SRINIVAS UNIVERSITY
INSTITUTE OF ENGINEERING AND TECHNOLOGY
MUKKA, MANGALURU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NOTES

SOFTWARE ENGINEERING

SUBJECT CODE: 19SCS44

COMPILED BY:

Ms. SHIFANA BEGUM
Assistant Professor,

Module-1

1. INTRODUCTION, SOFTWARE PROCESSES, REQUIREMENTS ENGINEERING

INTRODUCTION

Software Crisis

→ Software crisis is a term used in the early days of computing science for the difficulty of writing useful and efficient computer programs in the required time.

→ The software crisis was due to the rapid increases in computer power and the complexity of the problems that could be tackled.

→ With the increase in the complexity of the software, many software problems arose because existing methods were neither sufficient nor up to the mark.

→ The causes of the software crisis were linked to the overall complexity of hardware and the software development process.

→ The crisis manifested itself in several ways:

* Projects running over-budget
* Projects running over-time
* Software was very inefficient
* Software was of low quality
* Software often did not meet requirements
* Projects were unmanageable and code difficult to maintain
* Software was never delivered

Need for Software Engineering

→ The need of software engineering arises because of higher rate of change in user requirement and environment on which the software is working.

→ The following factors contribute to the need of software engineering.

* **Large Software:** As the size of software becomes large, engineering has to step to give it a scientific process.
* **Scalability:** If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing

* **Cost:** As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.
* **Dynamic Nature:** The always growing and adapting nature of software hugely depends upon the environment in which user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.
* **Quality Management:** Better process of software development provides better and quality software product.

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 are normally relevant for personal software development.

→ Software is not just the programs themselves but also all associated documentation and configuration data that is required to make these programs operate correctly.

→ A professionally developed software system is often more than a single program.

→ The system usually consists of a number of separate programs and configuration files that are used to set up these programs.

→ It may include system documentation, which describes the structure of the system; user documentation, which explains how to use the system, and websites for users to download recent product information

→ Fig 1.1 gives frequently asked questions about software.

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.
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. You can't, therefore, say that one method is better than another.
What differences has the Web made to software engineering?	The Web has led to the availability of software services and the possibility of developing highly distributed service-based systems. Web-based systems development has led to important advances in programming languages and software reuse.

Fig1.1: Frequently asked questions about software

→ There are two fundamental types of software product:

1. **Generic Products:** These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them. Examples of this type of product include software for PCs such as databases, word processors, drawing packages and project management tools.
2. **Customized (or bespoke) Products:** These are systems which are commissioned by a particular customer. A software contractor develops 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.

→ Fig 1.2 gives the essential characteristics of a professional software system.

Product characteristics	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable, and compatible with other systems that they use.

Fig 1.2: Essential attributes of good software

Software Engineering

→ Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.

→ In this definition, there are two key phrases:

1. **Engineering discipline:** Engineers make things work. They apply theories, methods, and tools where these are appropriate. However, they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods
2. **All aspects of software production:** Software engineering is not just concerned with the technical processes of software development. It also includes activities such as software project management and the development of tools, methods, and theories to support software production.

→ Software engineering is important for two reasons:

1. More and more, individuals and society rely on advanced software systems. It helps users to produce reliable and trustworthy systems economically and quickly.

2. It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project.

→ There are four fundamental activities that are common to all software processes.

These activities are:

1. **Software Specification**, where customers and engineers define the software that is to be produced and the constraints on its operation.
2. **Software Development**, where the software is designed and programmed.
3. **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.

→ Software engineering is related to both computer science and systems engineering:

1. **Computer Science** is concerned with the theories and methods that underlie computers and software systems, whereas software engineering is concerned with the practical problems of producing software.
2. **System Engineering** is concerned with all aspects of the development and evolution of complex systems where software plays a major role. System engineering is therefore concerned with hardware development, policy and process design and system deployment, as well as software engineering. System engineers are involved in specifying the system, defining its overall architecture, and then integrating the different parts to create the finished system. They are less concerned with the engineering of the system components (hardware, software, etc.)

→ There are three general issues that affect many different types of software:

1. **Heterogeneity:** Here it becomes necessary to integrate new software with older legacy systems written in different programming languages. The challenge here is to develop techniques for building dependable software that is flexible enough to cope with this heterogeneity.
2. **Business and Social Change:** Business and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and to rapidly develop new software.

3. **Security and Trust:** As software is intertwined with all aspects of our lives, it is essential that we can trust that software. This is especially true for remote software systems accessed through a web page or web service interface.

Software Engineering Diversity

→ There are many different types of application including:

1. **Stand-alone applications:** These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to a network. Examples of such applications are office applications on a PC, CAD programs, photo manipulation software, etc.
2. **Interactive transaction-based applications:** These are applications that execute on a remote computer and that are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications where users can interact with a remote system to buy goods and services.
3. **Embedded control systems:** These are software control systems that control and manage hardware devices. Examples of embedded systems include the software in a mobile (cell) phone, software that controls anti-lock braking in a car, and software in a microwave oven to control the cooking process.
4. **Batch processing systems:** These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs. Examples of batch systems include periodic billing systems, such as phone billing systems, and salary payment systems.
5. **Entertainment systems:** These are systems that are primarily for personal use and which are intended to entertain the user. Most of these systems are games of one kind or another. The quality of the user interaction offered is the most important distinguishing characteristic of entertainment systems.
6. **Systems for modeling and simulation:** These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects. These are often computationally intensive and require high-performance parallel systems for execution.
7. **Data collection systems:** These are systems that collect data from their environment using a set of sensors and send that data to other systems for

processing. The software has to interact with sensors and often is installed in a hostile environment such as inside an engine or in a remote location.

8. **Systems of systems:** These are systems that are composed of a number of other software systems. Some of these may be generic software products, such as a spreadsheet program. Other systems in the assembly may be specially written for that environment.

→ There are software engineering fundamentals that apply to all types of software system:

* They should be developed using a managed and understood development process. The organization developing the software should plan the development process and have clear ideas of what will be produced and when it will be completed.
* Dependability and performance are important for all types of systems. Software should behave as expected, without failures and should be available for use when it is required. It should be safe in its operation and, as far as possible, should be secure against external attack. The system should perform efficiently and should not waste resources.
* Understanding and managing the software specification and requirements are important. It is important to understand what different customers and users of the system expect from it and then it is needed to manage their expectations so that a useful system can be delivered within budget and to schedule.
* Existing resources must be used efficiently. This means that, where appropriate, you should reuse software that has already been developed rather than write new software.

Software engineering and the Web

→ The next stage in the development of web-based systems was the notion of web services.

→ Web services are software components that deliver specific, useful functionality and which are accessed over the Web. - Applications are constructed by integrating these web services, which may be provided by different companies.

→ In principle, this linking can be dynamic so that an application may use different web services each time that it is executed.

