DEPT OF ISE, RNSIT

SOFTWARE ENGINEERING

SUBJECT CODE: 15CS42

NOTES

By

Prof. T S Bhagavath Singh & Harshitha K Raj

SOFTWARE ENGINEERING

[As per Choice Based Credit System (CBCS) scheme]
(Effective from the academic year 2016 -2017)
SEMESTER – IV

Subject Code	15CS42	IA Marks	20
Number of Lecture Hours/Week	04	Exam Marks	80
Total Number of Lecture Hours	50	Exam Hours	03

CREDITS – 04

Course objectives: This course will enable students to

- Outline software engineering principles and activities involved in building large software programs.
- Identify ethical and professional issues and explain why they are of concern to software engineers.
- Describe the process of requirements gathering, requirements classification, requirements specification and requirements validation.
- Differentiate system models, use UML diagrams and apply design patterns.
- Discuss the distinctions between validation testing and defect testing.
- Recognize the importance of software maintenance and describe the intricacies involved in software evolution.
- Apply estimation techniques, schedule project activities and compute pricing.
- Identify software quality parameters and quantify software using measurements and metrics.
- List software quality standards and outline the practices involved.
- Recognize the need for agile software development, describe agile methods, apply agile practices and plan for agility.

Module 1	**Teaching Hours**
Introduction: Software Crisis, Need for Software Engineering. Professional Software Development, Software Engineering Ethics. Case Studies. **Software Processes**: Models: Waterfall Model **(Sec 2.1.1)**, Incremental Model **(Sec 2.1.2)** and Spiral Model **(Sec 2.1.3)**. Process activities. **Requirements Engineering**: Requirements Engineering Processes **(Chap 4)**. Requirements Elicitation and Analysis **(Sec 4.5).** Functional and non-functional requirements **(Sec 4.1)**. The software Requirements Document **(Sec 4.2)**. Requirements Specification **(Sec 4.3)**. Requirements validation **(Sec 4.6)**. Requirements Management **(Sec 4.7)**.	**12 Hours**
Module 2	
System Models: Context models **(Sec 5.1)**. Interaction models **(Sec 5.2)**. Structural models **(Sec 5.3)**. Behavioral models **(Sec 5.4)**. Model-driven engineering **(Sec 5.5)**. **Design and Implementation**: Introduction to RUP **(Sec 2.4)**, Design Principles **(Chap 17)**. Object-oriented design using the UML **(Sec 7.1)**. Design patterns **(Sec 7.2)**. Implementation issues **(Sec 7.3)**. Open source development **(Sec 7.4)**.	**11 Hours**
Module 3	
Software Testing: Development testing **(Sec 8.1)**, Test-driven development **(Sec 8.2)**, Release testing **(Sec 8.3)**, User testing **(Sec 8.4)**. Test Automation **(Page no 42, 70,212, 231,444,695)**. **Software Evolution**: Evolution processes **(Sec 9.1)**. Program evolution dynamics **(Sec**	**9 Hours**

9.2). Software maintenance **(Sec 9.3)**. Legacy system management **(Sec 9.4)**.	
Module 4	
Project Planning: Software pricing **(Sec 23.1)**. Plan-driven development **(Sec 23.2)**. Project scheduling **(Sec 23.3)**: Estimation techniques **(Sec 23.5)**. **Quality management**: Software quality **(Sec 24.1)**. Reviews and inspections **(Sec 24.3)**. Software measurement and metrics **(Sec 24.4)**. Software standards **(Sec 24.2)**	**10 Hours**
Module 5	
Agile Software Development: Coping with Change **(Sec 2.3)**, The Agile Manifesto: Values and Principles. Agile methods: SCRUM (Ref "**The SCRUM Primer, Ver 2.0**") and Extreme Programming **(Sec 3.3)**. Plan-driven and agile development **(Sec 3.2)**. Agile project management **(Sec 3.4),** Scaling agile methods **(Sec 3.5)**:	**8 Hours**

Course Outcomes: After studying this course, students will be able to:

- Design a software system, component, or process to meet desired needs within realistic constraints.
- Assess professional and ethical responsibility
- Function on multi-disciplinary teams
- Use the techniques, skills, and modern engineering tools necessary for engineering practice
- Analyze, design, implement, verify, validate, implement, apply, and maintain software systems or parts of software systems.

Graduate Attributes

- Project Management and Finance
- Conduct Investigations of Complex Problems
- Modern Tool Usage
- Ethics

Question paper pattern:

The question paper will have ten questions.
There will be 2 questions from each module.
Each question will have questions covering all the topics under a module.
The students will have to answer 5 full questions, selecting one full question from each module.

Text Books:

1. Ian Sommerville: Software Engineering, 9th Edition, Pearson Education, 2012.
(Listed topics only from Chapters 1,2,3,4, 5, 7, 8, 9, 23, and 24)
2. The SCRUM Primer, Ver 2.0, http://www.goodagile.com/scrumprimer/scrumprimer20.pdf

Reference Books:

1. Roger S. Pressman: Software Engineering-A Practitioners approach, 7th Edition, Tata McGraw Hill.
2. Pankaj Jalote: An Integrated Approach to Software Engineering, Wiley India

Web Reference for eBooks on Agile:

1. http://agilemanifesto.org/
2. http://www.jamesshore.com/Agile-Book/

Module 1

Introduction: Software Crisis, Need for Software Engineering. Professional Software Development, Software Engineering Ethics, Case Studies.
Software Processes: Models: Waterfall Model, Incremental Model and Spiral Model, Process activities.
Requirements Engineering: Requirements Engineering Processes, Requirements Elicitation and Analysis**,** Functional and non-functional requirement, The software Requirements Document, Requirements Specification **,** Requirements validation, Requirements Management.

Introduction

1.1 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 insufficient.
- The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

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
- The main cause is that improvements in computing power had outpaced the ability of programmers to effectively utilize those capabilities.

It is believed that the only satisfactory solution to the present software crisis can possibly come from a spread of software engineering practices among the engineers, coupled with further advancements in the software engineering discipline itself.

1.2 Need for Software engineering

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

- **Large software -** It is easier to build a wall than to a house or building, likewise, as the size of software become 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 one.
- **Cost-** As hardware industry has shown its skills and huge manufacturing has lower down he 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.

- **Software systems are abstract and intangible.** They are not constrained by the properties of materials, governed by physical laws, or by manufacturing processes.
- This simplifies software engineering, as there are no natural limits to the potential of software.
- There are many different types of software systems, from simple embedded systems to complex, worldwide information systems. It is pointless to look for universal notations, methods, or techniques for software engineering because different types of software require different approaches.
- There are still many reports of software projects **going wrong and 'software failures'**.
- Software engineering is criticized as inadequate for modern software development.

Many of these so-called software failures are a consequence of two factors:

1. ***Increasing demands*** As new software engineering techniques help us to build larger, more complex systems, the demands change. Systems have to be built and delivered more quickly; larger, even more complex systems are required; systems have to have new capabilities that were previously thought to be impossible.
2. ***Low expectations*** It is relatively easy to write computer programs without using software engineering methods and techniques. Many companies have drifted into software development as their products and services have evolved. They do not use software engineering methods in their everyday work. Consequently, their software is often more expensive and less reliable than it should be.

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.

Figure 1.1: Frequently asked questions about software engineering

1.3 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.

- Many people think that software is simply another word for computer programs. when we are talking about software engineering
- **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.

- One of the important differences between professional and a amateur software development. If you are writing a program for yourself, no one else will use it and you don't have to worry about writing program guides, documenting the program design, etc. If you are writing software that other people will use and other engineers will change then you usually have to provide additional information as well as the code of the program.

Types of Software products

- **Generic products**-Stand-alone systems that are marketed and sold to any customer who wishes to buy them.
  Examples – PC software such as graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists.

- **Customized products** -Software that is commissioned by a specific customer to meet their own needs.
  Examples – Attendance management system for colleges, embedded control systems, air traffic control software and traffic monitoring systems.

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.

Figure 1.2: Essential attributes of good software

The specific set of attributes that you might expect from a software system obviously depends on its application. Therefore, a banking system must be secure, an interactive game must be responsive, a telephone switching system must be reliable, and so on.

1.3.1 Software engineering

Software engineering is an **engineering discipline** that is 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, 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. Engineers also recognize that they must work to organizational and financial constraints so they look for solutions within these constraints.
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.

Importance of software engineering

1. 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**.
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. For most types of system, the majority of costs are the costs of changing the software after it has gone into use.

Software process activities

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

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. Some knowledge of computer science is essential for software engineers in the same way that some knowledge of physics is essential for electrical engineers.
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.

There are Three General issues that affect most software

1. **Heterogeneity**
   Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices.
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.

1.3.2 Software engineering diversity

- There are many different types of software system and there is no universal set of software techniques that is applicable to all of these.
- The software engineering methods and tools used depend on the type of application being developed, the requirements of the customer and the background of the development team.

Application types

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
Interactive transaction-based applications	Applications that execute on a remote computer and are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications.
Embedded control systems	These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system.
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.

Entertainment systems	These are systems that are primarily for personal use and which are intended to entertain the user.
Systems for modelling and simulation	These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects.
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.
Systems of systems	These are systems that are composed of a number of other software systems.

Software engineering fundamentals that apply to all types of software system,

1. 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.

2. **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.

3**. Understanding and managing the software specification and requirements** (what the software should do) are important.

4. **Reuse software that** has already been developed rather than write new software.

1.3.3 Software engineering and the web

- The Web is now a platform for running application and organizations are increasingly developing web-based systems rather than local systems.
- Web services allow application functionality to be accessed over the web.
- Cloud computing is an approach to the provision of computer services where applications run remotely on the 'cloud'.
    - Users do not buy software buy pay according to use.

Web software engineering

- **Software reuse** is 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.
- **Web-based systems** should be developed and delivered incrementally.

- It is now generally recognized that it is impractical to specify all the requirements for such systems in advance.

- **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.

1.4 Software engineering ethics

- Software engineering involves wider responsibilities than simply the application of technical skills.
- Software engineers must behave in an honest and ethically responsible way if they are to be respected as professionals.
- Ethical behaviour is more than simply upholding the law but involves following a set of principles that are morally correct.

➢ **Issues of professional responsibility**

Confidentiality
- Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.

Competence
- Engineers should not misrepresent their level of competence. They should not knowingly accept work which is out with their competence**.**

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

Computer misuse
- Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

ACM/IEEE Code of Ethics

The professional societies in the US have cooperated to produce a code of ethical practice.

✓ Members of these organisations sign up to the code of practice when they join.

✓ The Code contains eight Principles related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession.

The ACM/IEEE Code of Ethics

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

Ethical dilemmas

- Disagreement in principle with the policies of senior management.
- Your employer acts in an unethical way and releases a safety-critical system without finishing the testing of the system.
- Participation in the development of military weapons systems or nuclear systems.

1.5 Case studies

- A personal insulin pump
  - An embedded system in an insulin pump used by diabetics to maintain blood glucose control.
- A mental health case patient management system
  - A system used to maintain records of people receiving care for mental health problems.
- A wilderness weather station
  - A data collection system that collects data about weather conditions in remote areas.

Case study 1: Insulin pump control system

- **An insulin pump is a medical system** that simulates the operation of the pancreas (an internal organ). 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.**
- People who suffer from diabetes use the system. 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.**
- The conventional **treatment of diabetes involves regular injections** of genetically engineered insulin. Diabetics measure their blood sugar levels using an external meter and then calculate the dose of insulin that they should inject.
- The problem with this treatment is that the **level of insulin required does not just depend on the blood glucose** level but also on the time of the last insulin injection.
- This can lead to very low levels of blood glucose (if there is too much insulin) or very high levels of blood sugar (if there is too little insulin).
- **Low blood glucose is,** in the short term, a more serious condition as it can result in temporary **brain malfunctioning and, ultimately, unconsciousness and death.**
- **High blood glucose** can lead to **eye damage, kidney damage, and heart problems.**
- These systems monitor blood sugar levels and deliver an appropriate dose of insulin when required. Insulin delivery systems like this already exist for the treatment of hospital patients.
- **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. Figure 1.3 shows the hardware components and organization of the insulin pump.

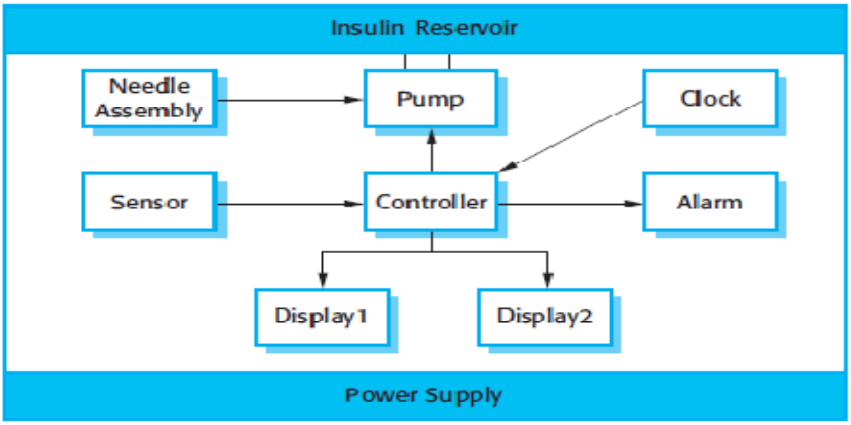


Figure 1.3: Insulin pump hardware architecture

- The insulin pump delivers one unit of insulin in response to a single pulse from a controller.
- Therefore, to deliver 10 units of insulin, the controller sends 10 pulses to the pump.
- Figure 1.4 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. Clearly, this is a safety-critical system.

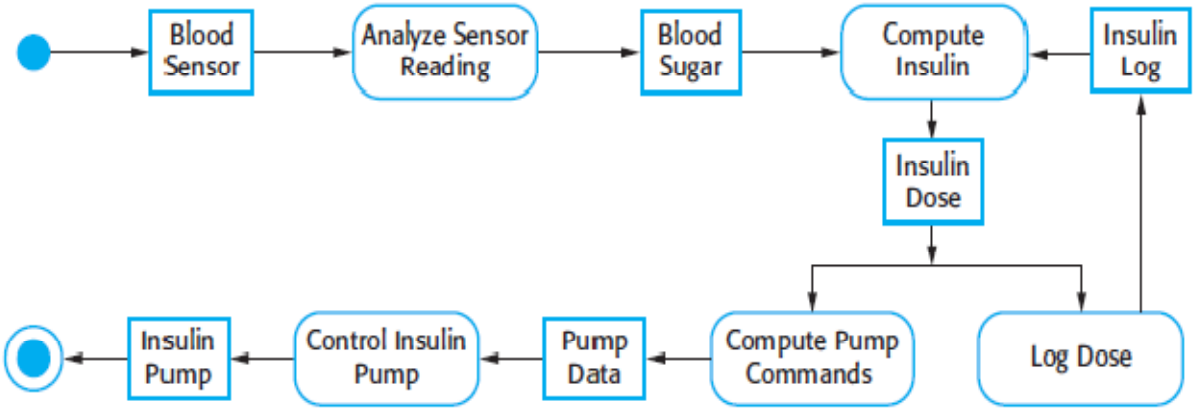


Figure 1.4: Activity model of the insulin pump

- If the pump fails to operate or does not operate correctly, then the user's health may be damaged or they may fall into a coma because their blood sugar levels are too high or too low.
- There are, therefore, two essential high-level requirements that this system must meet:
  - The system shall be available to deliver insulin when required.

Essential high-level requirements

- 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.
- The system must therefore be designed and implemented to ensure that the system always meets these requirements.

Case study 2: 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 overall system is as shown in figure 1.5.
- 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 centres.

MHC-PMS

- **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.

MHC-PMS 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.

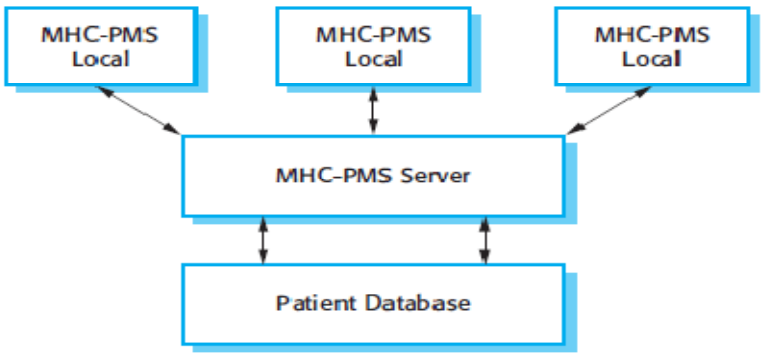


Figure 1.5: The organization of the MHC-PMS

MHC-PMS key features

- **Individual care management**
  - Clinicians can create records for patients, edit the information in the system, view patient history, etc. The system supports data summaries so that doctors can quickly learn about the key problems and treatments that have been prescribed.

- **Patient monitoring**
  - The system monitors the records of patients that are involved in treatment and issues warnings if possible problems are detected.
- **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.

MHC-PMS concerns

- **Privacy**
  - It is essential that patient information is confidential and is never disclosed to anyone apart from authorised medical staff and the patient themselves.
- **Safety**
  - Some mental illnesses cause patients to become suicidal or a danger to other people. Wherever possible, the system should warn medical staff about potentially suicidal or dangerous patients.
  - The system must be available when needed otherwise safety may be compromised and it may be impossible to prescribe the correct medication to patients.

Case study 3: Wilderness weather station

- The government of a country with **large areas of wilderness decides** to deploy several hundred **weather stations in remote areas.**
- Weather stations collect data from a set of instruments that measure **temperature and pressure, sunshine, rainfall, wind speed and wind direction.**
- The weather station includes a **number of instruments** that measure **weather parameters** such as the **wind speed and direction, the ground and air temperatures, the barometric pressure and the rainfall** over a 24-hour period. Each of these instruments is controlled by a software system that takes parameter readings periodically and manages the data collected from the instruments.

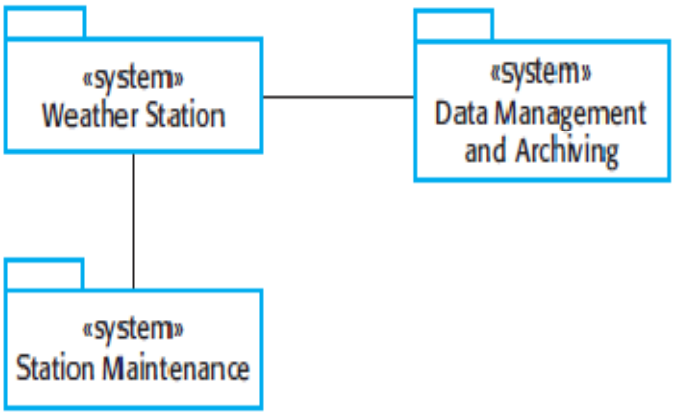


Figure 1.6: The weather station's environment

Weather information system

- **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.
- **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.
- **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.

2. Software process

- A structured set of activities required to develop a software system.
- Many different software processes but all involve:
  - **Specification** – defining what the system should do;
  - **Design and implementation** – defining the organization of the system and implementing the system;
  - **Validation** – checking that it does what the customer wants;
  - **Evolution** – changing the system in response to changing customer needs.
- A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc., and the ordering of these activities. However, as well as activities, process descriptions may also include:

i. **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.
ii. **Roles,** which reflect the responsibilities of the people involved in the process. Examples of roles are project manager, configuration manager, programmer, etc.
iii. **Pre- and post-conditions,** which are statements that are true before and after a process activity has been enacted or a product produced.

