UNIT 1

INTRODUCTION TO SOFTWARE ENGINEERING

UNIT STRUCTURE

1.0 Introduction

1.1 Unit Objectives

1.2 Professional software development

1.3 Legal Aspects of software engineering

1.4 Unit Summary

1.5 Unit Activities

1.0 Introduction

Software Engineering is about methods, tools and techniques used for developing software. This chapter is concerned with the reasons for having a field of study called software engineering and the problems encountered in developing software. This manual explains various techniques that attempt to solve problems and meet software engineering goals.

1.1 Unit Objectives

By the end of this unit, you should be able to do the following:

- Understand what software engineering is and why it is necessary;
- recognize that different types of software systems may necessitate the use of extra software engineering techniques;
- be aware of legal and professional issues that affect software engineers;

1.2 Professional Software Development

Software engineering has the purpose to support professional software development compared to other type of programming. It includes techniques such as specification, design, and evolution, none of which usually are relevant for personal software development.

A system can be made up of multiple different program files that are used to set up these programs. It could comprise system documentation, which explains how the system works, user documentation, how to use the system, and websites where users can get the most up-to-date product information.

Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production from initial conception to operation and maintenance.
What are the fundamental software engineering activities?	Software specification, software development, software validation and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are the best software engineering techniques and methods?	While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. There are no methods and techniques that are good for everything.
What differences has the Internet made to software engineering?	Not only has the Internet led to the development of massive, highly distributed, service-based systems, it has also supported the creation of an "app" industry for mobile devices which has changed the economics of software.

Fig 1.1: Frequently asked questions about software engineering

Software engineers are concerned with developing software products, that is, software that can be sold to a customer. There are two kinds of software products:

1. Generic products: These are stand-alone systems produced by a development organisation and sold to customers who can buy them on the free market. Examples of this product include mobile devices, software for PCs such as databases, word processors, drawing packages, and project management tools. This software also includes "vertical" applications designed for a specific market.

2. Customised (or bespoke) software: These are developed for a particular customer. A software contractor designs and implements the software especially for that customer.

1.2.1 Software Engineering

Software engineering is an engineering discipline concerned with all aspects of software production, from the early stages of system specification to maintaining the system after its use. In this definition, Software engineering is essential for 2 reasons:

- ✓ Individuals and society rely on technology.
- ✓ In the long term, it is usually cheaper to use software engineering methods and techniques for professional software systems rather than write programs as a personal programming project.

The systematic approach used in software engineering is sometimes called a software process. The latter is a sequence of activities that produce a software product, and four critical activities are common to all processes.

- ✓ Customers and engineers describe the software that will be developed as well as the constraints that will govern its functioning in a software specification.
- ✓ Software development is the process of designing and programming software.
- ✓ Program validation is the process of ensuring that the software meets the needs of the customer.
- ✓ Software evolution is the process of modifying software to meet changing consumer and market needs.

1.2.2 Internet Software Engineering

The development of the Internet and the World Wide Web has profoundly affected all of our lives. At first, the web was primarily a universally accessible information store, and it had little effect on software systems. These systems ran on local computers and were only accessible from within an organisation. This led to developing a vast range of new system products that delivered innovative services accessed over the web. These are often funded by adverts displayed on the user's screen and do not involve direct payment.

This change in software organisation has significantly affected software engineering for web-based systems:

- ✓ Software reuse has become the critical approach for constructing web-based systems. When building the systems, you shoud think about how you can assemble them from previous existing software components and systems, often bundled together in a framework.

- ✓ It is now generally recognised that it is impractical to specify all the requirements for such systems in advance. Web-based systems are constantly developed and delivered incrementally.

- ✓ Interface development technology such as AJAX (Holdener 2008) and HTML5 (Freeman 2011) have emerged that support creates rich interfaces within a web browser.

1.3 Legal Aspects of Software Engineering

Software Engineering is carried out within a social and legal framework. As a software engineer, you must accept that your job involves broader responsibilities than simply applying technical skills. You must also behave ethically and morally responsible if you are to be respected as a professional engineer. However, there are acceptable behaviour are not bound by laws but by the more tenuous notion of professional responsibility. Some of these are:

- Regardless of whether or not a non-disclosure agreement has been signed, you should generally preserve the confidentiality of your employers or clients.

- Competence: You should never exaggerate your level of expertise. You should never accept work that you are not qualified for.

- Intellectual property rights: Be aware of local regulations that control the use of intellectual property, such as patents and copyright. You must take care to preserve the intellectual property of your companies and clients.

- Using your technological talents to abuse other people's computers is not a good idea. Computer misuse can range from the seemingly innocuous (playing games on an employer's computer) to the egregious (playing games on an employer's computer).

1.4 Unit Summary

- Software engineering is an engineering profession dealing with all aspects of software creation, including all electronic documentation required by system users, quality assurance personnel, and developers.
- Maintainability, dependability, and security, as well as efficiency and acceptance, are essential software product features.
- There are numerous sorts of systems, each of which necessitates the use of certain software engineering tools and methodologies. Only a few, if any, design and implementation strategies are applicable to all types of systems.
- All forms of software systems can benefit from the fundamental notions of software engineering. Managed software processes, software dependability and security, requirements engineering, and software reuse are among the essentials.

1.5 Unit Activities

1. Describe why professional software generated for a customer differs from previously developed and provided products.
2. What is the primary distinction between the production of generic software products and the development of custom software? What does this signify for users of generic software products in practice?
3. Explain why using software engineering methodologies and techniques for software systems is usually less expensive in the long term.
4. Software engineering addresses not just system heterogeneity, business and social change, trust, and security, but also ethical considerations that affect the domain. Give some examples of ethical concerns in the software engineering field.

UNIT 2

SOFTWARE PROCESSES

UNIT STRUCTURE

2.0 Introduction

2.1 Unit Objectives

2.2 Software Process Models

2.3 Software Specifications

2.4 Unit Summary

2.5 Unit Activities

2.0 Introduction

A software process is a collection of interconnected operations that results in the creation of a software product. These activities could include creating software from the ground up in a common programming language. This is not always how business applications are created. Extending and altering existing systems, as well as configuring and integrating off-the-shelf software or system components, are frequently used to create new business software.

2.1 Unit Objectives

By the end of this unit, you should be able to do the following:

- Understand the concepts of software processes and software process models; be familiar with three generic software process models and when they might be used;
- be familiar with the fundamental process activities of software requirements engineering, software development, testing, and evolution; and
- be aware of how the Rational Unified Process integrates good software engineering practice to create adaptable software processes.
- Understand why agile software development methodologies are used, the agile manifesto, and the differences between agile and plan-driven development;

2.2 Software Process Models

Each process model depicts a process from a specific viewpoint, and hence only gives a portion of the information about that process. A process activity model, for example, may depict the activities and their sequence but not the responsibilities of the people involved in these activities.

The process model covered here are:

1. **The waterfall model**: Definition, development, validation, and evolution are represented as independent process steps, such as requirements specification, software design, implementation, testing, and so on.

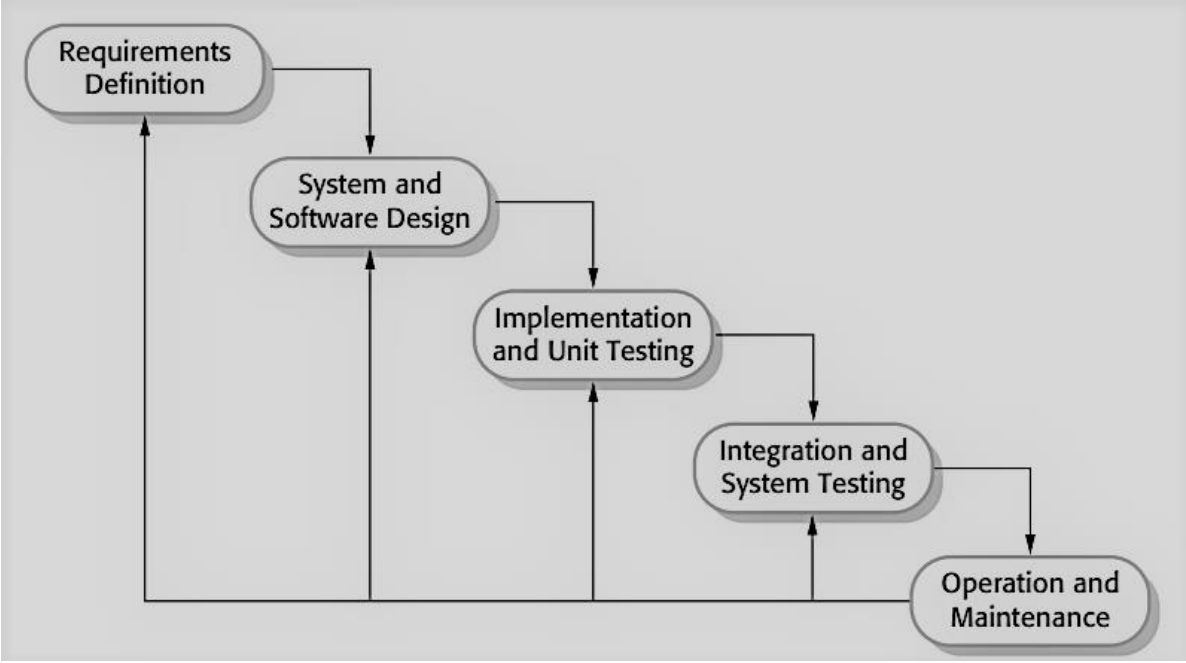

Fig 1.2 The waterfall model

The principal stages of the waterfall model directly reflect the fundamental development activities:

- ✓ **Requirements analysis and definition:** Consultation with system users determines the system's services, restrictions, and goals. They're then detailed specified and used to create a system specification.