→ Changes in the software organization, led to changes in the ways that web-based systems are engineered. 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.
* It is now generally recognized that it is impractical to specify all the requirements for such systems in advance. Web-based systems should be developed and delivered incrementally.
* User interfaces are constrained by the capabilities of web browsers. Web forms with local scripting are more commonly used. Application interfaces on web-based systems are often poorer than the specially designed user interfaces on PC system products.

Software Engineering Ethics

→ Some professional responsibilities includes:

- **Confidentiality:** You should normally respect the confidentiality of your employers or clients irrespective 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.

Software Engineering Code of Ethics and Professional Practice

ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. PUBLIC – Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER – Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT – Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT – Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT – Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION – Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES – Software engineers shall be fair to and supportive of their colleagues.
8. SELF – Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Fig 1.3: ACM/IEEE Code of Ethics

Case Studies

→ 3 types of systems used as case studies are:

1. **An Embedded System:**
   - This is a system where the software controls a hardware device and is embedded in that device.
   - Issues in embedded systems typically include physical size, responsiveness, power management, etc.
   - The example of an embedded system used here is a software system to control a medical device.
2. **An Information System:**
   - This is a system whose primary purpose is to manage and provide access to a database of information.
   - Issues in information systems include security, usability, privacy, and maintaining data integrity.
   - The example of an information system t used here is a medical records system.
3. **A Sensor-based Data Collection System:**

- This is a system whose primary purpose is to collect data from a set of sensors and process that data in some way.
- The key requirements of such systems are reliability, even in hostile environmental conditions, and maintainability.
- The example of a data collection system used here is a wilderness weather station.

An insulin pump control system

→ An insulin pump is a medical system that simulates the operation of the pancreas.

→ The software controlling this system is an embedded system, which collects information from a sensor and controls a pump that delivers a controlled dose of insulin to a user.

→ Diabetes is a relatively common condition where the human pancreas is unable to produce sufficient quantities of a hormone called insulin.

→ Insulin metabolises glucose (sugar) in the blood.

→ A software-controlled insulin delivery system might work by using a micro sensor embedded in the patient to measure some blood parameter that is proportional to the sugar level.

→ This is then sent to the pump controller.

→ This controller computes the sugar level and the amount of insulin that is needed. It then sends signals to a miniaturized pump to deliver the insulin via a permanently attached needle.

→ Fig 1.4 shows the insulin pump hardware.

→ Fig 1.5 is a UML activity model that illustrates how the software transforms an input blood sugar level to a sequence of commands that drive the insulin pump.

→ Two essential high-level requirements that this system must meet includes:

- The system shall be available to deliver insulin when required.
- The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar

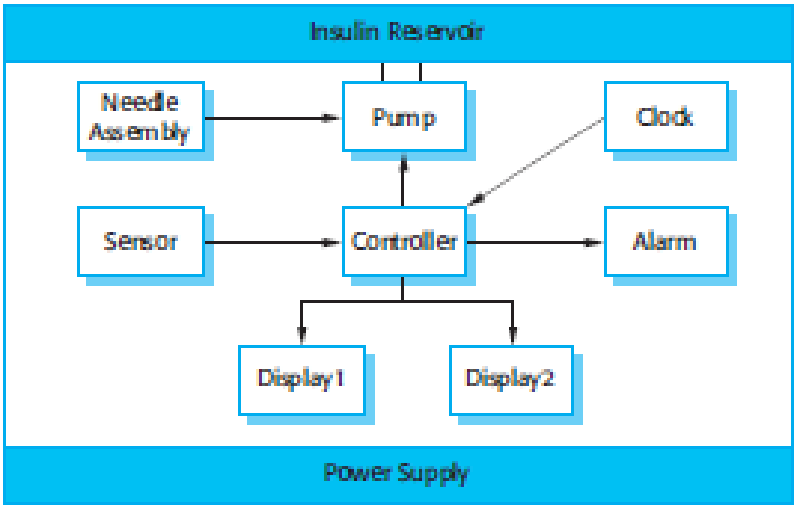

Fig 1.4: Insulin pump hardware

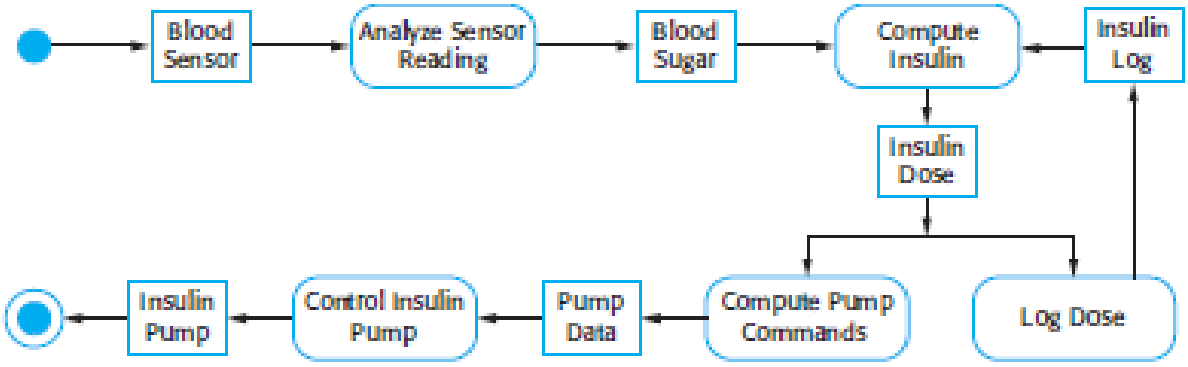

Fig 1.5: Activity model of the insulin pump

A patient information system for mental health care

→ A patient information system to support mental health care is a medical information system that maintains information about patients suffering from mental health problems and the treatments that they have received.

→ Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems.

→ To make it easier for patients to attend, these clinics are not just run in hospitals.

→ They may also be held in local medical practices or community centers.

→ The MHC-PMS (Mental Health Care-Patient Management System) is an information system that is intended for use in clinics. It makes use of a centralized database of patient information but has also been designed to run on a PC, so that it may be accessed and used from sites that do not have secure network connectivity.

→ When the local systems have secure network access, they use patient information in the database but they can download and use local copies of patient records when they are disconnected.

→ The system is not a complete medical records system so does not maintain information about other medical conditions.

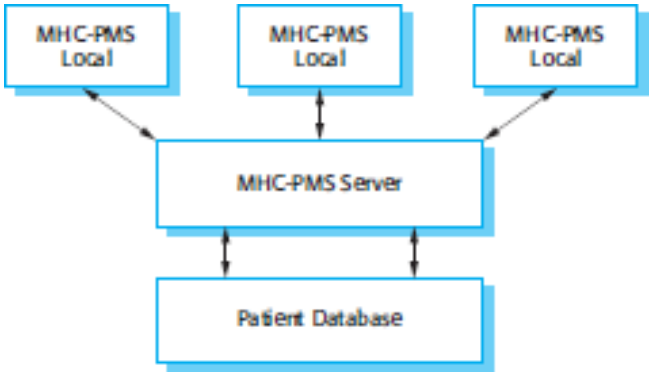

Fig 1.6: The organization of the MHC-PMS

→ The MHC-PMS has two overall goals:

* To generate management information that allows health service managers to assess performance against local and government targets.
* To provide medical staff with timely information to support the treatment of patients.