Software processes are complex and, like all intellectual and creative processes, rely on people making decisions and judgments. There is no ideal process and most organizations have developed their own software development processes.

2.1 Software process models

- **The waterfall model**
  - Plan-driven model. Separate and distinct phases of specification and development.
- **Incremental development**
  - Specification, development and validation are interleaved. May be plan-driven or agile.
- **Reuse-oriented software engineering**
  - The system is assembled from existing components. May be plan-driven or agile.
- **Spiral model**
  - Process is represented as a spiral rather than as a sequence of activities with backtracking.

In practice, most large systems are developed using a process that incorporates elements from all of these models.

2.1.1 The waterfall model or Software Development life cycle

Cascade from one phase to another, this model is known as the **waterfall model or software life cycle.** The principal stages of the model map onto 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 system design process partitions the requirements to either **hardware or software systems.**
  - **Software design** involves **identifying and describing** the fundamental **software system abstractions and their relationships.**

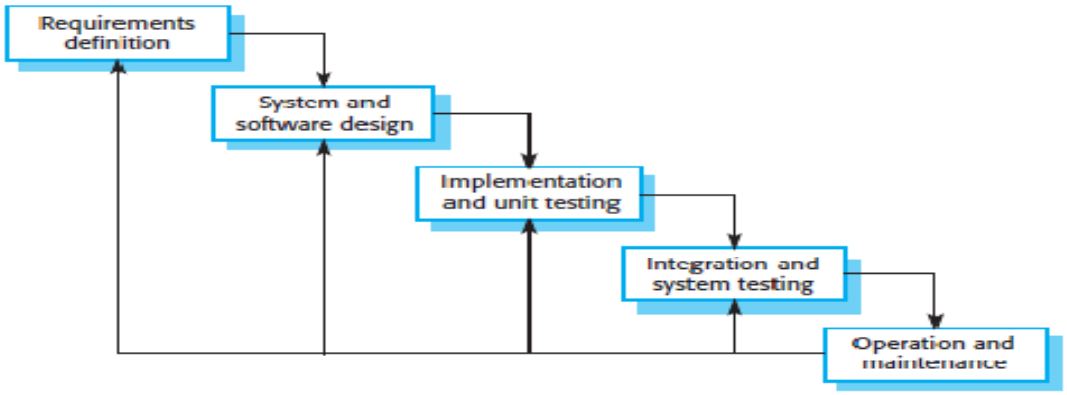


Figure 2.1: The waterfall model

- **Implementation and unit testing**
  - 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**
  - 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 result of each phase is **one or more documents** that are approved .The following **phase should not start** until the **previous phase has finished**. In practice, these stages **overlap and feed information** to each other.

During the **final life-cycle phase (operation and maintenance)**, the software is put into use. Errors and omissions in the original software requirements are discovered. Program and design errors emerge and the need for new functionality is identified. The system must therefore evolve to remain useful. Making these changes (software maintenance) may involve repeating previous process stages.

- **Advantages:**

1. The documentation is produced at each phase and that it fits with other engineering process models.
2. Easy to manage due to the rigidity of the model – each phase has **specific deliverables and a review process.**
3. Phases are processed and completed one at a time.
4. Works well for smaller projects where requirements are very well understood.

- **Disadvantages:**

1. Its **inflexible partitioning** of the project into distinct stages.
2. **Premature freezing of requirements** may mean that the system won't do what the user wants. It may also lead to badly structured systems as design problems are circumvented by implementation tricks.
3. **Commitments must be made at an early stage in the process**, which makes it difficult to respond to changing customer requirements.
4. Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage.

2.1.2 Incremental development

Incremental development is based on the idea of **developing an initial implementation**, exposing this to **user comment** and **refining** it through **many versions** until an adequate system has been developed (Figure 2.2). **Specification, development and validation activities are interleaved** rather than separate, with rapid feedback across activities.

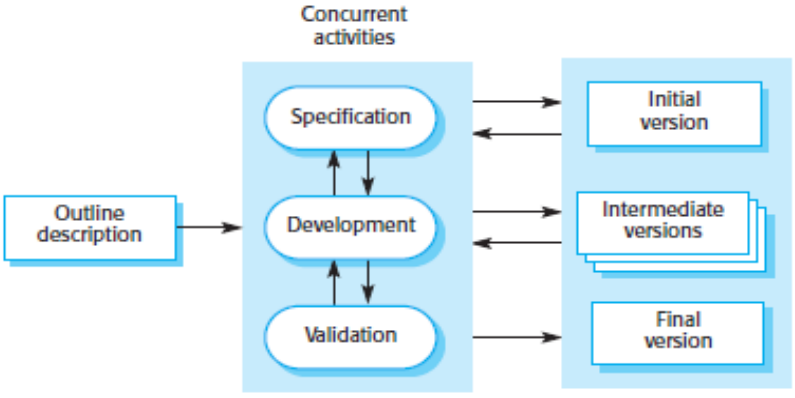


Figure 2.2: Incremental development

➢ **Incremental development benefits**

- 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.
- More **rapid delivery and deployment of useful software** to the customer is possible.
  - Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

➢ **Incremental development 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.

2.1.3 Reuse Oriented Software Engineering

- In the majority of software projects, there is some software reuse. This informal reuse takes place irrespective of the development process that is used.
- Reuse-oriented approaches rely on a large base of reusable software components and an integrating framework for the composition of these components.

A general process model for reuse-based development is shown in Figure 2.3.

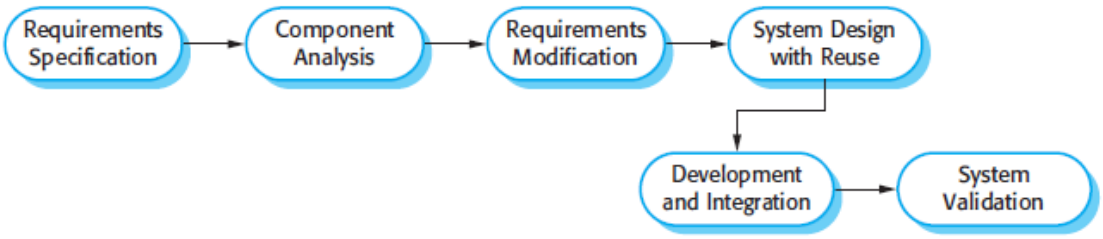


Figure 2.3: Reuse –Oriented Software Engineering

Although the initial requirements specification stage and the validation stage are comparable with other software processes, the intermediate stages in a reuse oriented process are different. These stages are:

1. ***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.

2. ***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.
3. ***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.
4. ***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:

i. Web services that are developed according to service standards and which are available for remote invocation.
ii. Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
iii. Stand-alone software systems that are configured for use in a particular environment.

Advantages:

- Reducing the amount of software to be developed and so reducing cost and risks.
- It also leads to faster delivery of the software.
- Requirements compromises are inevitable and this may lead to a system that does not meet the real needs of users.

2.1.4 Spiral development

The **spiral model of the software process** (Figure 2.4) was originally proposed by **Boehm**. Rather than represent the software process as a sequence of activities with some backtracking from one activity to another, the process is represented as a spiral.

Each loop in the spiral represents a phase of the software process. Each loop in the spiral is split into four sectors:

1. ***Objective setting***
- Specific objectives for that phase of the project is 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, if user interface risks are dominant, an appropriate development model might be evolutionary prototyping.
- If safety risks are the main consideration, development based on formal transformations may be the most appropriate and so on.
- The waterfall model may be the most appropriate development model if the main identified risk is sub-system integration.

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.

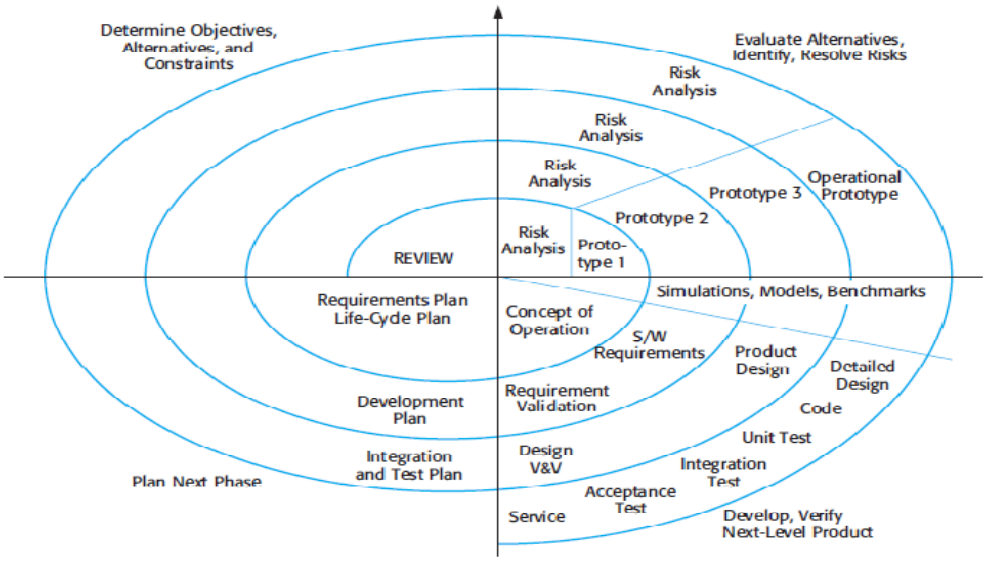


Figure 2.4: Boehm's spiral model of the software process

The main difference between the spiral model and other software process models is the explicit recognition of risk in the spiral model. For example, if the intention is to use a new programming language, a risk is that the available compilers are unreliable or do not produce sufficiently efficient object code.

- ✓ A cycle of the spiral begins by elaborating objectives such as performance and functionality.
- ✓ Alternative ways of achieving these objectives and the constraints imposed on each of them are then enumerated.
- ✓ Each alternative is assessed against each objective and sources of project risk are identified.
- ✓ The next step is to resolve these risks by information-gathering activities such as more detailed analysis, prototyping and simulation.

- ✓ Once risks have been assessed, some development is carried out, followed by a planning activity for the next phase of the process.

Advantages

1. High amount of risk analysis hence, avoidance of Risk is enhanced.
2. Good for large and mission-critical projects.
3. Strong approval and documentation control.
4. Additional Functionality can be added at a later date.
5. Software is produced early in the software life cycle.
6. Project estimates in terms of schedule, cost etc become more and more realistic as the project moves forward and loops in spiral get completed.

Disadvantages

1. Can be a costly model to use.
2. Risk analysis requires highly specific expertise.
3. Project's success is highly dependent on the risk analysis phase.
4. Doesn't work well for smaller projects.
5. It is not suitable for low risk projects.
6. May be hard to define objective, verifiable milestones.
7. Spiral may continue indefinitely.

2.2 Process activities

- Real software processes are interleaved sequences of technical, collaborative, and managerial activities with the overall goal of specifying, designing, implementing, and testing a software system.

There are 4 major activates involved in developing a good software,

1. Software Specification
2. Software Design and Implementation
3. Software Validation
4. Software Evolution

2.2.1 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.

There are four main phases 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.**
   - A feasibility study should be relatively **cheap and quick.**
   - The result should inform the decision of whether to go ahead with a more detailed analysis.

2. ***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**.
3. ***Requirements specification***
   - 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.

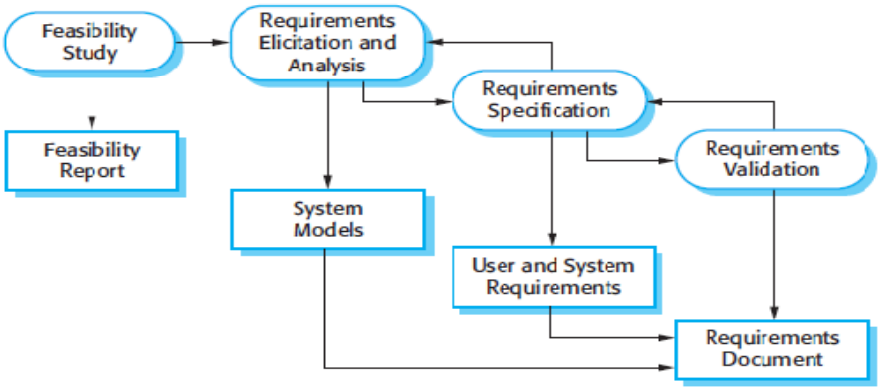


Figure 2.5: The requirements engineering process

2.2.2 Software design and implementation

The implementation stage of software development is the process of **converting a system specification into an executable system.** It always involves processes of software design and programming.

Figure 2.6 is a model of this process showing the design descriptions that may be produced at various stages of design. This diagram suggests that the stages of the design process are sequential. In fact, design process activities are interleaved. Feedback from one stage to another and consequent design rework is inevitable in all design processes.

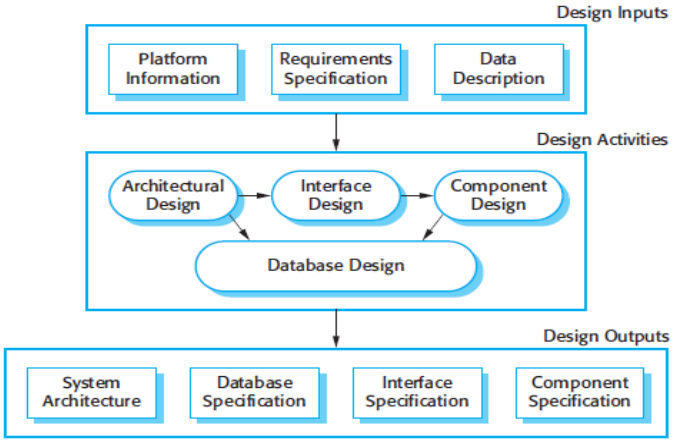


Figure 2.6: A general model of the design process

The specific design process activities are:

1. *Architectural design*
- Architectural design is a creative process where you try to establish a system organization that will satisfy the functional and non-functional system requirements.
- The sub-systems making up the system and their relationships are identified and documented.

2. *Abstract specification*
- For each sub-system, an abstract specification of its services and the constraints under which it must operate is produced.

3. *Interface design*
- For each sub-system, its interface with other sub-systems is designed and documented.
- This interface specification must be unambiguous as it allows the sub-system to be used without knowledge of the sub-system operation.

4. *Component design*
- Services are allocated to components and the interfaces of these components are designed.

5. *Data structure design*
- The data structures used in the system implementation are designed in detail and specified.

Possible adaptations are:

1. The last two stages of design data structure and algorithm design may be delayed until the implementation process.
2. If an exploratory approach to design is used, the system interfaces may be designed after the data structures have been specified.
3. The abstract specification stage may be skipped, although it is usually an essential part of critical systems design.

2.2.3 Software validation

Software validation or more generally, verification and validation (V & V) is intended to show that a system conforms to its specification and that the system meets the expectations of the customer buying the system.
Figure 2.7 shows a three-stage testing process where system components are tested, the integrated system is tested and, finally, the system is tested with the customer's data.

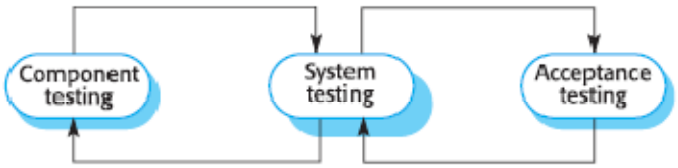


Figure 2.6: The testing process

The stages in the testing process are:

1. Development testing (*unit testing)*

- Individual components are tested to ensure that they operate correctly.
- 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*

- The components are integrated to make up the system.
- This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems. I
- t is also concerned with validating that the system meets its functional and non-functional requirements and testing the emergent system properties.
- For large systems, this may be a multistage process where components are integrated to form sub-systems that are individually tested before they are themselves integrated to form the final system.

3. *Acceptance testing*

- Acceptance testing 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.
- When a plan-driven software process is used (e.g., for critical systems development), testing is driven by a set of test plans.

- Figure 2.8 illustrates how test plans are the link between testing and development activities. This is sometimes called the V-model of development (turn it on its side to see the V).

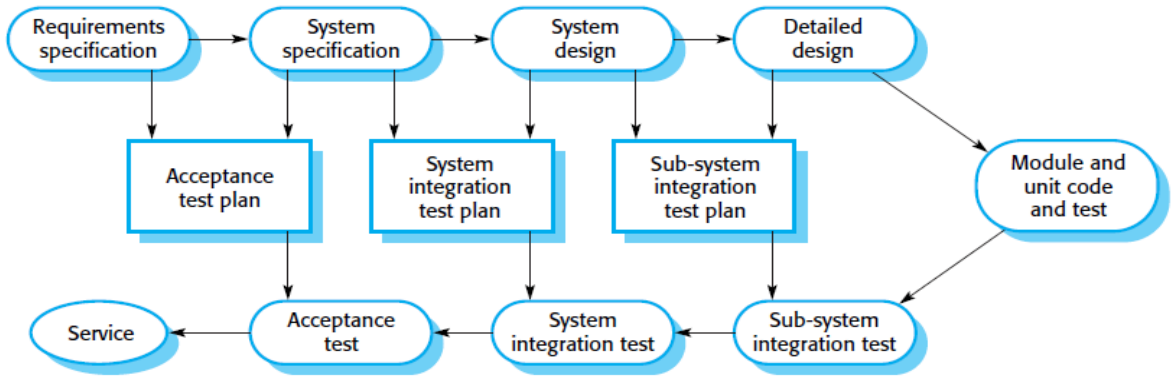


Figure 2.8: Testing phases in the software process

Acceptance testing has two types

1. **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 system requirements.
2. **Beta testing:** When a system is to be **marketed as a software product**, involves delivering a system to a number of **potential customers** who agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors. After this feedback, the system is modified and released either for further beta testing or for general sale.

2.2.4 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.
- Changes can be made to software at any time during or after the system development.
- Few software systems are now completely new systems, and it makes much more sense to see development and maintenance as a continuum.
- Rather than two separate processes, it is more realistic to think of software engineering as an evolutionary process (Figure 2.9) where software is continually changed over its lifetime in response to changing requirements and customer needs.

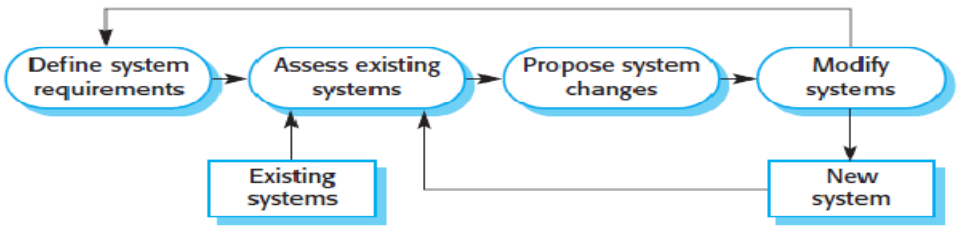


Figure 2.9: System Evolution

Requirements engineering

The requirements for a system are the descriptions of what the system should do— the services that it provides and the constraints on its operation. These **requirements reflect the needs of customers for a system** that serves a certain purpose such as controlling a device, placing an order, or finding information. The process of finding out, analyzing, documenting and checking these services and constraints is called **requirements engineering (RE).**

Types of requirement

- **User requirements**
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.
- **System requirements**
  - A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

The Figure 3.1 shows that the **user requirement is quite general**. The **system requirements provide more specific information about the services and functions of the system** that is to be implemented.