- ✓ **System and software design:** The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.

- ✓ **Implementation and unit testing:** During this stage, the software design is realised as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

- ✓ **Integration and system testing:** To confirm that the software criteria have been met, the separate program modules or programs are combined and tested as a whole system. The software system is supplied to the customer once it has been thoroughly tested.

- ✓ **Operation and maintenance:** This is usually (but not always) the longest period of the life cycle. The system has been installed and put to use. Maintenance entails rectifying mistakes that were not identified earlier in the life cycle, improving system unit implementation, and extending the system's services when new requirements emerge.

The waterfall approach is similar to other engineering process models, and documentation is generated at each stage. This makes the process visible, allowing managers to keep track of how things are doing in relation to the development strategy. Its most serious flaw is the project's rigid division into stages. It's tough to respond to changing client requirements since commitments must be made early in the process. In general, the waterfall model should be utilized only when the requirements are well defined and unlikely to change significantly throughout system development. The waterfall model, on the other hand, is representative of the procedure utilized in other engineering projects. Software processes based on the waterfall model are still widespread since it is easier to apply a common management approach for the entire project.

2. **Incremental Development:** Incremental development is based on the idea of developing an initial implementation, exposing this to user comment and evolving it through several versions until an adequate system has been developed (Figure 1.3). Specification, development, and validation activities are interleaved rather than separate, with rapid feedback across activities.

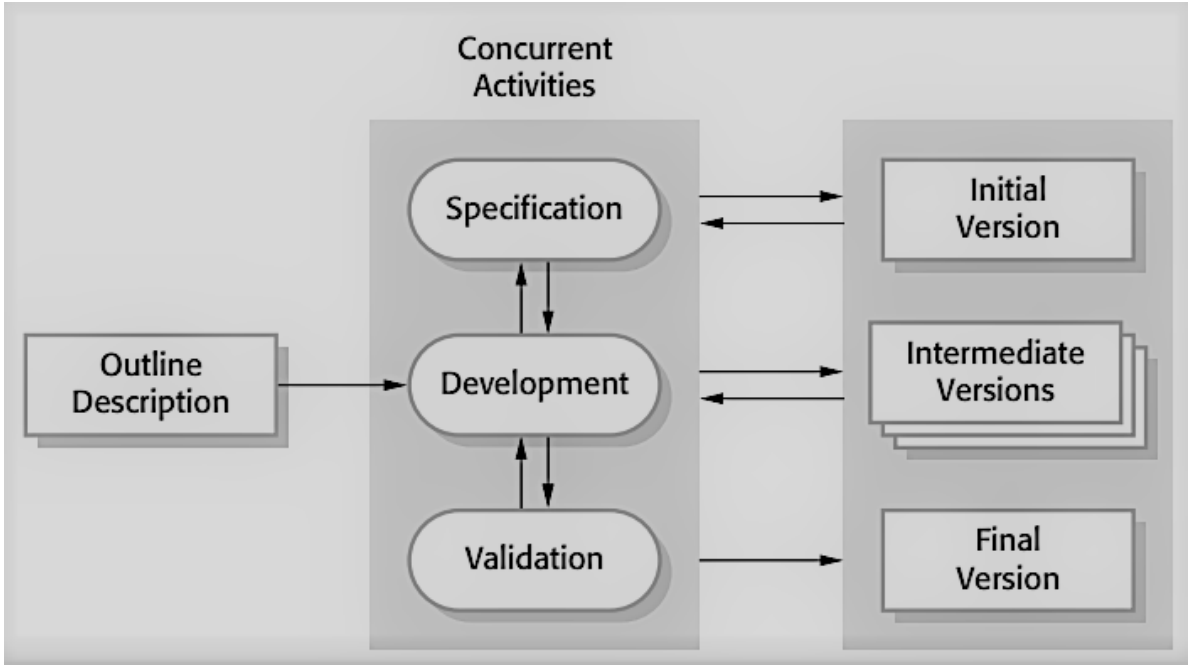

Fig 1.3 Incremental development

Incremental software development, which is a fundamental part of agile approaches, is better than a waterfall approach for most business, e-commerce, and personal systems. Incremental development reflects the way that we solve problems. By developing the software incrementally, it is cheaper and easier to make changes in the software as it is being developed.

Incremental development has three important benefits, compared to the waterfall model:

- ✓ The cost of accommodating changing customer requirements is reduced. The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.

- ✓ It is much easier to obtain client feedback on previously completed development work. Customers can leave feedback on software demonstrations and observe how much has been incorporated. Customers have a hard time judging progress based on software design papers.

- ✓ Even if all of the functionality is not provided, more rapid delivery and deployment of valuable software to the client is achievable. Customers can use and benefit from the product more sooner than they might with a waterfall process.

From a management perspective, the incremental approach has two problems:

- ✓ The procedure is undetectable. Managers require regular deliverables in order to track progress. When systems are developed quickly, producing papers that reflect every version of the system is not cost-effective.

- ✓ As additional increments are added, the system structure tends to deteriorate. Regular change tends to destroy the software's structure unless time and money are spent on refactoring to improve it. Further software modifications become more difficult and expensive to incorporate.

3. **Reuse-oriented software engineering:** There is some software reuse in the majority of software initiatives. This frequently occurs informally when project team members are aware of designs or code that are comparable to what is desired. They seek them out, tweak them as needed, and integrate them into their system.

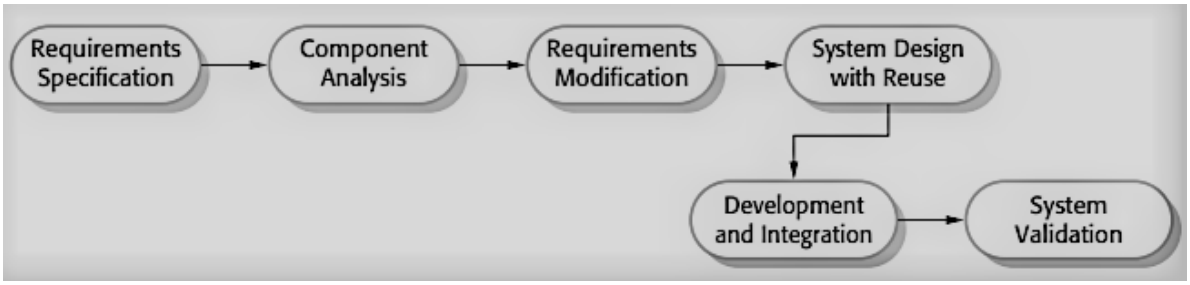

Fig 1.4 Reuse-oriented software engineering

A general process model for reuse-based development is shown in Figure 2.3. Although the initial requirements specification stage and the validation stage are comparable with other software processes, the intermediate stages in a reuseoriented process are different. These stages are:

- ✓ Analyze the components A search for components to implement the requirements specification is conducted based on the requirements specification. In most cases, there is no exact match, and the components that can be employed only give a portion of the desired functionality.

- ✓ Modification of the requirements Using the knowledge about the components that have been discovered, the requirements are analyzed at this step. They're then tweaked to reflect the components that are available. If changes aren't possible, the component analysis activity can be re-entered to look for other options.

- ✓ System design with reuse During this phase, the framework of the system is designed or an existing framework is reused. The designers take into account the components that are reused and organise the framework to cater for this. Some new software may have to be designed if reusable components are not available.

- ✓ Integration and development The components and COTS systems are merged to build the new system, and software that cannot be obtained outside is produced. In this concept, system integration might be a part of the development process rather than a distinct operation.

There are three types of software component that may be used in a reuse-oriented process:

- ✓ Web services that are developed according to service standards and which are available for remote invocation.

- ✓ Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.

- ✓ Stand-alone software systems that are configured for use in a particular environment.

2.3 Software Specification

Software requirements or specifications Understanding and specifying what services are required from the system, as well as determining the limits on the system's functioning and evolution, is the process of engineering. Requirements engineering is a crucial stage in the software development process since failures here invariably lead to difficulties later in the system design and execution.

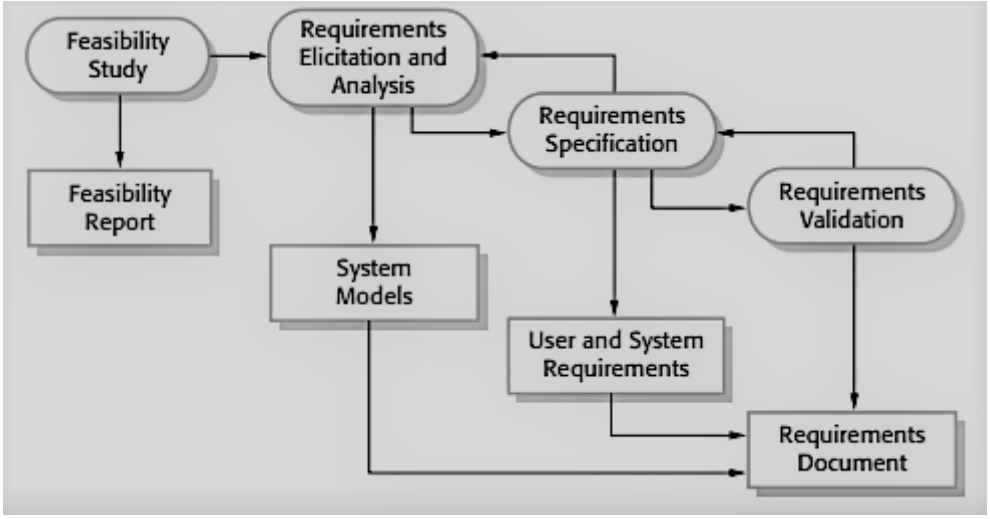

Fig 1.4 The requirement engineering process

There are four main activities in the requirements engineering process:

- ✓ Feasibility analysis The indicated user needs are assessed to see if they can be met with current software and hardware solutions. The analysis examines whether the proposed system is cost-effective from a business standpoint and whether it can be constructed given current budgetary restrictions. A feasibility study should be inexpensive and quick to complete. The outcome should help you decide whether or not to conduct a more thorough investigation.