→ The system is used to record information about patients (name, address, age, next of kin, etc.), consultations (date, doctor seen, subjective impressions of the patient, etc.), conditions, and treatments.

→ Reports are generated at regular intervals for medical staff and health authority managers

→ The key features of the system are:

1. **Individual care Management:** Clinicians can create records for patients, edit the information in the system, view patient history, etc.
2. **Patient Monitoring:** The system regularly monitors the records of patients that are involved in treatment and issues warnings if possible problems are detected.
3. **Administrative Reporting:** The system generates monthly management reports showing the number of patients treated at each clinic, the number of patients who have entered and left the care system, number of patients sectioned, the drugs prescribed and their costs, etc.

A wilderness weather station

→ Wilderness weather stations are part of a larger system (Figure 1.5.4), which is a weather information system that collects data from weather stations and makes it available to other systems for processing.

→ The systems in Fig 1.7 are:

1. **The weather station system:** This is responsible for collecting weather data, carrying out some initial data processing, and transmitting it to the data management system.
2. **The data management and archiving system:** This system collects the data from all of the wilderness weather stations, carries out data processing and analysis, and archives the data in a form that can be retrieved by other systems, such as weather forecasting systems.
3. **The station maintenance system:** This system can communicate by satellite with all wilderness weather stations to monitor the health of these systems and provide reports of problems. It can update the embedded software in these systems. In the event of system problems, this system can also be used to remotely control a wilderness weather system

→ Each weather station is battery-powered and must be entirely self-contained—there are no external power or network cables available.

→ All communications are through a relatively slow-speed satellite link and the weather station must include some mechanism (solar or wind power) to charge its batteries.

→ As they are deployed in wilderness areas, they are exposed to severe environmental conditions and may be damaged by animals.

→ The station software is therefore not just concerned with data collection.

→ It must also:

* Monitor the instruments, power, and communication hardware and report faults to the management system.
* Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit but also that generators are shut down in potentially damaging weather conditions, such as high wind.

* Allow for dynamic reconfiguration where parts of the software are replaced with new versions and where backup instruments are switched into the system in the event of system failure.

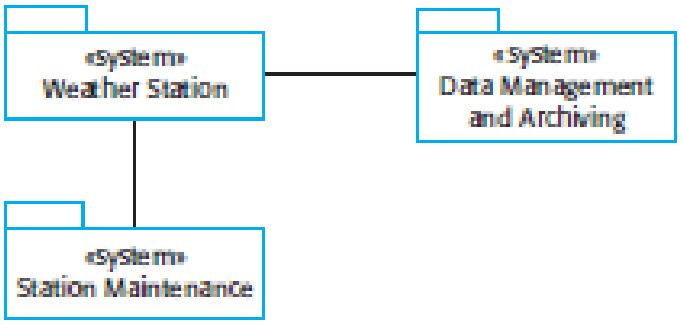

Fig 1.7: The weather station's environment

SOFTWARE PROCESSES

→ 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 like Java or C.

→ 4 activities that are fundamental to software engineering:

1. **Software Specification:** The functionality of the software and constraints on its operation must be defined.
2. **Software Design and Implementation:** The software to meet the specification must be produced
3. **Software Validation:** The software must be validated to ensure that it does what the customer wants.
4. **Software Evolution:** The software must evolve to meet changing customer needs.

→ Process descriptions may also include:

1. **Products**, which are the outcomes of a process activity. For example, the outcome of the activity of architectural design may be a model of the software architecture.
2. **Roles**, which reflect the responsibilities of the people involved in the process. Examples of roles are project manager, configuration manager, programmer, etc.

3. **Pre and post-conditions**, which are statements that are true before and after a process activity has been enacted or a product produced. For example, before architectural design begins, a pre-condition may be that all requirements have been approved by the customer; after this activity is finished, a post-condition might be that the UML models describing the architecture have been reviewed

Models

→ A software process model is a simplified representation of a software process.

→ Each process model represents a process from a particular perspective, and thus provides only partial information about that process.

The Waterfall Model

→ Because of the cascade from one phase to another, this model is known as the 'waterfall model' or software life cycle.

→ The waterfall model is an example of a plan- driven process.

→ The principal stages of the waterfall model [Fig 1.8] directly reflect the fundamental development activities:

1. **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.
2. **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.
3. **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.
4. **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.
5. **Operation and maintenance:** 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.

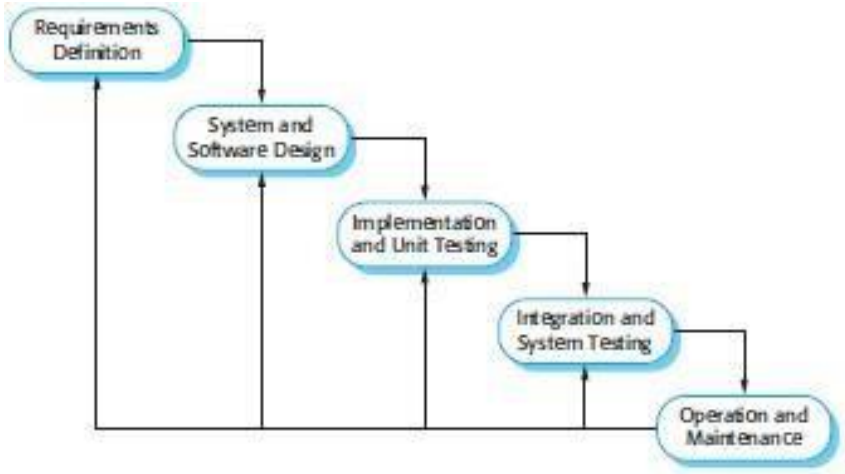

Fig 1.8: The Waterfall Model

→ 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

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 [Fig 1.9].

→ 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.
* More rapid delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included.

→ 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.

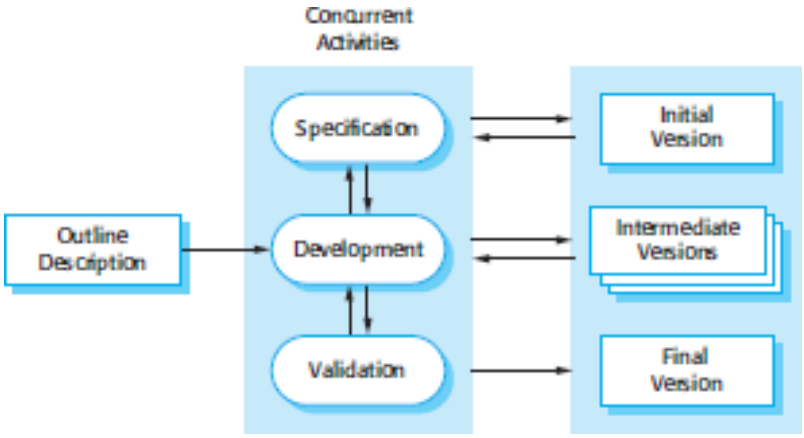

Fig 1.9: Incremental Development

Boehm's Spiral Model

→ Here, the software process is represented as a spiral, rather than a sequence of activities with some backtracking from one activity to another [Fig 1.10].