User Requirement Definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.
1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.
1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

Figure 3.1: User and System Requirements

Figure 3.2 shows possible **readers of the user and system requirements**. The readers of the user requirements are not usually concerned with how the system will be implemented and may be managers who are not interested in the detailed facilities of the system.

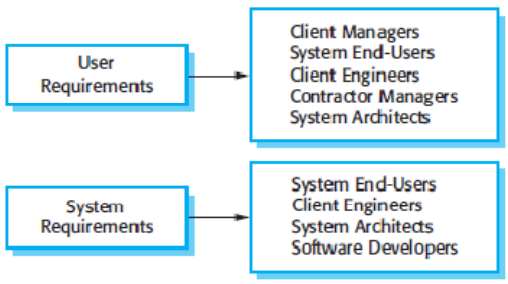


Figure 3.2: Readers of different types of requirements specification

3.1 Requirements engineering processes

- The processes used for requirement engineering vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- In practice, RE is an iterative activity in which these processes are interleaved.
- However, there are a number of generic activities common to all processes
  - Requirements elicitation;
  - Requirements analysis;
  - Requirements validation;
  - Requirements management.
- **Figure 3.3 shows this interleaving**. The activities are organized **as an iterative process around a spiral,** with the output being a system requirements document.
- 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.
- Some people consider requirements engineering to be the process of applying a structured analysis method, such as object-oriented analysis.
- Involves analyzing the system and developing a set of graphical system models, such as use case models, which then serve as a system specification.

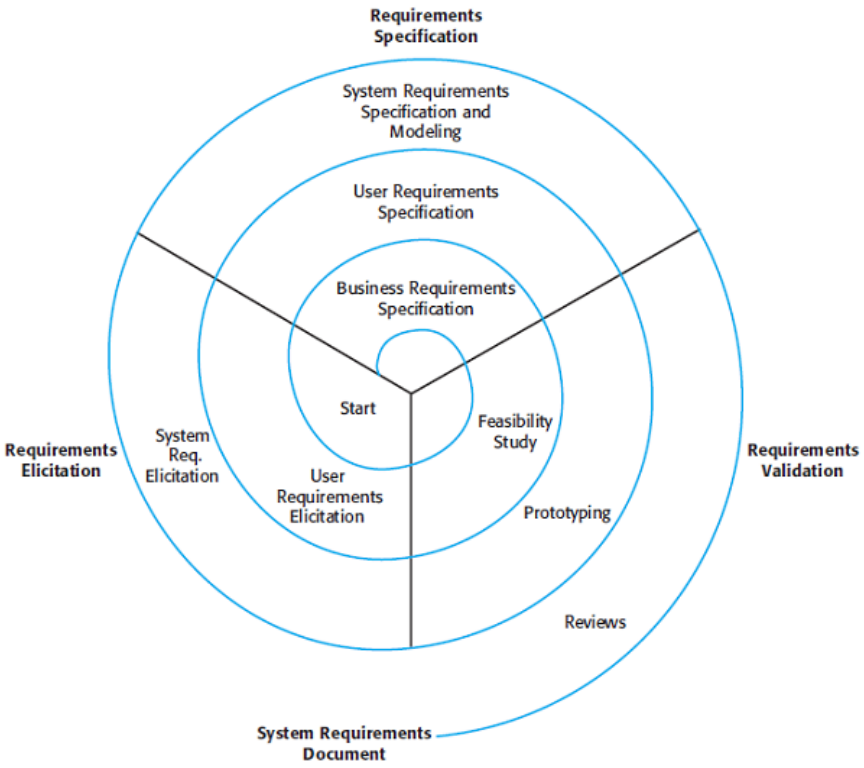


Figure 3.3: A spiral view of the requirements engineering process

3.2 Requirements elicitation and analysis

- Requirements elicitation and analysis may involve a variety of **different kinds of people in an organization.**
- Sometimes **called requirements elicitation or requirements discovery.**
- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called ***stakeholders.***
- 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 is an iterative process including requirements elicitation, specification and validation.
- Requirements elicitation and analysis is an iterative process that can be represented as a spiral of activities – requirements discovery, requirements classification and organization, requirements negotiation and requirements documentation.

Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.

Stages include:

- **Requirements discovery**
  - Interacting with stakeholders to discover their requirements.
  - Domain requirements from stakeholders and documentation are also discovered during this activity.
- **Requirements classification and organisation**
  - Groups related requirements and organises 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.
- **Prioritisation and negotiation**
  - Prioritising requirements and resolving requirements conflicts.
  - This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation.
- **Requirements specification**
  - Requirements are documented and input into the next round of the spiral.
  - Formal or informal requirements documents may be produced.

Problems of requirements elicitation

- Stakeholders don't know what they really want.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have conflicting requirements.
- Organisational and political factors may influence the system requirements.
- The requirements change during the analysis process. New stakeholders may emerge and the business environment change.

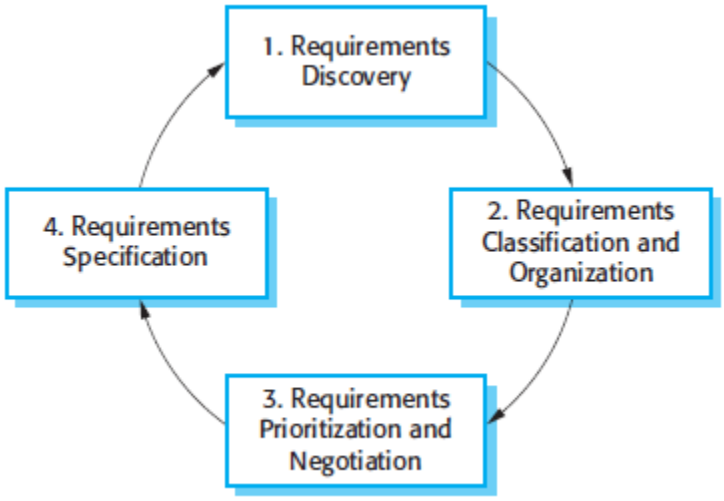


Figure 3.4: The requirements elicitation and analysis process

3.2.1 Requirements discovery

- The process of gathering information about the required and existing systems and distilling the user and system requirements from this information.
- Interaction is with system stakeholders from managers to external regulators.
- Systems normally have a range of stakeholders.
- Stakeholders range from end-users of a system through managers to external stakeholders such as regulators, who certify the acceptability of the system.

EXAMPLE: In MHC-PMS Stakeholders,

- 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.
- Stakeholders in the MHC-PMS
- A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
- Health care managers who obtain management information from the system.
- Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

3.2.2 Interviewing

- Formal or informal interviews with stakeholders are part of most RE processes.

Types of interview

- **Closed interviews** based on pre-determined list of questions
- **Open interviews** where various issues are explored with stakeholders.

Effective interviewing

- Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
- Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.

- Normally a mix of closed and open-ended interviewing.

Interviews are not good for understanding domain requirements

- Requirements engineers cannot understand specific domain terminology;
- Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating.

3.2.3 Scenarios: Scenarios are real-life examples of how a system can be used.

- They can understand and criticize a scenario of how they might interact with a software system.
- Scenarios can be particularly useful for adding detail to an outline requirements description. 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.
- A scenario starts with an outline of the interaction. During the elicitation process, details are added to this to create a complete description of that interaction.
- Scenarios should include
  - A description of the starting situation
  - A description of the normal flow of events
  - A description of what can go wrong
  - Information about other concurrent activities
  - A description of the state when the scenario finishes.
- Scenarios may be written as text, supplemented by diagrams, screen shots, etc.
- The Figure 3.5 consider how the MHC-PMS may be used to enter data for a new patient

INITIAL ASSUMPTION:
The patient has seen a medical receptionist who has created a record in the system and collected the patient's personal information (name, address, age, etc.). A nurse is logged on to the system and is collecting medical history.

NORMAL:
The nurse searches for the patient by family name. If there is more than one patient with the same surname, the given name (first name in English) and date of birth are used to identify the patient.

The nurse chooses the menu option to add medical history.

The nurse then follows a series of prompts from the system to enter information about consultations elsewhere on mental health problems (free text input), existing medical conditions (nurse selects conditions from menu), medication currently taken (selected from menu), allergies (free text), and home life (form).

WHAT CAN GO WRONG:
The patient's record does not exist or cannot be found. The nurse should create a new record and record personal information.

Patient conditions or medication are not entered in the menu. The nurse should choose the 'other' option and enter free text describing the condition/medication.

Patient cannot/will not provide information on medical history. The nurse should enter free text recording the patient's inability/unwillingness to provide information. The system should print the standard exclusion form stating that the lack of information may mean that treatment will be limited or delayed. This should be signed and handed to the patient.

OTHER ACTIVITIES:
Record may be consulted but not edited by other staff while information is being entered.

SYSTEM STATE ON COMPLETION:
User is logged on. The patient record including medical history is entered in the database, a record is added to the system log showing the start and end time of the session and the nurse involved.

Figure 3.5: Scenario for collecting medical history in MHC-PMS

3.2.4 Use cases

- Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.
- 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.** Optionally, arrowheads may be added to lines to show how the interaction is initiated. This is illustrated in Figure 3.6, which shows some of the use cases for the patient information system.
- Use cases identify the individual interactions between the system and its users or other systems. Each use case should be documented with a textual description.
- A set of use cases should describe all possible interactions with the system.

- High-level graphical model supplemented by more detailed tabular description Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.
- Use cases for the MHC-PMS

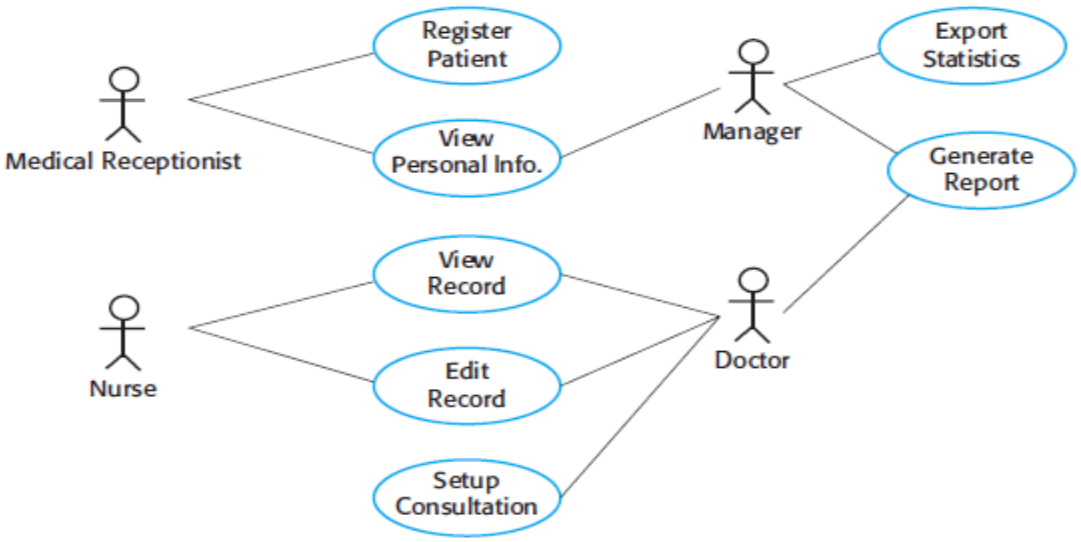


Figure 3.6: Use Case For The MHC-PMS

ACTORS	ROLE
Medical receptionist	Manages the patients register and personal information related to patients
Nurse	Record management related to patients health status.
Manager	Report management about the patient's condition and framing the statistics for analysis.
Doctor	Monitor the patients frame a case sheet and prescriptions.

3.2.5 Ethnography

- **Ethnography is an observational technique** that can be used to understand operational processes and help derive support requirements for these processes.
- An analyst immerses himself or herself in the working environment where the system will be used.
- The day-to-day work is observed and notes made of the actual tasks in which participants are involved.
- A social scientist spends a considerable time observing and analysing how people actually work.
- People do not have to explain or articulate their work.
- Social and organisational factors of importance may be observed.
- Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.

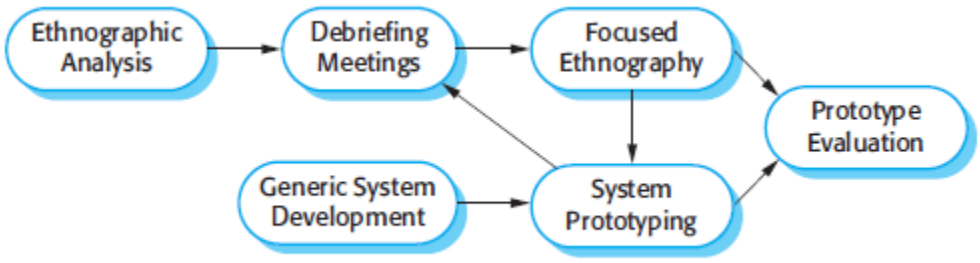


Figure: 3.7: Ethnography And Prototyping For Requirements Analysis

- **Ethnography is particularly effective for discovering two types of requirements:**
  - Requirements that are derived from the way that people actually work rather than the way I which process definitions suggest that they ought to work.
  - Requirements that are derived from cooperation and awareness of other people's activities.
- Awareness of what other people are doing leads to changes in the ways in which we do things.
- Ethnography is effective for understanding existing processes but cannot identify new features that should be added to a system.
- Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques.

3.3 Functional and non-functional requirements

- **Functional requirements**
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  - May state what the system should not do.
- **Non-functional requirements**
  - Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  - Often apply to the system as a whole rather than individual features or services.
- **Domain requirements**
  - Constraints on the system from the domain of operation

3.3.1 Functional requirements

- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do.
- Functional system requirements should describe the system services in detail.
- Requirements, which are related to functional aspect of software fall into this category.
- They define functions and functionality within and from the software system.

Example: Functional requirements for the MHC-PMS

- 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 8-digit employee number.

Imprecision

- Problems arise when requirements are not precisely stated.
- Ambiguous requirements may be interpreted in different ways by developers and users.
- Consider the term 'search' in requirement
  - User intention – search for a patient name across all appointments in all clinics;
  - Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.

Requirements completeness and consistency

- Complete
  - They should include descriptions of all facilities required.
- Consistent
  - There should be no conflicts or contradictions in the descriptions of the system facilities.

4.3.2 Non-functional requirements

- Non-functional requirements, as the name suggests, 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.
- System users can usually find ways to work around a system function that doesn't really meet their needs.

For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly.

- Although it is often possible to identify which system components implement
- Requirements, which are not related to functional aspect of software, fall into this category. They are implicit or expected characteristics of software, which users make assumption of.

There are 2 main reasons:

- Non-functional requirements may affect the overall architecture of a system rather than the individual components. **To ensure that performance requirements are met,** you may have to organize the system to minimize communications between components.
- A **single non-functional requirement, such as a security requirement**, may generate a number of related functional requirements that define system services that are required. It may also generate requirements that restrict existing requirements.

Types/ Classification of non-functional requirements (Figure 3.8)

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.

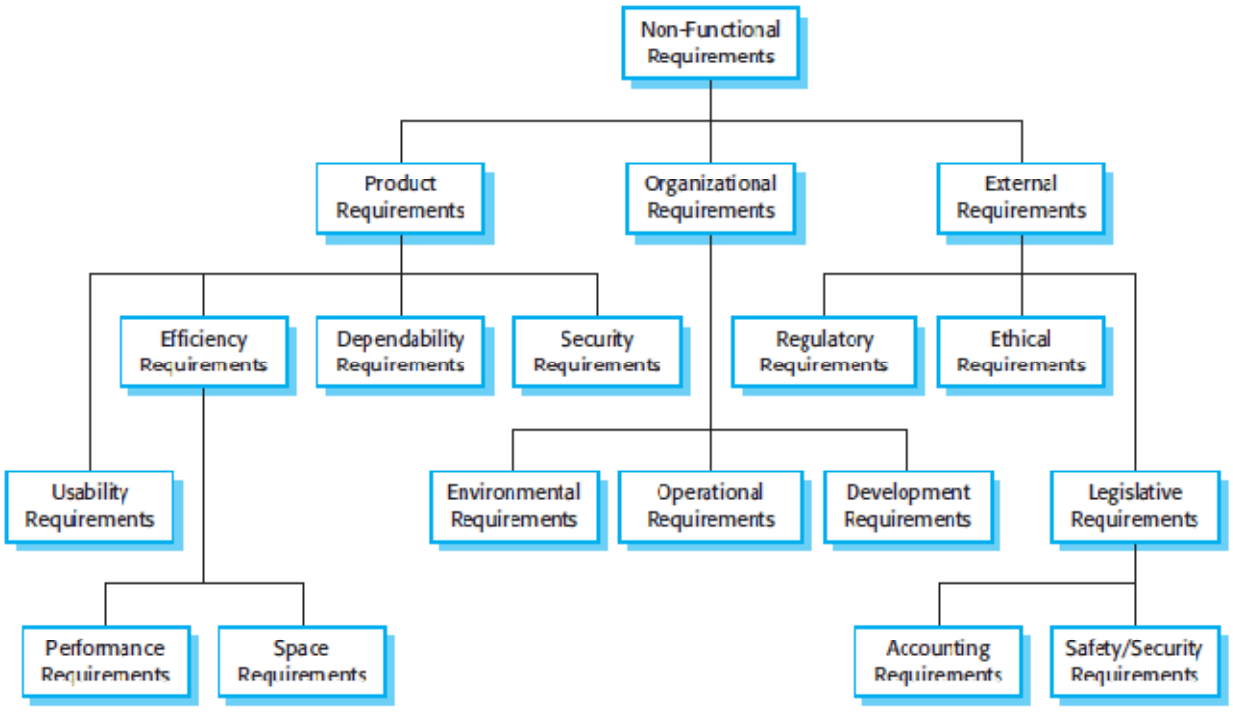


Figure 3.8: Types of non-functional requirements

3. ***External requirements***
   - Requirements that are derived from factors external to the system and its development process.
   - **Regulatory requirements** set out what must be done for the system to be approved for use by a regulator, such as a central bank
   - **Legislative requirements** that must be followed to ensure that the system operates within the law.
   - **Ethical requirements** are requirements placed on a system to ensure that it will be acceptable to its users and the general public.

PRODUCT REQUIREMENT
The MHC-PMS shall be available to all clinics during normal working hours (Mon–Fri, 08.30–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

ORGANIZATIONAL REQUIREMENT
Users of the MHC-PMS system shall authenticate themselves using their health authority identity card.

EXTERNAL REQUIREMENT
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Figure 3.9: Examples of nonfunctional requirements in the MHC-PMS

- Figure 3.9 shows examples of product, organizational, and external requirements taken from the MHC-PMS.
- **The product requirement** is an availability requirement that defines when the system has to be available and the allowed down time each day.
- It says nothing about the functionality of MHC-PMS and clearly identifies a constraint that has to be considered by the system designers.
- **The organizational requirement specifies** how users authenticate themselves to the system.
- **The external requirement is derived** from the need for the system to conform to privacy legislation. Privacy is obviously a very important issue in healthcare systems and the requirement specifies that the system should be developed in accordance with a national privacy standard.
- Figure 3.10 shows metrics that you can use to specify non-functional system properties.
- You can measure these characteristics when the system is being tested to check whether or not the system has met its nonfunctional requirements.

Goals and requirements

- Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.
- Goal
  - A general intention of the user such as ease of use.
- Verifiable non-functional requirement
  - A statement using some measure that can be objectively tested.
- Goals are helpful to developers as they convey the intentions of the system users.

