1.3. The changes needed to transform each base lined file to the next version are stored and are called deltas. The main reason behind storing the deltas rather than storing the full version files is to save disk space. The change control facilities provided by SCCS and RCS include the ability to incorporate restrictions on the set of individuals who can create new versions, and facilities for checking components in and out (i.e. reserve and restore operations).

3.3 SOFTWARE MAINTENANCE

Software maintenance is an integral part of a software life cycle. The term maintenance, when associated to software, assumes a meaning profoundly different from the meaning it assumes in any other engineering discipline. In fact, in most engineering disciplines the term maintenance refers to the process of keeping something in working order i.e. in repair. The key concept is the deterioration of an equipment or machinery due to use and passage of time; the aim of maintenance is therefore to keep the object's functionality in line with that defined and registered at the time of release. This view of maintenance does not apply to software as software does not deteriorate with the use and the passage of time. Nevertheless, the need for modifying a piece of software after delivery has been with us since the very beginning of electronic computing. Hence Software Maintenance as defined by The Institute of Electrical and Electronics Engineers (IEEE) stands as:

"Software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment."

3.3.1 Need for Software Maintenance

Maintenance is needed to ensure that the system continues to satisfy user requirements. Maintenance is applicable to systems developed using any software development model (e.g., spiral). The need for Software maintenance arises due to corrective and non-corrective software actions. Maintenance must be performed due to some of the following reasons:

- ***Hardware Obsolescence***: The rate of hardware obsolescence in comparison to the immortal software creates a demand of the user community to see the existing software products run on newer platforms, in newer environments, and/or with enhanced features.
- ***Hardware Evolution:*** As changes are introduced in the hardware platform a software product performing some low-level functions requires maintenance.
- ***Environmental Changes:*** Changes in the support environment of a software product, require rework in the software to cope up with the newer interface.
- ***Correct Errors:*** As the software is modified, maintenance is required to make the software error free.
- ***Enhance Features:*** Enhancements requested in the software by the customer from time to time require an ongoing maintenance activity.
- ***Software Portability:*** Ensuring portability of software across platforms, both existing and upcomimg require maintenance.
- ***Correct Requirements and Design Flaws:*** The ever changing requirements of the customer during the development process and even after implementation need to be handled as maintenance effort.
- ***Interface with other systems:*** Convert programs so that different hardware, software, system features, and telecommunications facilities can be used.
- ***Migrate legacy systems.***
- ***Retire systems.***

3.3.7 Types of software maintenance

The categories of maintenance defined by ISO/IEC are as follows:

- **Corrective maintenance:** Reactive modification of a software product performed after delivery to correct discovered problems.
- **Adaptive maintenance**: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.
- **Perfective maintenance**: Modification of a software product after delivery to improve performance or maintainability.
- **Preventive maintenance:** Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.
- **Enhancive maintenance:** Modifications to the software after delivery to satisfy the customer expectations of extended functionalities and features to support other additional applications.

Adaptive, Perfective and Enhancive maintenance can be called as enhancements; Corrective and Preventive maintenance as corrections. Preventive maintenance, the newest category, is defined as maintenance performed for the purpose of preventing problems before they occur. Preventive maintenance is most often performed on software products where safety is critical. The distribution of approximate effort spent during the life time of a software project in maintenance has been depicted in Figure 3.1.

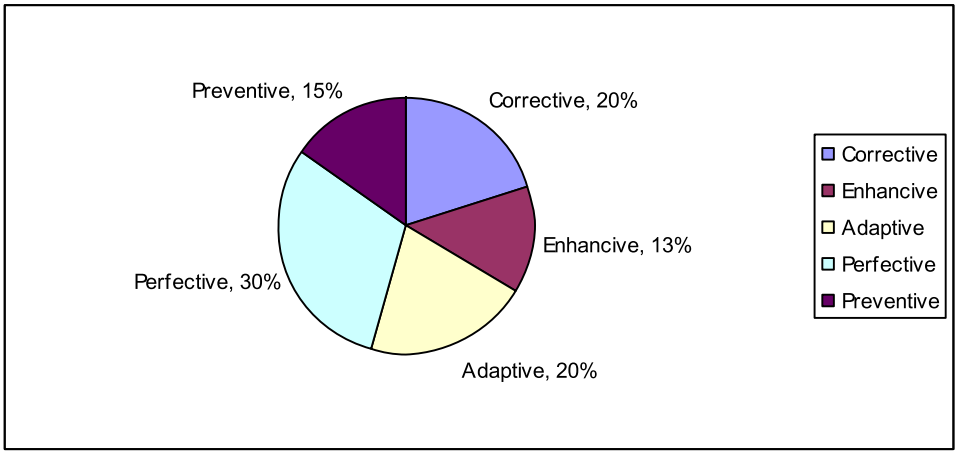


Fig: 3.1: Distribution of Maintenance Effort

3.3.3. Software Maintenance Process

As software is due for maintenance owing to any of the reasons stated in section 3.3.1, the software is subjected to the maintenance process which is divided into the following five phases:

Phase 0: Determine Maintenance Objective: This phase involves identifying the objective of maintenance i.e. correcting program errors, enhancing capabilities, software portability etc.