→ Each loop in the spiral represents a phase of the software process.

→ Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design, and so on.

→ Each loop in the spiral is split into four sectors:

1. **Objective Setting:** Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified.
2. **Risk Assessment and Reduction:** For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

3. **Development and Validation:** After risk evaluation, a development model for the system is chosen. For example, throwaway prototyping may be the best development approach if user interface risks are dominant.
4. **Planning:** The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

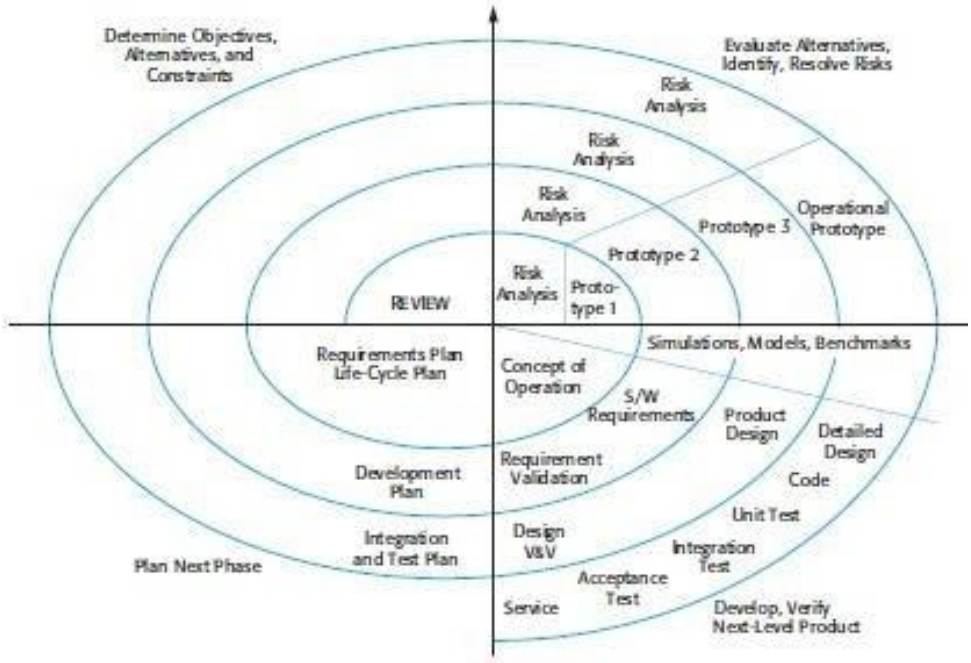

Fig 1.10: Boehm's spiral model of the software process

Process Activities

→ The four basic process activities of specification, development, validation, and evolution are organized differently in different development processes

Software Specification

→ It 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.

→ The requirements engineering process aims (Fig 1.11) to produce an agreed requirements document that specifies a system satisfying stakeholder requirements.

→ Requirements are usually presented at two levels of detail.

→ End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification.

→ There are four main activities in the requirements engineering process:

1. **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.
2. **Requirements Elicitation and Analysis***:*
   - Process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on.
   - May involve the development of one or more system models and prototypes.
3. **Requirements Specification**:
   - It 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.
4. **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.

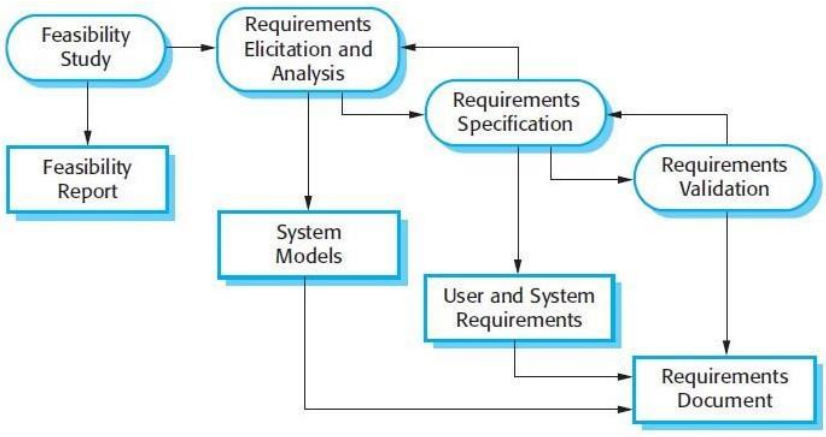

Fig 1.11: The Requirements Engineering Process

Software Design and Implementation

→ The implementation stage of software development is the process of converting a system specification into an executable system.

→ A software design is a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and, sometimes, the algorithms used .

→ Fig 1.12 shows an abstract model of this process showing the inputs to the design process, process activities, and the documents produced as outputs from this process.

→ It shows four activities that may be part of the design process for information systems:

1. **Architectural design,** where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships, and how they are distributed.
2. **Interface design**, where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component can be used without other components having to know how it is implemented. Once interface specifications are agreed, the components can be designed and developed concurrently.
3. **Component design**, where you take each system component and design how it will operate. This may be a simple statement of the expected functionality to be implemented, with the specific design left to the programmer. The design model may be used to automatically generate an implementation.
4. **Database design**, where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created

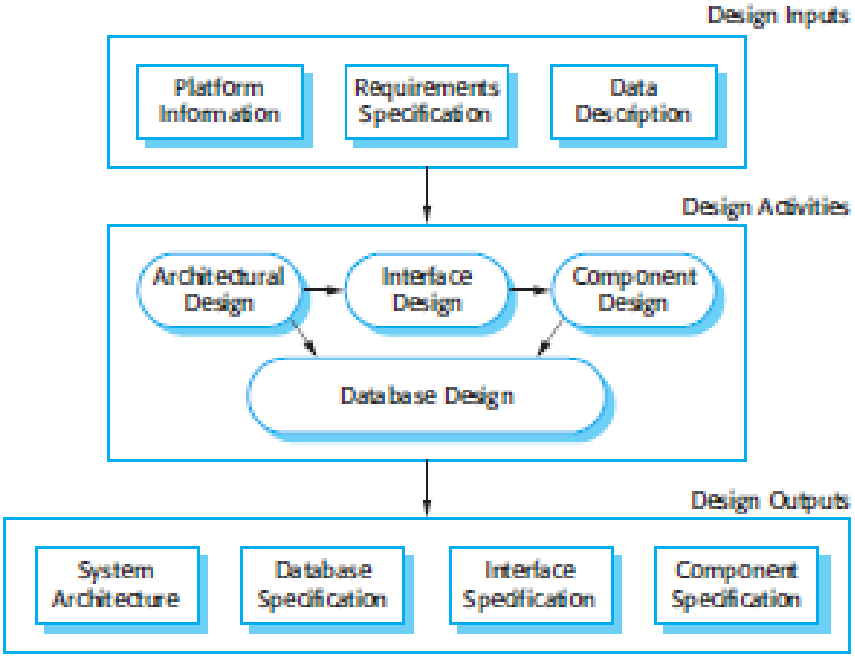

Fig 1.12: A general model of the design process

Software Validation

→ Software validation or, more generally, verification and validation (V&V) is intended to show that a system both conforms to its specification and that it meets the expectations of the system customer.