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

Figure 3.10: Metrics for specifying nonfunctional requirements

Usability requirements

- The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized. (Goal)
- Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use. (Testable non-functional requirement)

Domain requirements

- The system's operational domain imposes requirements on the system.
  - For example, a train control system has to take into account the braking characteristics in different weather conditions.
- Domain requirements be new functional requirements, constraints on existing requirements or define specific computations.
- If domain requirements are not satisfied, the system may be unworkable.

Train protection system

- This is a domain requirement for a train protection system:
- The deceleration of the train shall be computed as:
  - Dtrain = Dcontrol + Dgradient
  - where Dgradient is 9.81ms2 * compensated gradient/alpha and where the values of 9.81ms2 /alpha are known for different types of train.
- It is difficult for a non-specialist to understand the implications of this and how it interacts with other requirements.

Domain requirements problems

- **Understandability**
  - Requirements are expressed in the language of the application domain;
  - This is often not understood by software engineers developing the system.
- **Implicitness**
  - Domain specialists understand the area so well that they do not think of making the domain requirements explicit.

3.4 The software requirements document (SRS- System requirements specification)

- The **software requirements document** is the official statement of what is required of the system developers.
- Should include both a **definition of user requirements and a 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 figure 3.11 shows possible users of the document and how they use it.

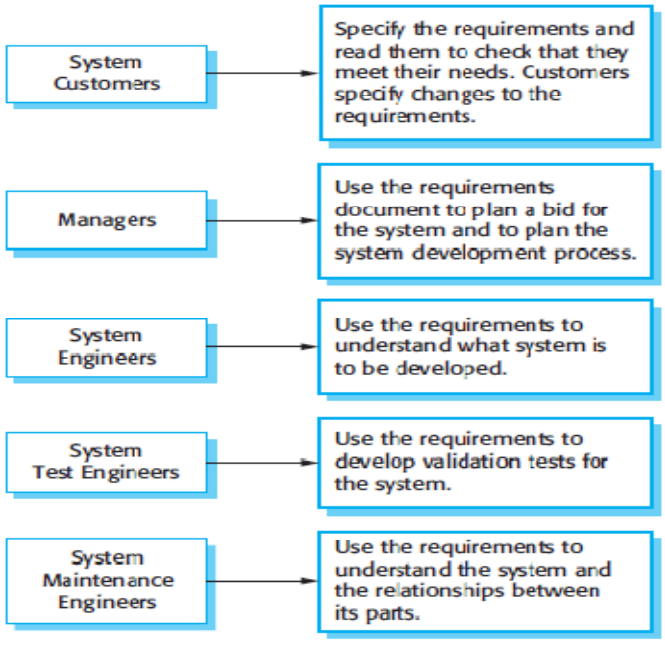


Figure 3.11: Users of a requirements document

- The diversity of possible users means that the requirements document has to be a compromise between communicating the requirements to customers, defining the

requirements in precise detail for developers and testers including information about possible system evolution.

- The detail that you should include in a requirements document depends on the type of system that is being developed and the development process used.
- Critical systems need to have detailed requirements because safety and security have to be analyzed in detail.
- Figure 3.12 shows one possible organization for a requirements document that is based on an IEEE standard for requirements documents (IEEE, 1998). This standard is a generic standard that can be adapted to specific uses.
- This information helps the maintainers of the system and allows designers to include support for future system features.
- The focus will be on defining the user requirements and high-level, non-functional system requirements.

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.

Figure 3.12: The structure of a requirements document

3.5 Requirements specification

- Requirements specification is the **process of writing down the user and system requirements** in a requirements document.
- The **user and system requirements should be clear, unambiguous, easy to understand,** complete, and consistent.
- The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge.
- The requirements document **should not include details of the system architecture or design.** Consequently, if you are writing user requirements, you should not use software jargon, structured notations, or formal notations.
- User requirements should be **written in natural language**, with simple tables, forms, and intuitive diagrams.
- System requirements may also be **written in natural language** but other notations based **on forms, graphical system models, or mathematical system models** can also be used.
- 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.
- They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system.
- The system requirements should simply describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented.
- Figure 3.13 summarizes the possible notations that could be used for writing system requirements.

There are several reasons

1. 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.
2. In most cases, systems **must interoperate with existing systems**, which constrain the design and impose requirements on the new system.
3. The use of a specific architecture to satisfy non-functional requirements may be necessary. An external regulator who needs to certify that the system is safe may specify that an already certified architectural design be used.

- In principle, requirements should state what the system should do and the design should describe how it does this.
- In practice, requirements and design are inseparable
    - A system architecture may be designed to structure the requirements;
    - The system may inter-operate with other systems that generate design requirements;
    - The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.

- This may be the consequence of a regulatory requirement.

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.

Figure 3.13: Ways of writing system requirements specification

3.5.1 Natural language specification

- Natural language has been used to write requirements for software since the beginning of software engineering.
- It is expressive, intuitive, and universal.
- It is also potentially vague, ambiguous, and its meaning depends on the background of the reader.

Guidelines for writing requirements

1. Invent a standard format and use it for all requirements. Standardizing the format makes omissions less likely and requirements easier to check.
2. Use language in a consistent way. Use shall distinguish between mandatory requirements and desirable requirements.
3. Use text highlighting to identify key parts of the requirement.
4. Do not assume that readers understand technical software engineering language. Therefore avoid the use of jargon, abbreviations, and acronyms.
5. Whenever possible, you should try to associate a rationale with each user requirement. The rationale should explain why the requirement has been included.

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.*)

Figure 3.14: Example requirements for the insulin pump software system

Problems with natural language

- **Lack of clarity**
  - Precision is difficult without making the document difficult to read.
- **Requirements confusion**
  - Functional and non-functional requirements tend to be mixed-up.
- **Requirements amalgamation**
  - Several different requirements may be expressed together.

3.5.2 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.
- This approach maintains most of the **expressiveness and understandability of natural language** but ensures that some uniformity is imposed on the specification.
- Structured language notations use **templates to specify system requirements**.
- The specification may use programming language constructs to show alternatives and iteration, and may highlight key elements using shading or different fonts.
- This is similar to the approach used in the example of a structured specification shown in Figure 3.15

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.

Figure 3.15: A structured specification of a requirement for an insulin pump

Form-based specifications

- Description of the function or entity.
- Description of inputs and where they come from.
- Description of outputs and where they go to.
- Information about the information needed for the computation and other entities used.
- Description of the action to be taken.
- Pre and post conditions (if appropriate).
- The side effects (if any) of the function.

- Using structured specifications removes some of the problems of natural language specification.
- Variability in the specification is reduced and requirements are organized more effectively.

Tabular specification

- Used to supplement natural language.
- Particularly useful when you have to define a number of possible alternative courses of action.
- For example, the insulin pump systems bases its computations on the rate of change of blood sugar level and the tabular specification explains how to calculate the insulin requirement for different scenarios.

Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 − r1) < (r1 − r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing ((r2 − r1) ≥ (r1 − r0))	CompDose = round ((r2 − r1)/4) If rounded result = 0 then CompDose = MinimumDose

Figure 3.16: Tabular specification of computation for an insulin pump

3.6 Requirements validation

- Concerned with demonstrating that the requirements define the system that the customer really wants.
- Requirements error costs are high so validation is very important
  - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements check

1. **Validity Checks**: Does the system provide the functions which best support the customer's needs
2. **Consistency Checks**: Are there any requirements conflicts
3. **Completeness Checks:** Are all functions required by the customer included
4. **Realism Checks**: Can the requirements be implemented given available budget and technology
5. **Verifiability:** To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable.

- Regular reviews should be held while the requirements definition is being formulated. Both client and contractor staff should be involved in reviews.
- Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.

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. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

3.8 Requirements management

- Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- New requirements emerge as a system is being developed and after it has gone into use.
- You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You need to establish a formal process for making change proposals and linking these to system requirements.
- During the software process, the stakeholders' understanding of the problem is constantly changing (Figure 3.17).

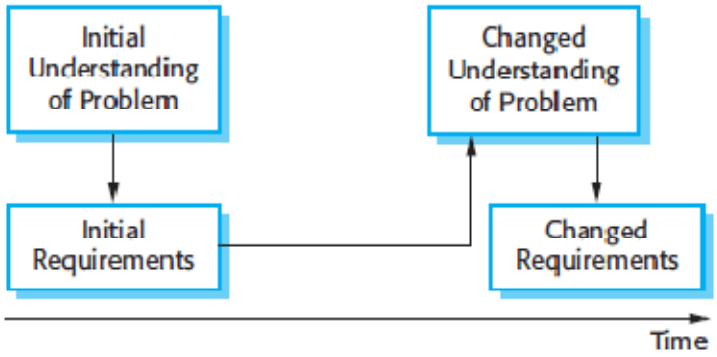


Figure 3.17: Requirements Evolution

Changing requirements

- **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 (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.
- **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.**
  - The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

3.8.1 Requirements management planning

Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.

1. ***Requirements identification*** Each requirement must be uniquely identified so that it can be cross-referenced with other requirements.
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.
4. ***Tool support*** *Tools* that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

- Requirements management needs automated support and the software tools for this should be chosen during the planning phase. You need tool support for:

  1. Requirements storage The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.

  2. *Change management* The process of change management (Figure 3.18) is simplified if active tool support is available.

  3. *Traceability management* As discussed above, 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.

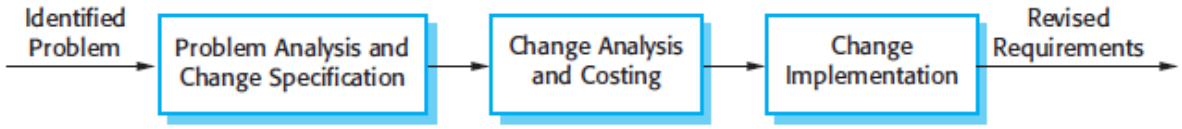


Figure 3.18: Requirements change Management

3.8.2 Requirements change management

Deciding if a requirements change should be accepted

1) **Problem analysis and change specification:** 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. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.

3) Change implementation: The requirements document and, where necessary, the system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented.

MODULE- 2

System Models: Context models, Interaction models, Structural models, Behavioral models Model-driven engineering.

Design and Implementation: Introduction to RUP, Design Principles, Object-oriented design using the UML, Design patterns, Implementation issues, Open source development.

SYSTEM MODELS

1. System Modeling

- **Definition:** 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 now come to mean representing a system using some kind of graphical notation, which is now almost always based on notations in the **Unified Modeling Language (UML).**
- System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.
- Models are used during the requirements engineering process to help derive the requirements for a system, during the design process to describe the system to engineers implementing the system and after implementation to document the system's structure and operation.

❖ **You may develop models of both the existing system and the system to be developed:**

1. Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.
2. Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation.

- In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.

❖ **System Perspectives:**

- **An external perspective**, where you model the context or environment of the system.
- **An interaction perspective**, where you model the interactions between a system and its environment, or between the components of a system.
- **A structural perspective**, where you model the organization of a system or the structure of the data that is processed by the system.
- **A behavioral perspective**, where you model the dynamic behavior of the system and how it responds to events.

Types of System Models

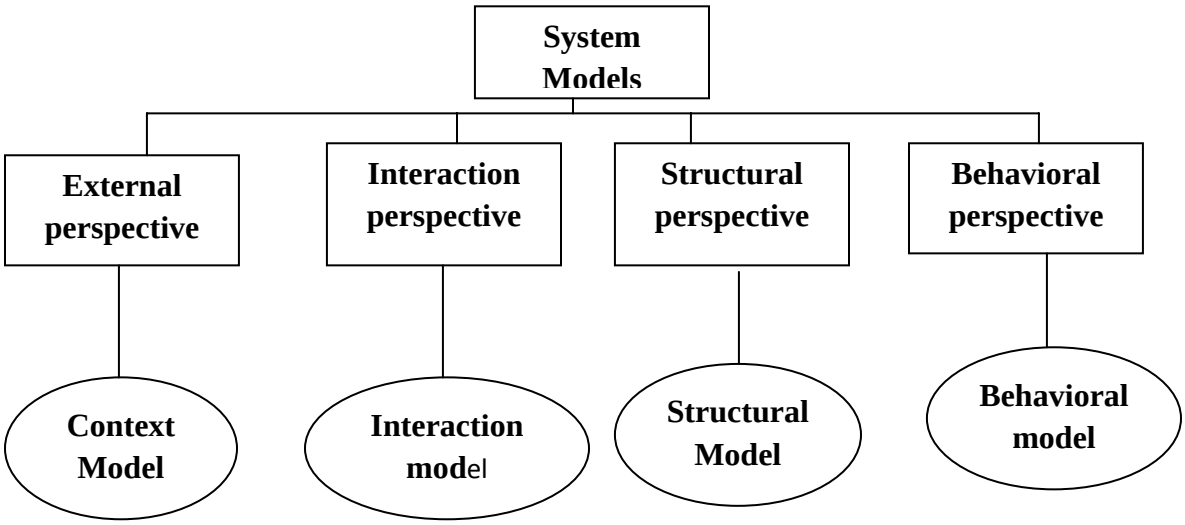


Figure 1.1: Types of System Models

❖ **UML (Unified Modeling Language)Diagram Types**

- **Activity diagrams**: which show the activities involved in a process or in data processing.
- **Use case diagrams:** which show the interactions between a system and its environment?
- **Sequence diagrams**: which show interactions between actors and the system and between system components?
- **Class diagrams**: which show the object classes in the system and the associations between these classes?
- **State diagrams**: which show how the system reacts to internal and external events?

❖ **Use Of Graphical Models**

- As a means of facilitating discussion about an existing or proposed system.
  - Incomplete and incorrect models are OK as their role is to support discussion.
- As a way of documenting an existing system
  - Models should be an accurate representation of the system but need not be complete.
- As a detailed system description that can be used to generate a system implementation
  - Models have to be both correct and complete.

1.1 Context Models

- Context models are used to illustrate the **operational context** of a system - they show what lies outside the system boundaries.
- Social and organisational concerns may affect the decision on where to position system boundaries.
- Architectural models show the system and its relationship with other systems.

1.1.1 System Boundaries

- System boundaries are established to define what is **inside and what is outside the system**.
  - They show other systems that are used or depend on the system being developed.
- The position of the system boundary has a profound effect on the system requirements.
- Defining a system boundary is a political judgment
  - There may be pressures to develop system boundaries that increase / decrease the influence or workload of different parts of an organization.
- If you are developing the specification for the patient information system for mental healthcare. This system is intended to manage information about patients attending mental health clinics and the treatments that have been prescribed.
- In developing the specification for this system, you have to decide whether the system should focus exclusively on collecting information about consultations (using other systems to collect personal information about patients) or whether it should also collect personal patient information.

Advantage:

- The advantage of relying on other systems for patient information is that you avoid duplicating data.

Disadvantage:

- However, is that using other systems may make it slower to access information?
- If these systems are unavailable, then the MHC-PMS cannot be used.

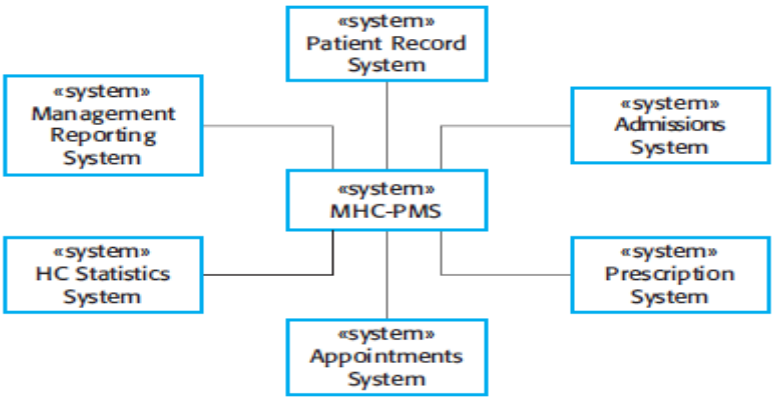


Figure 1.2: The context model of the MHC-PMS

- Figure 1.2 is a simple context model that shows the patient information system and the other systems in its environment. It is connected to an appointments system and a more general patient record system with which it shares data.
- The system is also connected to systems for management reporting and hospital bed allocation and a statistics system that collects information for research. Finally, it makes use of a prescription system to generate prescriptions for patients' medication.
- 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.

- Therefore, simple context models are used along with other models, such as business process models. These describe human and automated processes in which particular software systems are used.

1.1.2 Process Model of Involuntary Detention

- **Process model** is a model of an important system process that shows the processes in which the MHC-PMS is used. Sometimes, patients who are suffering from mental health problems may be a danger to others or to themselves. They may therefore have to be detained against their will in a hospital so that treatment can be administered.
- Such detention is subject to strict legal safeguards—**for example**, the decision to detain a patient must be regularly reviewed so that people are not held indefinitely without good reason. One of the functions of the MHC-PMS is to ensure that such safeguards are implemented.
- **Figure 1.3 is a UML activity diagram.** Activity diagrams are intended to show the activities that make up a system process and the flow of control from one activity to another.
- **Workflow:**
  1. The start of a process is indicated by a **filled circle.**
  2. End by a filled circle inside another **circle**.
  3. **Rectangles with round corners** represent activities, that is, the specific sub-processes that must be carried out.

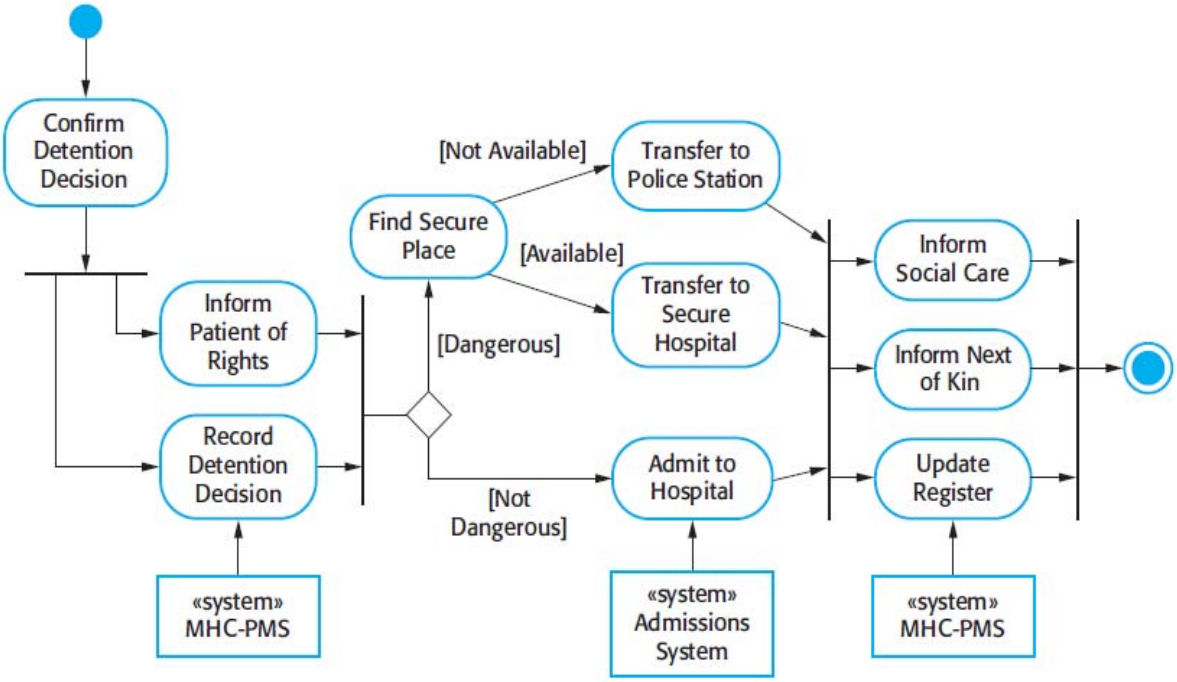


Figure 1.3: Process Model of Involuntary Detention

4. **Arrows** represent the flow of work from one activity to another.
5. **A solid bar** is used to indicate activity coordination. When the flow from more than one activity leads to a solid bar then all of these activities must be complete before progress is possible. When the flow from a solid bar leads to a number of activities, these may be executed in parallel. Therefore, in Figure 1.2, the activities to inform social care and the patient's next of kin(family or friends) and to update the detention register may be concurrent.
6. Arrows may be **annotated with guards** that indicate the condition when that flow is taken.
7. Guards showing the flows for patients who are dangerous and not dangerous to society. Patients who are dangerous to society must be detained in a secure facility. However,
8. Patients who are suicidal and so are a danger to themselves may be detained in an appropriate ward in a hospital.