Phase 1: Program Understanding: The first phase consists of analyzing the program in order to understand. This phase involves analyzing the program complexity, documentation and the program descriptiveness.

Phase 2: Generating Particular Maintenance Proposal: The second phase consists of generating a particular maintenance proposal to accomplish the implementation of the maintenance objective.

Phase 3: Ripple Effect: The third phase consists of accounting for all of the ripple effect as a consequence of program modifications.

Phase 4: Modified Program Testing: The fourth phase consists of testing the modified program to ensure that the modified program has at least the same reliability level as before.

The process can be depicted in Figure 3.2.

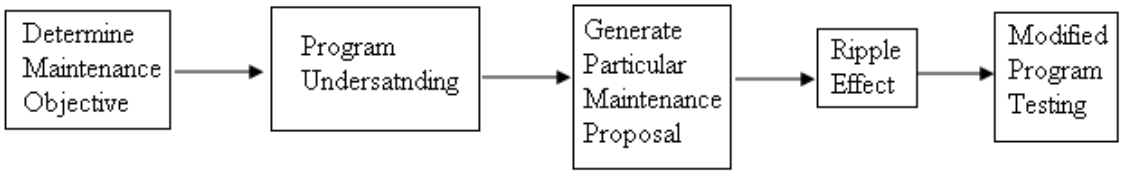


Figure 3.2: Software Maintenance Process

3.3.4. Software Maintenance Models

- **Quick-Fix Model**

This is basically an adhoc approach to maintaining software. It is a fire fighting approach, waiting for the problem to occur and then trying to fix it as quickly as possible.

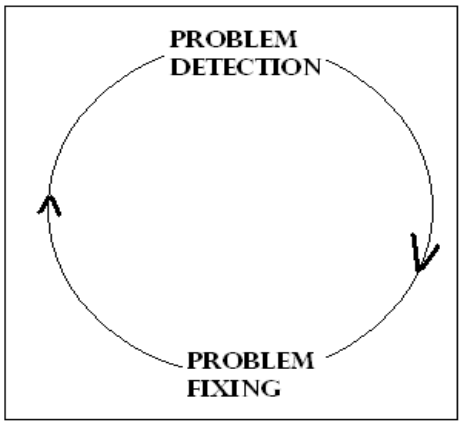


Figure 3.3: Quick-Fix Model

- **Iterative Enhancement Model**

The Model consists of three stages in each iteration:
1. Analysis of the Existing system: This includes study of the existing system.
2. Characterization of proposed modifications.
3. Redesign of the current version and implementation.

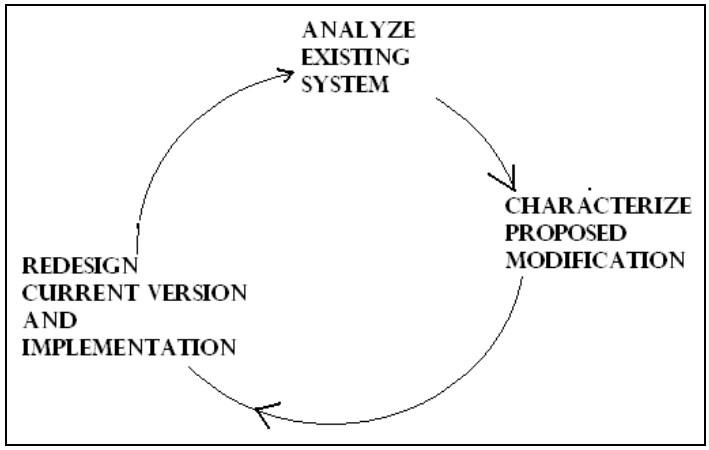


Figure 3.4: Iterative Enhancement Model

- **Reuse Oriented Model**

The model emphasizes component reuse in the following way.
1. Identification of the parts of the old system that are candidates for reuse.
2. Understanding these system parts.
3. Modification of the old system parts appropriate to the new requirements.
4. Integration of the modified parts into the new system.

