UNIT 1

# INTRODUCTION TO SOFTWARE ENGINEERING

**UNIT STRUCTURE**



## 1.0 Introduction

Software Engineering is about methods, tools and techniques used for developing software. This particular 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 a variety of 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 essential;
- understand that the development of different types of a software system may require additional software engineering techniques;
- understand legal and professional issues that are important for software engineers;

## 1.2 Professional Software Development

Software engineering is intended to support professional software development rather than individual programming. It includes techniques that support program specification, design, and evolution, none of which usually are relevant for personal software development. To help you get a broad view of software engineering, I have frequently asked questions about the subject in Figure 1.1. Many people think that software is simply another word for computer programs. However, when we are talking about software engineering, the software is not just the programs themselves but also all associated documentation, libraries, support websites, and configuration data needed to make these programs useful. A professionally developed software system is often more than a single program.

A system may consist of several separate programs and configuration files that are used to set up these programs. It may include system documentation, which describes the system's structure, user documentation, which explains how to use the system, and websites for users to download recent 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 on the open market to customers who can buy them. Examples of this type of 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, such as library information systems, accounting systems, or systems for maintaining dental records.

**2. Customised (or bespoke) software:** These are commissioned by and developed for a particular customer. A software contractor designs and implements the software especially for that customer. Examples of this type of software include control systems

for electronic devices, systems written to support a particular business process, and air traffic control systems.

### 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 it has gone into use. In this definition, Software engineering is essential for two reasons:

- ✓ More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.

- ✓ In the long run, it is usually cheaper to use software engineering methods and techniques for professional software systems rather than write programs as a personal programming project. Failure to use software engineering methods leads to higher costs for testing, quality assurance, and long-term maintenance.

The systematic approach that is used in software engineering is sometimes called a software process. A software process is a sequence of activities that leads to the production of a software product. Four fundamental activities are common to all software processes.

- ✓ Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation.

- ✓ Software development, where the software is designed and programmed.

- ✓ Software validation, where the software is checked to ensure that it is what the customer requires. 4. Software evolution, where the software is modified to reflect changing customer and market requirements.

### 1.2.2 Internet Software Engineering

The development of the Internet and the World Wide Web has profoundly affected all of our lives. Initially, 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. Around 2000, the web started to evolve, and more and more functionality was added to browsers. This meant that web-based systems could be developed where, instead of a special-purpose user interface, these systems could be accessed using a web browser. This led to the development of 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 had a significant effect on software engineering for web-based systems. For example:

- ✓ Software reuse has become the dominant approach for constructing web-based systems. When building these systems, you think about how you can assemble them from pre-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.

- ✓ Software may be implemented using service-oriented software engineering, where the software components are stand-alone web services.

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

The fundamental ideas of software engineering, discussed in the previous section, apply to web-based software, as they do to other types of software. Web-based systems are getting larger and larger, so software engineering techniques that deal with scale and complexity are relevant for these systems.

## 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 areas where standards of acceptable behaviour are not bound by laws but by the more tenuous notion of professional responsibility. Some of these are:

- ✓ Confidentiality: You should generally respect the confidentiality of your employers or clients regardless of whether or not a formal confidentiality agreement has been signed.

- ✓ Competence: You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.

- ✓ Intellectual property rights: You should be aware of local laws governing the use of intellectual property, such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.

- ✓ Computer misuse: You should not use your technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's

## 1.4 Unit Summary

■ Software engineering is an engineering discipline that is concerned with all aspects of software production.

■ Software is not just a program or program but also includes all electronic documentation that system users need, quality assurance staff, and developers. Essential software product attributes are maintainability, dependability and security, efficiency, and acceptability.

■ There are many different types of systems, and each requires appropriate software engineering tools and techniques for their development. Few, if any, specific design and implementation techniques apply to all kinds of systems.

■ The fundamental ideas of software engineering apply to all types of the software system. These fundamentals include managed software processes, software dependability and security, requirements engineering, and software reuse.

■ Software engineers have responsibilities to the engineering profession and society. They should not simply be concerned with technical issues but should be aware of their work's ethical problems.

## 1.5 Unit Activities

1. Explain why professional software that is developed for a customer is not simply the programs that have been developed and delivered.
2. What is the most important difference between generic software product development and custom software development? What might this mean in practice for users of generic software products?
3. Briefly discuss why it is usually cheaper in the long run to use software engineering methods and techniques for software systems.
4. Software engineering is not only concerned with issues like system heterogeneity, business and social change, trust, and security, but also with ethical issues affecting the domain. Give some examples of ethical issues that have an impact on the software engineering domain.
5. Based on your own knowledge of some of the application types discussed , explain, with examples, why different application types require specialized software engineering techniques to support their design and development.

UNIT 2

# SOFTWARE PROCESSES

**UNIT STRUCTURE**



## 2.0 Introduction

A software process is a set of related activities that leads to the production of a software product. These activities may involve the development of software from scratch in a standard programming language. However, business applications are not necessarily developed in this way. New business software is now often developed by extending and modifying existing systems or by configuring and integrating off-the-shelf software or system components.

## 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;

■ have been introduced to three generic software process models and when they might be used;

■ know about the fundamental process activities of software requirements engineering, software development, testing, and evolution;

■ understand how the Rational Unified Process integrates good software engineering practice to create adaptable software processes.

■ understand the rationale for agile software development methods, the agile manifesto, and the differences between agile and plandriven development;

## 2.2 Software Process Models

Each process model represents a process from a particular perspective, and thus provides only partial information about that process. For example, a process activity model shows the activities and their sequence but may not show the roles of the people involved in these activities.

The process model covered here are:

1. **The waterfall model**: This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, testing, and so on.