❖ **Context model of inventory control system**

First of all using the requirements of the system the **system boundaries** are decided and dependencies of the system are specified in order to define the system environment. As shown in the figure 1.4, system boundaries are clearly shown and the environmental factors are defined. The inventory control system is connected to various systems such as inventory monitoring system, accounting system is connected to various system such as accounting system, report generator system, maintenance system

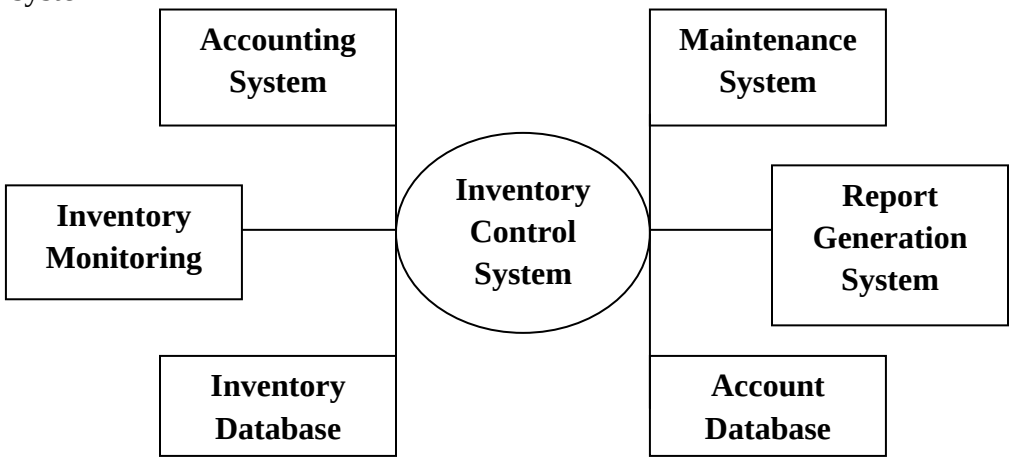
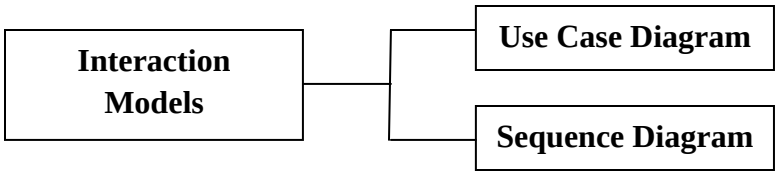


Figure 1.4: context model of the inventory control system.

- Context models are architectural models on which environment of the system is shown.
- The relationship that exists with other system is shown but nature of these relationships is not mentioned in the context model. All these defined relationships help in finding the requirements of the system.
- To detail out such architectural model some process model can be used in conjunction with this .The process model is again graphical representation of system processes.

1.2 Interaction Models

- All systems involve interaction of some kind. This can be user interaction, which involves user inputs and outputs, interaction between the systems being developed and other systems or interaction between the components of the system.



- Modeling user interaction is important as it helps to identify user requirements.
- Modeling system-to-system interaction highlights the communication problems that may arise.
- Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.
- Use case diagrams and sequence diagrams may be used for interaction modelling.

1.2.1 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).
- Use cases were developed originally to support requirements **elicitation and analysis** activity.
- **Each use case** represents a discrete task that involves external interaction with a system.
- Actors in a use case may be people or other systems.
- Represented diagrammatically to provide an overview of the use case and in a more detailed textual form.

Transfer-data use case of MHC-PMS

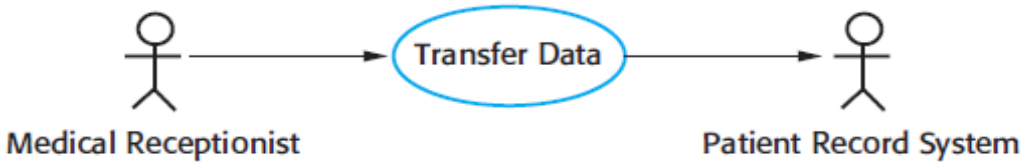


Figure 1.5: Transfer-data use case

- The above figure shows a use case from the MHC-PMS that represents the task of uploading data from the MHC-PMS to a more general patient record system.
- This maintains summary data about a patient rather than the data about each consultation, which is recorded in the MHC-PMS.
- Notice that there are two actors in this use case: the operator who is transferring the data and the patient record system.
- The stick figure notation was originally developed to cover human interaction but it is also now used to represent other external systems and hardware.
- Formally, use case diagrams should use lines without arrows as arrows in the UML indicate the direction of flow of messages. Obviously, in a use case messages pass in both directions.

Table 1.1: Tabular description of the 'Transfer data' use-case

MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the MHC-PMS to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

Use cases in the MHC-PMS involving the role 'Medical Receptionist'

Medical receptionist is an actor in MHC-PMS system, list of activities are listed from admitting a n mental patient to discharge of an patient.

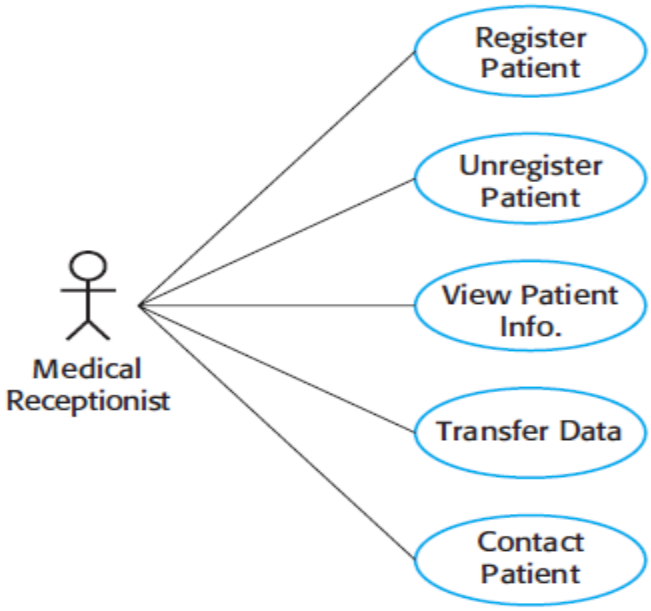


Figure 1.6: Use cases in the MHC-PMS involving the role 'Medical Receptionist'

1.2.2 Sequence diagrams

- Sequence diagrams are part of the UML and are used to model the interactions between the actors and the objects within a system.
- A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
- 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.
- A box named alt is used with the conditions indicated in square brackets.

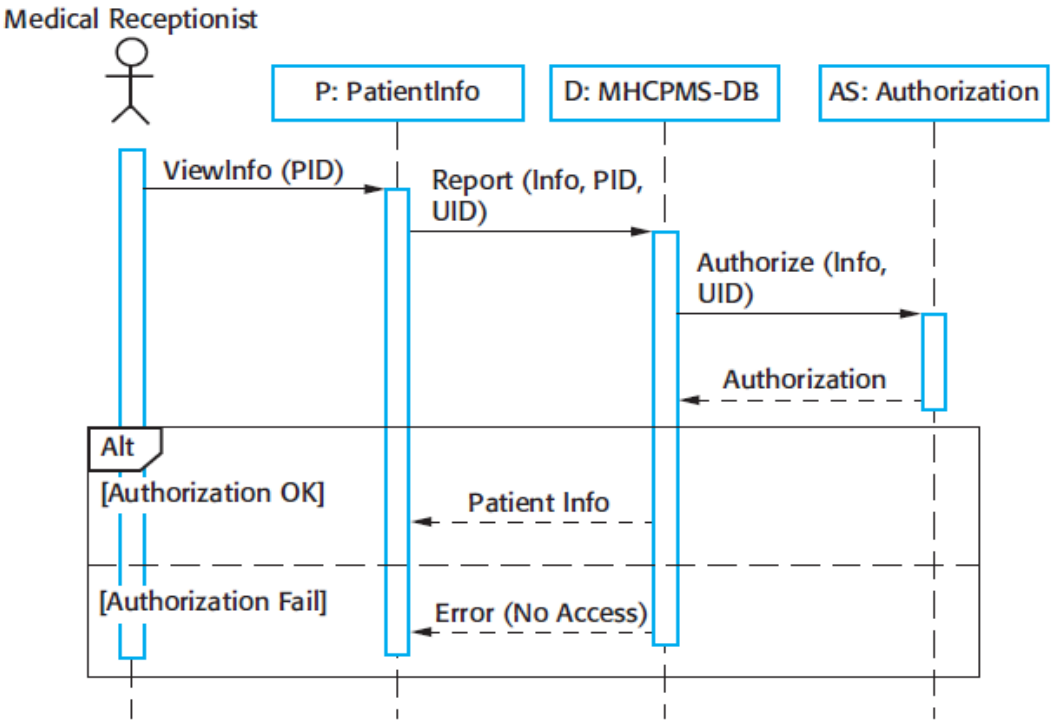


Figure 1.7: Sequence diagram for View patient information

Workflow:

1. The medical receptionist triggers the View Info method in an instance P of the Patient Info 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.

Sequence Diagram for Transfer Data

Figure 1.8 is a second example of a sequence diagram from the same system that **illustrates two additional features**. These are the direct communication between the actors in the system and the creation of objects as part of a sequence of operations. In this example, an object of type Summary is created to hold the summary data that is to be uploaded to the PRS (patient record system). You can read this diagram as follows:

Workflow:

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.

From Figure 1.8 the decision on how to get the user's identifier to check authorization is one that can be delayed. In an implementation, this might involve interacting with a User object but this is not important at this stage and so need not be included in the sequence diagram.

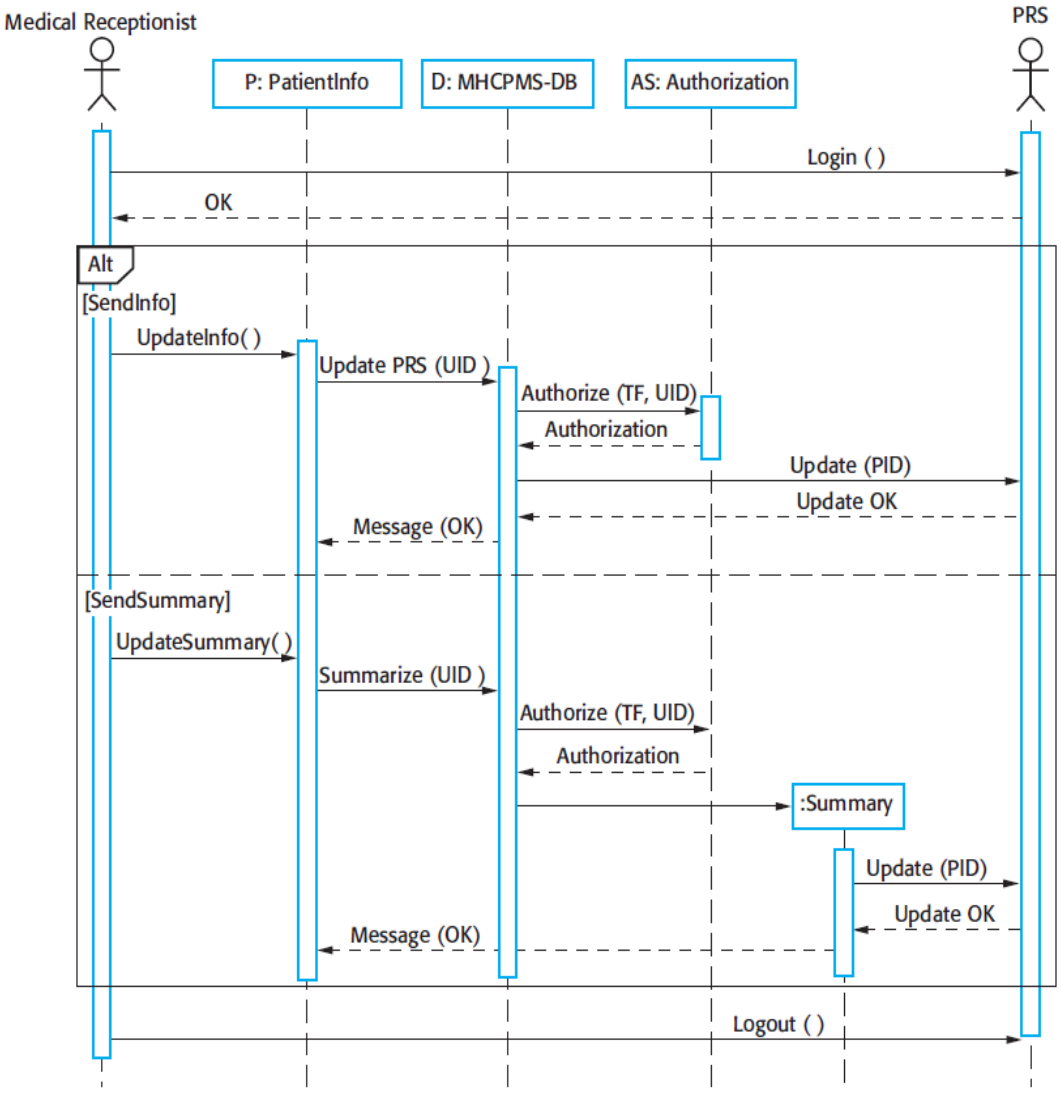


Figure 1.8: Sequence diagram for Transfer Data

1.3 Structural Models

- Structural models represent the **organization of a system** in terms of the components that make up that system and their relationships.
- Structural models may be static models, which show the structure of the system design, as well as **dynamic models**, which show the organization of the system when it is executing.
- You create structural models of a system when you are discussing and designing the system architecture.
- The **class diagram** is used to represent the structural model.

1.3.1 Class Diagrams

- Class diagram is a diagram in which various classes and the relationship among these classes is represented and the associations between these classes.
- 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 some relationship between these classes.
- When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.
- 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 by drawing a line between classes.
- For example, the below diagram is a simple class diagram showing two classes: Patient and Patient Record with an association between them.

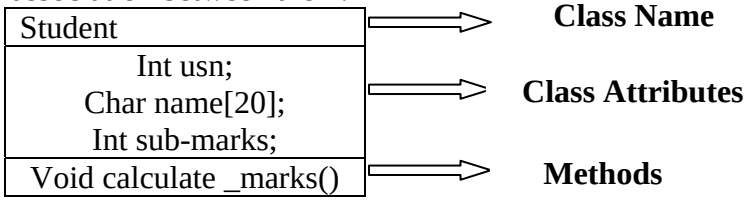


Figure1.9: General Example Of Class Diagram Of Student Class

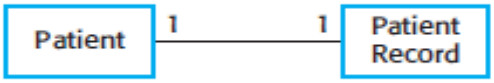


Figure 1.10: UML classes and association of patient record

- Figure 1.10, illustrates a further feature of class diagrams—the ability to show how many objects are involved in the association. In this example, each end of the association is annotated with a 1, meaning that there is a **1:1 relationship** between objects of these classes. That is, each patient has exactly one record and each record maintains information about exactly one patient. As you can see from later examples, other multiplicities are possible. You can define that an exact number of objects are involved

or, by using a *, as shown in Figure 1.9, that there are an indefinite number of objects involved in the association.

- Figure 1.10 develops this type of class diagram to show that objects of class Patient are also involved in relationships with a number of other classes. In this example, show that name associations to give the reader an indication of the type of relationship that exists. The UML also allows the role of the objects participating in the association to be specified.
- 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.
- 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, use the UML to represent a semantic data model. You can think of entities in a semantic data model as simplified object classes

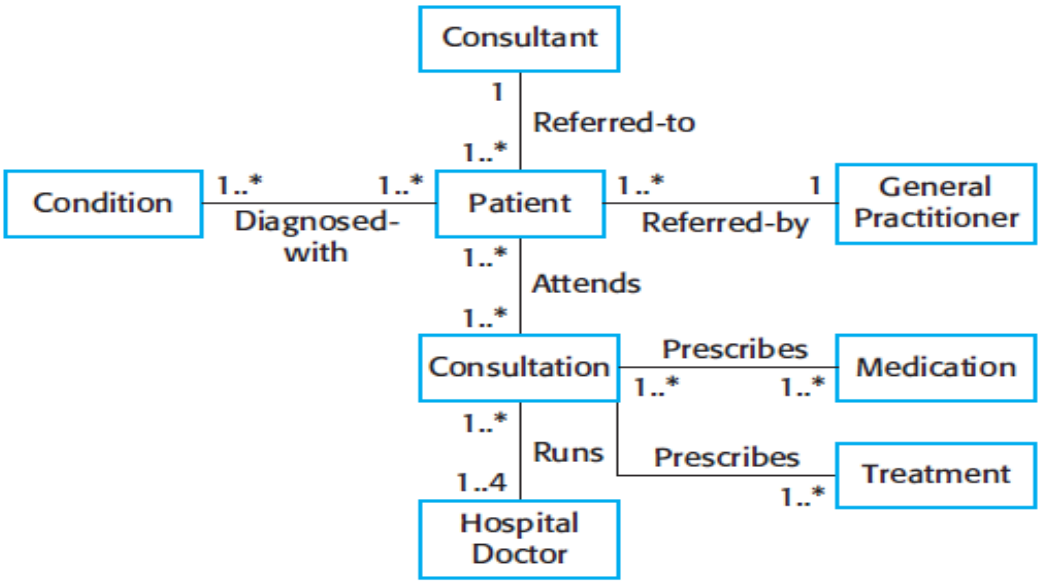


Figure 1.11: Classes and associations in the MHC-PMS

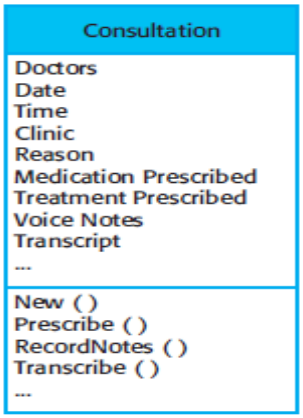


Figure 1.12: The consultation class of doctor

1.3.2 Generalization

- Generalization is an everyday technique that we use to manage complexity.
- It is a relationship between **parent class and child class.**
- Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes.
- In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.
- In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.
- In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.
- The **lower-level classes are subclasses** inherit the attributes and operations from their super classes. These lower-level classes then add more specific attributes and operations.