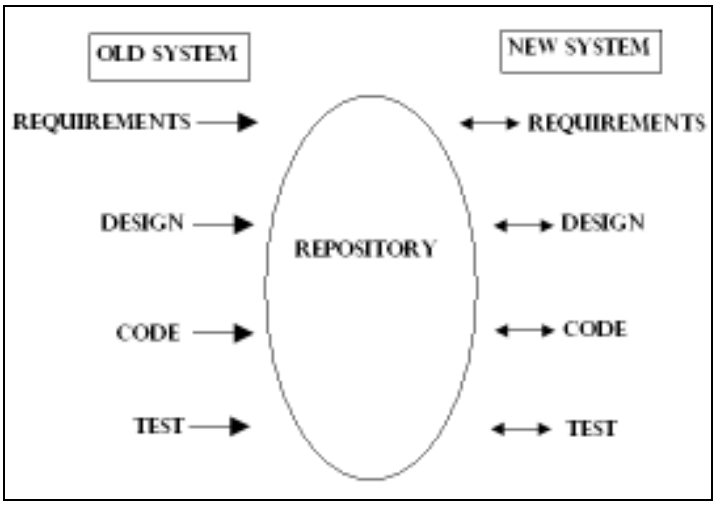


Figure 3.5: Reuse Oriented Model

- **Boehm's Model**

The maintenance process model proposed by Boehm is based on the economic models and principles. He represents the maintenance process as a closed loop cycle as depicted in Figure 3.6.

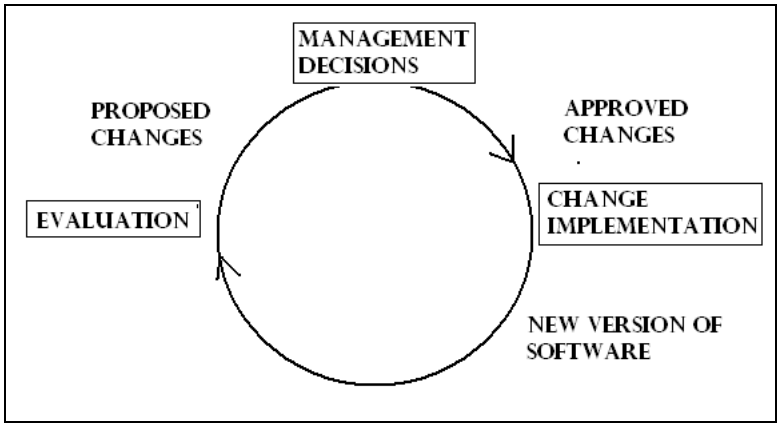


Figure 3.6.: Boehm's Model

- **Taute Maintenance Model**

The model depicted in Figure 3.7.consists of eight phases:
1. Change request phase: Changes are accepted from the customer.
2. Estimate phase: The effort and cost estimates are computed for the requested changes.
3. Schedule phase: The schedule for the work to be done is prepared for incorporating the changes.
4. Programming phase: The changes are implemented.
5. Test phase: The modified software is tested.
6. Documentation phase: The documentation for the tested software is prepared.
7. Release phase: The modified software is released.
8. Operation phase: The modified software is finally fully operational after correcting the post release bugs.

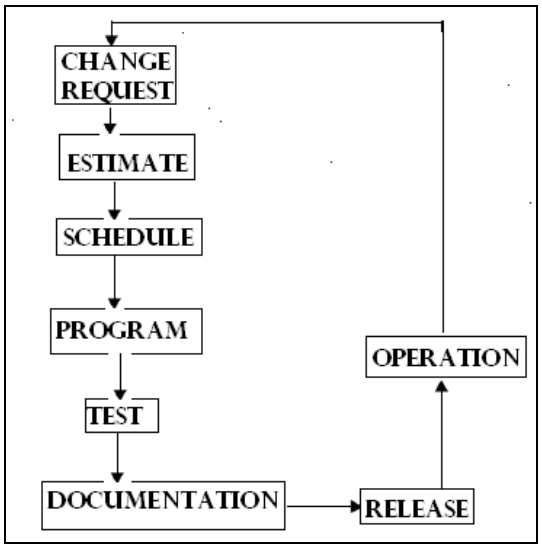


Figure 3.7: Taute Maintenance Model

3.3.5 Reverse Engineering

Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at higher levels of abstraction. Accordingly, reverse engineering is a process of examination, not a process of change, and therefore it does not involve changing the software under examination. One type of reverse engineering is redocumentation. ***Redocumentation*** is the recreation of a semantically equivalent representation within the same relative abstraction level. Another type is design recovery. ***Design recovery*** entails identifying and extracting meaningful higher level abstractions beyond those obtained directly from examination of the source code. This may be achieved from a combination of code, existing design documentation, personal experience, and knowledge of the problem and application domains. ***Refactoring***, a program transformation that reorganizes a program without changing its behavior, is now being used in reverse engineering to improve the structure of object oriented programs.