- ✓ Elicitation and analysis of requirements This is the process of determining system needs by looking at current systems, talking to possible users and procurers, doing task analysis, and so on. This could entail creating one or more system models and prototypes. These aid in the comprehension of the system to be detailed.

- ✓ Specification of requirements The activity of turning the information acquired during the analysis activity into a document that describes a set of requirements is known as requirements specification. This document may contain two sorts of requirements. System requirements are a more specific description of the functionality to be delivered; user requirements are abstract assertions of the system needs for the customer and end-user of the system.

- ✓ Validation of requirements This activity verifies that all of the standards for realism, consistency, and completeness have been met. Errors in the requirements document will surely be detected during this procedure. It must then be changed to address these issues.

2.4 Unit Summary

- The operations that go into creating a software system are referred to as software processes. Abstract representations of these processes are known as software process models.
- The organization of software processes is described by general process models. The waterfall model, incremental development, and reuse-oriented development are all examples of broad models.
- The process of creating a software specification is known as requirements engineering. The purpose of specifications is to communicate the customer's system requirements to the system developers.
- The transformation of a requirements specification into an executable software system is the focus of the design and implementation phases. As part of this change, systematic design methodologies may be applied.

2.5 Unit Activities

1. Giving reasons for your answer based on the type of system being developed, suggest the most appropriate generic software process model that might be used as a basis for managing the development of the following systems:

- A system to control anti-lock braking in a car
- A virtual reality system to support software maintenance
- A university accounting system that replaces an existing system
- An interactive travel planning system that helps users plan journeys with the lowest environmental impact

2. Explain why incremental development is the most effective approach for developing business software systems. Why is this model less appropriate for real-time systems engineering?

3. Describe the main activities in the software design process and the outputs of these activities. Using a diagram, show possible relationships between the outputs of these activities.

UNIT 3

REQUIREMENTS ENGINEERING

UNIT STRUCTURE

3.0 Introduction

3.1 Unit Objectives

3.2 Functional and non-functional requirements

3.3 Types of functional requirements and their specifications

3.4 Software Requirements Specification Document

3.5 Unit Summary

3.6 Unit Activities

3.0 Introduction

Prior to the execution of most major systems, there is still a clearly recognizable requirements engineering phase. The end result is a set of requirements that may be included in the system development contract. Naturally, the requirements are subject to change, and user requirements may be developed into more detailed system requirements.

3.1 Unit Objectives

By the end of this unit, you should be able to do the following:

■ understand the concepts of user and system requirements and why these requirements should be written in different ways;

■ understand the differences between functional and nonfunctional software requirements;

■ understand how requirements may be organised in a software requirements document;

3.2 Functional and Non Functional Requirements

Software system requirements are often classified as functional requirements or nonfunctional requirements:

1. Functional requirements: These are assertions about the services the system should deliver, how it should respond to specific inputs, and how it should behave in specific situations. The functional requirements may also specify explicitly what the system should not perform in particular instances.

2. Non-functional requirements: These are limitations on the system's services or functions. They include time restrictions, development process constraints, and standards-based constraints. Non-functional requirements are frequently applied to the entire system rather than specific system features or services.

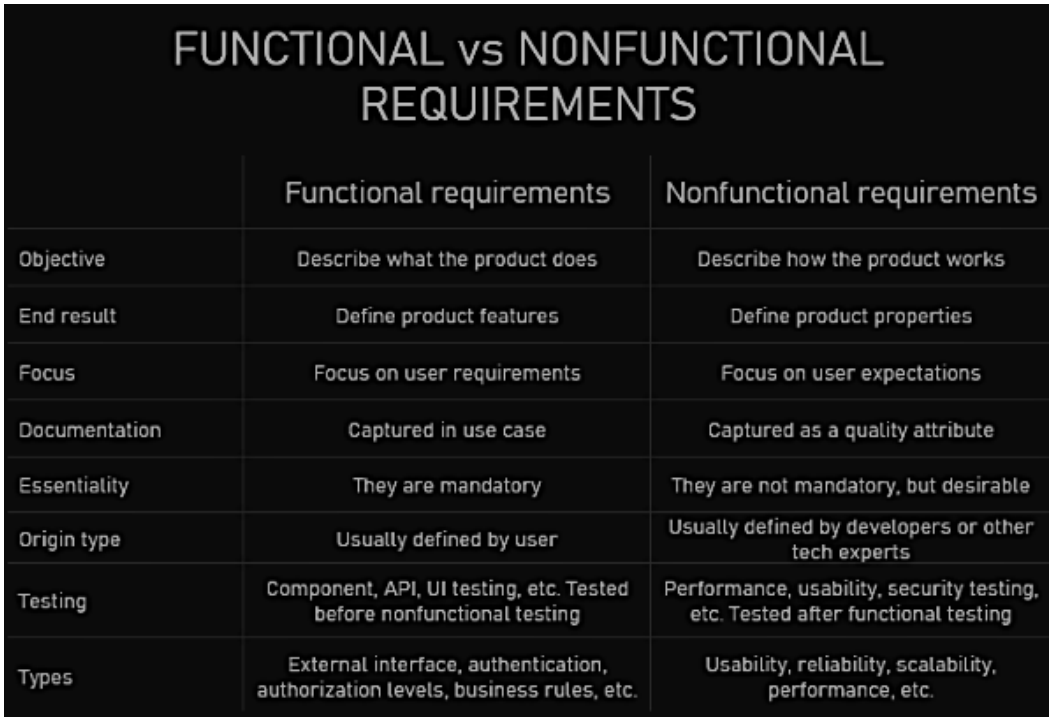

Fig 1.5 Functional vs Non Functional Requirements

3.3 Types of functional requirements and their specifications

Functional requirements can be classified according to different criteria. For example, we can group them on the basis of the *functions* a given feature must perform in the end product. Of course, they would differ depending on the product being developed, but for the sake of an example, the types of functional requirements might be

- Authentication
- Authorisation levels
- Compliance to laws or regulations
- External interfaces
- Transactions processing
- Reporting
- Business rules

Requirements are usually written in text, especially for Agile-driven projects. However, they may also be visuals. Here are the most common formats and documents:

- Software requirements specification document
- Use cases
- User stories
- Work Breakdown Structure (WBS), or functional decomposition
- Prototypes
- Models and diagrams

3.4 Software Requirements Specification Document

The software requirements specification (SRS) document can be used to formalize both functional and nonfunctional needs. Read our article on software documentation to learn more about it in general. The SRS offers descriptions of the product's required functions and capabilities. Constraints and assumptions are also defined in the paper. The SRS can be a standalone document that communicates functional requirements, or it can be used in conjunction with other software documents such as user stories and use cases.

1. SRS must include the following sections:

Purpose. Definitions, system overview, and background.

Overall description. Assumptions, constraints, business rules, and product vision.

Specific requirements. System attributes, functional requirements, and database requirements.

It's essential to make the SRS readable for all stakeholders. You also should use templates with visual emphasis to structure the information and aid in understanding it. If you have requirements stored in some other document formats, provide a link to them so that readers can find the needed information.

Below is an example of a concise list of SRS contents:

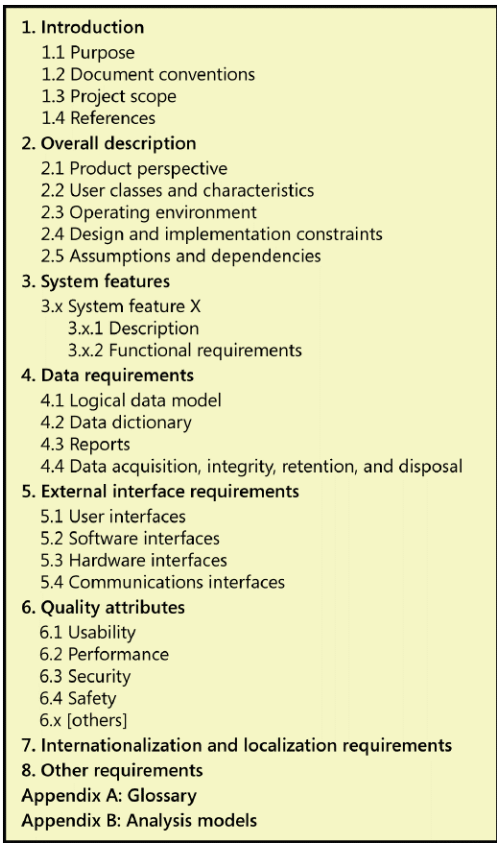

Fig 1.6 Template for SRS document

2. Use cases

Use cases describe the interaction between the system and external users that leads to achieving particular goals.

Each use case includes three main elements:

Actors. These are the external users that interact with the system.

System. The system is described by functional requirements that define an intended behavior of the product.

Goals. The purposes of the interaction between the users and the system are outlined as goals.