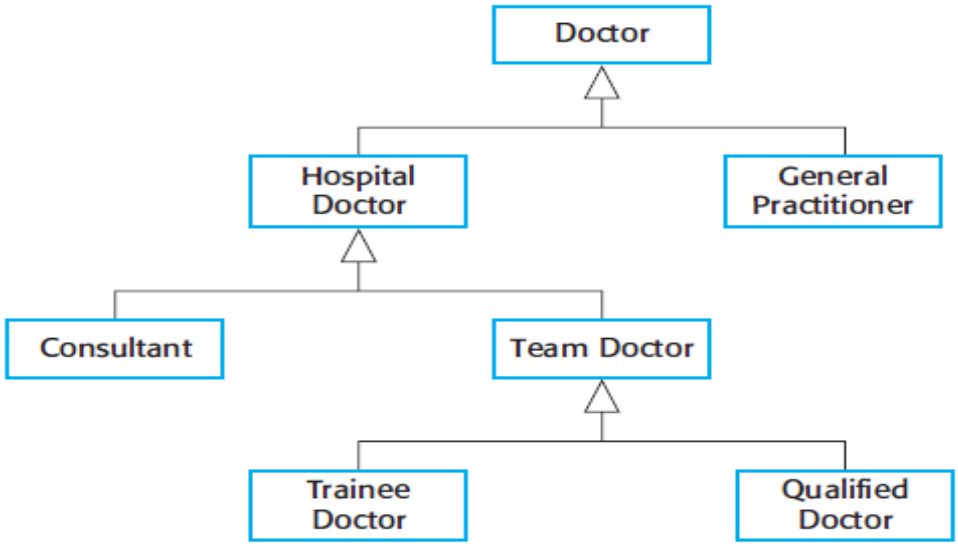


Figure 1.13: Generalization Hierarchy

- In the figure 1.13 generalizations is shown as an arrowhead pointing up to the more general class. This shows that general practitioners and hospital doctors can be generalized as doctors and that there are three types of Hospital Doctor— those that have just graduated from medical school and have to be supervised (Trainee Doctor), those that can work unsupervised as part of a consultant's team (Registered Doctor); and consultants, who are senior doctors with full decision making responsibilities.
- In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.
- In essence, the lower-level classes are subclasses inherit the attributes and operations from their super classes.
- These lower-level classes then add more specific attributes and operations. For example, all doctors have a name and phone number; all hospital doctors have a staff number and a department but general practitioners don't have these attributes as they work independently. They do however; have a practice name and address. This is illustrated in Figure 1.14, which

shows part of the generalization hierarchy that extended with class attributes. The operations associated with the class Doctor are intended to register and de-register that doctor with the MHC-PMS.

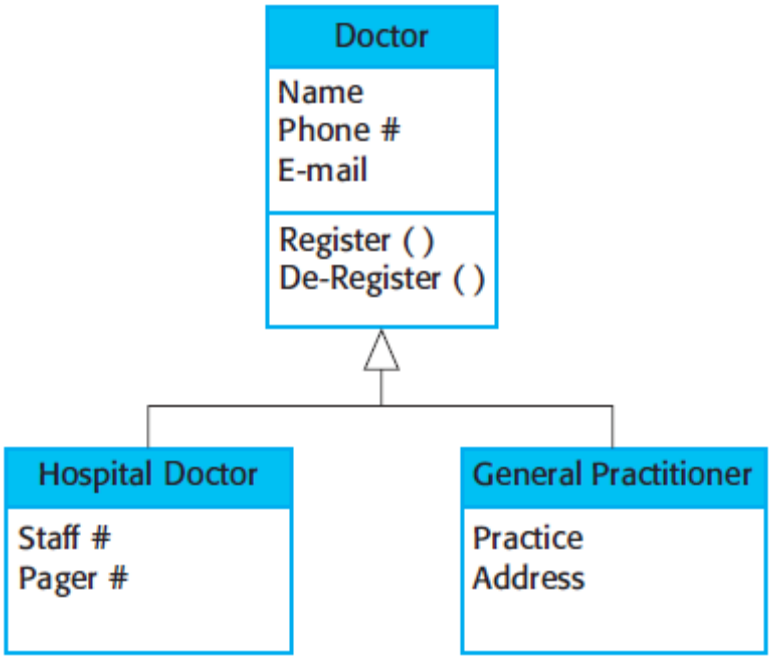


Figure 1.14: A generalization hierarchy with added detail

1.3.3 Aggregation

- Aggregation is one type of association.
- An aggregation model shows how classes that are collections are composed of other classes.
- Represents a **whole-part** relationship. Denoted by 
- Aggregation models are similar to the part-of relationship in semantic data models.

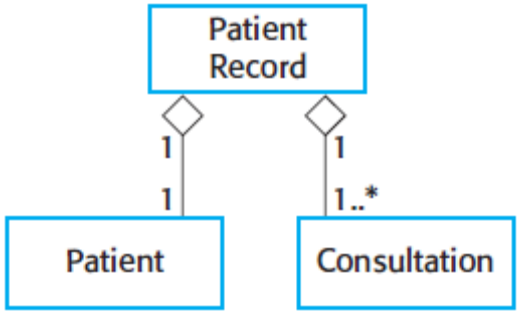


Figure 1.15: Aggregation association of patient and doctors consultation

1.4 Behavioral models

- Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.
- **Stimuli are of two types:**
    1. Some **data** arrives that has to be processed by the system.
    2. Some **events** happens that triggers system processing. Events may have associated data, although this is not always the case.

1.4.1 Data-driven modeling

- Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing. Their processing involves a sequence of actions on that data and the generation of an output.
- Data-driven models show the sequence of actions involved in processing input data and generating an associated output.
- They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.
- Data-driven models were amongst the first graphical software models. In the 1970s, structured methods such as DeMarco's Structured Analysis (DeMarco, 1978) introduced data-flow diagrams (DFDs) as a way of illustrating the processing steps in a system.
- Data-flow models are useful because tracking and documenting how the data associated with a particular process moves through the system helps analysts and designers understand what is going on.
- Data-flow diagrams are simple and intuitive and it is usually possible to explain them to potential system users who can then participate in validating the model.
- The UML does not support data-flow diagrams as they were originally proposed and used for modeling data processing.

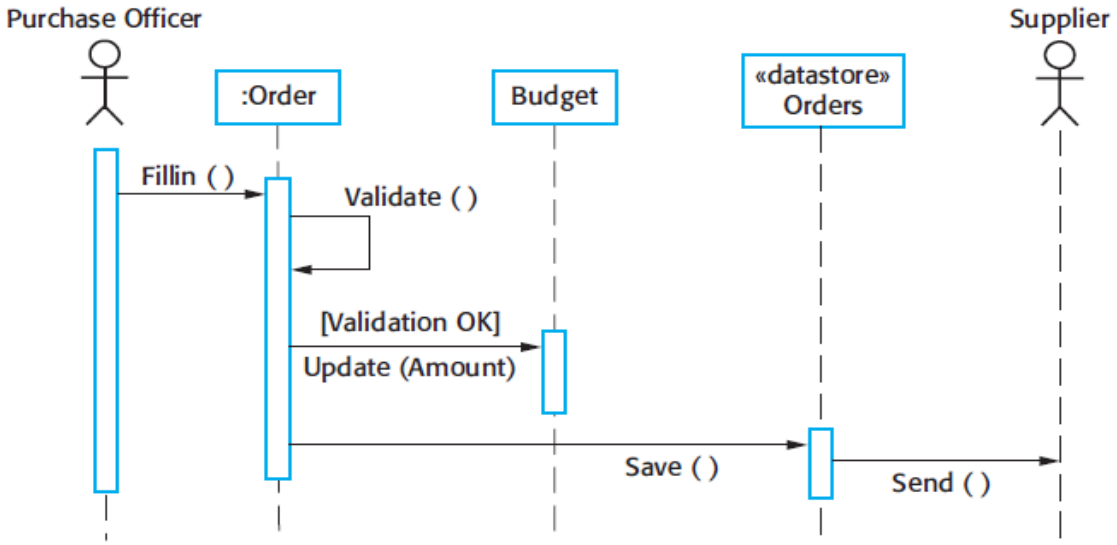


Figure 1.16: An activity model of the insulin pump's operation

- The above diagram shows the chain of processing involved in the insulin pump software. In this diagram, you can see the processing steps (represented as activities) and the data flowing between these steps (represented as objects).
- An alternative way of showing the sequence of processing in a system is to use UML sequence diagrams.

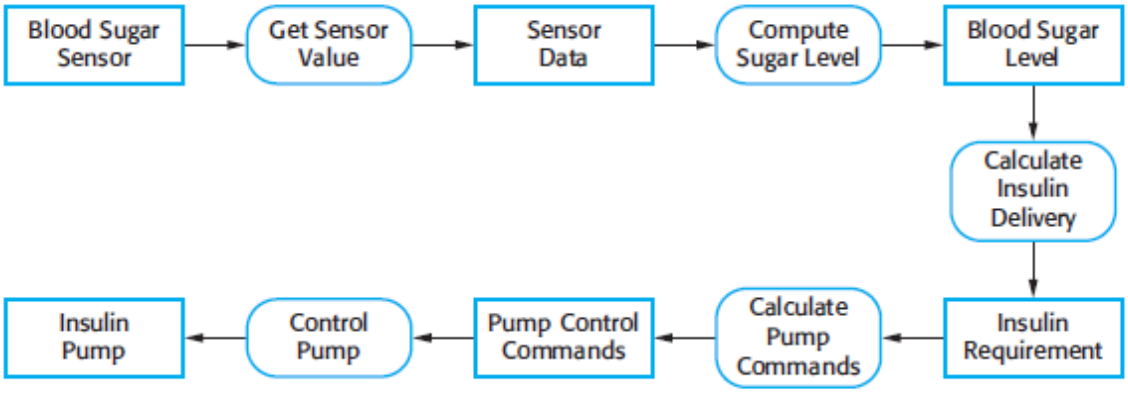


Figure 1.17: Order Processing

- The above diagram shows using a sequence model of the processing of an order and sending it to a supplier.
- Sequence models highlight objects in a system, whereas data-flow diagrams highlight the functions.

1.4.2 Event-Driven Modeling

- Real-time systems are often event-driven, with minimal data processing. For example, a landline phone switching system responds to events such as 'receiver off hook' by generating a dial tone.
- Event-driven modeling shows how a system responds to external and internal events.
- It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.
- The UML supports event-based modeling using state diagrams, which were based on Statecharts (Harel, 1987, 1988). State diagrams show system states and events that cause transitions from one state to another.
- They do not show the flow of data within the system but may include additional information on the computations carried out in each state.

State Machine Models

- These model the behaviour of the system in response to external and internal events.
- They show the system's responses to stimuli so are often used for modelling real-time systems.
- State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- Statecharts are an integral part of the UML and are used to represent state machine models.

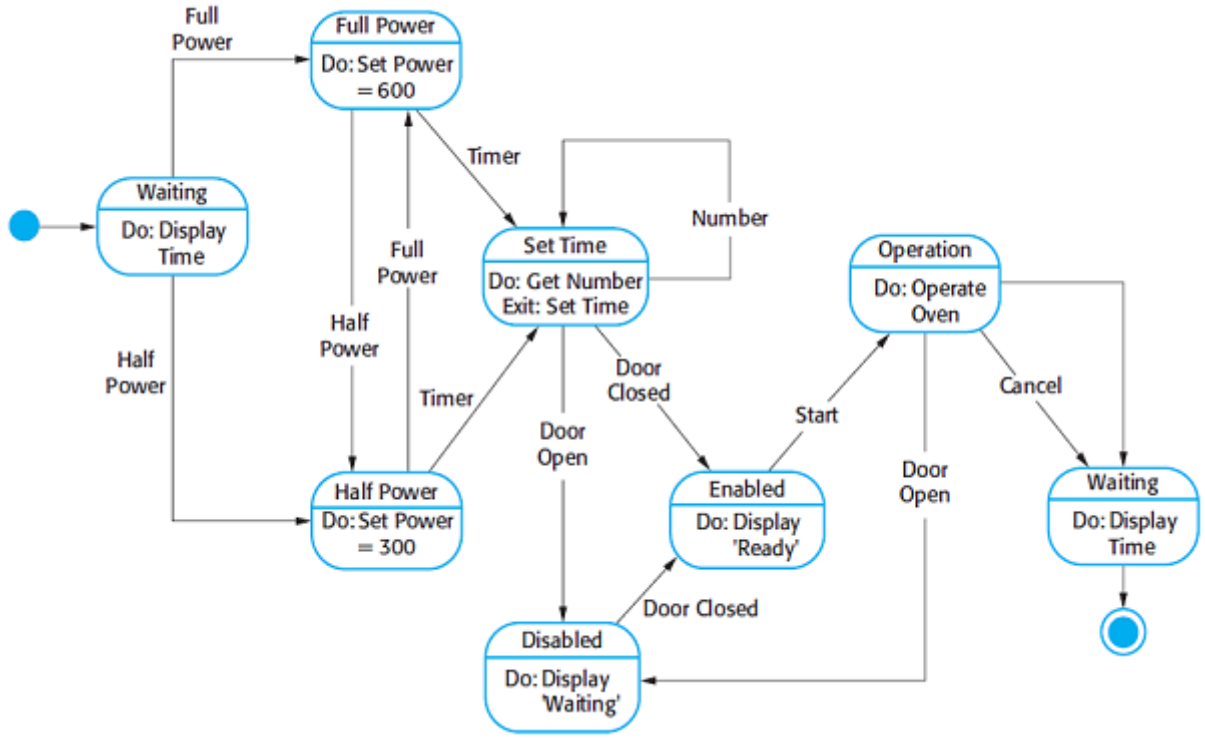


Figure 1.18: State diagram of a Microwave Oven

- An example of control software for a very simple microwave oven is used to illustrate **event-driven modeling**. Real microwave ovens are actually much more complex than this system but the simplified system is easier to understand.
- This simple microwave has a switch to select full or half power, a numeric keypad to input the cooking time, a start/stop button, and an alphanumeric display.
- **The sequence of actions in using the microwave is**:
    1. Select the power level (either half power or full power).
    2. Input the cooking time using a numeric keypad.
    3. Press Start and the food is cooked for the given time.
- For **safety reasons**, the oven should not operate when the door is open and, on completion of cooking, a buzzer is sounded.
- The oven has a very simple **alphanumeric display** that is used to display various alerts and warning messages.
- In UML state diagrams, **rounded rectangles represent system states**. They may include a brief description (following 'do') of the actions taken in that state.
- The **labeled arrows** represent stimuli that force a transition from one state to another.
- Indicate **start** and **end** states using **filled circles**, as in activity diagrams.
- From Figure 1.18, it is clear that the system starts in a waiting state and responds initially to either the **full-power or the half-power** button. Users can change their mind after selecting one of these and press the other button. The time is set and, if the door is closed, the Start button is enabled. Pushing this button starts the **oven operation** and cooking takes place for the specified time. This is the end of the cooking cycle and the system returns to the waiting state.

- The UML notation lets you indicate the activity that takes place in a state. In a detailed system specification you have to provide more detail about both the stimuli and the system states. **Table 1.2**, illustrate this in which shows a tabular description of each state and how the stimuli that force state transitions are generated.
- The **problem with state-based modeling** is that the number of possible states increases rapidly. For large system models, therefore, you need to hide detail in the models. One way to do this is by **using the notion of a super-state that encapsulates a number of separate states**. This super state looks like a single state on a high-level model but is then expanded to show more detail on a separate diagram. To illustrate this concept, consider the Operation state in Figure 1.18. This is a super state that can be expanded, as illustrated in table 1.2.
- The Operation state includes a number of sub-states. It shows that operation starts with a status check and that if any problems are discovered an alarm is indicated and operation is disabled. Cooking involves running the microwave generator for the specified time; on completion, a buzzer is sounded. If the door is opened during operation, the system moves to the disabled state, as shown in figure 1.19.

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.
Stimulus	**Description**
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.

Table 1.2: States and stimuli for the microwave oven

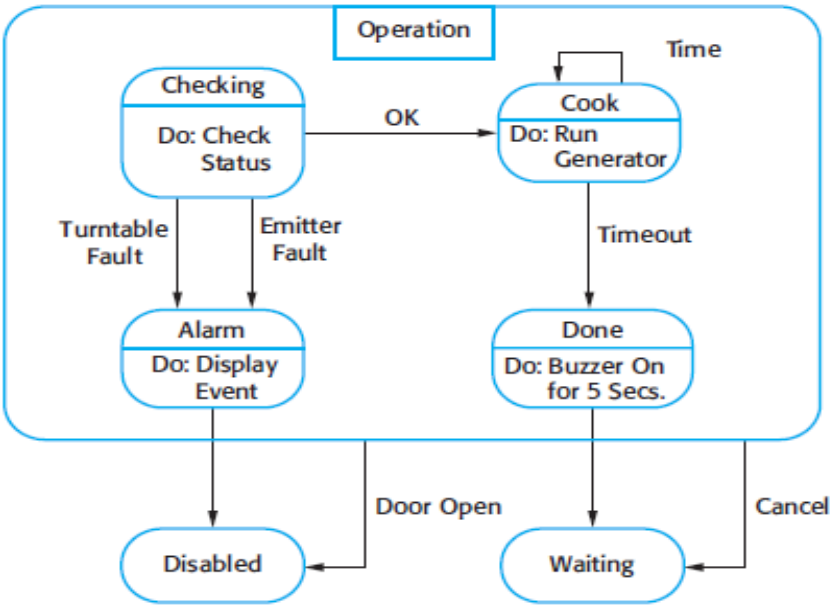


Figure 1.19: Microwave Oven Operations

1.5 Model-Driven Engineering

- Model-driven engineering (MDE) is an approach to software development where models rather than programs are the principal outputs of the development process.
- The programs that execute on a hardware/software platform are then generated automatically from the models.
- Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.
- Model-driven engineering has its roots in model-driven architecture (MDA) which was proposed by the Object Management Group (OMG) in 2001 as a new software development paradigm.

Usage of Model-Driven Engineering

- Model-driven engineering is still at an early stage of development, and it is unclear whether or not it will have a significant effect on software engineering practice.
- **Pros**
    1. Allows systems to be considered at higher levels of abstraction
    2. Generating code automatically means that it is cheaper to adapt systems to new platforms.
- **Cons**
    1. Models for abstraction and not necessarily right for implementation.
    2. Savings from generating code may be outweighed by the costs of developing translators for new platforms.

1.5.1 Model-driven architecture (MDA) was the precursor of more general model-driven engineering.

- MDA is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system.
- Models at different levels of abstraction are created. From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.

Types of model

- **A computation independent model (CIM):** This type of model represents the important domain abstractions used in a system. CIMs are sometimes called domain models.
- **A platform independent model (PIM):** These models the operation of the system without reference to its implementation. The PIM is usually described using **UML models** that show the static system structure and how it responds to external and internal events.
- **Platform specific models (PSM)***:*This model is used during application platform. The platform independent model is transformed into the platform specific model.These are layers of platform specific models that are added to model architecture. In each of these layers the **platform specific information** is added.