Although software reverse engineering originated in software maintenance, it is applicable to many problem areas. Six key objectives of reverse engineering can be:

- coping with complexity,
- generating alternate views,
- recovering lost information,
- detecting side effects,
- synthesizing higher abstractions, and
- facilitating reuse.

Reverse engineering has been recommended as a key supporting technology to deal with systems that have the source code as the only reliable representation. Examples of problem areas where reverse engineering has been successfully applied include identifying reusable assets, finding objects in procedural programs, discovering architectures, deriving conceptual data models, detecting duplications, transforming binary programs into source code, renewing user interfaces, parallelizing sequential programs, and translating, downsizing, migrating, and wrapping legacy code. Reverse engineering principles have

also been applied to business process reengineering to create a model of an existing enterprise.

Reverse engineering has been viewed as a two step process: information extraction and abstraction. ***Information extraction*** analyses the subject system artifacts – primarily the source code – to gather row data, whereas ***information abstraction*** creates user-oriented documents and views.

The IEEE Standard for Software Maintenance suggests that the process of reverse engineering evolves though six steps:

- dissection of source code into formal units;
- semantic description of formal units and creation of functional units;
- description of links for each unit (input/output schematics of units);
- creation of a map of all units and successions of consecutively connected units (linear circuits);
- declaration and semantic description of system applications, and;
- creation of an anatomy of the system.

The first three steps concern local analysis on a unit level, while the other three steps are for global analysis on a system level.

3.3.6 Re-engineering

Re-engineering is defined as the examination and alteration of the subject system to reconstitute it in a new form, and the subsequent implementation of the new form. In other words re-engineering is the activity of better understanding and maintaining a software system. Reengineering is often not undertaken to improve maintainability but is used to replace aging legacy systems. Formally defining "Software Re-engineering is any activity that: (1) improves one's understanding of software, or (2) prepares or improves the software itself, usually for increased maintainability, reusability, or evolvability."

Re-engineering entails some form of reverse engineering to create a more abstract view of a system, a regeneration of this abstract view followed by forward engineering activities to realize the system in the new form. This process is illustrated in Figure 3.8. The presence of a reverse engineering step distinguishes re-engineering from restructuring, the latter consisting of transforming an artifact from one form to another at the same relative level of abstraction.

Software re-engineering has proven important for several reasons. Seven main reasons that demonstrate the relevance of re-engineering:

- Re-engineering can help reduce an organization's evolution risk;
- Re-engineering can help an organization recoup its investment in software;
- Re-engineering can make software easier to change;
- Re-engineering is a big business;
- Re-engineering capability extends CASE toolsets;
- Re-engineering is a catalyst for automatic software maintenance;
- Re-engineering is a catalyst for applying artificial intelligence techniques to solve software re-engineering problems.

Examples of scenarios in which re-engineering has proven useful include migrating a system from one platform to another, downsizing, translating, reducing maintenance costs, improving quality, and migrating and re-engineering data.

Software re-engineering is a complex process that re-engineering tools can only support, not completely automate. There is a good deal of human intervention with any software reengineering project. Re-engineering tools can provide help in moving a system to a new maintenance environment, for example one based on a repository, but they cannot define such an environment nor the optimal path along which to migrate the system to it. These are activities that only human beings can perform.
Another problem that re-engineering tools only marginally tackle is the creation of an adequate testbed to prove that the end product of reengineering is fully equivalent to the original system. This still involves much hand-checking, partially because very rarely an application is re-engineered without existing functions being changed and new functions being added.
Finally, re-engineering tools often fail to take into account the unique aspects of a system, such as the use of a JCL or a TP-Monitor, the access to a particular DBMS or the presence of embedded calls to modules in other languages.