→ Program testing, where the system is executed using simulated test data, is the principal validation technique.

→ Validation may also involve checking processes, such as inspections and reviews, at each stage of the software process from user requirements definition to program development.

→ Fig 1.13 shows a three-stage testing process in which system components are tested then the integrated system is tested and, finally, the system is tested with the customer's data.

→ The stages in the testing process are:

1. **Development Testing:**
   - The components making up the system are tested by the people developing the system.
   - Each component is tested independently, without other system components.

* Components may be simple entities such as functions or object classes, or may be coherent groupings of these entities.

2. **System Testing:**
   * System components are integrated to create a complete system.
   * This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems.
   * It is also concerned with showing that the system meets its functional and non-functional requirements, and testing the emergent system properties.
3. **Acceptance Testing:**
   * This is the final stage in the testing process before the system is accepted for operational use.
   * The system is tested with data supplied by the system customer rather than with simulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data.

→ Fig 1.14 illustrates how test plans are the link between testing and development activities. This is sometimes called the V-model of development.

→ Acceptance testing is sometimes called **'alpha testing'.**

→ Custom systems are developed for a single client.

→ The alpha testing process continues until the system developer and the client agree that the delivered system is an acceptable implementation of the requirements.

→ When a system is to be marketed as a software product, a testing process called **'beta testing'** is often used.

→ Beta testing involves delivering a system to a number of potential customers who agree to use that system. They report problems to the system developers.

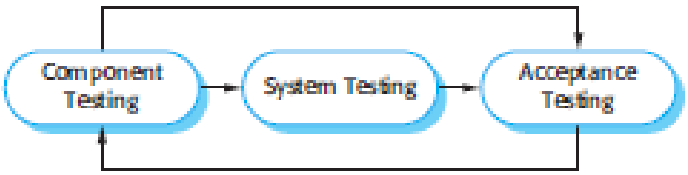

Fig 1.13: Stages of testing

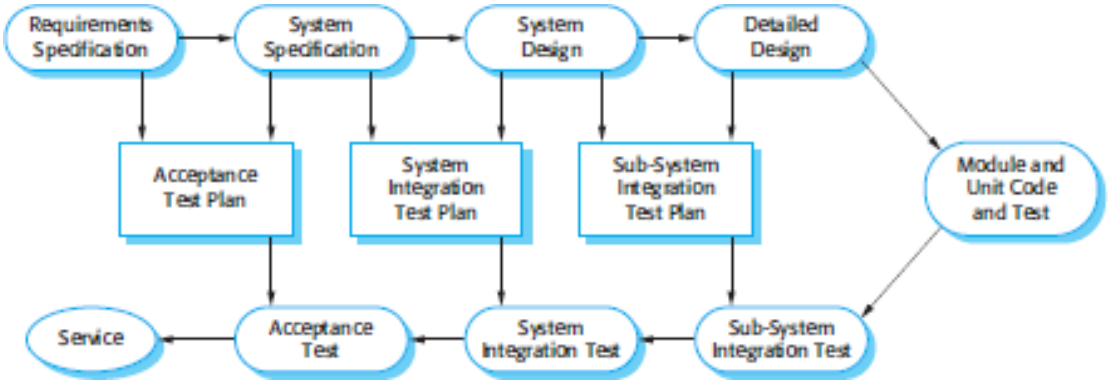

Fig 1.14: Testing phases in a plan-driven software process

Software Evolution

→ The flexibility of software systems is one of the main reasons why more and more software is being incorporated in large, complex systems.

→ Once a decision has been made to manufacture hardware, it is very expensive to make changes to the hardware design.

→ However, changes can be made to software at any time during or after the system development.

→ Even extensive changes are still much cheaper than corresponding changes to system hardware.

→ The fig 1.15 shows that software is continually changed over its lifetime in response to changing requirements and customer needs

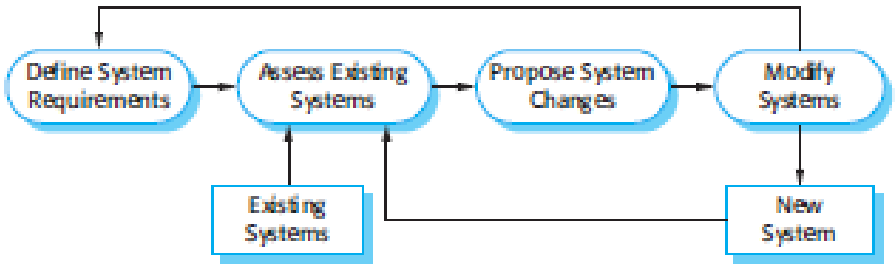

Fig 1.15: System Evolution

REQUIREMENTS ENGINEERING

Requirements Engineering Processes

→ Requirements engineering processes may include four high-level activities.

→ These focus on assessing if the system is useful to the business (feasibility study), discovering requirements (elicitation and analysis), converting these requirements into

some standard form (specification), and checking that the requirements actually define the system that the customer wants (validation).

→ Fig 1.16 below shows this interleaving. The activities are organized as an iterative process around a spiral, with the output being a system requirements document.

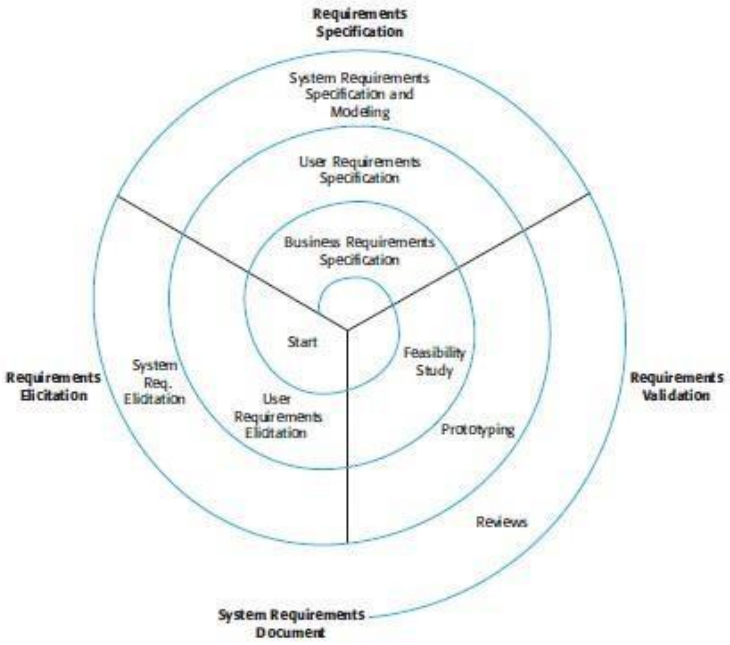

Fig 1.16: A spiral view of the requirements engineering process.