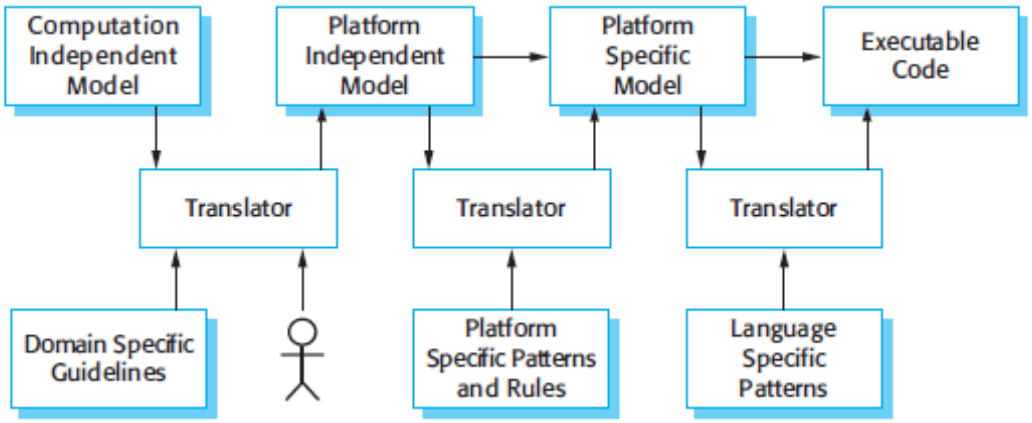


Figure 1.20: MDA platform-specific models

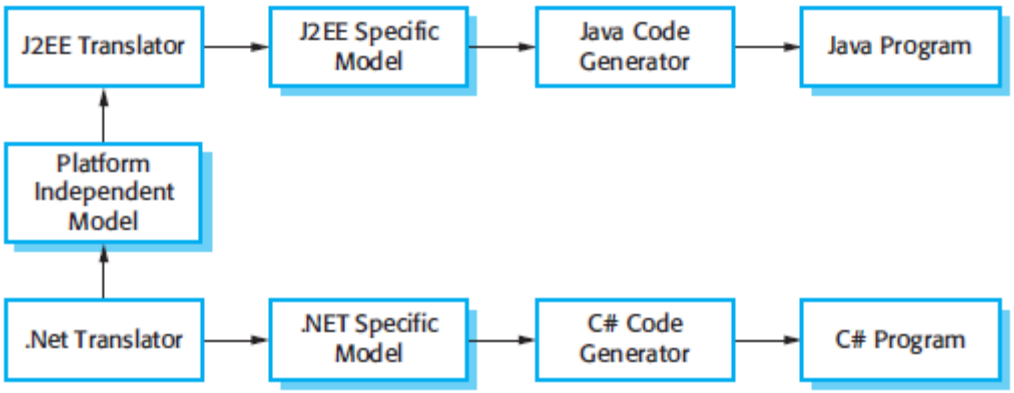


Figure 1.21: Multiple platform-specific models

1.5.2 Executable UML

- The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible.
- This is possible using a subset of UML 2, called Executable UML or XUML.

➢ **Features of executable UML**

- To create an executable subset of UML, the number of model types has therefore been dramatically reduced to these **3 key types**:
  - **Domain models** that identify the principal concerns in a system. They are defined using UML class diagrams and include objects, attributes and associations.
  - **Class models** in which classes are defined, along with their attributes and operations.
  - **State models** in which a state diagram is associated with each class and is used to describe the life cycle of the class.
- The dynamic behaviour of the system may be specified declaratively using the object constraint language (OCL), or may be expressed using UML's action language.

Design and Implementation

2.1 The Rational Unified Process (RUP)

- The Rational Unified Process (RUP) is an example of a modern process model and it is a good example of a **hybrid process model.**
- It brings together elements from all of the generic process models, supports iterative and incremental by nature.
- The RUP recognises that conventional process models present a single view of the process. In contrast, the RUP is normally described from **three perspectives**:

1. **A dynamic perspective** that shows the phases of the model over time.
2. **A static perspective** that shows the process activities that are enacted.
3. **A practice perspective** that suggests good practices to be used during the process.

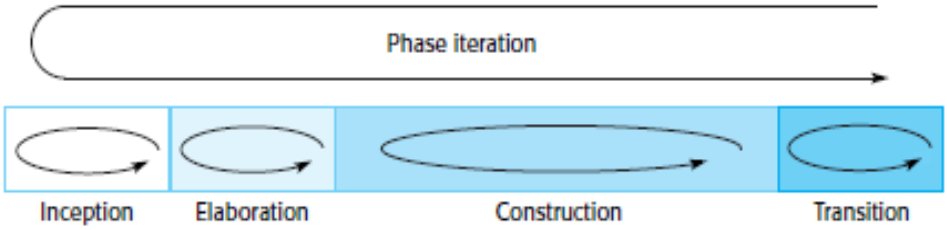


Figure 2.1: Phases in the Rational Unified Process

- The RUP is a phased model that identifies **four discrete phases** in the software process.
- Figure 2.1 shows the phases in the RUP. These are:
  - **Inception** The goal of the inception phase is to **establish a business case** for the system. Identify all external entities (people and systems) that will interact with the system and define these interactions. If this contribution is minor, then the project may be cancelled after this phase. Using the use cases sequence of actions can be identified.
  - **Elaboration** The goals of the elaboration phase are to **develop an understanding** of the **problem domain**, establish an **architectural framework** for the system, develop the project plan and **identify key project risks**.
  - **Construction** The construction phase is essentially concerned with **system design**, **programming and testing**. Parts of the system are developed in parallel and integrated during this phase.
  - **Transition** The final phase of the RUP is concerned with moving the system from the **development community to the user community** and making it work in a real environment.

Inception Phase	Elaboration Phase	Construction Phase	Transition Phase
Initial use case model	Use case model	Design model	Delivered software increments
Initial risk assessment	Requirements analysis model	Software components	Best test reports
Project plan	Architecture model	Test plans	User feedback report
	Preliminary design model	Test cases	
	Iterative project plan	Installation manual	

Figure 2.2: Various work products from RUP

- The core engineering and support workflows are described in below table

Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
Testing	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users and installed in their workplace.
Configuration and change management	This supporting workflow manages changes to the system (see Chapter 29).
Project management	This supporting workflow manages the system development (see Chapter 5).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

Table 2.1: Static workflows in Rational Unified Process

The practice perspective on the RUP describes **good software engineering practices** that are recommended for use in systems development. **Six fundamental best practices** are recommended:

1. Develop software iteratively. **Plan increments** of the system based on customer priorities and develop and deliver the **highest priority** system features early in the development process.

2. Manage requirements. Explicitly **document the customer's requirements** and **keep track of changes to these requirements.** Analyse the impact of changes on the system before accepting them. **Workflow Description**

3. Use component-based architectures. Structure the **system architecture into components** as discussed earlier in this chapter.

4. Visually model software. Use **graphical UML models** to present **static and dynamic views** of the software.

5. Verify software quality. Ensure that the software meets the **organisational quality standards**.

6. <u>Control changes to software.</u> Manage changes to the software using a change management system and configuration management procedures and tools.

3. Design Principle

1. **Abstraction –** allows designers to focus on solving a problem without being concerned about irrelevant lower level details (*procedural abstraction* - named sequence of events and *data abstraction* – named collection of data objects)
2. **Software Architecture** – overall structure of the software components and the ways in which that structure provides conceptual integrity for a system
    - Structural models – architecture as organized collection of components
    - Framework models – attempt to identify repeatable architectural patterns
    - Dynamic models – indicate how program structure changes as a function of external events
    - Process models – focus on the design of the business or technical process that system must accommodate
    - Functional models – used to represent system functional hierarchy
3. **Design Patterns** – description of a design structure that solves a particular design problem within a specific context and its impact when applied
4. **Separation of concerns** – any complex problem is solvable by subdividing it into pieces that can be solved independently
5. **Modularity** - the degree to which software can be understood by examining its components independently of one another
6. **Information Hiding** – information (data and procedure) contained within a module is inaccessible to modules that have no need for such information
7. **Functional Independence** – achieved by developing modules with single-minded purpose and an aversion to excessive interaction with other models
    - Cohesion - qualitative indication of the degree to which a module focuses on just one thing
    - Coupling - qualitative indication of the degree to which a module is connected to other modules and to the outside world
8. **Refinement** – process of elaboration where the designer provides successively more detail for each design component
9. **Aspects** – a representation of a cross-cutting concern that must be accommodated as refinement and modularization occur
10. **Refactoring** – process of changing a software system in such a way internal structure is improved without altering the external behavior or code design

4. Design and Implementation

Design and implementation

- Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- Software design and implementation activities are invariably interleaved.
- Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
- Implementation is the process of realizing the design as a program.
- Design and implementation are closely linked and you should normally take implementation issues into account when developing a design.
- In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
    - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.
- When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

4.1 Object-oriented design using the UML

- An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state.
- The representation of the state is private and cannot be accessed directly from outside the object.
- Object-oriented design processes involve designing object classes and the relationships between these classes.
- To develop a system design from concept to detailed, object-oriented design, there are several things that you need to do:
    - Understand and define the context and the external interactions with the system.
    - Design the system architecture.
    - Identify the principal objects in the system.
    - Develop design models.
    - Specify interfaces.

4.1.1 System context and interactions

- Understanding the **relationships between the software** that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- Understanding of the **context also** lets you establish the boundaries of the system.
- Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

- System context models and interaction models present complementary views of the relationships between a system and its environment:
  - A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
  - An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

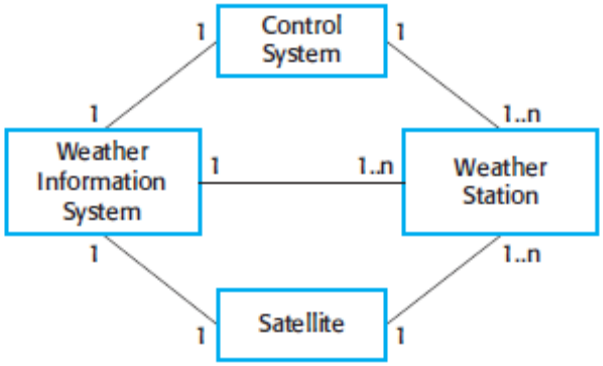


Figure 4.1: System context for the weather station

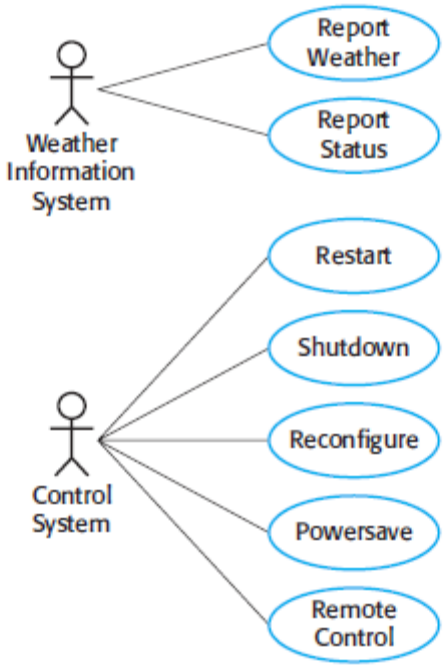


Fig 4.2: Use case description—Report weather

- A system context model is a structural model that demonstrates the other systems in the environment of the system being developed. An interaction model is a dynamic model that shows how the system interacts with its environment as it is used

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Dat	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data are sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.

Figure 4.3: Use case description—Report weather

4.1.2 Architectural Design

- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client-server model.
- The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure as shown below.

.

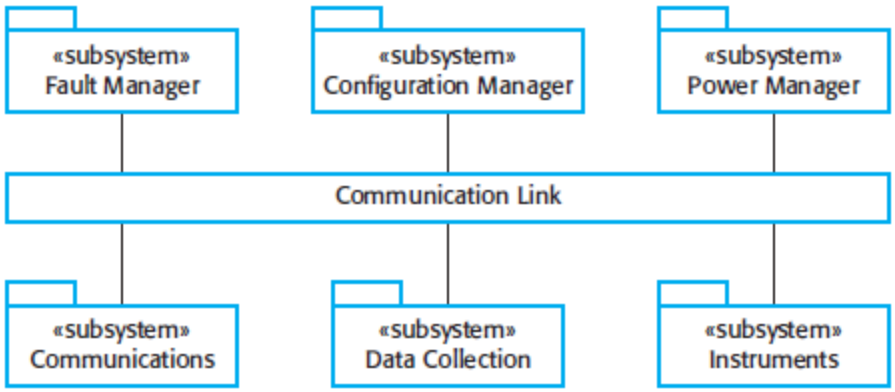


Figure 4.4: High-level architecture of the weather station

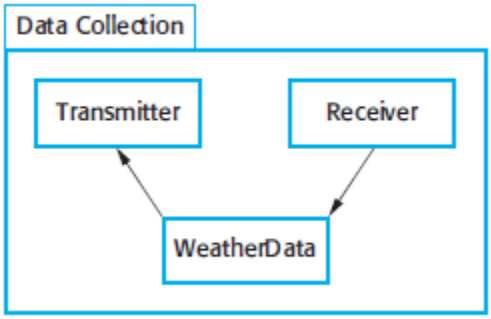


Figure 4.5: Architecture of data collection system

4.1.3 Object Class Identification

- Identifying object classes is too often a difficult part of object oriented design.
- There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first time.

Approaches to identification

1. **Use a grammatical analysis** of a natural language description of the system to be constructed. Objects and attributes are nouns; operations or services are verbs.
2. **Use tangible entities** (things) in the application domain such as aircraft, roles such as manager or doctor, events such as requests, interactions such as meetings, locations such as offices, organizational units such as companies, and so on.
3. **Use a scenario-based analysis** where various scenarios of system use are identified and analyzed in turn. As each scenario is analyzed, the team responsible for the analysis must identify the required objects, attributes, and operations.

Example : Weather station description

- A weather station is a package of software controlled instruments which collects data, performs some data processing and transmits this data for further processing. The instruments include air and ground thermometers, an anemometer, a wind vane, a barometer and a rain gauge. Data is collected periodically.
- When a command is issued to transmit the weather data, the weather station processes and summarises the collected data. The summarised data is transmitted to the mapping computer when a request is received.

Weather station object classes

- In the below figure The Ground thermometer, Anemometer, and Barometer objects are application domain objects, and the WeatherStation and WeatherData objects have been identified from the system description and the scenario (use case) description:
  1. The **WeatherStation object** class provides the basic interface of the weather station with its environment. Its operations reflect the interactions.

2. The **WeatherData object** class is responsible for processing the report weather command. It sends the summarized data from the weather station instruments to the weather information system.
3. The Ground thermometer, Anemometer, and Barometer object classes are directly related to instruments in the system. They reflect tangible hardware entities in the system and the operations are concerned with controlling that hardware. These objects operate autonomously to collect data at the specified frequency and store the collected data locally. This data is delivered to the WeatherData object on request.

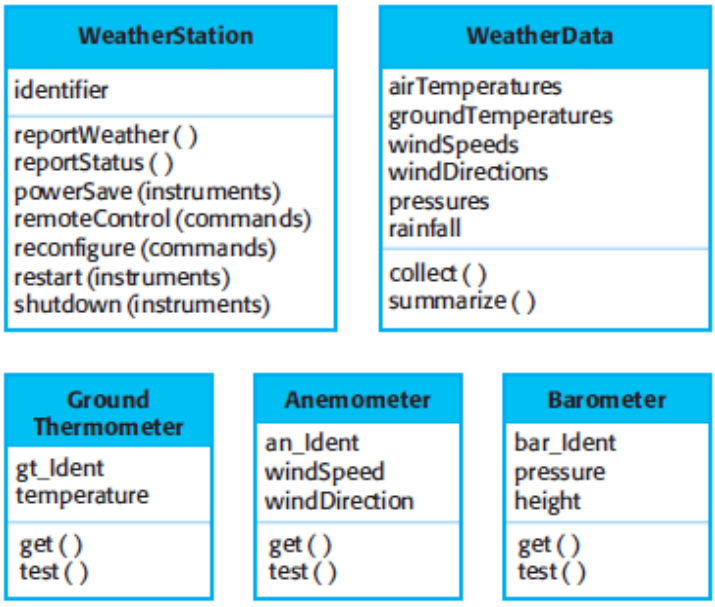


Figure 4.6: Weather station object classes

4.1.4 Design models

- Design models are the models created to represent the **objects and object classes and relationships between these entities**.
- These models are the bridge between the system requirements and the implementation of a system.
- An important step in the design process, therefore, is to decide on the design models that you need and the level of detail required in these models. This depends on the type of system that is being developed.
- The design models can be classified **into 2 types**
    1. **Structural models,** which describe the static structure of the system using, object classes and their relationships. Important relationships that may be documented at this stage are generalization (inheritance) relationships, uses/used-by relationships, and composition relationships.
    2. **Dynamic models,** which describe the dynamic structure of the system and show the interactions between the system objects. Interactions that may be documented include the sequence of service requests made by objects and the state changes that are triggered by these object interactions.

Examples of Design Models

1. **Subsystem models,** which that show logical groupings of objects into coherent subsystems. These are represented using a form of class diagram with each subsyste shown as a package with enclosed objects. Subsystem models are static (structural) models.
2. **Sequence models**, which show the sequence of object interactions. These are represented using a UML sequence or a collaboration diagram. Sequence models are dynamic models.
3. **State machine model,** which show how individual objects change their state in response to events. These are represented in the UML using state diagrams. State machine models are dynamic models.

Sequence model of weather information system:

- Sequence models show the sequence of object interactions that take place
  - **Objects** are arranged horizontally across the top;
  - **Time** is represented vertically so models are read top to bottom;
  - **Interactions** are represented by labelled arrows, Different styles of arrow represent different types of interaction;
  - A **thin rectangle** in an object lifeline represents the time when the object is the controlling object in the system.

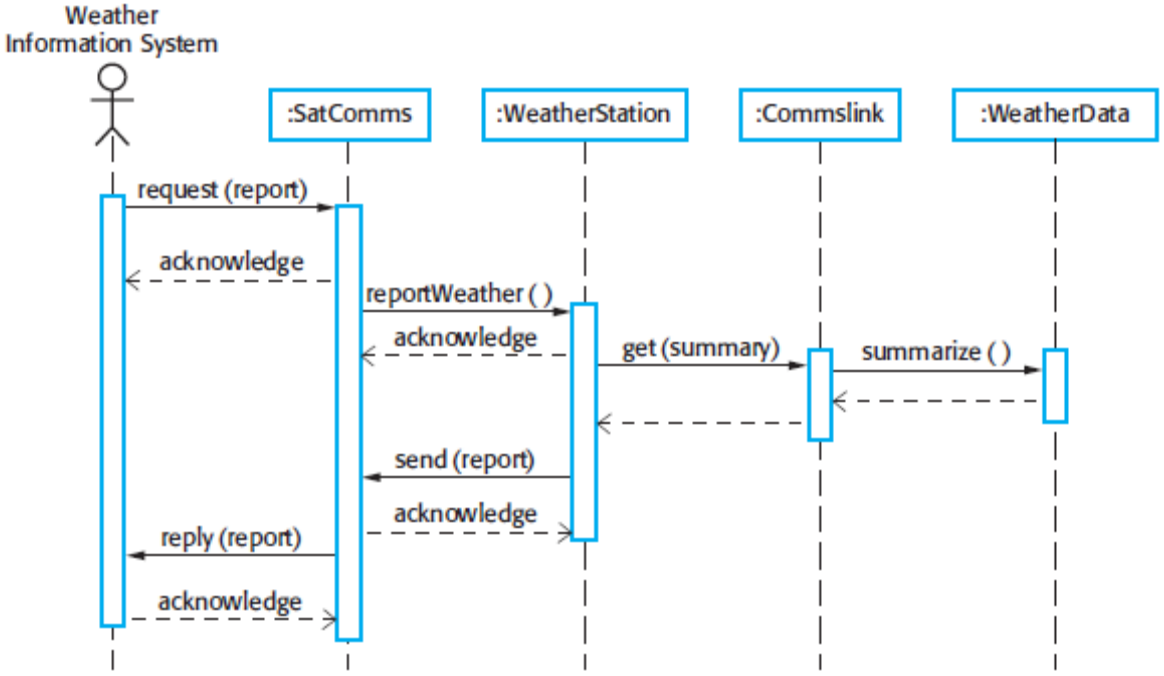


Figure 4.7: Sequence diagram describing data collection

You read sequence diagrams from top to bottom:

1. The **SatComms object receives** a request from the weather information system to collect a weather report from a weather station. It acknowledges receipt of this request. The stick arrowhead on the sent message indicates that the external system does not wait for a reply but can carry on with other processing.