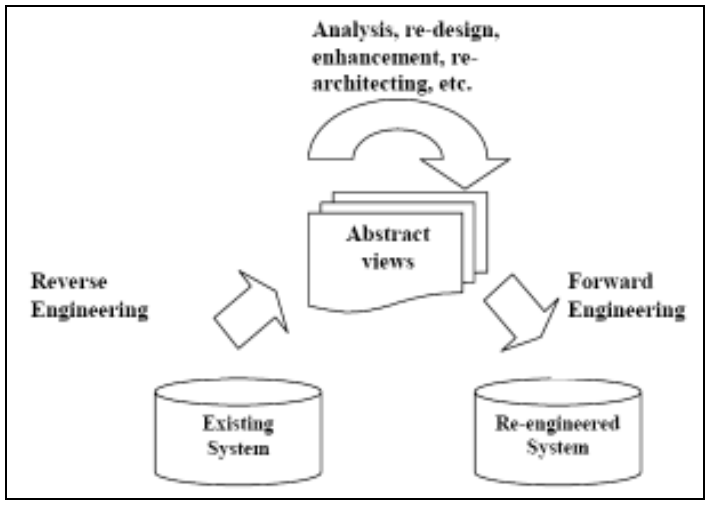


Figure 3.8 Reverse Engineering and Re-engineering

3.3.7. Estimation of Approximate Maintenance Cost

It is well known that maintenance efforts require about 60% of the total life cycle cost for a typical software product. However, maintenance costs vary widely from one application domain to another. For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost.
Boehm proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model. Boehm's maintenance cost estimation is made in terms of a quantity called the Annual Change Traffic (ACT). Boehm defined ACT as the fraction of a software product's source instructions which undergo change during a typical year either through addition or deletion.

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}}$$

where, $KLOC_{added}$ is the total kilo lines of source code added during maintenance. $KLOC_{deleted}$ the total KLOC deleted during maintenance. Thus, the code that is changed,

should be counted in both the code added and the code deleted. The annual change traffic (ACT) is multiplied with the total development cost to arrive at the maintenance cost:

$$\text{Maintenance Cost} = \text{ACT} \times \text{development cost.}$$

Most maintenance cost estimation models, however, yield only approximate results because they do not take into account several factors such as experience level of the engineers, and familiarity of the engineers with the product, hardware requirements, software complexity, etc.

Exercises

Q.1. Differentiate between Software Re-engineering and Reverse Engineering.

Q.2. If the development cost of a software product is Rs. 20,000,000/-, compute the annual maintenance cost given that every year approximately 10% of the code needs modification.

3.4 SOFTWARE QUALITY ASSURANCE (SQA)

3.4.1. Quality Concept

The term software quality refers to conformance to explicitly stated requirements and standards, as well as implicit characteristics that customers assume will be present in any professionally developed software. The SQA group must look at software from the customer's perspective, as well as assessing its technical merits. Software Quality Assurance controls variation among products. Software engineers are concerned with controlling the variation in their processes, resource expenditures, and the quality attributes of the end products. The activities performed by the SQA group involve quality planning, oversight, record keeping, analysis and reporting. An elaborate definition of SQA can be given as:

"A systematic, planned set of actions necessary to provide adequate confidence that the software development process or the maintenance process of the software system product conforms to established functional technical requirements as well as with the managerial requirements of keeping the schedule and operating within budgetary confines."

Software Quality Assurance (SQA) consists of a means of monitoring the software engineering processes and methods used to ensure quality. It does this by means of audits of the quality management system under which the software system is created. These audits are backed by one or more standards, usually ISO 9000.

It is distinct from software quality control. Quality Control (QC) is a set of activities (including reviewing requirements documents, and software testing) carried out with the main objective of withholding products from shipment if they do not qualify. Quality Assurance (QA) is meant to minimize the costs of quality by introducing a variety of activities throughout the development process and maintenance process in order to prevent the causes of errors, detect them, and correct them in the early stages of the development. As a result, quality assurance substantially reduces the rate of non-qualifying products. Software quality control is a control of products, software quality assurance is a control of processes.

3.4.2. Software Errors, Faults and Failures

Software Error: Section of the code that are incorrect as a result of grammatical, logical or other mistakes made by a system analyst, a programmer, or another member of the software development team. Software Errors can arise due to the following reasons:

- Faulty requirements
- Client- developer communication failures
- Deliberate deviations from software requirements
- Logical design errors
- Coding errors
- Non compliance with documentation and coding instructions
- Shortcoming of the testing process
- Procedure errors
- Documentation errors

Software Fault: Software faults are software errors that causes the incorrect functioning of the software during a specific application.

Software Failure: Software faults become software failures only when they are "activated.

3.4.3. Cost of Quality

Cost of quality includes all costs incurred in the pursuit of quality or in performing quality related

activities. Cost of quality studies are conducted to provide a baseline for the current. Cost of quality, to identify opportunities for reducing the cost of quality and ;to provide a normalized basis of comparison. The basis of normalization is almost always dollars. Once we have normalized quality costs on a dollar basis, we have the necessary data to evaluate where the opportunities lie to improve our processes. Furthermore, we can evaluate the affect of changes in dollar-based terms. Quality costs may be divided into costs associated with prevention, appraisal, and failure.

Prevention costs include:

- Quality planning
- Formal technical reviews (discussed in detail in Section 3.4.6)
- Test equipment
- Training

Appraisal costs include activities to gain insight into product condition "first time through" each process. Examples of appraisal costs include:

- In-process and inter process inspection
- Equipment calibration and maintenance
- Testing

Failure costs are costs that would disappear if no defects appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs.