→ The amount of time and effort devoted to each activity in each iteration depends on the stage of the overall process and the type of system being developed.

→ This spiral model accommodates approaches to development where the requirements are developed to different levels of detail.

→ The number of iterations around the spiral can vary so the spiral can be exited after some or all of the user requirements have been elicited

Requirements Elicitation and Analysis

→ After an initial feasibility study, the next stage of the requirements engineering process is requirements elicitation and analysis.

→ In this activity, software engineers work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.

→ A process model of the elicitation and analysis process is shown in fig 1.17.

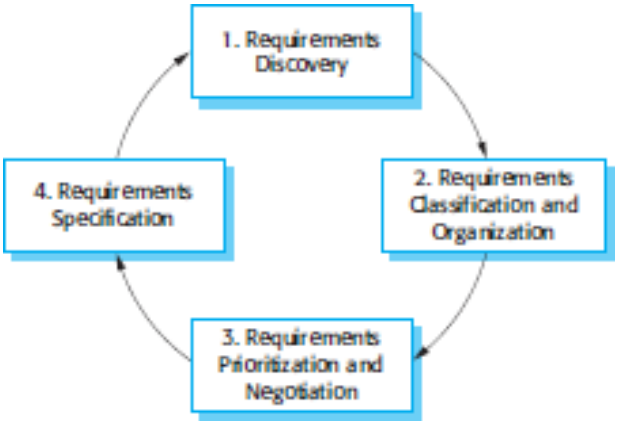

Fig 1.17: The requirements elicitation and analysis process

→ The process activities are:

1. **Requirements discovery**: This is the processof interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity.
2. **Requirements classification and organization:** This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system.
3. **Requirements prioritization and negotiation**: This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.
4. **Requirements specification**: The requirements are documented and input into the next round of the spiral. Formal or informal requirements documents may be produced.

→ Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

* Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.
* Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers,

without experience in the customer's domain, may not understand these requirements.

* Different stakeholders have different requirements and they may express these in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.
* Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.
* The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

Requirements Discovery

→ Requirements discovery (sometime called requirements elicitation) is the process of gathering information about the required system and existing systems, and distilling the user and system requirements from this information.

→ For example, system stakeholders for the mental healthcare patient information system include:

* Patients whose information is recorded in the system.
* Doctors who are responsible for assessing and treating patients.
* Nurses who coordinate the consultations with doctors and administer some treatments.
* Medical receptionists who manage patients' appointments.
* IT staff who are responsible for installing and maintaining the system. These different requirements sources (stakeholders, domain, systems) can all be represented as system viewpoints with each viewpoint showing a subset of the requirements for the system.

Interviewing

→ In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed.

→ Interviews may be of two types:

1. Closed interviews, where the stakeholder answers a pre-defined set of questions.
2. Open interviews, in which there is no pre-defined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develop a better understanding of their needs.

→ It can be difficult to elicit domain knowledge through interviews for two reasons:

1. All application specialists use terminology and jargon that are specific to a domain. It is impossible for them to discuss domain requirements without using this terminology.
2. Some domain knowledge is so familiar to stakeholders that they either find it difficult to explain or they think it is so fundamental that it isn't worth mentioning. For example, for a librarian, it goes without saying that all acquisitions are catalogued before they are added to the library. However, this may not be obvious to the interviewer, and so it isn't taken into account in the requirements.

→ Effective interviewers have two characteristics:

1. They are open-minded, avoid pre-conceived ideas about the requirements, and are willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, then they are willing to change their mind about the system.
2. They prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system. They find it much easier to talk in a defined context rather than in general terms

Scenarios

→ They are descriptions of example interaction sessions.

→ Each scenario usually covers one or a small number of possible interactions.

→ Different forms of scenarios are developed and they provide different types of information at different levels of detail about the system.

→ At its most general, a scenario may include:

* A description of what the system and users expects when the scenario starts.
* A description of the normal flow of events in the scenario.

- A description of what can go wrong and how this is handled.
- Information about other activities that might be going on at the same time.
- A description of the system state when the scenario finishes.

Use Cases

→ A use case identifies the actors involved in an interaction and names the type of interaction.

→ Use cases are documented using a high-level use case diagram.

→ The set of use cases represents all of the possible interactions that will be described in the system requirements.

→ Actors in the process, who may be human or other systems, are represented as stick figures.

→ Each class of interaction is represented as a named ellipse.

→ Lines link the actors with the interaction.

→ Arrowheads may be added to lines to show how the interaction is initiated.

→ Each use case should be documented with a textual description. These can then be linked to other models in the UML that will develop the scenario in more detail.

→ For example, a brief description of the Setup Consultation use case from fig 1.18 below might be:

Setup consultation allows two or more doctors, working in different offices, to view the same record at the same time. One doctor initiates the consultation by choosing the people involved from a drop-down menu of doctors who are online.

The patient record is then displayed on their screens but only the initiating doctor can edit the record. In addition, a text chat window is created to help coordinate actions. It is assumed that a phone conference for voice communication will be separately set up.

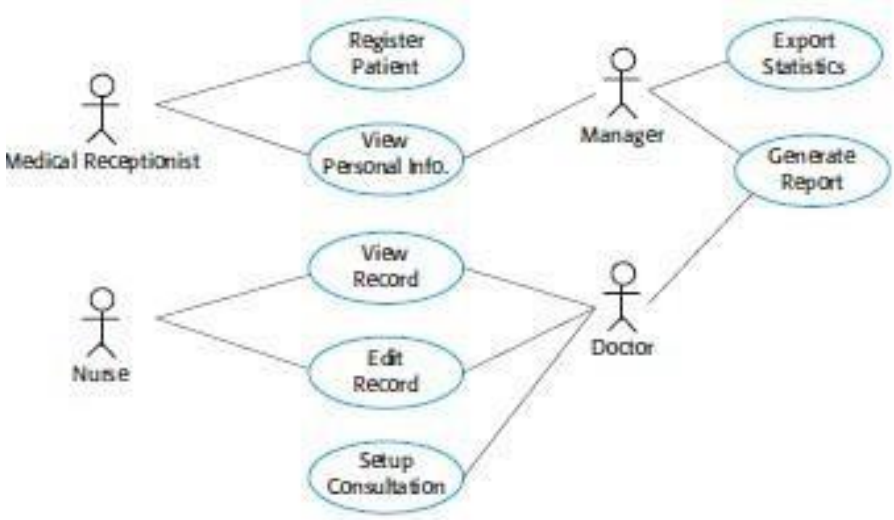

Fig 1.18 Use cases for the MHC-PMS

Ethnography

→ Ethnography is an observational technique that can be used to understand operational processes and help derive support requirements for these processes.