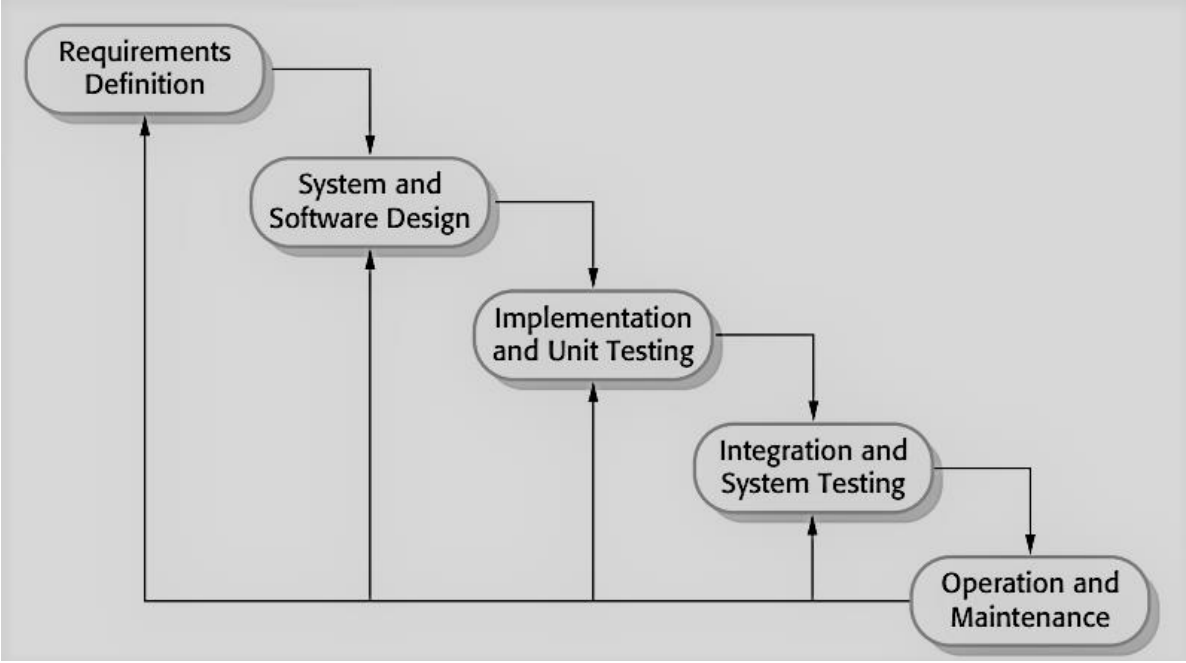


*Fig 1.2 The waterfall model*

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

- ✓ **Requirements analysis and definition:** The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as 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 realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

- ✓ **Integration and system testing:** The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

- ✓ **Operation and maintenance:** Normally (although not necessarily), this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

The waterfall model is consistent with other engineering process models and documentation is produced at each phase. This makes the process visible so managers can monitor progress against the development plan. Its major problem is the inflexible partitioning of the project into distinct stages. Commitments must be made at an early stage in the process, which makes it difficult to respond to changing customer requirements. In principle, the waterfall model should only be used when the requirements are well understood and unlikely to change radically during system development. However, the waterfall model reflects the type of process used in other engineering projects. As is easier to use a common management model for the whole project, software processes based on the waterfall model are still commonly used.

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.

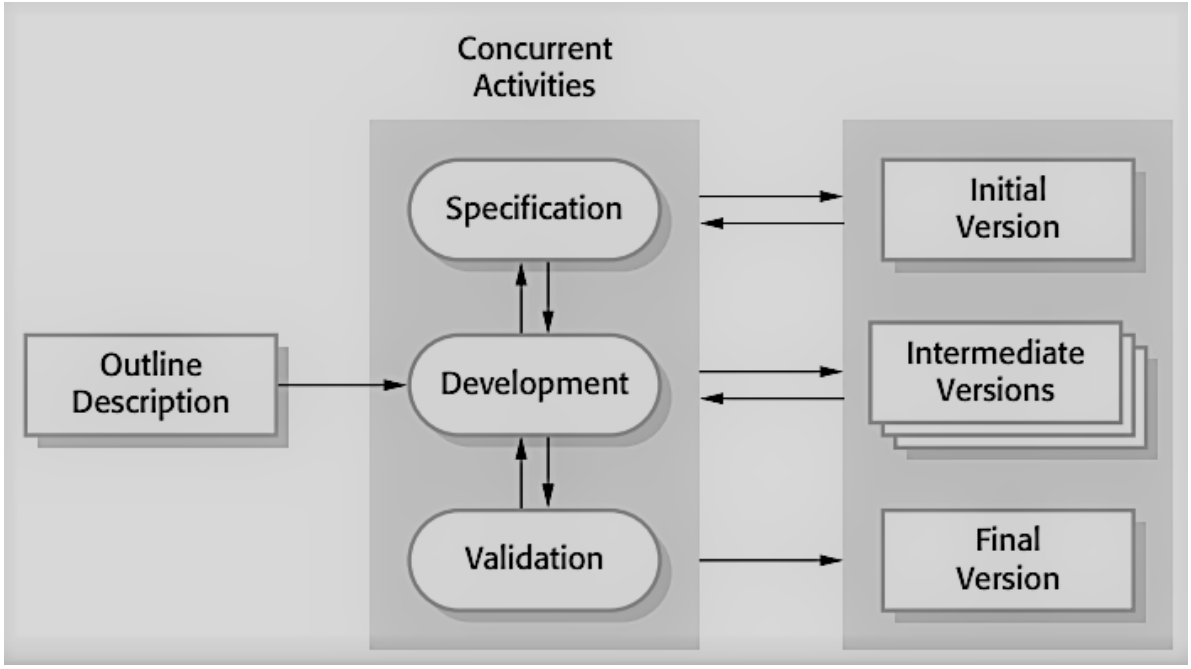


*Fig 1.3 Incremental development*

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 easier to get customer feedback on the development work that has been done. Customers can comment on demonstrations of the software and see how much has been implemented. Customers find it difficult to judge progress from software design documents.

- ✓ More rapid delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

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

- ✓ The process is not visible. Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.

- ✓ System structure tends to degrade as new increments are added. Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

3. **Reuse-oriented software engineering:** In the majority of software projects, there is some software reuse. This often happens informally when people working on the project know of designs or code that are similar to what is required. They look for these, modify them as needed, and incorporate them into their system.

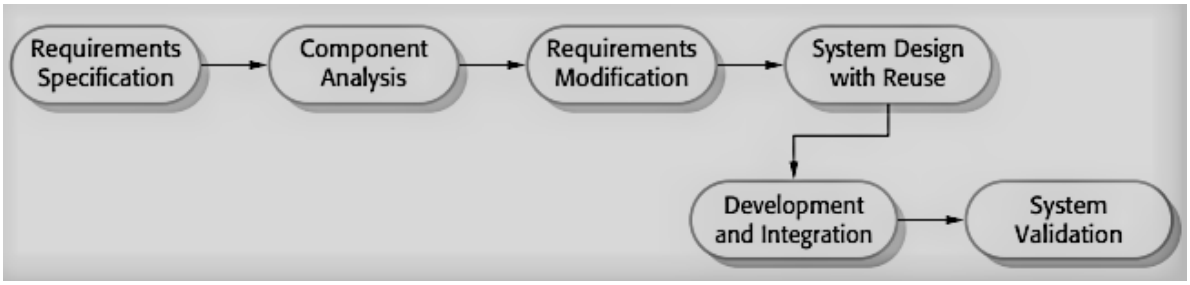


*Fig 1.4 Reuse-oriented software engineering*

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:

- ✓ Component analysis Given the requirements specification, a search is made for components to implement that specification. Usually, there is no exact match and the components that may be used only provide some of the functionality required.

- ✓ Requirements modification During this stage, the requirements are analyzed using information about the components that have been discovered. They are then modified to reflect the available components. Where modifications are impossible, the component analysis activity may be re-entered to search for alternative solutions.

- ✓ 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 organize the framework to cater for this. Some new software may have to be designed if reusable components are not available.

- ✓ Development and integration Software that cannot be externally procured is developed, and the components and COTS systems are integrated to create the new system. System integration, in this model, may be part of the development process rather than a separate activity.