- ***Internal failure costs*** are the costs incurred when we detect an error in our product prior to shipment . Internal failure costs include:
    - Rework

    - Repair
    - Failure mode analysis
- ***External failure costs*** are the costs associated with defects found after the product has been shipped to the customer. Examples of external failure s costs are:
    - Complaint resolution
    - Product return and replacement
    - Help line support
    - Warranty work

As expected, the relative costs to find and repair a defect increase dramatically as we go from prevention to detection and from internal failure to external failure.

3.4.4. Software Quality Models

Software quality can be described by specific quality models. Two such models have been described in this section.

3.4.4.2 McCall Software Quality Model

The concept of software quality and the efforts to understand it in terms of measurable quantities i.e quality factors and quality criteria was attempted by McCall. A ***Quality Factor*** represents a behavioral characteristic of a system. Quality factors are external attributes of a software system. Customers, software developers, and quality assurance engineers are interested in different quality factors to different extents. For example, customers may want an efficient and reliable software with less concern for portability. The developers strive to meet customer needs by making their system efficient and reliable, at the same time making the product portable and reusable to reduce the cost of software development. The software quality assurance team is more interested in the testability of a system so that some other factors, such as correctness, reliability, and efficiency, can be easily verified through testing. McCall identified 11 quality factors. The 11 quality factors have been grouped into three broad categories, depicted in Figure 3.7, as follows:

1. **Product operation:** Factors which are related to the operation of a product are combined. These five factors are related to operational performance, convenience, ease of usage and its correctness. These factors play a very significant role in building customer's satisfaction. The factors are:
    - *Correctness*: Extent to which a program satisfies its specifications and fulfills the user's mission objectives.
    - *Efficiency:* Amount of computing resources and code required by a program to perform a function.
    - *Integrity:* Extent to which access to software or data by unauthorized persons can be controlled.
    - *Reliability:* Extent to which a program can be expected to perform its intended function with required precision.
    - *Usability:* Effort required to learn, operate, prepare input and interpret output of a program.
2. **Product Revision:** These factors pertain to the testing & maintainability of software. They give us idea about ease of maintenance, flexibility and testing effort. Hence, they are combined under the umbrella of product revision.

- *Maintainability:* Effort required for locating and fixing a defect in an operational program.
- *Flexibility:* Effort required for modifying an operational program.
- *Testability:* Effort required for testing a program to ensure that it performs its intended functions.

3. **Product Transition:** We may have to transfer a product from one platform to another platform or from one technology to another technology. The factors related to such a transfer are combined and given below:
   - *Portability:* Effort required to transfer a program from one hardware and/or software environment to another
   - *Reusability:* Extent to which parts of a software system can be reused in other applications.
   - *Interoperability:* Effort required to couple one system with another.

It may be noted that the above three categories relate more to post-development activities expectations and less to in-development activities. In other words, McCall's quality factors emphasize more on the quality levels of a product delivered by an organization and the quality levels of a delivered product relevant to product maintenance.

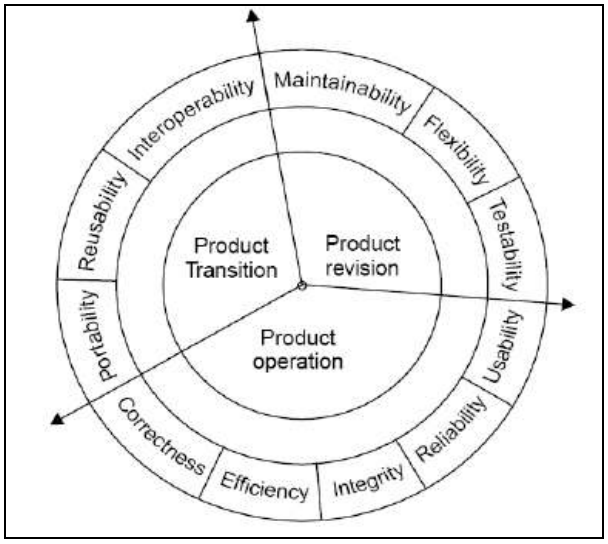


Figure 3.7 Mc Calls Software Quality Factors

3.4.4.2. ISO 9126 Software Quality Model

To define a general framework for software quality, an expert group, under the aegis of the ISO, standardized a software quality document, namely, ISO 9126, which defines six broad, independent categories of quality characteristics as follows:

1. *Functionality*: A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.
2. *Reliability*: A set of attributes that bear on the capability of software to maintain its performance level under stated conditions for a stated period of time.
3. *Usability*: A set of attributes that bear on the effort needed for use and on the individual assessment of such use by a stated or implied set of users.