There are two formats to represent use cases:

- Use case specification structured in textual format
- Use case diagram

A **use case specification** represents the sequence of events along with other information that relates to this use case. A typical use case specification template includes the following information:

- Description
- Pre- and Post- interaction condition
- Basic interaction path
- Alternative path
- Exception path

Overview	
Title	[Title of the basic flow use case]
Description	[Short description of the basic flow]
Actors and Interfaces	[Identifies the Actors and Interfaces to components and services that participate in the use case]
Initial Status and Preconditions	[A pre-condition (of a use case) is the state of the system that must be present prior to a use case being performed]
Basic Flow	
STEP 1: … STEP 2: …	
Post Condition	
[A post-condition (of a use case) is a list of possible states the system can be in immediately after a use case has finished]	
Alternative Flow(s)	
[Alternative flows are described here if needed]	

Fig 1.7 Template for Use Case Specification

A **use case diagram** doesn't contain a lot of details. It shows a high-level overview of the relationships between actors, different use cases, and the system.

The use case diagram includes the following main elements:

- **Use cases.** Usually drawn with ovals, use cases represent different interaction scenarios that actors might have with the system (*log in, make a purchase, view items,* etc.).
- **System boundaries.** Boundaries are outlined by the box that groups various use cases in a system.
- **Actors.** These are the figures that depict external users (people or systems) that interact with the system.
- **Associations.** Associations are drawn with lines showing different types of relationships between actors and use cases.

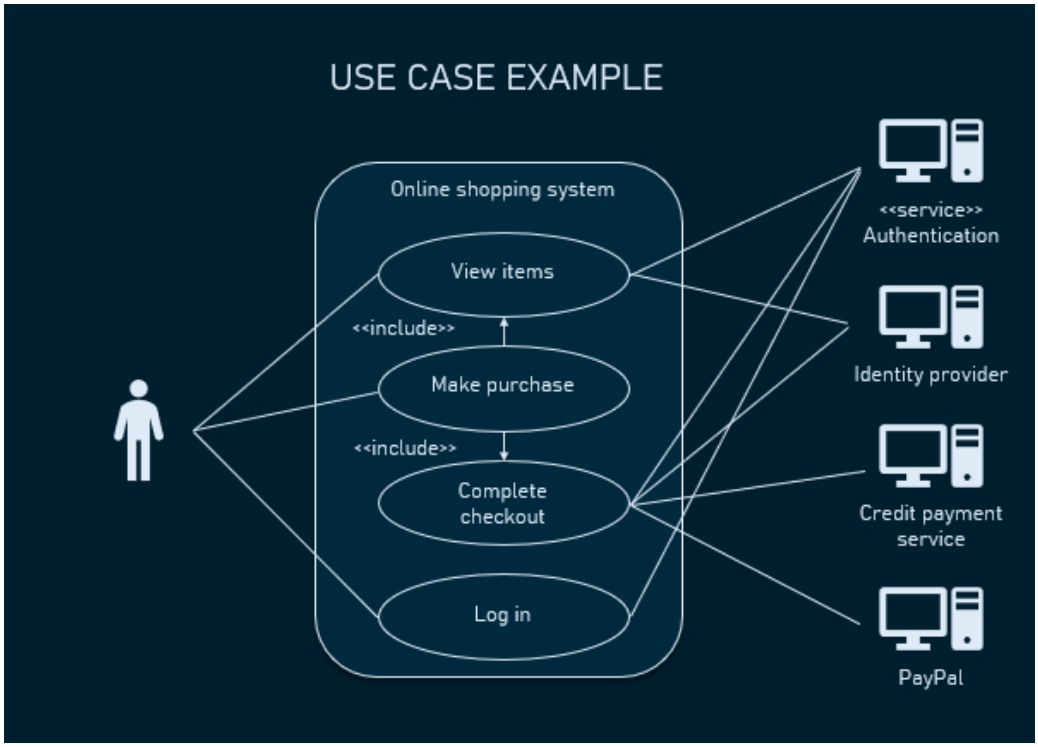

Fig 1.7 Use Case Example

3. Work Breakdown Structure (WBS)

A functional decomposition, often known as a WBS, is a diagram that shows how complicated operations are broken down into their constituent parts. WBS is a useful tool for allowing each part to be analyzed independently. WBS also aids in obtaining a complete view of the project.

The decomposition process may look like this:

High Level Function ->Sub-function -> Process -> Activity

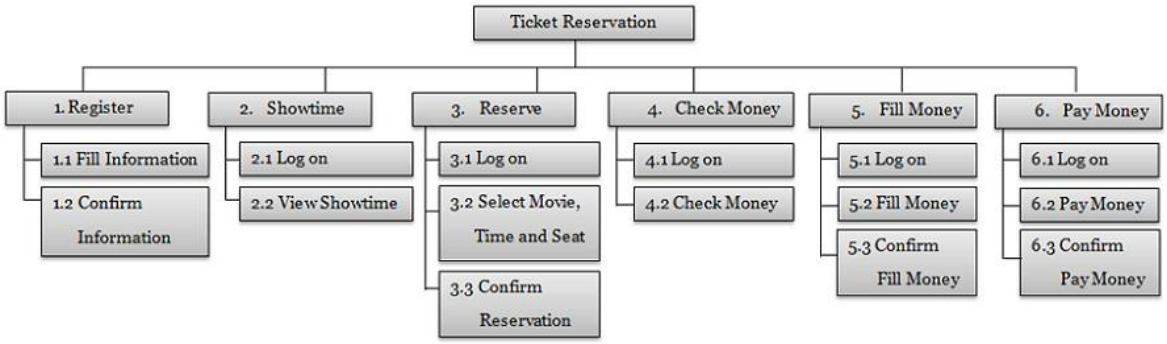

Fig 1.8 WBS Example

3.5 Non Functional Requirements

The definition of non-functional requirements is quality attributes that describe ways your product should behave. The list of basic non-functional requirements includes:

3.5.1 Usability

- **Efficiency of use:** the average time it takes to accomplish a user's goals, how many tasks a user can complete without any help, the number of transactions completed without errors, etc.
- **Intuitiveness:** how simple it is to understand the interface, buttons, headings, etc.
- **Low perceived workload:** how many attempts users need to accomplish a particular task.

Example: Usability requirements can consider language barriers and localisation tasks: People with no understanding of French must be able to use the product. Or you may set accessibility requirements: Keyboard users who navigate a website using <tab>, must be able to reach the "Add to cart" button from a product page within 15 <tab> clicks.

3.5.2 Security

Security standards ensure that the software is safeguarded against unauthorized access to the system and its data. It takes into account various degrees of authorisation and authentication for various user roles. Data privacy, for example, is a security feature that outlines who can generate, see, copy, alter, or remove data. Security also includes anti-virus and anti-malware protection.

Example: Access permissions for the particular system information may only be changed by the system's data administrator.

3.5.3 Reliability

The term "reliability" refers to the likelihood that software will work without fail for a specific amount of time. Because of defects in the programming, hardware failures, or issues with other system components, reliability suffers. You can count the percentage of actions that are completed correctly or track the average duration the system operates before failing to measure software dependability.

Example: The database update process must roll back all related updates when any update fails.

3.5.4 Performance

Performance is a quality attribute that describes how responsive a system is to different types of user interactions. User experience suffers as a result of poor performance. When the system is overloaded, it also puts the system's safety at jeopardy.

Example: The front-page load time must be no more than 2 seconds for users that access the website using an LTE mobile connection.

3.5.5 Availability

The amount of time that the system's functionality and services are available for use with all operations is measured by availability. As a result, scheduled maintenance periods have a direct impact on this parameter. It's also crucial to determine how to reduce the impact of maintenance. The team must describe the most crucial system components that must be available at all times while defining the availability requirements. You should also have user notifications ready in case the system or one of its components goes down.

Example: New module deployment mustn't impact front page, product pages, and check out pages availability and mustn't take longer than one hour. The rest of the pages that may experience problems must display a notification with a timer showing when the system is going to be up again.

3.5.6 Scalability

The system's scalability criteria outline how it must grow without compromising its performance. This implies more people will be served, more data will be processed, and more transactions will be completed. Both hardware and software are affected by scalability. You can boost scalability by adding RAM, servers, or disk space, for example. You can, on the other hand, compress data, apply optimization algorithms, and so on.

Example: The website attendance limit must be scalable enough to support 200,000 users at a time.

3.6 Unit Summary

■ Requirements for a software system set out what the system should do and define constraints on its operation and implementation.

■ Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.

■ Non-functional requirements often constrain the system being developed and the development process being used. These might be product requirements, organisational requirements, or external requirements. They often relate to the emergent properties of the system and therefore apply to the system as a whole.

■ The software requirements document is an agreed statement of the system requirements. It should be organised so that both system customers and software developers can use it.

■ The requirements engineering process includes a feasibility study, requirements elicitation and analysis, requirements specification, requirements validation, and requirements management.

3.7 Unit Activities

1. Identify and briefly describe four types of requirement that may be defined for a computerbased system.
2. Write a set of non-functional requirements for the ticket-issuing system, setting out its expected reliability and response time.
3. Who should be involved in a requirements review? Draw a process model showing how a requirements review might be organised.

UNIT 4

SYSTEM MODELING

UNIT STRUCTURE

4.0 Introduction

4.1 Unit Objectives

4.2 Context Models

4.3 Use Case Models

4.4 Sequence Diagrams

4.5 Class Diagrams

4.6 Unit Summary

4.7 Unit Activities

4.0 Introduction

The practice of creating abstract models of a system, each of which presents a different view or perspective on that system, is known as system modeling. System modeling has come to entail expressing a system with some type of graphical notation, which is nearly always based on the Unified Modeling Language notations (UML).

4.1 Unit Objectives

By the end of this unit, you should be able to do the following:

■ understand how graphical models can be used to represent software systems;

■ understand why different types of model are required and the fundamental system modeling perspectives of context, interaction, structure, and behavior;

■ have been introduced to some of the diagram types in the Unified Modeling Language (UML) and how these diagrams may be used in system modeling;

■ be aware of the ideas underlying model-driven engineering, where a system is automatically generated from structural and behavioral models.