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 specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development. Requirements engineering is a particularly critical stage of the software process as errors at this stage inevitably lead to later problems in the system design and implementation.

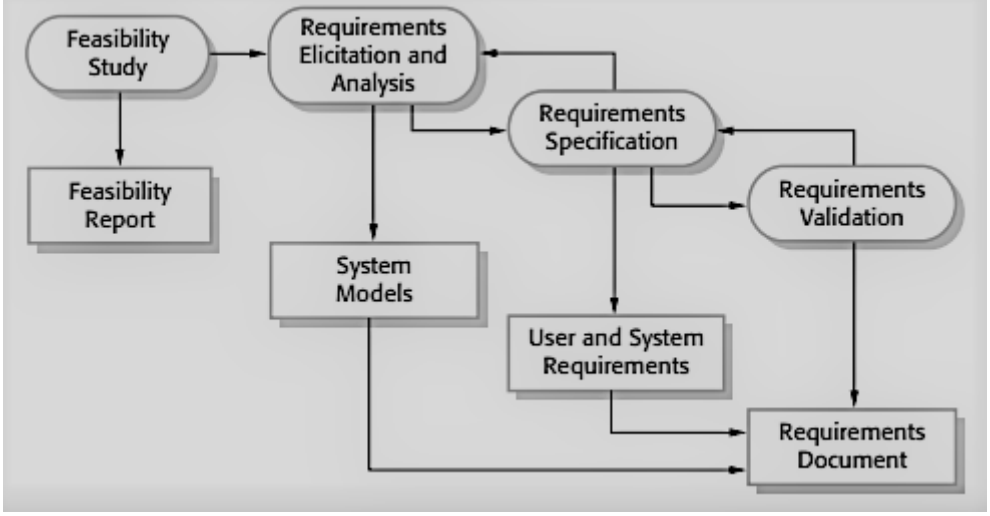


*Fig 1.4 The requirement engineering process*

There are four main activities in the requirements engineering process:

- ✓ Feasibility study An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and if it can be developed within existing budgetary constraints. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether or not to go ahead with a more detailed analysis.

- ✓ Requirements elicitation and analysis This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on. This may involve the development of one or more system models and prototypes. These help you understand the system to be specified.

- ✓ Requirements specification Requirements specification is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. Two types of requirements may be included in this document. User requirements are abstract statements of the system requirements for the customer and end-user of the system; system requirements are a more detailed description of the functionality to be provided.

- ✓ Requirements validation This activity checks the requirements for realism, consistency, and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.

## 2.4 Unit Summary

■ Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.

■ General process models describe the organization of software processes. Examples of these general models include the waterfall model, incremental development, and reuse-oriented development.

■ Requirements engineering is the process of developing a software specification. Specifications are intended to communicate the system needs of the customer to the system developers.

■ Design and implementation processes are concerned with transforming a requirements specification into an executable software system. Systematic design methods may be used as part of this transformation.

## 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

For most large systems, it is still the case that there is a clearly identifiable requirements engineering phase before the implementation of the system begins. The outcome is a requirements document, which may be part of the system development contract. Of course, there are usually subsequent changes to the requirements and user requirements may be expanded 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 organized 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 statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.

2. **Non-functional requirements**: These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services.

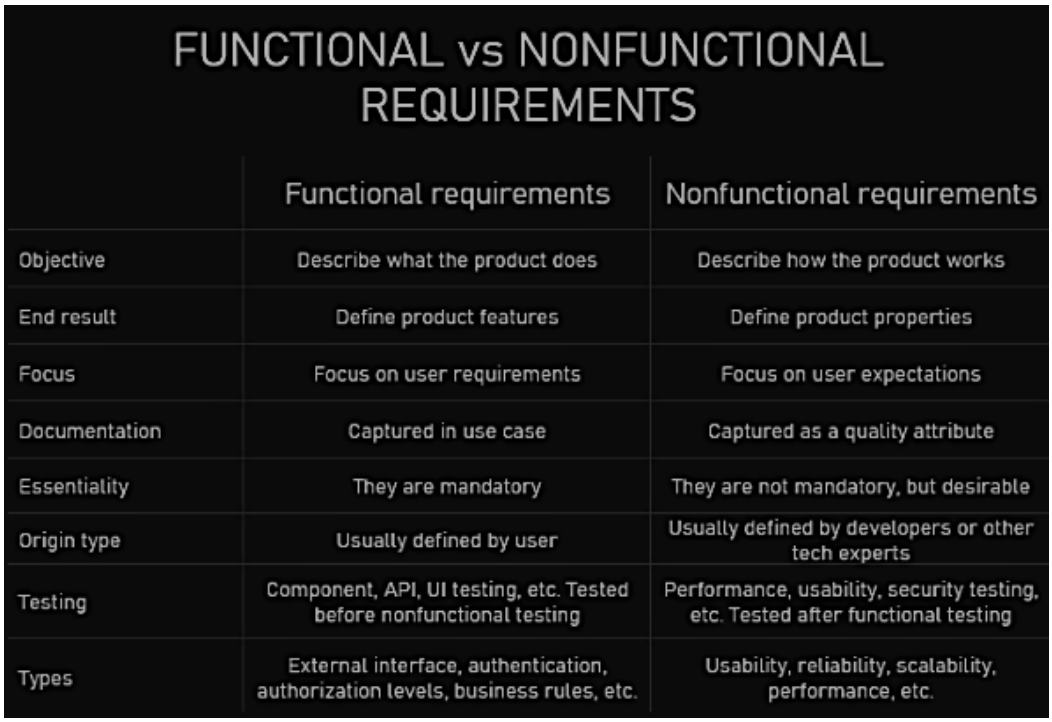


*Fig 1.5 Functional vs Non Functional Requirements*

## 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
- Authorization 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

Both functional and nonfunctional requirements can be formalized in the *software requirements specification (SRS)* document. To learn more about software documentation in general, read our article on that topic. The SRS contains descriptions of functions and capabilities that the product must provide. The document also defines constraints and assumptions. The SRS can be a single document communicating functional requirements or it may accompany other software documentation like 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:

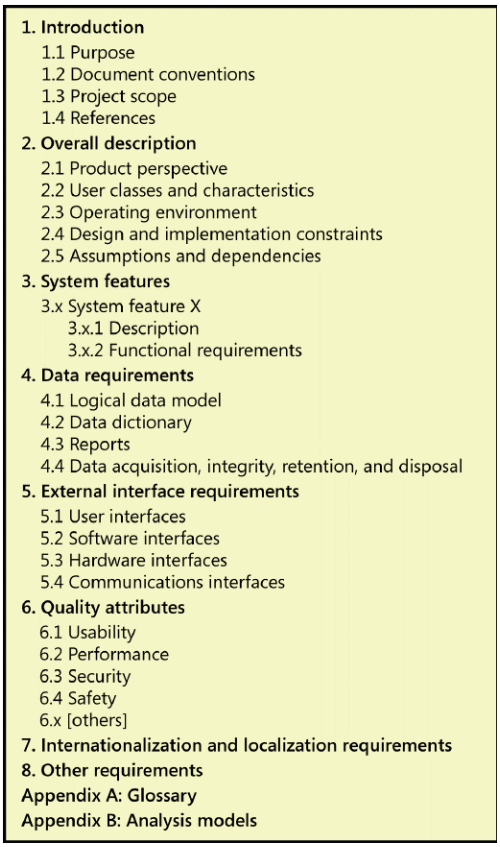


*Fig 1.6 Template for SRS document*

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: …<br>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.