→ The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.

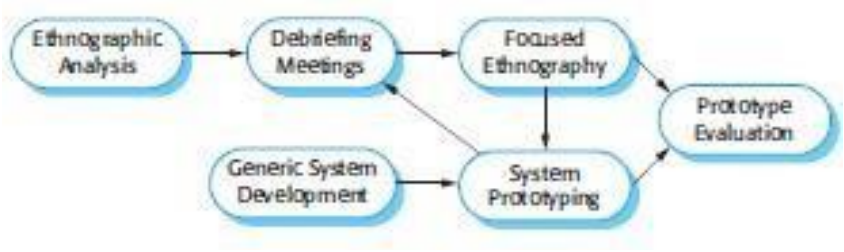

Fig 1.19: Ethnography and Prototyping for Requirements Analysis

→ Ethnography is particularly effective for discovering two types of requirements:

1. Requirements that are derived from the way in which people actually work, rather than the way in which process definitions say they ought to work
2. Requirements that are derived from cooperation and awareness of other people's activities.

→ Ethnography can be combined with prototyping as shown in fig 1.19.

Functional and Non functional Requirements

→ Software system requirements are often classified as functional requirements or non-functional 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.
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.

Functional Requirements

→ The functional requirements for a system describe what the system should do.

→ These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements.

→ When expressed as user requirements, functional requirements are usually described in an abstract way that can be understood by system users.

→ Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems.

→ Examples for functional requirements for MHC-PMS system includes:

* A user shall be able to search the appointments lists for all clinics.
* The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
* Each staff member using the system shall be uniquely identified by his or her eight- digit employee number.

→ The functional requirements specification of a system should be both complete and consistent.

→ Completeness means that all services required by the user should be defined.

→ Consistency means that requirements should not have contradictory definitions.

Non-Functional Requirements

→ They are requirements that are not directly concerned with the specific services delivered by the system to its users.

→ They may relate to emergent system properties such as reliability, response time, and store occupancy.

→ Non-functional requirements, such as performance, security, or availability, usually specify or constrain characteristics of the system as a whole.

→ Non-functional requirements are often more critical than individual functional requirements

→ The implementation of these requirements may be diffused throughout the system. There are two reasons for this:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components.
2. A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define new system services that are required. The figure below shows the classification of non-functional requirements

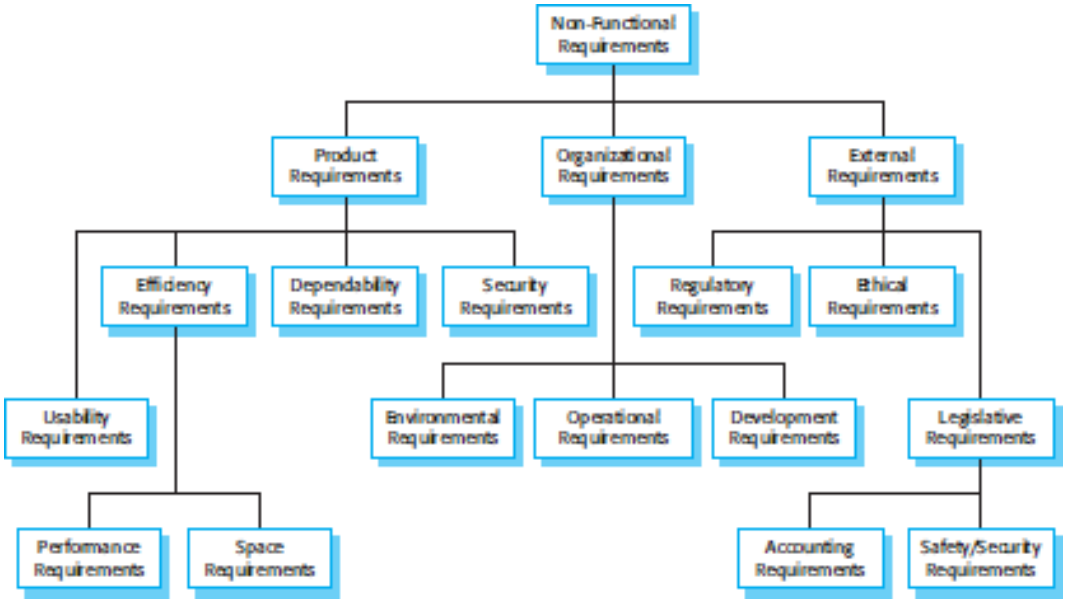

Fig 1.20: Types of Non-Functional Requirement

→ Fig 1.20 is a classification of non-functional requirements.

→ The various types includes:

1.* Product Requirements*:

* These requirements specify or constrain the behavior of the software.
* Examples include performance requirements on how fast the system must execute and how much memory it requires, reliability requirements that set out the acceptable failure rate, security requirements, and usability requirements.

2. **Organizational Requirements**:
   - These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organization.
   - Examples include operational process requirements that define how the system will be used, development process requirements that specify the programming language, the development environment or process standards to be used, and environmental requirements that specify the operating environment of the system.
3. **External requirements:**
   - This broad heading covers all requirements that are derived from factors external to the system and its development process.
   - These may include regulatory requirements that set Out what must be done for the system to be approved for use by a regulator, such as a central bank.

→ The fig 1.21 below shows the metric used for specifying non-functional requirements

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Fig 1.21: Metrics for specifying non functional requirements

The Software Requirements Document

→ The software requirements document (sometimes called the software requirements specification or SRS) is an official statement of what the system developers should

→ It should include both the user requirements for a system and a detailed specification of the system requirements.

→ The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software.

→ The users of requirements document is as shown below in fig 1.22.