4.2 Context Models

You should decide on the system boundaries early in the system specification process. Working with system stakeholders to determine what functionality should be included in the system and what the system's environment provides is part of this process. You may determine that some business operations should have automated support, while others should be human or supported by distinct systems. You should consider any functional similarities with current systems before deciding where new functionality should be deployed. These decisions should be taken early in the process to keep system costs down and the time it takes to understand and build the system down.

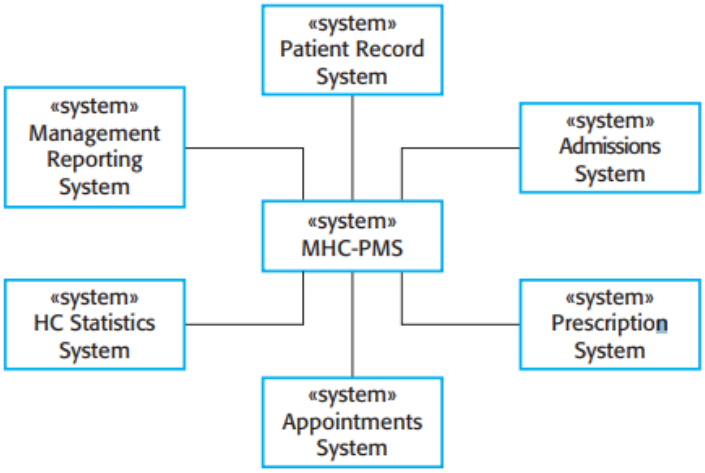

Fig 1.9 The context diagram for PMS

Context models typically show the presence of multiple different automated systems in the environment. They do not, however, illustrate the types of relationships that exist between the environment's systems and the system that is being specified. External systems may generate data for the system or consume data from it. They may share data with the system, or they may be directly connected, via a network, or not at all. They could be in the same building or in different buildings. All of these relationships can have an impact on the requirements and design of the system being created, so they must be considered.

4.3 Use case Modeling

Use case modeling was originally developed by Jacobson et al. (1993) in the 1990s and was incorporated into the first release of the UML (Rumbaugh et al., 1999). A use case can be taken as a simple scenario that describes what a user expects from a system.

Fig 2.0 Transfer Data Use Case

Use case diagrams provide a quick overview of an interaction, so you'll need to go into more detail to fully comprehend what's going on. As explained below, this detail can be a basic textual description, a structured description in a table, or a sequence diagram. Depending on the use case and the level of detail you believe is required in the model, you chose the most appropriate format. The most useful format for me is a regular tabular format.

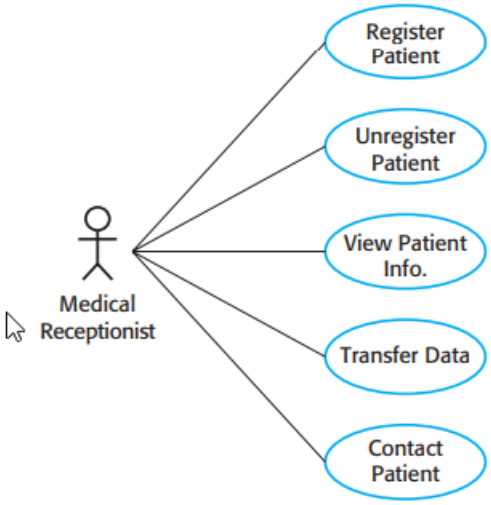

Fig 2.1 Use cases involving the role' medical receptionist'

4.4 Sequence Diagrams

In the UML, sequence diagrams are generally used to model interactions between actors and objects in a system, as well as interactions between items. The UML includes a comprehensive syntax for sequence diagrams, allowing for the modeling of a wide range of interactions.

The objects and actors involved are mentioned at the top of the diagram, with a vertical dotted line traced from them. Annotated arrows show how items interact with each other. The lifeline of the object is indicated by the rectangle on the dotted lines (i.e., the time that object instance is involved in the computation). From top to bottom, you read the chain of exchanges. The calls to the objects, their parameters, and their return values are shown by the annotations on the arrows. I also show the notation used to represent alternatives in this case. The conditions in square brackets are utilized in a box called alt.

1. The medical receptionist triggers the ViewInfo method in an instance P of the PatientInfo object class, supplying the patient's identifier, PID. P is a user interface object, which is displayed as a form showing patient information.

2. The instance P calls the database to return the information required, supplying the receptionist's identifier to allow security checking (at this stage, we do not care where this UID comes from).

3. The database checks with an authorisation system that the user is authorised for this action.

4. If authorised, the patient information is returned and a form on the user's screen is filled in. If authorisation fails, then an error message is returned.

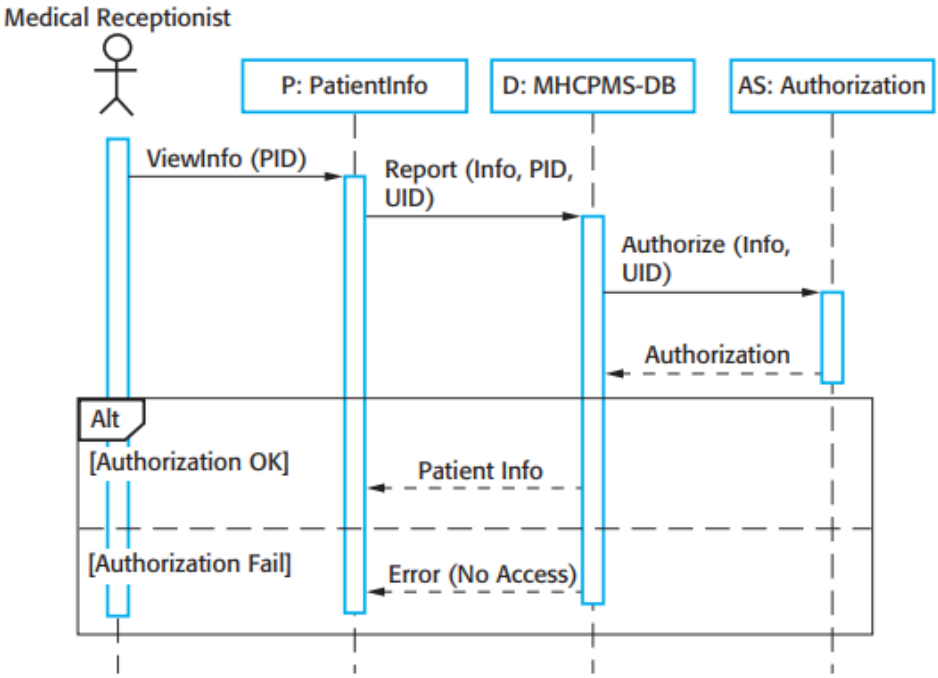

Fig 2.2 Sequence diagram for View patient information

You can read this diagram as follows:

1.The receptionist logs on to the PRS for the first time.

2. There are two possibilities for you to choose from. These enable direct patient information updates to the PRS as well as the transfer of summary health data from the MHC-PMS to the PRS.

3. The permissions of the receptionist are checked using the authorisation system in each situation.

4. Personal data can be sent from the user interface object to the PRS directly. Alternatively, the database can be used to build a summary record, which is then transferred.

5. The PRS sends a status message when the transfer is complete, and the user logs off.

4.5 Class Diagrams

Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes. Loosely, an object class can be thought of as a general definition of one kind of system object. An association is a link between classes that indicates that there is a relationship between these classes. Consequently, each class may have to have some knowledge of its associated class.

In the UML, class diagrams can be expressed at various levels of depth. When creating a model, the initial step is to observe the world, determine the most important items, and represent them as classes. The most straightforward method is to write the class name in a box. You can also simply make a note of the existence of a link.

Class diagrams resemble semantic data models at this degree of depth. In database design, semantic data models are utilized. They display data entities, their associated attributes, and the relationships that exist between them. Chen (1976) proposed this approach to modeling in the mid-1970s, and since then, other modifications have been developed (Codd, 1979; Hammer and McLeod, 1981; Hull and King, 1987), all having the same fundamental shape.

Because it assumes an object-oriented development process and models data using objects and their relationships, the UML does not provide a special notation for database modeling. The UML, on the other hand, can be used to represent a semantic data model. Entities in a semantic data model can be thought of as simplified object classes.

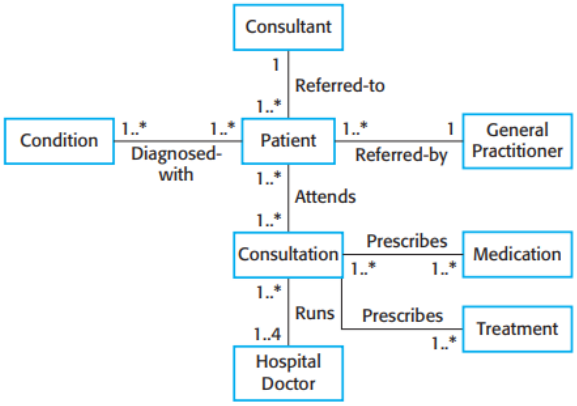

Fig 2.3 Classes and associations in the MHC-PMS

4.6 Unit Summary

- A model is a simplified representation of a system that ignores some of the system's specifics. Complementary system models can be created to depict the context, interactions, structure, and behavior of the system.

- Context models depict how a represented system interacts with other systems and processes in its surroundings. They aid in the definition of the system's limits.

- Use case diagrams and sequence diagrams are used to depict the interactions between users and the system under development. Use cases illustrate interactions between a system and external actors; sequence diagrams expand on this by depicting interactions among system objects.