4. *Efficiency*: A set of attributes that bear on the relationship between the software's performance and the amount of resource used under stated conditions.
5. *Maintainability*: A set of attributes that bear on the effort needed to make specified modifications (which may include corrections, improvements, or adaptations of software to environmental changes and changes in the requirements and functional specifications).
6. *Portability*: A set of attributes that bear on the ability of software to be transferred from one environment to another (this includes the organizational, hardware or, software environment).

The ISO 9126 standard further decomposes the quality characteristics into more concrete sub characteristics.

Functionality has been subdivided into:

- *Suitability*: The capability of the software to provide an adequate set of functions for specified tasks and user objectives.
- *Accuracy*: The capability of the software to provide the right or agreed-upon results or effects.
- *Interoperability*: The capability of the software to interact with one or more specified systems.
- *Security*: The capability of the software to prevent unintended access and resist deliberate attacks intended to gain unauthorized access to confidential information or to make unauthorized modifications to information or to the program so as to provide the attacker with some advantage or so as to deny service to legitimate users.

Reliability has been subdivided into:

- *Maturity*: The capability of the software to avoid failure as a result of faults in the software.
- *Fault Tolerance*: The capability of the software to maintain a specified level of performance in case of software faults or of infringement of its specified interface.
- *Recoverability*: The capability of the software to reestablish its level of performance and recover the data directly affected in the case of a failure.

Usability has been subdivided into:

- *Understandability*: The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.
- *Learnability*: The capability of the software product to enable the user to learn its applications.
- *Operability*: The capability of the software product to enable the user to operate and control it.
- *Attractiveness*: The capability of the software product to be liked by the user.

Efficiency has been subdivided into:

- *Time Behavior*: The capability of the software to provide appropriate response and processing times and throughput rates when performing its function under stated conditions.

- *Resource Utilization*: The capability of the software to use appropriate resources in an appropriate time when the software performs its function under stated condition.

Maintanability has been subdivided into:

- *Analyzability*: The capability of the software product to be diagnosed for deficiencies or causes of failures in the software or for the parts to be modified to be identified.
- *Changeability*: The capability of the software product to enable a specified modification to be implemented.
- *Stability* : The capability of the software to minimize unexpected effects from modifications of the software.
- *Testability* : The capability of the software product to enable modified software to be validated.

Portability has been subdivided into:

- *Adaptability*: The capability of the software to be modified for different specified environments without applying actions or means other than those provided for this purpose for the software considered.
- *Installability* : The capability of the software to be installed in a specified environment.
- *Coexistence*: The capability of the software to coexist with other independent software in a common environment sharing common resources.
- *Replaceability*: The capability of the software to be used in place of other specified software in the environment of that software.

Differences between McCall's quality model and ISO 9126 model.

- The ISO 9126 model emphasizes characteristics *visible* to the users, whereas the McCall model considers *internal* qualities as well. For example, reusability is an internal characteristic of a product. Product developers strive to produce reusable components, whereas its impact is not perceived by customers.
- In McCall's model, one quality criterion can impact several quality factors, whereas in the ISO 9126 model, one sub-characteristic impacts exactly one quality characteristic.
- A high-level quality factor, such as testability, in the McCall model is a low-level sub-characteristic of maintainability in the ISO 9126 model.

3.4.5. SQA Activities

Software quality assurance comprises of a variety of tasks associated with two different groups i.e. the software engineers who do technical work and the SQA group that has responsibility for quality assurance planning, over sight, record keeping, analysis and reporting. Software engineers address quality and perform quality assurance by applying technical methods and measures, conducting formal technical reviews, and performing well-planned software testing. The SQA group assists the software engineering team in obtaining a high quality end product.

The Software Engineering Institute recommends a set of SQA activities that address quality assurance planning. These activities are performed by an independent SQA group.

The SQA plan is developed during project planning and is reviewed by all interested parties. The plan identifies:

- Evaluations to be performed
- Audits and reviews to be performed
- Standards that are applicable to the project
- Procedures for error reporting and tracking
- Documents to be produced by the SQA group
- Amount of feedback provided to software project team

3.4.6. Formal Technical Reviews

A ***Formal Technical Review (FTR)*** is a software quality assurance activity performed by software engineers. The objectives of the FTR are:

1. to cover errors in function logic, or implementation for any representation of the software;
2. to verify that the software under review meets its requirements
3. to ensure that the software has been represented according to predefined standards;
4. to achieve software that is developed in a uniform manner; and
5. to make projects more manageable.

In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation. The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen.

The FTR is actually a class of reviews that include walkthroughs, inspections, round-robin reviews, and other small group technical assessments of software. Each FTR is conducted as a meeting and is successful only if it is properly planned, controlled, and attended.