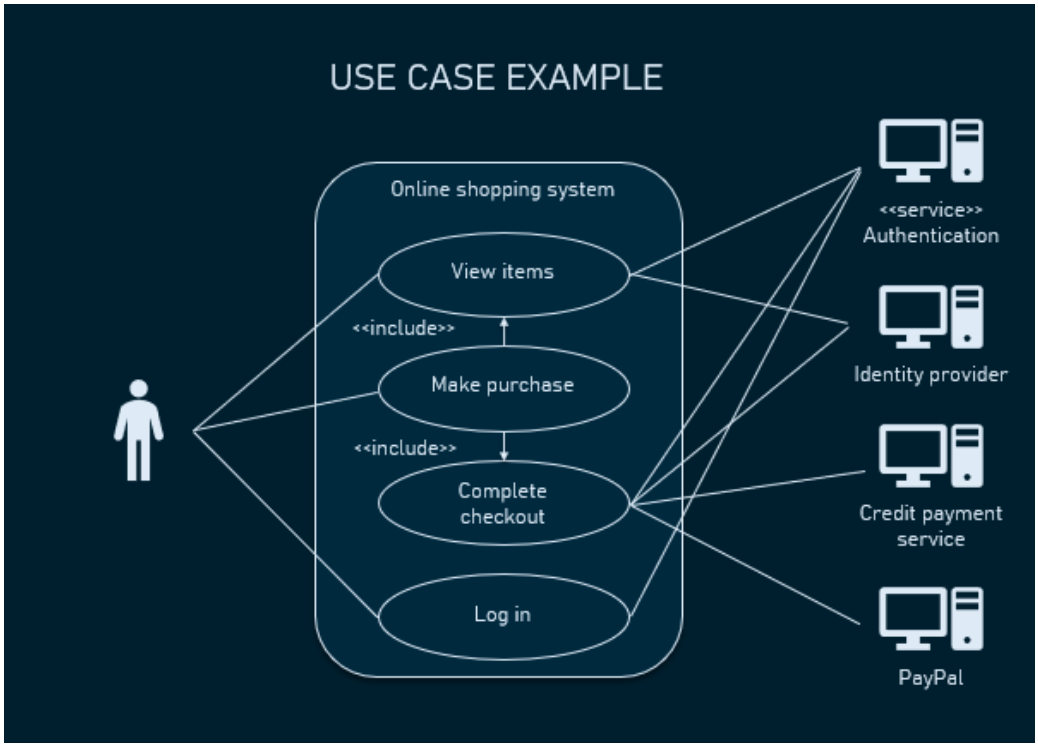


*Fig 1.7 Use Case Example*

3. Work Breakdown Structure (WBS)

A functional decomposition or WBS is a visual document that illustrates how complex processes break down into their simpler components. WBS is an effective approach to allow for an independent analysis of each part. WBS also helps capture the full picture of the project.

The decomposition process may look like this:

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

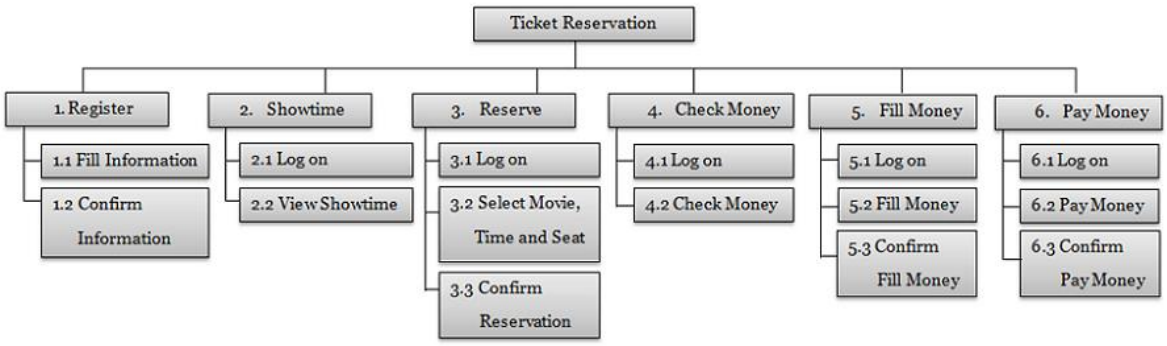


*Fig 1.8 WBS Example*

## 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 localization 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 requirements ensure that the software is protected from unauthorized access to the system and its stored data. It considers different levels of authorization and authentication across different users roles. For instance, data privacy is a security characteristic that describes who can create, see, copy, change, or delete information. Security also includes protection against viruses and malware attacks.

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

### 3.5.3 Reliability

Reliability defines how likely it is for the software to work without failure for a given period of time. Reliability decreases because of bugs in the code, hardware failures, or problems with other system components. To measure software reliability, you can count the percentage of operations that are completed correctly or track the average period of time the system runs before failing.

*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 the responsiveness of the system to various user interactions with it. Poor performance leads to negative user experience. It also jeopardizes system safety when it's overloaded.

*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

Availability is gauged by the period of time that the system's functionality and services are available for use with all operations. So, scheduled maintenance periods directly influence this parameter. And it's important to define how the impact of maintenance can be minimized. When writing the availability requirements, the team has to define the most critical components of the system that must be available at all times. You should also prepare user notifications in case the system or one of its parts becomes unavailable.

*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

Scalability requirements describe how the system must grow without negative influence on its performance. This means serving more users, processing more data, and doing more transactions. Scalability has both hardware and software implications. For instance, you can increase scalability by adding memory, servers, or disk space. On the other hand, you can compress data, use optimizing algorithms, etc.

*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, organizational 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 organized 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 organized.