4.7 Unit Activities

1. 1. Describe why it is critical to model the context of a system in development. Give two examples of problems that could occur if software engineers are unaware of the system's context.
2. 2. How might you make use of an existing system model? Explain why having a complete and correct system model isn't always necessary. Would the same be true if you were designing a new system's model? In a requirements review, who should be involved? Create a flowchart to show how a requirements review might be conducted.
3. 3. You've been tasked with creating a system to assist in the organizing of large-scale events and parties, such as weddings, graduation parties, and birthday parties.parties such as weddings, graduation celebrations, birthday parties, etc. Using an activity diagram, model the process context for such a system that shows the activities involved in planning a party (booking a venue, organising invitations, etc.) and the system elements that may be used at each stage.

UNIT 5

SOURCE CODE MANAGEMENT

UNIT STRUCTURE

5.0 Introduction

5.1 Unit Objectives

5.2 Importance of source code management tools

5.3 Benefits of source code management

5.4 Source code management best practices

5.5 Unit Summary

5.6 Unit Activities

5.0 Introduction

The term source code management (SCM) refers to the process of tracking changes to a source code repository. SCM keeps track of a code base's history of changes and assists in resolving conflicts when merging updates from various contributors. Version control is also referred to as SCM. As the number of lines of code and contributors on a software project grows, so do the expenses of communication overhead and management complexity. SCM is a vital instrument for reducing the strain on organizations caused by rising development expenses.

5.1 Unit Objectives

By the end of this unit, you should be able to do the following:

- understand the importance of source code tools
- understand the benefits of source code management
- have been introduced to source code management best practices

5.2 Importance of source code management tools

When multiple developers are working within a shared codebase it is a common occurrence to make edits to a shared piece of code. Separate developers may be working on a seemingly isolated feature, however this feature may use a shared code module. Therefore developer 1 working on Feature 1 could make some edits and find out later that Developer 2 working on Feature 2 has conflicting edits.

Before the adoption of SCM this was a nightmare scenario. Developers would edit text files directly and move them around to remote locations using FTP or other protocols. Developer 1 would make edits and Developer 2 would unknowingly save over Developer 1's work and wipe out the changes. SCM's role as a protection mechanism against this specific scenario is known as Version Control.

Version control protections were added to SCM to prevent work loss due to conflict overwriting. These safeguards function by tracking each developer's modifications, detecting areas of conflict, and prohibiting overwrites. SCM will then notify the developers of these points of contention so that they can safely evaluate and resolve them.

This basic conflict resolution process also serves as a form of passive communication for the development team. The team can then keep track of the work in progress that the SCM is keeping track of and discuss it. The SCM keeps track of all modifications to the code base throughout time. This enables developers to investigate and review adjustments that may have resulted in problems or regressions.

5.3 Benefits of source code management

SCM provides a suite of other useful capabilities in addition to version control to make collaborative code development more user friendly. SCM creates a detailed historical record of a project's life once it begins tracking all changes to it over time. Changes to the codebase can then be 'undone' using this historical record. The SCM can instantaneously rollback the codebase to a previous version. This is highly useful for avoiding update regressions and rectifying mistakes.

The SCM archive of every modification made during the life of a project is invaluable for keeping track of release version notes. Release notes can be interchanged with a clean and well-maintained SCM history log. This provides clarity and insight.

SCM reduces a team's communication overhead and speeds up release cycles. Without SCM, development takes longer because contributors must make an extra effort to plan a non-overlapping release sequence. Developers can work on separate branches of feature development independently with SCM, eventually merging them together.

Overall, SCM is a significant help to engineering teams, as it allows engineering personnel to execute more efficiently, lowering development expenses. In today's world of software development, SCM is a must-have. Version control is used by professional teams, and it should be used by yours as well.

5.4 Source code management best practices

5.4.1 Commit often

Commits are inexpensive and simple to create. They should be done on a regular basis to capture changes to a code base. Each commit is a snapshot of the codebase that can be rolled back if necessary. Many opportunities to rollback or undo work are provided by frequent commits. To clarify the development log, a rebase can be used to consolidate multiple commits into a single commit.

5.4.2 Ensure you are working from latest version

Multiple developers can quickly update a project using SCM. It's very easy for a local copy of the codebase to become out of sync with the global copy. Before making any changes, be sure you git pull or retrieve the most recent code. This will aid in the avoidance of conflicts throughout the merge process.

5.4.3 Make detailed notes

There is a log record for each commit. This log entry is filled with a message at the time of commit creation. It's critical to leave descriptive commit log entries that explain what's going on. The "why" and "what" of the commit's content should be explained in these commit log messages. These log messages serve as the project's canonical history, leaving a trail for future contributors to follow.

5.4.4 Review changes before committing

SCM's offer a 'staging area'. The staging area can be used to collect a group of edits before writing them to a commit. The staging area can be used to manage and review changes before creating the commit snapshot. Utilising the staging area in this manner provides a buffer area to help refine the contents of the commit.

5.4.5 Use Branches

Branching is a strong SCM feature that allows developers to construct a new development line. Because branches are quick and inexpensive, they should be utilized regularly. Branches allow different developers to work on various lines of code in simultaneously. Different product characteristics are often the focus of these development lines. When a branch's development is finished, it is merged into the main development line.

5.4.6 Agree on Workflow

By default, SCMs provide relatively open-ended contribution techniques. It's critical for teams to develop shared collaboration patterns. Patterns and techniques for branch merging are established by SCM workflows. When it comes time to merge branches, if a team doesn't agree on a shared workflow, it might lead to inefficient communication overhead.

5.6 Unit Summary

■ SCM is an invaluable tool for modern software development. The best software teams use SCM, and your team should be too.

■ CM is straightforward to set up on a new project, and the return on investment is high.

■ Atlassian offers some of the best SCM integration tools in the world that will help you get started.

5.7 Unit Activities

1. Explain why it is important to use tools for source code management.
2. What are the benefits of source code management.
3. List down the code management best practices.

UNIT 6

AGILE SOFTWARE DEVELOPMENT

UNIT STRUCTURE

6.0 Introduction

6.1 Unit Objectives

6.2 What is Agile

6.3 The Agile Manifesto

6.4 Agile Software Development

6.5 What is Sprint

6.7 Unit Summary

6.8 Unit Activities

6.0 Introduction

Businesses currently operate in a global, fast-paced world. They must adjust to new markets and possibilities, shifting economic conditions, and the advent of competitive products and services. Because software is used in practically every aspect of business, new software is developed swiftly to capitalize on new opportunities and adapt to competitive pressure. As a result, rapid development and delivery is increasingly frequently the most important need for software systems.

6.1 Unit Objectives

By the end of this unit, you should be able to do the following:

- understand the agile methodology
- understand the agile methodologies
- understand the concept of scrum
- understand the concept of sprints

6.1 What is Agile?

Agile project management and software development is an iterative method that helps teams offer value to clients faster and with fewer headaches. An agile team delivers work in tiny, digestible increments rather than relying on a "big bang" launch. Teams have a natural mechanism for adjusting to change fast because requirements, strategies, and results are evaluated on a regular basis.

6.2 The Agile Manifesto

Agile software development is an umbrella term for a set of frameworks and practices based on the values and principles expressed in the Manifesto for Agile Software Development and the 12 Principles behind it. When you approach software development in a particular manner, it's generally good to live by these values and principles and use them to help figure out the right things to do given your particular context.

We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value:
Individuals and interactions *over processes and tools*
Working software *over comprehensive documentation*
Customer collaboration *over contract negotiation*
Responding to change *over following a plan*
That is, while there is value in the items on the right, we value the items on the left more.

The following 12 Principles are based on the Agile Manifesto:

1	Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.	7	Working software is the primary measure of progress.
2	Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.	8	Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
3	Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.	9	Continuous attention to technical excellence and good design enhances agility.
4	Business people and developers must work together daily throughout the project.	10	Simplicity–the art of maximizing the amount of work not done–is essential.
5	Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.	11	The best architectures, requirements, and designs emerge from self-organizing teams.
6	The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.	12	At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Fig 2.4 12 Principles of Agile Manifesto

6.3 What is Agile Software Development?

Agile software development is more than frameworks such as Scrum, Extreme Programming, or Feature-Driven Development (FDD). Agile software development is more than practices such as pair programming, test-driven development, stand-ups, planning sessions, and sprints.

One thing that separates Agile from other approaches to software development is the focus on the people doing the work and how they work together. Solutions evolve through collaboration between self-organising cross-functional teams utilising the appropriate practices for their context.

The Agile software development community places a strong emphasis on cooperation and the self-organizing team.

That isn't to say there aren't any managers. It suggests that groups are capable of figuring out how to handle problems on their own.
Those teams are cross-functional, in other words. Those teams don't need to have distinct jobs; instead, when they get together, they should make sure they have all of the necessary skill sets.

6.4 What are Agile Methodologies?

What does it mean if Agile is a mindset? What does it mean if Agile approaches are a mindset? It may be helpful to have a clear definition of methodology to address this question.
A methodology, according to Alistair Cockburn, is a collection of conventions that a group agrees to follow. That means that each team will have its own methodology, which will differ from the methodologies of the other teams in tiny or big ways.
As a result, Agile techniques are the conventions that a team decides to follow while adhering to Agile values and principles.

6.5 What is Scrum?

Scrum is a framework for facilitating teamwork. Scrum encourages teams to learn via experiences, self-organize while working on an issue, and reflect on their victories and losses to continuously improve, much like a rugby team (from which it gets its name).
While the scrum I'm referring to is most commonly utilized by software development teams, the concepts and lessons it teaches may be applied to any type of teamwork. One of the reasons scrum is so popular is because of this. Scrum is a combination of meetings, tools, and roles that work together to help teams structure and manage their work. It's sometimes referred to as an agile project management framework.