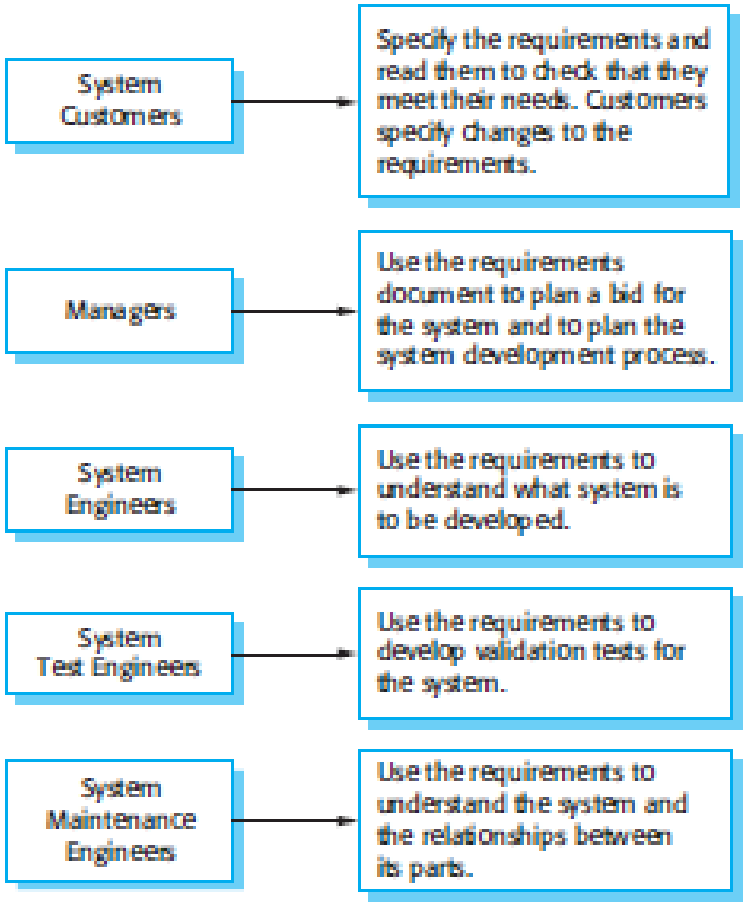

Fig 1.22: Users of a Requirement Document

→ Figure 1.23 shows one possible organization for a requirements document that is based on an IEEE standard for requirements documents

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The non-functional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components, the system, and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Fig 1.23: The Structure of a Requirements Document

Requirements Specification

→ Requirements specification is the process of writing down the user and system requirements in a requirements document.

→ System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design.

→ They add detail and explain how the user requirements should be provided by the system.

→ It is practically impossible to exclude all design information. There are several reasons for this:

→ You may have to design an initial architecture of the system to help structure the requirements specification.

→ The system requirements are organized according to the different sub-systems that make up the system

→ In most cases, systems must interoperate with existing systems, which constrain the design and impose requirements on the new system.

→ The use of a specific architecture to satisfy non-functional requirements may be necessary.

→ The fig 1.24 below shows the different ways of writing system requirement specification.

Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract.

Fig 1.24 Ways of writing a system Requirements specification

Natural Language Specification

→ To minimize misunderstandings when writing natural language requirements, there are some simple guidelines to be followed:

1. Invent a standard format and ensure that all requirement definitions adhere to that format.
2. Use language consistently to distinguish between mandatory and desirable requirements.
3. Use text highlighting (bold, italic, or color) to pick out key parts of the requirement.
4. Do not assume that readers understand technical software engineering language. It is easy for words like ‘architecture’ and ‘module’ to be misunderstood. You should, therefore, avoid the use of abbreviations, and acronyms.

5. Whenever possible, you should try to associate a rationale with each user requirement.

→ Fig 1.25 illustrates how these guidelines may be used. It includes two requirements for the embedded software for the automated insulin pump

3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)

3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

Fig 1.25: Example requirements for the insulin pump software system

Structured Specifications

→ Structured natural language is a way of writing system requirements where the freedom of the requirements writer is limited and all requirements are written in a standard way.

→ Structured language notations use templates to specify system requirements.

→ An example of a form-based specification, that defines how to calculate the dose of insulin to be delivered when the blood sugar is within a safe band, as shown in fig 1.26.

Insulin Pump/Control Software/SRS/3.3.2

Function	Compute insulin dose: Safe sugar level.
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.
Inputs	Current sugar reading (r2), the previous two readings (r0 and r1).
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose—the dose in insulin to be delivered.
Destination	Main control loop.
Action	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.
Requirements	Two previous readings so that the rate of change of sugar level can be computed.
Pre-condition	The insulin reservoir contains at least the maximum allowed single dose of insulin.
Post-condition	r0 is replaced by r1 then r1 is replaced by r2.
Side effects	None.

Fig 1.26: A structured specification of a requirement for an insulin pump

→ When a standard form is used for specifying functional requirements, the following information should be included:

* A description of the function or entity being specified.
* A description of its inputs and where these come from.
* A description of its outputs and where these go to.

- Information about the information that is needed for the computation or other entities in the system that are used (the 'requires' part).
- A description of the action to be taken.
- If a functional approach is used, a pre-condition setting out what must be true before the function is called, and a post-condition specifying what is true after the function is called.
- A description of the side effects (if any) of the operation.

Requirements Validation

→ Requirements validation is the process of checking that requirements actually define the system that the customer really wants.

→ It overlaps with analysis as it is concerned with finding problems with the requirements.

→ During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document.

→ These checks include:

1. **Validity Checks*:*** A user may think that a system is needed to perform certain functions.
2. **Consistency Checks:** Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.
3. **Completeness Checks:** The requirements document should include requirements that define all functions and the constraints intended by the system user.
4. **Realism Checks:** Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented.
5. **Verifiability:** To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

→ There are a number of requirements validation techniques that can be used individually or in conjunction with one another:

1. **Requirements Reviews**: The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.
2. **Prototyping**: In this approach to validation, an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.
3. **Test-Case Generation**: Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems.

Requirements Management

→ The requirements for large software systems are always changing.

→ Once a system has been installed and is regularly used, new requirements inevitably emerge.

→ There are several reasons why change is inevitable:

* The business and technical environment of the system always changes after installation. New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change
* The people who pay for a system and the users of that system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.
* Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.

Requirements Management Planning

→ Planning is an essential first stage in the requirements management process. The planning stage establishes the level of requirements management detail that is required.

→ During the requirements management stage, a decision is to be taken on:

1. **Requirements Identification:**
   * Each requirement must be uniquely identified so that it can be cross-referenced with other requirements & used in traceability assessments.

2. **A Change Management Process:**
   - This is the set of activities that assess the impact and cost of changes.
3. **Traceability Policies:**
   - These policies define the relationships between each requirement and between the requirements and the system design that should be recorded.
   - The traceability policy should also define how these records should be maintained.
4. **Tool Support:**
   - Requirements management involves the processing of large amounts of information about the requirements.
   - Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.
   - Tool supports might be needed for:
     a. ***Requirements Storage:*** The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.
     b. ***Change Management:*** The process of change management is simplified if active tool support is available as shown in fig 1.27.

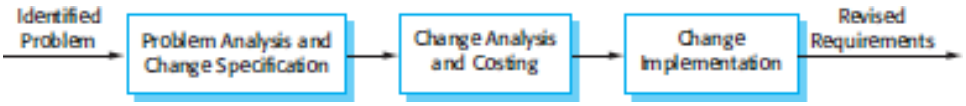

Fig 1.27: Requirements Change Management

     c. ***Traceability Management:*** Tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.

→ There are three principal stages to a change management process:

1. **Problem Analysis and Change Specification**:
   - The process starts with an identified requirements problem or, sometimes, with a specific change proposal.
   - During this stage, the problem or the change proposal is analyzed to check that it is valid.

- This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.

2. **Change Analysis and Costing:**
   - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements.
3. **Change Implementation:**
   - The requirements document and, where necessary, the system design and implementation, are modified.
   - Requirements document will have to be organized so that changes can be made to it without extensive rewriting or reorganization.