UNIT 4

# SYSTEM MODELING

**UNIT STRUCTURE**



## 4.0 Introduction

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. System modeling has generally come to mean representing the system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (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

At an early stage in the specification of a system, you should decide on the system boundaries. This involves working with system stakeholders to decide what functionality should be included in the system and what is provided by the system's environment. You may decide that automated support for some business processes should be implemented but others should be manual processes or supported by different systems. You should look at possible overlaps in functionality with existing systems and decide where new functionality should be implemented. These decisions should be made early in the process to limit the system costs and the time needed for understanding the system requirements and design**.**

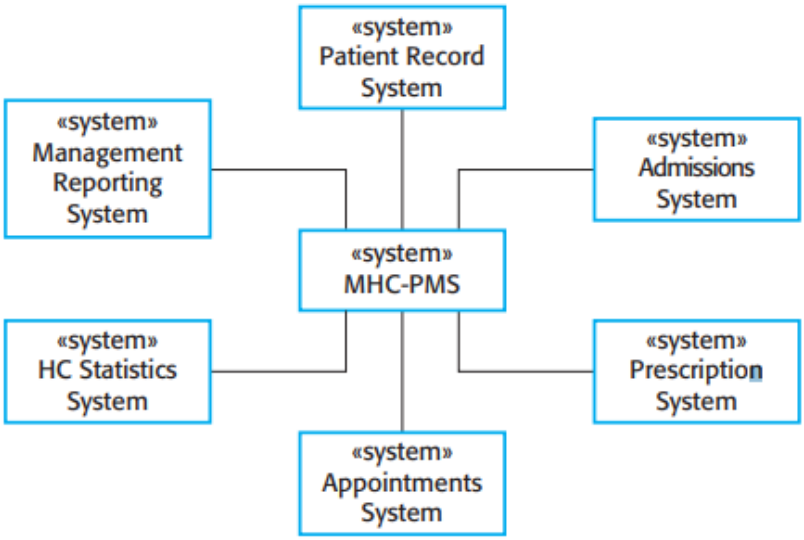


*Fig 1.9 The context diagram for PMS*

Context models normally show that the environment includes several other automated systems. However, they do not show the types of relationships between the systems in the environment and the system that is being specified. External systems might produce data for or consume data from the system. They might share data with the system, or they might be connected directly, through a network or not connected at all. They might be physically co-located or located in separate buildings. All of these relations may affect the requirements and design of the system being defined and must be taken into account.

## 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 give a fairly simple overview of an interaction so you have to provide more detail to understand what is involved. This detail can either be a simple textual description, a structured description in a table, or a sequence diagram as discussed below. You chose the most appropriate format depending on the use case and the level of detail that you think is required in the model. I find a standard tabular format to be the most useful.

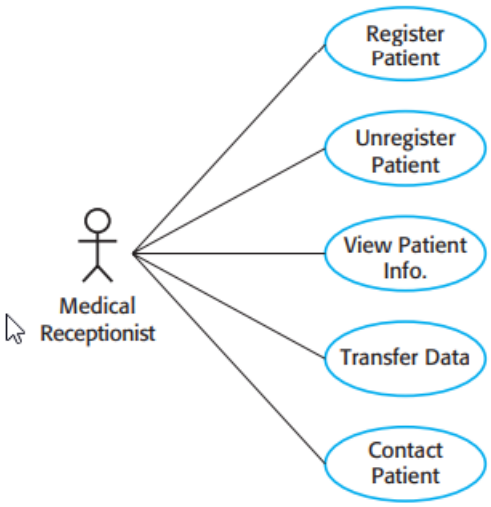


*Fig 2.1 Use cases involving the role 'medical receptionist'*

## 4.4 Sequence Diagrams

Sequence diagrams in the UML are primarily used to model the interactions between the actors and the objects in a system and the interactions between the objects themselves. The UML has a rich syntax for sequence diagrams, which allows many different kinds of interaction to be modeled.

The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these. Interactions between objects are indicated by annotated arrows. The rectangle on the dotted lines indicates the lifeline of the object concerned (i.e., the time that object instance is involved in the computation). You read the sequence of interactions from top to bottom. The annotations on the arrows indicate the calls to the objects, their parameters, and the return values. In this example, I also show the notation used to denote alternatives. A box named alt is used with the conditions indicated in square brackets.

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 authorization system that the user is authorized for this action.

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

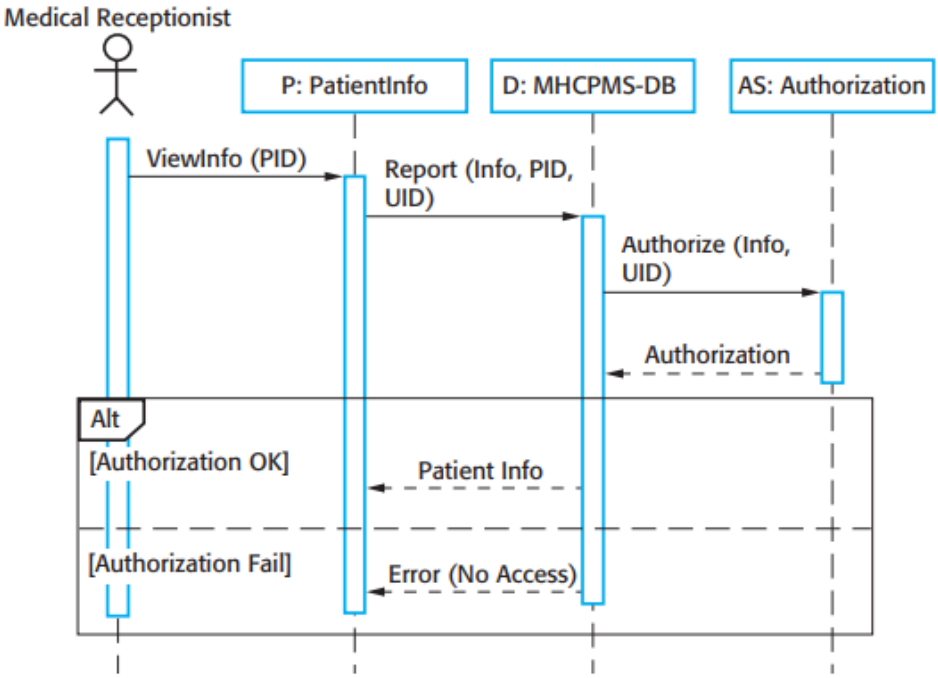


*Fig 2.2 Sequence diagram for View patient information*

You can read this diagram as follows:

1. The receptionist logs on to the PRS.
2. There are two options available. These allow the direct transfer of updated patient information to the PRS and the transfer of summary health data from the MHC-PMS to the PRS.
3. In each case, the receptionist's permissions are checked using the authorization system.
4. Personal information may be transferred directly from the user interface object to the PRS. Alternatively, a summary record may be created from the database and that record is then transferred.
5. On completion of the transfer, the PRS issues a status message 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.

Class diagrams in the UML can be expressed at different levels of detail. When you are developing a model, the first stage is usually to look at the world, identify the essential objects, and represent these as classes. The simplest way of writing these is to write the class name in a box. You can also simply note the existence of an association.

At this level of detail, class diagrams look like semantic data models. Semantic data models are used in database design. They show the data entities, their associated attributes, and the relations between these entities. This approach to modeling was first proposed in the mid-1970s by Chen (1976); several variants have been developed since then (Codd, 1979; Hammer and McLeod, 1981; Hull and King, 1987), all with the same basic form.

The UML does not include a specific notation for this database modeling as it assumes an object-oriented development process and models data using objects and their relationships. However, you can use the UML to represent a semantic data model. You can think of entities in a semantic data model as simplified object classes.