6.6 What are Sprints?

A sprint is a condensed period of time during which a scrum team works to complete a specific amount of work. Scrum and agile approaches are built around sprints, and getting sprints right can help your agile team ship better product with fewer issues.

Due to the many parallels between agile values and scrum procedures, a reasonable comparison can be made. Sprints assist teams in adhering to the agile principle of "often delivering working software" as well as the agile value of "responding to change over with a plan." Transparency, inspection, and adaptation are key to the scrum values of transparency, inspection, and adaptability.

6.6.1 How to plan and execute scrum sprints?

The product owner, scrum master, and development team work together to choose the relevant work items for a sprint. The sprint goal is discussed, as well as the product backlog items that, when completed, will achieve the sprint goal.

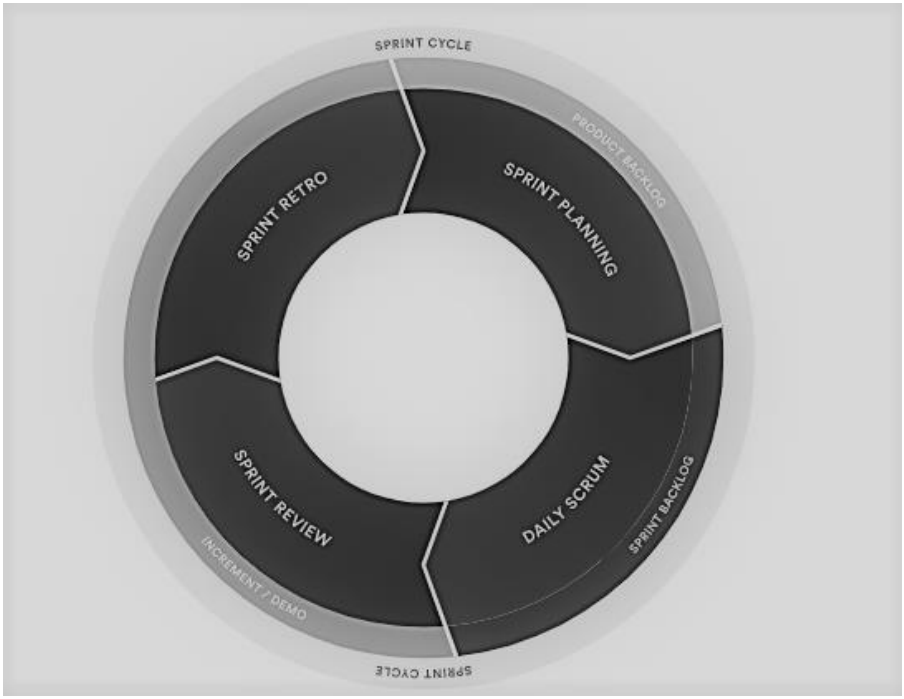

Fig 2.5 12 Phases of Scrum

The team then devises a strategy for completing the backlog items and declaring them "Done" before the sprint ends. The sprint backlog is the list of work items chosen and the plan for completing them. The team is ready to start working on the sprint backlog by the end of sprint planning, moving things from the backlog to "In-progress" and "Done."

During a sprint, the team checks in on how the work is moving during the daily scrum, or standup. The purpose of this discussion is to identify any roadblocks or problems that may interfere with the team's ability to fulfill the sprint target.

During the sprint review, the team shows what they've accomplished during the sprint. This is it.

6.6.2 Sprint Planning

Sprint planning is a scrum event that starts the sprint. The goal of sprint planning is to figure out what can be accomplished in a given sprint and how it will be done. The scrum team as a whole collaborates on sprint planning. In scrum, a sprint is a predetermined amount of time during which all of the work is completed. However, before you can take action, you must first prepare for the race. You'll need to select on the length of the time limit, the sprint target, and where you'll begin.The sprint planning session sets the agenda and emphasis for the sprint. It also provides a motivating environment for the team if done effectively.

6.6.3 The Product Backlog

The roadmap and associated requirements are used to create a product backlog, which is a prioritized list of work for the development team. The most critical things are displayed at the top of the product backlog, allowing the team to prioritize what should be delivered first. The development team is not moving through the backlog at the same rate as the product owner, and the product owner is not assigning work to the development team. Instead, the development team draws work from the product backlog when capacity allows, either on a continuous basis (Kanban) or iteratively (scrum).

6.6.4 Sprint Reviews

Sprint reviews aren't the same thing as retrospectives. The goal of a sprint review is to show off the efforts of the entire team, including designers, developers, and the product owner. For informal demos, team members gather around a desk and describe the work they've done for that iteration. It's a great opportunity to ask questions, try out new features, and provide feedback. Building an agile team necessitates sharing in the success.

6.7 Unit Summary

■ Agile software development is an umbrella term for a set of frameworks and practices based on the values and principles expressed in the Manifesto for Agile Software Development and the 12 Principles behind it.

■ Scrum is a framework that helps teams work together. Much like a rugby team (where it gets its name) training for the big game, scrum encourages teams to learn through experiences, self-organise while working on a problem, and reflect on their wins and losses to continuously improve

6.8 Unit Activities

1. Explain why it is important to understand the agile manifesto.
2. What is scrum?
3. What is a sprint?
4. Explain the important of a sprint review.

UNIT 7

SOFTWARE TESTING

UNIT STRUCTURE

7.0 Introduction
7.1 Unit Objectives
7.2 Validation and Verification
7.3 Black Box Testing
7.4 White Box Testing
7.5 Integration Testing
7.6 Unit Summary
7.7 Unit Activities

7.0 Introduction

The aim of the testing process is to identify all defects existing in a software product. However for most practical systems, even after satisfactorily carrying out the testing phase, it is not possible to guarantee that the software is error free. This is because of the fact that the input data domain of most software products is very large. It is not practical to test the software exhaustively with respect to each value that the input data may assume.

7.1 Unit Objectives

By the end of this unit, you should be able to do the following:

■ understand the stages of testing from testing, during development to acceptance testing by system customers;

■ have been introduced to techniques that help you choose test cases that are geared to discovering program defects;

■ understand test-first development, where you design tests before writing code and run these tests automatically;

■ know the important differences between component, system, and release testing and be aware of user testing processes and techniques.

7.3 Validation vs Verification

Testing is done to ensure that a software accomplishes what it's supposed to do and to find bugs before it's placed into production. When testing software, you run a program with fictitious data. You look for mistakes, anomalies, or information regarding the program's non-functional attributes in the test results. The purpose of the testing procedure is twofold:

- Demonstrate to the developer and customer that the software fits their specifications. For custom software, this means that each need in the requirements document should have at least one test. It indicates that there should be tests for all of the system features, as well as combinations of these characteristics, that will be included in the product release for generic software products.

- To find circumstances when the software's behavior is erroneous, unpleasant, or does not meet its specifications. These are the results of software flaws. Defect testing looks for issues such system crashes, inappropriate interactions with other systems, inaccurate computations, and data corruption.

Barry Boehm, a pioneer of software engineering, succinctly expressed the difference between them (Boehm, 1979):

- 'Validation: Are we building the right product?'
- 'Verification: Are we building the product right?'

Verification and validation techniques ensure that the software being produced satisfies its specifications and provides the capabilities that the people who are paying for it anticipate. These processes begin as soon as requirements are made accessible and continue throughout the development phase.

The ultimate goal of verification and validation processes is to establish trust in the software system's suitability for its intended use. This implies that the system must be adequate for the task at hand. The required level of confidence is determined by the

system's purpose, the system users' expectations, and the existing marketing environment for the system:

- **Software purpose:** The more vital the software is, the more important it is to be dependable. The level of confidence necessary for software used to control a safety-critical system, for example, is substantially higher than that required for a prototype produced to illustrate new product concepts.

- **User expectations:** Many users have low expectations of software quality as a result of their encounters with unstable, unreliable software. When their program fails, they are unsurprised. Users may tolerate failures when a new system is implemented because the advantages of use outweigh the costs of failure recovery.

- **Marketing environment:** When a system is being marketed, the sellers must consider rival products, the price that buyers are willing to pay for a system, and the delivery schedule that is required. In a competitive market, a software firm may decide to release a program before it has been thoroughly tested and debugged in order to be the first to market.

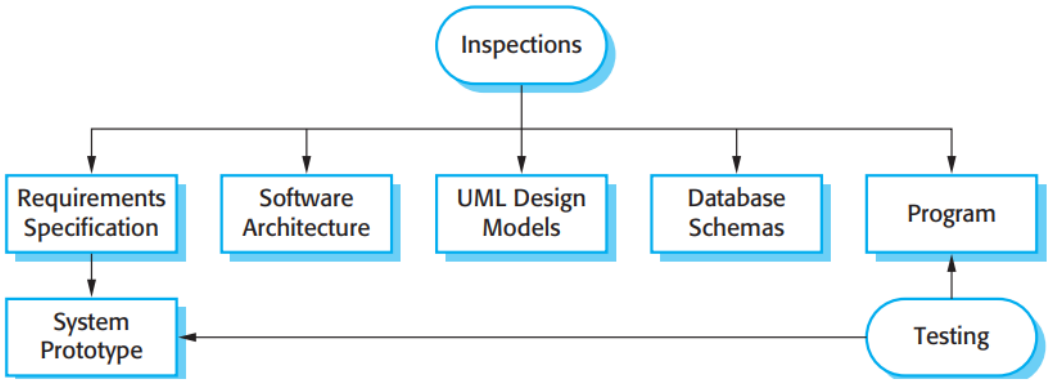

Fig 2.6 12 Inspection and testing

7.4 Black Box Testing

7.4.1 Testing in the large vs. testing in the small

Software products are normally tested first at the individual component (or unit) level. This is referred to as testing in the small. After testing all the components individually, the components are slowly integrated and tested at each level of integration (integration testing). Finally, the fully integrated system is tested (called system testing). Integration and system testing are known as testing in the large.