The Review Meeting

The FTR focuses on a specific and small part of the overall software. For example rather than attempting to review an entire design, walkthroughs are conducted for each module or small group of modules. With a narrower focus, the FTR has a higher likelihood of uncovering errors. Regardless of the chosen FTR format, every review meeting should abide by the following constraints:

- The review should involve between three to five people;
- Advance preparations must be made by each person; and
- The duration of the review meeting should be less than two hours.

The focus of the FTR is on a work product-a component of the software. The individual who has developed the work product informs the project leader that the work product is complete and that a review is required. The project leader contacts a review leader, who evaluates the work product for readiness. Generates copies, and distributes them to two or three reviewers for advance preparation. Each reviewer is expected to spend between one and two hours reviewing the work product, making notes and otherwise becoming familiar with the work. Concurrently, the review leader also reviews the work product and prepares for the review meeting which is typically scheduled for the next day.

The review meeting is attended by the review leader, all reviewers and the producer. One of the reviewers takes on the role of the recorder i.e. the individual who records (in writing) all important issues raised during the review. The FTR begins with an

introduction of the agenda and a brief introduction by the producer. The producer then proceeds to "walk through" the work product, explaining the material, while reviewers raise issues based on their advance preparation. When valid problems or errors are discovered the recorder notes each. At the end of the review, all attendees of the FTR must decide whether to:
1) accept the work product without further modification,
2) reject the work product due to severe errors (once corrected, another review must be performed) or
3) accept the work product provisionally (minor errors have been encountered and must be corrected, but no additional review will be required).
The decision is finally agreed upon and all FTR attendees complete a sign-off indicating their participation in the review and their concurrence with the review team's findings.

Review Reporting and Record Keeping
During the FTR a reviewer (the recorder) actively records all issues that have been raised. These are summarized at the end of the review meeting and a review issues list is produced. In addition, a simple review summary report is completed. A review summary report answers three questions:
1. What was reviewed?
2. Who reviewed it?
3. What were the findings and conclusions?
The review summary report is typically a single page form (with possible attachments). It becomes part of the project historical record and may be distributed to the project leader and other interested parties. The review issues list serves two purposes: (1) to identify problem areas within the product and (2) to serve as an action item checklist that guides the producer as corrections are made. An issues list is normally attached to the summary report. It is important to establish a follow-up procedure to ensure that items on the issues list have been properly corrected. The responsibility for follow-up is assigned to the review leader.

Review Guidelines
Guidelines for the conduct of formal technical reviews must be established in advance, distributed to all reviewers, agreed upon and then followed. A review that is uncontrolled can often be worse than no review at all. Following are a set of guidelines for formal technical reviews:
1. *Review the product, not the producer:* An FTR involves people and egos. Conducted properly, the FTR should leave all participants with a warm feeling of accomplishment. Errors should be pointed out gently; the tone of the meeting should be loose and constructive; and the intent should not be to embarrass or belittle. The review leader should conduct the review meeting to ensure that the proper tone and attitude are maintained and should immediately halt a review that has gotten out of control.
2. *Set an agenda and maintain it:* An FTR must be kept on track and on schedule.
3. *Limit debate and rebuttal*: When an issue is raised by a reviewer, there may not be universal
agreement on its impact. Rather than spending time debating the question, the issue should be recorded for further discussion off-line.

4. *Enunciate problem areas, but don't attempt to solve every problem noted*: A review is not a problem solving session. The solution of a problem can often be accomplished by the producer alone or with the help of only one other individual. Problem solving should be postponed until after the review meeting.
5. *Take written notes*: It is sometimes a good idea for the recorder to make notes on as wall board, so that wording and prioritization can be assessed by other reviewers as information is recorded.
6. *Limit the number of participants and insist upon advance preparation*: Keep the number of people involved to the necessary minimum. However, all review team members must prepare in advance.
7. *Develop a checklist for each work product that is likely to be reviewed*: Checklists should be developed for analysis, design, coding, and even test documents.
8. *Allocate resources and time schedule for FTR.*
9. *Conduct meaningful training for all reviewers*. To be effective all review participants should receive some formal training.
10. *Review your early reviews*.

Exercises

Q.1. Describe the concept of Software Quality. How is quality of software products different from that of hardware products?

Q.2. Describe the importance of Formal Technical Reviews in the Software Quality Assurance process.

3.5. Summary

In this chapter the concepts of Software Configuration Management have been elaborated. SCM is a set of activities that helps in incorporating changes in software. Maintenance refers to the set of activities that occur after the software has been delivered to the customer. SQA is a set of activities that provide the data necessary about product quality, thereby gaining insight and confidence that product quality is meeting its goals.