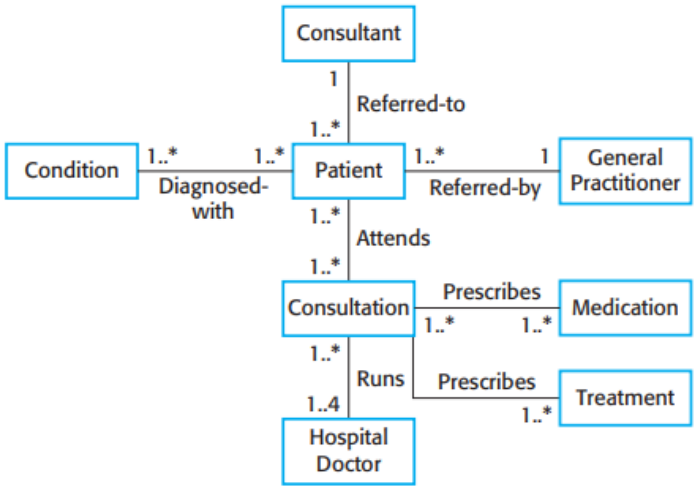


*Fig 2.3 Classes and associations in the MHC-PMS*

## 4.6 Unit Summary

■ A model is an abstract view of a system that ignores some system details. Complementary system models can be developed to show the system's context, interactions, structure, and behavior.

■ Context models show how a system that is being modeled is positioned in an environment with other systems and processes. They help define the boundaries of the system to be developed.

■ Use case diagrams and sequence diagrams are used to describe the interactions between user the system being designed and users/other systems. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.

## 4.7 Unit Activities

1. Explain why it is important to model the context of a system that is being developed. Give two examples of possible errors that could arise if software engineers do not understand the system context.
2. How might you use a model of a system that already exists? Explain why it is not always necessary for such a system model to be complete and correct. Would the same be true if you were developing a model of a new system? Who should be involved in a requirements review? Draw a process model showing how a requirements review might be organized.
3. You have been asked to develop a system that will help with planning large-scale events and 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, organizing invitations, etc.) and the system elements that may be used at each stage.

UNIT 5

# SOURCE CODE MANAGEMENT

**UNIT STRUCTURE**



## 5.0 Introduction

Source code management (SCM) is used to track modifications to a source code repository. SCM tracks a running history of changes to a code base and helps resolve conflicts when merging updates from multiple contributors. SCM is also synonymous with Version control. As software projects grow in lines of code and contributor head count, the costs of communication overhead and management complexity also grow. SCM is a critical tool to alleviate the organizational strain of growing development costs.

## 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.

SCM brought version control safeguards to prevent loss of work due to conflict overwriting. These safeguards work by tracking changes from each individual developer and identifying areas of conflict and preventing overwrites. SCM will then communicate these points of conflict back to the developers so that they can safely review and address.

This foundational conflict prevention mechanism has the side effect of providing passive communication for the development team. The team can then monitor and discuss the work in progress that the SCM is monitoring. The SCM tracks an entire history of changes to the code base. This allows developers to examine and review edits that may have introduced bugs or regressions.

## 5.3 Benefits of source code management

In addition to version control SCM provides a suite of other helpful features to make collaborative code development a more user friendly experience. Once SCM has started tracking all the changes to a project over time, a detailed historical record of the projects life is created. This historical record can then be used to 'undo' changes to the codebase. The SCM can instantly revert the codebase back to a previous point in time. This is extremely valuable for preventing regressions on updates and undoing mistakes.

The SCM archive of every change over a project's life time provides valuable record keeping for a project's release version notes. A clean and maintained SCM history log can be used interchangeably as release notes. This offers insight and transparency into the progress of a project that can be shared with end users or non-development teams.

SCM will reduce a team's communication overhead and increase release velocity. Without SCM development is slower because contributors have to take extra effort to plan a non-overlapping sequence of develop for release. With SCM developers can work independently on separate branches of feature development, eventually merging them together.

Overall SCM is a huge aid to engineering teams that will lower development costs by allowing engineering resources to execute more efficiently. SCM is a must have in the modern age of software development. Professional teams use version control and your team should too.

## 5.4 Source code management best practices

### 5.4.1 Commit often

Commits are cheap and easy to make. They should be made frequently to capture updates to a code base. Each commit is a snapshot that the codebase can be reverted to if needed. Frequent commits give many opportunities to revert or undo work. A group of commits can be combined into a single commit using a rebase to clarify the development log.

### 5.4.2 Ensure you are working from latest version

SCM enables rapid updates from multiple developers. It's easy to have a local copy of the codebase fall behind the global copy. Make sure to git pull or fetch the latest code before making updates. This will help avoid conflicts at merge time.

### 5.4.3 Make detailed notes

Each commit has a corresponding log entry. At the time of commit creation, this log entry is populated with a message. It is important to leave descriptive explanatory commit log messages. These commit log messages should explain the "why" and "what" that encompass the commits content. These log messages become the canonical history of the project's development and leave a trail for future contributors to review.

### 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. Utilizing 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 powerful SCM mechanism that allows developers to create a separate line of development. Branches should be used frequently as they are quick and inexpensive. Branches enable multiple developers to work in parallel on separate lines of development. These lines of development are generally different product features. When development is complete on a branch it is then merged into the main line of development.

### 5.4.6 Agree on Workflow

By default SCMs offer very free form methods of contribution. It is important that teams establish shared patterns of collaboration. SCM workflows establish patterns and processes for merging branches. If a team doesn't agree on a shared workflow it can lead to inefficient communication overhead when it comes time to merge branches.

## 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

Businesses now operate in a global, rapidly changing environment. They have to respond to new opportunities and markets, changing economic conditions, and the emergence of competing products and services. Software is part of almost all business operations so new software is developed quickly to take advantage of new opportunities and to respond to competitive pressure. Rapid development and delivery is therefore now often the most critical requirement 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 is an iterative approach to project management and software development that helps teams deliver value to their customers faster and with fewer headaches. Instead of betting everything on a "big bang" launch, an agile team delivers work in small, but consumable, increments. Requirements, plans, and results are evaluated continuously so teams have a natural mechanism for responding to change quickly.

## 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-organizing cross-functional teams utilizing the appropriate practices for their context.

There's a big focus in the Agile software development community on collaboration and the self-organizing team.

That doesn't mean that there aren't managers. It means that teams have the ability to figure out how they're going to approach things on their own.

It means that those teams are cross-functional. Those teams don't have to have specific roles involved so much as that when you get the team together, you make sure that you have all the right skill sets on the team.

## 6.4 What are Agile Methodologies?

If Agile is a mindset, then what does that say about the idea of Agile methodologies? To answer this question, you may find it helpful to have a clear definition of methodology.

Alistair Cockburn suggested that a methodology is the set of conventions that a team agrees to follow. That means that each team will have its own methodology, which will be different in either small or large ways from every other team's methodology.

So Agile methodologies are the conventions that a team chooses to follow in a way that follows Agile values and principles.