7.4.2 Unit Testing

After a module has been coded and properly reviewed, unit testing is performed. Unit testing (also known as module testing) is the process of testing individual components (or modules) of a system in isolation. To test a single module, you'll need a complete environment that includes everything you'll need to run the module. That is, in addition to the module under test, the following actions are required in order to test the module:

- The procedures belonging to other modules that the module under test calls.
- Nonlocal data structures that the module accesses.
- A procedure to call the functions of the module under test with appropriate parameters.

Modules are required to offer the necessary environment (which either calls or is called by the module under test), while stubs and drivers are designed to provide a module's complete environment. Figure 19.1 depicts the function of stub and driver modules. A fake procedure with the same I/O parameters as the given procedure but a greatly simplified behavior is known as a stub procedure. A stub process, for example, could use a simple table lookup mechanism to create the necessary behavior. A driver module would contain the nonlocal data structures accessed by the module under test, as well as the code to call the module's various functions with proper parameters.

7.4.3 Black Box Testing

Test cases are created using simply an evaluation of the input/output values in black-box testing, and no knowledge of design or code is necessary. The two basic techniques to creating black box test scenarios are as follows.

- Equivalence class portioning: In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behaviour of the program is similar for every input data belonging to the same equivalence class. The main idea behind defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class. Equivalence classes for a software can be designed by examining the input data and output data.
- Boundary Value Analysis: A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs. The reason behind such errors might purely be due to psychological factors. Programmers often fail to see the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for.

7.5 White Box Testing

If all sorts of mistakes found by the first testing strategy are also detected by the second testing strategy, and the second testing strategy also discovers some more types of errors, one white-box testing approach is considered to be stronger than the other. When two testing procedures uncover faults that are different, at least in some ways, they are referred to as complimentary.

7.5.1 White Box Testing Techniques

Statement Coverage**:** This technique requires every possible statement in the code to be tested at least once during the testing process of software engineering.

Branch Coverage: This technique checks every possible path (if-else and other conditional loops) of a software application.

Aside from the aforementioned coverage kinds, there are a slew of others, including Condition Coverage, Multiple Condition Coverage, Path Coverage, and Function Coverage. Each method has its own set of advantages and aims to test (cover) all aspects of software code. You can typically achieve 80-90 percent code coverage using Statement and Branch coverage, which is sufficient.

7.5.2 Advantages vs Disadvantages

- **Advantages**

  - Code optimisation by finding hidden errors.
  - White box tests cases can be easily automated.
  - Testing is more thorough as all code paths are usually covered.
  - Testing can start early in SDLC even if GUI is not available.

- **Disadvantages**

  - White box testing can be quite complex and expensive.
  - Developers who usually execute white box test cases detest it. The white box testing by developers is not detailed can lead to production errors.
  - White box testing requires professional resources, with a detailed understanding of programming and implementation.
  - White-box testing is time-consuming, bigger programming applications take the time to test fully.

7.6 Integration Testing

The fundamental goal of integration testing is to ensure that there are no errors in parameter passing when one module calls another. Different components of a system are integrated in a planned manner using an integration plan during integration testing. The integration plan lays out how modules will be joined and in what order to complete the system. The partially integrated system is tested after each integration phase. An important factor that guides the integration plan is the module dependency graph. The structure chart (or module dependency graph) denotes the order in which different modules call each other. By examining the structure chart the integration plan can be developed.

There are four different techniques to integration testing. To create the integration test plan, you can use any (or a combination) of the following methods. The following are some of the approaches:

- Big bang approach
- Bottom- up approach
- Top-down approach
- Mixed-approach

7.6.1 Big-Bang Integration Testing

It is the simplest integration testing approach, where all the modules making up a system are integrated in a single step. In simple words, all the modules of the system are simply put together and tested. However, this technique is practicable only for very small systems. The biggest issue with this strategy is that once an error is discovered during integration testing, it is extremely difficult to pinpoint where the error originated because it could be in any of the modules being merged. As a result, debugging issues discovered during big bang integration testing is quite costly.

7.6.2 Bottom-Up Integration Testing

Each subsystem is tested separately before the entire system is tested in bottom-up testing. A subsystem may be made up of several modules that communicate with one another via well-defined interfaces. The fundamental goal of each subsystem's testing is to check the interfaces between the numerous modules that make up the subsystem. Both the control and data interfaces are put through their paces. The test cases must be carefully chosen in order to put the interfaces through their paces in every way conceivable. Large software systems typically necessitate many levels of subsystem testing, with lower-level subsystems being joined to generate higher-level subsystems. Bottom-up integration testing has the advantage of allowing multiple disconnected subsystems to be tested at the same time. No stubs are necessary in pure bottom-up testing; only test-drivers are required. Bottom-up testing has one drawback.

7.6.2 Top-Down Integration Testing

Top-down integration testing starts with the main routine and one or two subordinate routines in the system. After the top-level 'skeleton' has been tested, the immediately subroutines of the 'skeleton' are combined with it and tested. Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test. A pure top-down integration does not require any driver routines. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, many times it may become difficult to exercise the top-level routines in the desired manner since the lower-level routines perform several low-level functions such as I/O.

7.6.3 Mixed Integration Testing

A mixed (also known as sandwiched) integration test combines top-down and bottom-up testing techniques. Testing can begin only once the top-level modules have been coded and unit tested in a top-down method. Similarly, bottom-up testing can only begin once the lowest-level modules are complete. This deficiency of the top-down and bottom-up approaches is addressed by the mixed approach. When using a mixed testing method, testing can begin as soon as modules are ready. As a result, this is one of the most widely utilized ways of integration testing.

7.7 Unit Summary

- Testing is done to ensure that a software accomplishes what it's supposed to do and to find bugs before it's placed into production. When testing software, you run a program with fictitious data.
- Verification and validation techniques ensure that the software being produced fits its specifications and provides the capabilities that the people paying for it anticipate.

7.8 Unit Activities

1. Explain what do you understand by testing
2. Distinguish between verification and validation.
3. What is Black box testing?
4. What is White box testing?
5. What do you understand by integration testing?

UNIT 8

DIGITAL TRANSFORMATION FOR SOFTWARE ENGINEERING

UNIT STRUCTURE

8.0 Introduction

8.1 Unit Objectives

8.2 Importance of source code management tools

8.3 Benefits of source code management

8.4 Source code management best practices

8.5 Unit Summary

8.6 Unit Activities

8.0 Introduction

Digital transformation is the process of integrating digital technology into all aspects of a company, radically altering how it operates and provides value to customers. It's also a cultural shift that necessitates organizations challenging the existing quo, experimenting, and becoming comfortable with failure.

8.1 Unit Objectives

By the end of this unit, you should be able to do the following:

- understand the stages of testing during development to acceptance testing
- have been introduced to techniques of test cases
- understand test-first development
- Know the essential differences between component, system, and release testing and be aware of user testing processes and techniques.

8.2 Digital Transformation

From tiny enterprises to large corporations, digital transformation is a must. Because every company's digital transformation will be distinct, it's difficult to come up with a universal description. However, we define digital transformation in broad terms as the integration of digital technology into all business domains, resulting in substantial changes in how firms function and give value to consumers. Beyond that, it's a culture shift that necessitates organizations challenging the status quo on a regular basis, experimenting frequently, and becoming comfortable with failure. This may include abandoning long-standing business processes on which businesses were founded in favor of relatively fresh practices that are still being defined.

8.3 Why Digital Transformation?

A company may undertake digital transformation for a variety of reasons. The most likely reason, however, is that they must: It's a matter of life and death. The ability of an organization to quickly react to supply chain interruptions, time to market challenges, and rapidly changing customer expectations has become important in the aftermath of the epidemic.

Furthermore, this fact is reflected in expenditure priorities. Despite the hurdles posed by the COVID-19 pandemic, investment on digital transformation (DX) of corporate methods, products, and organizations continues "at a solid pace," according to the May 2020 International Data Corporation (IDC) Worldwide Digital Transformation Spending Guide. According to IDC, global investment on DX technologies and services will increase 10.4% to $1.3 trillion in 2020. This compares to a 17.9% increase in the previous year.

It's too early to tell which long-term improvements in consumer behavior will remain. "Digital has been increasing across just about all categories on the consumer side," says Rodney Zemmel, global leader, McKinsey Digital at McKinsey & Company. One thing to keep an eye on is how forced change — three out of four Americans, for example, tried a new buying behavior — will revert as soon as possible, given today's emphasis on staying put.

8.4 Digital Transformation Framework

Depending on the organization's individual issues and demands, digital transformation will differ greatly. Existing case studies and published frameworks share a few constants and common elements that all business and technology leaders should evaluate when they begin on digital transformation.

For instance, these digital transformation elements :

- Customer experience
- Operational agility
- Culture and leadership
- Workforce enablement
- Digital technology integration

8.5 Unit Summary

■ Digital transformation is imperative for all businesses, from the small to the enterprise. That message comes through loud and clear from seemingly every keynote, panel discussion, article, or study related to how businesses can remain competitive and relevant as the world becomes increasingly digital.

■ A business may take on digital transformation for several reasons. But by far, the most likely reason is that they have to: It's a survival issue. In the wake of the pandemic, an organisation's ability to adapt quickly to supply chain disruptions, time to market pressures, and rapidly changing customer expectations has become critical.

■ Although digital transformation will vary widely based on organisation's specific challenges and demands, there are a few constants and common themes among existing case studies and published frameworks that all business and technology leaders should consider as they embark on digital transformation.

8.6 Unit Activities

- Explain what do you understand by Digital Transformation
- Why is it important to go for Digital Transformation during the Covid 19?