## 6.5 What is Scrum?

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-organize while working on a problem, and reflect on their wins and losses to continuously improve.

While the scrum I'm talking about is most frequently used by software development teams, its principles and lessons can be applied to all kinds of teamwork. This is one of the reasons scrum is so popular. Often thought of as an agile project management framework, scrum describes a set of meetings, tools, and roles that work in concert to help teams structure and manage their work.

## 6.6 What are Sprints?

A sprint is a short, time-boxed period when a scrum team works to complete a set amount of work. Sprints are at the very heart of scrum and agile methodologies, and getting sprints right will help your agile team ship better software with fewer headaches.

The many similarities between agile values and scrum processes lead to a fair association. Sprints help teams follow the agile principle of "delivering working software frequently," as well as live the agile value of "responding to change over following a plan." The scrum values of transparency, inspection, and adaptation are complementary to agile and central to the concept of sprints.

### 6.6.1 How to plan and execute scrum sprints?

Choosing the right work items for a sprint is a collaborative effort between the product owner, scrum master, and development team. The product owner discusses the objective that the sprint should achieve and the product backlog items that, upon completion, would achieve the sprint goal.

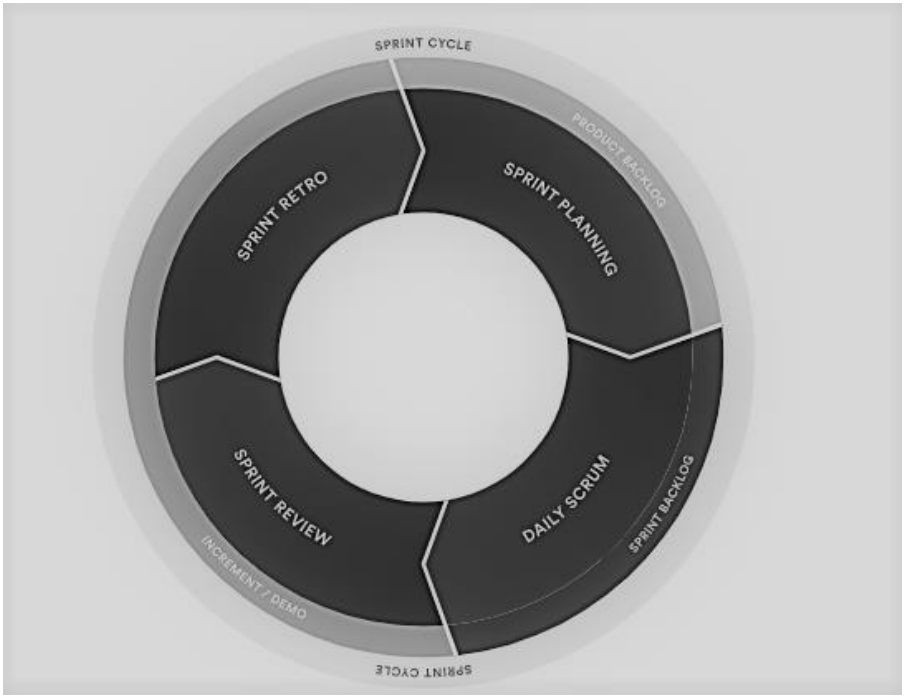


*Fig 2.5 12 Phases of Scrum*

The team then creates a plan for how they will build the backlog items and get them “Done” before the end of the sprint. The work items chosen and the plan for how to get them done is called the sprint backlog. By the end of sprint planning the team is ready to start work on the sprint backlog, taking items from the backlog, to “In-progress,” and “Done."

During a sprint, the team checks in during the daily scrum, or standup, about how the work is progressing. The goal of this meeting is to surface any blockers and challenges that would impact the team’s ability to deliver the sprint goal.

After a sprint, the team demonstrates what they’ve completed during the sprint review. This is your team’s opportunity to showcase their work to stakeholders and teammates before it hits production.

### 6.6.2 Sprint Planning

Sprint planning is an event in scrum that kicks off the sprint. The purpose of sprint planning is to define what can be delivered in the sprint and how that work will be achieved. Sprint planning is done in collaboration with the whole scrum team. In scrum, the sprint is a set period of time where all the work is done. However, before you can leap into action you have to set up the sprint. You need to decide on how long the time box is going to be, the sprint goal, and where you're going to start.

The sprint planning session kicks off the sprint by setting the agenda and focus. If done correctly, it also creates an environment where the team is motivated, challenged, and can be successful. Bad sprint plans can derail the team by setting unrealistic expectations.

### 6.6.3 The Product Backlog

A product backlog is a prioritized list of work for the development team that is derived from the roadmap and its requirements. The most important items are shown at the top of the product backlog so the team knows what to deliver first. The development team doesn't work through the backlog at the product owner's pace and the product owner isn't pushing work to the development team. Instead, the development team pulls work from the product backlog as there is capacity for it, either continually (Kanban) or by iteration (scrum).

### 6.6.4 Sprint Reviews

Sprint reviews are not retrospectives. A sprint review is about demonstrating the hard work of the entire team: designers, developers, and the product owner. Team members gather around a desk for informal demos and describe the work they've done for that iteration. It's a time to ask questions, try new features, and give feedback. Sharing in success is an important part of building an agile team.

## 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-organize 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

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 intended to show that a program does what it is intended to do and to discover program defects before it is put into use. When you test software, you execute a program using artificial data. You check the results of the test run for errors, anomalies, or information about the programs non-functional attributes. The testing process has two distinct goals:

- To demonstrate to the developer and the customer that the software meets its requirements. For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.

- To discover situations in which the behaviour of the software is incorrect, undesirable, or does not conform to its specification. These are a consequence of software defects. Defect testing is concerned with rooting out undesirable system behaviour such as system crashes, unwanted interactions with other systems, incorrect 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 processes are concerned with checking that software being developed meets its specification and delivers the functionality expected by the people paying for the software. These checking processes start as soon as requirements become available and continue through all stages of the development process.

The ultimate goal of verification and validation processes is to establish confidence that the software system is 'fit for purpose'. This means that the system must be good enough for its intended use. The level of required confidence depends on the system's purpose, the expectations of the system users, and the current marketing environment for the system:

- **Software purpose:** The more critical the software, the more important that it is reliable. For example, the level of confidence required for software used to control a safety-critical system is much higher than that required for a prototype that has been developed to demonstrate new product ideas.

- **User expectations:** Because of their experiences with buggy, unreliable software, many users have low expectations of software quality. They are not surprised when their software fails. When a new system is installed, users may tolerate failures because the benefits of use outweigh the costs of failure recovery.

- **Marketing environment:** When a system is marketed, the sellers of the system must take into account competing products, the price that customers are willing to pay for a system, and the required schedule for delivering that system. In a competitive environment, a software company may decide to release a program before it has been fully tested and debugged because they want to be the first into the market.

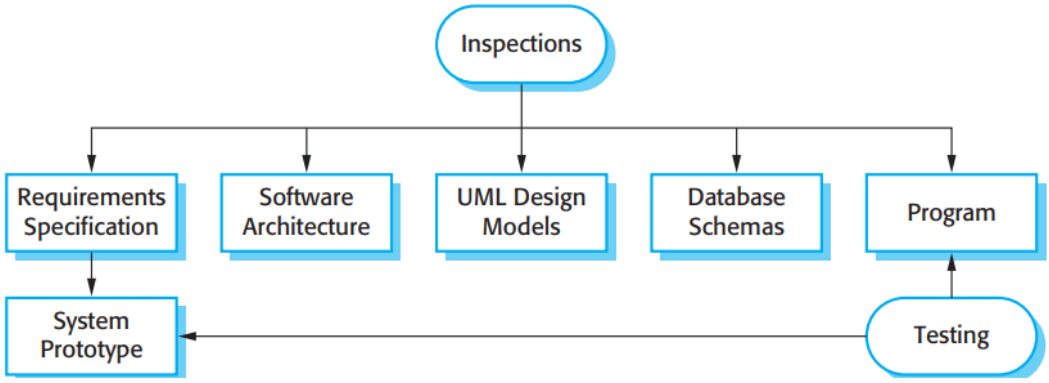


*Fig 2.6 12 Inspection and testing*

## 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

Unit testing is undertaken after a module has been coded and successfully reviewed. Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation. In order to test a single module, a complete environment is needed to provide all that is necessary for execution of the module. That is, besides the module under test itself, the following steps are needed in order to be able 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 provide the necessary environment (which either call or are called by the module under test) is usually not available until they too have been unit tested, stubs and drivers are designed to provide the complete environment for a module. The role of stub and driver modules is pictorially shown in fig. 19.1. A stub procedure is a dummy procedure that has the same I/O parameters as the given procedure but has a highly simplified behaviour. For example, a stub procedure may produce the expected behaviour using a simple table lookup mechanism. A driver module contain the nonlocal data structures accessed by the module under test, and would also have the code to call the different functions of the module with appropriate parameter values.

### 7.4.3 Black Box Testing

In the black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. The following are the two main approaches to designing black box test cases.

- 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

One white-box testing strategy is said to be stronger than another strategy, if all types of errors detected by the first testing strategy is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors. When two testing strategies detect errors that are different at least with respect to some types of errors, then they are called complementary.

### 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.

Apart from above, there are numerous coverage types such as Condition Coverage, Multiple Condition Coverage, Path Coverage, Function Coverage etc. Each technique has its own merits and attempts to test (cover) all parts of software code. Using Statement and Branch coverage you generally attain 80-90% code coverage which is sufficient.

### 7.5.2 Advantages vs Disadvantages

- **Advantages**

  - Code optimization by finding hidden errors.
  - White box tests cases can be easily automated.
  - Testing is more thorough as all code paths are usually covered.
  - 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 primary objective of integration testing is to test the module interfaces, i.e. there are no errors in the parameter passing, when one module invokes another module. During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realize the full system. After each integration step, the partially integrated system is tested. 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. Integration test approaches

There are four types of integration testing approaches. Any one (or a mixture) of the following approaches can be used to develop the integration test plan. Those approaches are the following:

- 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 main problem with this approach is that once an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated. Therefore, debugging errors reported during big bang integration testing are very expensive to fix.

### 7.6.2 Bottom-Up Integration Testing

In bottom-up testing, each subsystem is tested separately and then the full system is tested. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. The primary purpose of testing each subsystem is to test the interfaces among various modules making up the subsystem. Both control and data interfaces are tested. The test cases must be carefully chosen to exercise the interfaces in all possible manners Large software systems normally require several levels of subsystem testing; lower-level subsystems are successively combined to form higher-level subsystems. A principal advantage of bottom-up integration testing is that several disjoint subsystems can be tested simultaneously. In a pure bottom-up testing no stubs are required, only test-drivers are required. A disadvantage of bottom up testing is the complexity that occurs when the system is made up of a large number of small subsystems. The extreme case corresponds to the big-bang approach.

### 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 called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready. The mixed approach overcomes this shortcoming of the top down and bottom-up approaches. In the mixed testing approaches, testing can start as and when modules become available. Therefore, this is one of the most commonly used integration testing approaches.

## 7.7 Unit Summary

■ Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use. When you test software, you execute a program using artificial data.

■ Verification and validation processes are concerned with checking that software being developed meets its specification and delivers the functionality expected by the people paying for the software.

## 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

Digital transformation is the integration of digital technology into all areas of a business, fundamentally changing how you operate and deliver value to customers. It's also a cultural change that requires organizations to continually challenge the status quo, experiment, and get 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 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.

## 8.2 Digital Transformation

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. What's not clear to many business leaders is what digital transformation means. Is it just a catchy way to say moving to the cloud? What are the specific steps we need to take? Do we need to design new jobs to help us create a framework for digital transformation, or hire a consulting service? What parts of our business strategy need to change? Is it really worth it?

Because digital transformation will look different for every company, it can be hard to pinpoint a definition that applies to all. However, in general terms, we define digital transformation as the integration of digital technology into all areas of a business resulting in fundamental changes to how businesses operate and how they deliver value to customers. Beyond that, it's a cultural change that requires organizations to continually challenge the status quo, experiment often, and get comfortable with failure. This sometimes means walking away from long-standing business processes that companies were built upon in favour of relatively new practices that are still being defined.

## 8.3 Why Digital Transformation?

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 organization's ability to adapt quickly to supply chain disruptions, time to market pressures, and rapidly changing customer expectations has become critical.

And spending priorities reflect this reality. According to the May, 2020 International Data Corporation (IDC) Worldwide Digital Transformation Spending Guide, spending on the digital transformation (DX) of business practices, products, and organizations continues "at a solid pace despite the challenges presented by the COVID-19 pandemic." IDC forecasts that global spending on DX technologies and services will grow 10.4 percent in 2020 to $1.3 trillion. That compares to 17.9 percent growth in 2019, "but remains one of the few bright spots in a year characterized by dramatic reductions in overall technology spending," IDC notes.

It's early to guess which long-term consumer behavior changes will stick. However, Rodney Zemmel, global leader, McKinsey Digital of McKinsey & Company, says that on the consumer side "digital has been accelerating in just about all categories." An important factor to watch will be the degree to which forced change — three out of four Americans tried a new shopping behavior, for example — will revert when possible, post today's emphasis on stay-in-place.

McKinsey data shows that the accelerated shift towards streaming and online fitness is likely to stay permanently, Zemmel says. But the biggest shifts were around food. Both home cooking and online grocery shopping — a category that has been generally resistant to getting moved online — will probably stay more popular with consumers than in the past. Cashless transactions are also gaining steam. On the B2B side, McKinsey data shows remote selling is working.

## 8.4 Digital Transformation Framework

Although digital transformation will vary widely based on organization'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.

For instance, these digital transformation elements are often cited:

- 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 organization'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 organization'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 with the current covid Pandemic?