2. **SatComms sends a message to WeatherStation**, via a satellite link, to create a summary of the collected weather data. Again, the stick arrowhead indicates that SatComms does not suspend itself waiting for a reply.
3**. WeatherStation sends a message to a Commslink object** to summarize the weather data. In this case, the squared-off style of arrowhead indicates that the instance of the WeatherStation object class waits for a reply.
4. **Commslink calls th**e summarize method in the object **WeatherData** and waits for a reply.
5. The weather data summary is computed and **returned to WeatherStation via the Commslink object.**
6. **WeatherStation then calls the SatComms** object to transmit the summarized data to the weather information system, through the satellite communications system.

State diagram of Weather station

- State diagrams are used to show how objects respond to different service requests and the state transitions triggered by these requests.
- State diagrams are useful high-level models of a system or an object's run-time behavior.
- You don't usually need a state diagram for all of the objects in the system. Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.

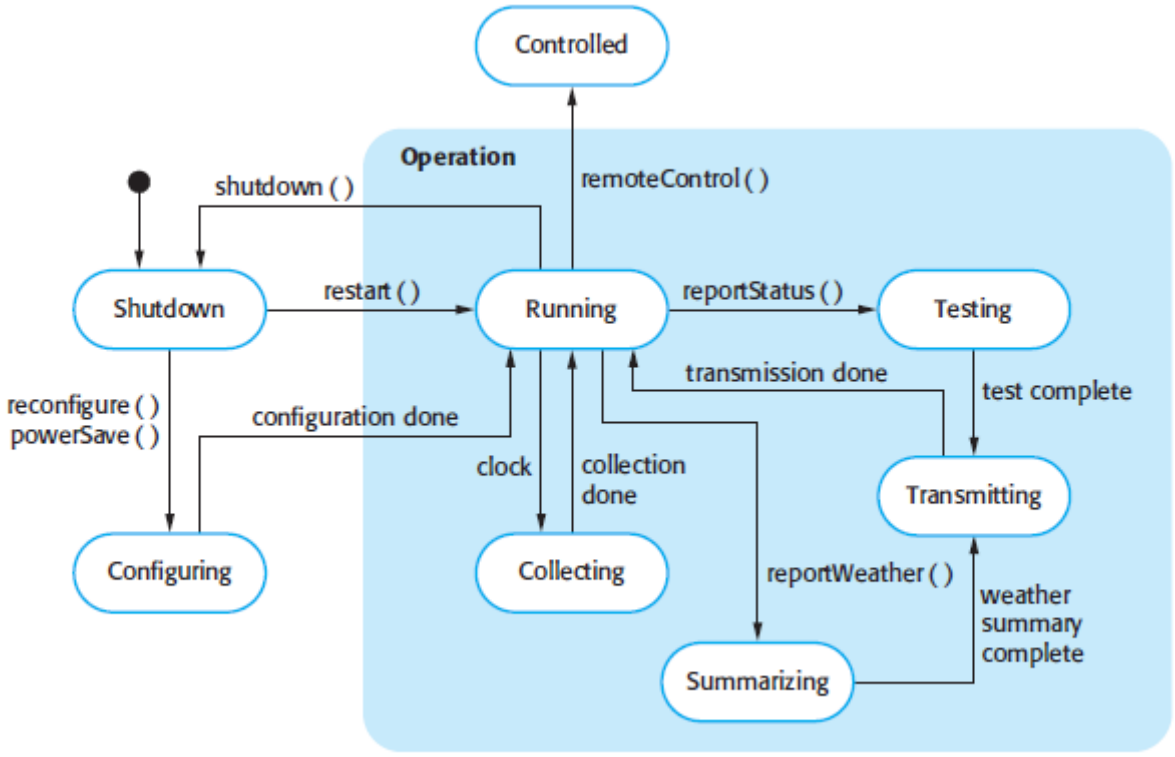


Figure 4.8: Weather station state diagram

You can read this diagram as follows:
1. If the system state is **Shutdown** then it can respond to a **restart(), a reconfigure(), or a powerSave()** message. The unlabeled arrow with the black blob indicates that the Shutdown state is the initial state. **A restart()** message causes a transition to normal operation. Both the **powerSave() and reconfigure()** messages cause a transition to a state in which the system reconfigures itself. The state diagram shows that reconfiguration is only allowed if the system has been shut down.
2. In the **Running state**, the system expects further messages. If a **shutdown()** message is received, the object returns to the shutdown state.
3. If a **reportWeather()** message is received, the system moves to the Summarizing state. When the summary is complete, the system moves to a **Transmitting state** where the information is transmitted to the remote system. It then returns to the Running state.
4. If a **reportStatus() message** is received, the system moves to the **Testing state**, then the **Transmitting state**, before returning to the **Running state.**
5. If a signal from the clock is received, the system moves to the **Collecting state**, where it collects data from the instruments. Each instrument is instructed in turn to collect its data from the associated sensors.
6. If a **remoteControl()** message is received, the system moves to a controlled state in which it responds to a different set of messages from the remote control room. These are not shown on this diagram.

Sub System Model

- These are static models.
- These show logical groupings of objects into logical subsystems.
- These are represented using a form of class diagram where subsystem is shown as a package.

4.1.5 Interface specification

- Object interfaces have to be specified so that the **objects and other components can be designed in parallel.**
- Designers should avoid designing the interface representation but should hide this in the object itself.
- Objects may have several interfaces which are viewpoints on the methods provided.
- The UML uses class diagrams for interface specification but Java may also be used example for this is shown below,

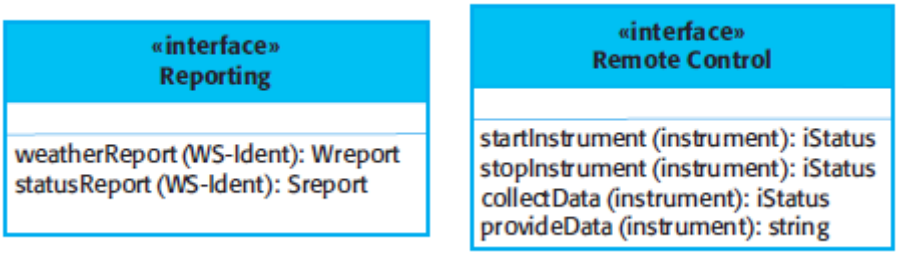


Figure 4.10: Weather station interfaces

4.2 Design Patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.
- A quote from the Hillside Group web site (http://hillside.net), which is dedicated to maintaining information about patterns, encapsulates their role in reuse:

Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.

- Patterns have made a huge impact on object-oriented software design. As well as being tested solutions to common problems, they have become a vocabulary for talking about a design.
- Design patterns are usually associated with object-oriented design.

Definition: Design pattern is general reusable solution to commonly occurring problem within a given context in software design.

The four essential elements of design patterns were defined by the 'Gang of Four' in their patterns book:

1. **A name** that is a meaningful reference to the pattern.
2. **A description** of the problem area that explains when the pattern may be applied.
3. **A solution** description of the parts of the design solution, their relationships, and their responsibilities. This is not a concrete design description. It is a template for a design solution that can be instantiated in different ways.
4. **A statement of the consequences**—the results and trade-offs—of applying the pattern. This can help designers understand whether or not a pattern can be used in a particular situation. The below figure shows the observer pattern.

Pattern name: Observer
Description: Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description: In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.

This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.
Solution description: This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.

The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
The UML model of the pattern is shown in Figure 7.12.
Consequences: The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Figure 4.11: The Observer Design pattern

- This pattern can be used in situations where different presentations of an object's state are required.
- It separates the object that must be displayed from the different forms of presentation
- This is illustrated in below figure which shows two graphical presentations of the same data set.

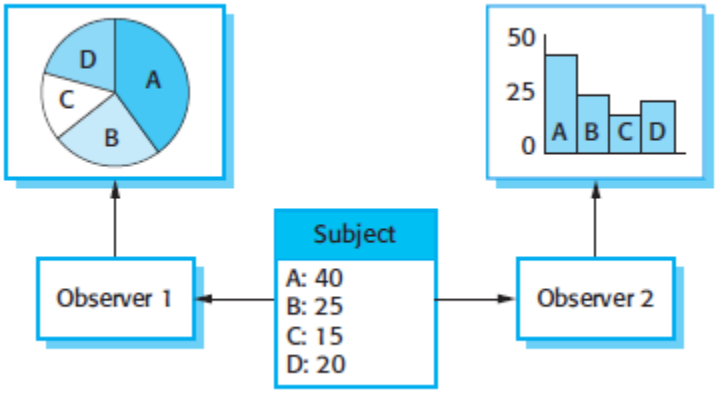


Figure 4.12: Multiple displays using the Observer pattern

- Graphical representations are normally used to illustrate the object classes in patterns and their relationships.
- These supplement the pattern description and add detail to the solution description. The below figure shows an UML representation of Observer Pattern

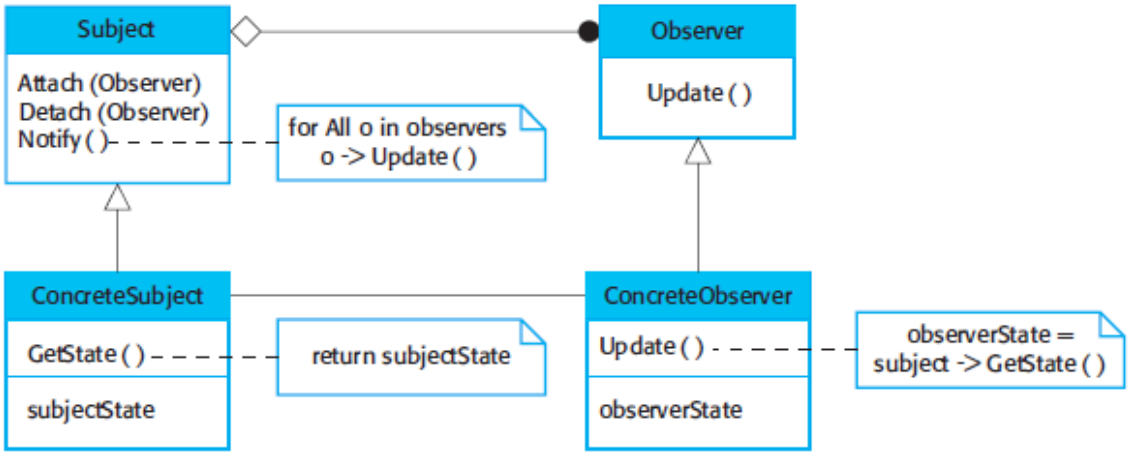


Figure 4.13: A UML model of the Observer pattern

- To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied.
  - Tell several objects that the state of some other object has changed (Observer pattern).

- Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
- Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).

Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).

4.3 Implementation issues

- Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
  - **Reuse** Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
  - **Configuration management** During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - **Host-target development** Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

4.3.1 Reuse

An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software

Reuse levels

- **The abstraction level:** At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.
- **The object level:** At this level, you directly reuse objects from a library rather than writing the code yourself. To implement this type of reuse, you have to find appropriate libraries and discover if the objects and methods offer the functionality that you need.
- **The component level:** Components are collections of objects and object classes that you reuse in application systems. You often have to adapt and extend the component by adding some code of your own.
- **The system level:** At this level, you reuse entire application systems. This usually involves some kind of configuration of these systems. This may be done by adding and modifying code (if you are reusing a software product line) or by using the system's own configuration interface.

Reuse costs

- The costs of the **time spent in** looking for software to reuse and assessing whether or not it meets your needs.
- Where applicable, the costs of **buying the reusable software**. For large off-the-shelf systems, these costs can be very high.
- The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- The costs of **integrating reusable software** elements with each other (if you are using software from different sources) and with the new code that you have developed.

4.3.2 Configuration Management

- Configuration management is the name given to the general process of managing a **changing software system.**
- The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made and compile and link components to create a system.

Configuration Management Activities

- **Version management:** where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- **System integration**: where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
- **Problem tracking**: where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

4.3.3 Host-Target Development

- Most software is **developed on one computer (the host),** but **runs on a separate machine (the target).**
- More generally, we can talk about a development platform and an execution platform.
  - A platform is more than just hardware.
  - It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- Development platform usually has different installed software than execution platform; these platforms may have different architectures.

➢ A software development platform should provide a range of tools to support software engineering processes. These may include:

1. **An integrated compiler** and syntax-directed editing system that allows you to create, edit, and compile code.
2. A **language debugging system**.
3. **Graphical editing tools**, such as tools to edit UML models.
4. **Testing tools, such as JUnit** (Massol, 2003) that can automatically run a set of tests on a new version of a program.
5. **Project support tools** that help you organize the code for different development projects.

- Software development tools are often grouped to create an integrated development environment (IDE).
- An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
- IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.

- As part of the development process, you need to make decisions about how the developed software will be deployed on the target platform.
  - Issues that you have to consider in making this decision are:
    1. ***The hardware and software requirements of a component*** If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
    2. ***The availability requirements of the system*** High-availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
    3. ***Component communications*** If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other.

4.4 Open source development

Definition: It is an approach in which the source code is freely available.

- Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.

Open source Systems:

- The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
- Other important open source products are Java, the Apache web server and the mySQL database management system.
  - For a company involved in software development, there are **two open source issues that have to be considered:**
    - ❖ Should the product that is being developed make use of open source components?
    - ❖ Should an open source approach be used for the software's development?

Open source business:

- More and more product companies are using an open source approach to development.
- Their business model is not reliant on selling a software product but on selling support for that product.
- They believe that involving the open source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.

4.4.1 Open source licensing

- A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
    - Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.
    - Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.
    - Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

The open source licenses are derived from one of the three License models

1. **The GNU General Public License (GPL).** This is a so-called **reciprocal** license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
2. **The GNU Lesser General Public License (LGPL)** is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
3. **The Berkley Standard Distribution (BSD)**. This is a **non-reciprocal license**, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

Guidelines for License management

1. Establish a system for maintaining information about open-source components that are downloaded and used.
2. Be aware of the different types of licenses and understand how a component is licensed before it is used.
3. Be aware of evolution pathways for components.
4. Educate people about open source.
5. Have auditing systems in place.
6. Participate in the open source community.

Question Bank
Module 1

Sl.no	Questions	Marks
1.	What are the needs for Software engineering?	5M
2.	What is Software? Explain essential attributes of good software.	6M
3.	What is Software Engineering? What are the fundamental activities of software engineering? Explain	6M
4.	What is the difference between computer science and software engineering?	2M
5.	What is the difference between software engineering and system engineering?	2M
6.	What are key challenges facing in software engineering? Explain	4M
7.	Mention two types of software products used in software engineering?	3M
8.	Mention the importance of software engineering?	2M
9.	Mention the different applications where software engineering is used? Explain.	8M
10.	Explain professional and ethical responsibilities. Also give the IEEE code of Ethics?	8M
11.	What are differences between generic and customized software products?	4M
12.	With neat diagrams explain an insulin pump control system?	8M
13.	Mention the 3 types of critical system where software engineering is used? Explain any one of them.	8M
14.	What is software process?	2M
15.	What is software process model? Mention the types of software process model.	4M
16.	Explain waterfall model mention its advantage and disadvantage.	8M
17.	Explain incremental development process model and also give advantages and disadvantages.	8M
18.	Explain different stages involved in Reuse- oriented software engineering?	6M
19.	Write spiral model of software process and explain with advantage	10M
20.	What are process activities? With a neat block diagram explain the requirements engineering process activities?	8M
21.	Describe the general model of the design process?	6M
22.	Explain different stages in testing process with neat block diagram?	8M
23.	Explain software evolution?	4M
24.	Differentiate between i) User and system requirements. ii) Functional and non functional requirements.	5M
25.	Describe functional and non functional requirement	10M
26.	Explain user and system requirement with example.	5M
27.	Explain the metrics to define non-functional requirements	6M
28.	With neat block diagram explain non functional requirement.	8M
29.	What is ethnography? How ethnography is effective in discovering the types of requirements?	8M

30.	**Explain the structure of requirement document Or Explain the IEEE standard for requirement document.**	**10M**
31.	**Write the users of requirement document.**	**5M**
32.	**What is requirement specification? Explain different notations used for writing system requirements Or explain different ways of writing a system requirement specification.**	**6M**
33.	**Write the guidelines in writing natural language requirement.**	**6M**
34.	**Explain different phases of requirement engineering process with neat diagram?**	**5M**
34.	**Explain the need for requirements elicitation and analysis. Explain different process activities involved.**	**8M**
35.	**Give different stake holders for** **i) Health care patient information system.** **ii) Automated university examination system.** **iii) Online shopping application (Amazon, flipkart etc…)**	**15M**
36.	**Explain Different techniques in Requirement Discovery** **Or** **Explain i) interviewing ii) Scenarios iii) Use cases.**	**8M**
37.	**Discuss the problems of using natural language for defining user and system requirements with examples? Explain how structured language overcome this.**	**8M**
38.	**Why requirements need to be validated? Explain checks made in requirements validated and also explain requirement validation technique, which can be used in conjunction or individual.**	**10M**
39.	**Mention reasons why changes are inevitable in requirement management?**	**4M**
40.	**Explain requirement management planning and 3 stages of requirement change management process.**	**7M**

Question bank
Module 2

Sl.no	Questions	Marks
1.	Explain Context models with example of MHC-PMS?	**6M**
2.	Write the process model of involuntary detention.	**5M**
3.	Explain different types of system models.	**10M**
4.	Mention two types of Interaction model? Explain any one.	**8M**
5.	Explain i) use case modeling ii) sequence diagrams(view patient information, transfer data)	**10M**
6.	Based on your experience with a bank ATM, draw a data flow diagram modeling the data processing involved when a customer withdraws cash from machine	**8M**
7.	Explain in detail Structural model?	**10M**
8.	List out and discuss two types of behavioral models.	**10M**
9.	Draw a state diagram (state machine) model of a simple microwave oven and also explain.	**10M**
10.	Explain the Model-driven Architecture	**8M**
11.	Explain RUP, describe various phases of RUP	**6M**
12.	Draw and explain the sequence and state diagram for a typical weather station	**10M**
13.	Draw a sequence diagram for withdrawing money from ATM.	**8M**
14.	Discuss in detail the various stages of object oriented design process, with a help of design models	**8M**
15.	What is design pattern? Explain four elements of design pattern.	**6M**
16.	Explain three types of implementation issues.	**10M**
17.	What is software reuse? State the general model of open source licenses.	**5M**
18.	Explain with a figure, the data flow model of an invoice processing system.	**10M**
19.	What is model driven engineering? State three types of abstract system model produced.	**5M**
20.	What are the things to be done for a design of object oriented system? How the objects are identified.	**6M**

NEW SCHEME

CS/IS62

USN | | | | | | | | | | |

Sixth Semester B.E. Degree Examination, July/August 2005

Computer Science / Information Science and Engineering

Software Engineering

me: 3 hrs.] [Max.Marks : 100

Note: *1. Answer any FIVE full questions.*
2. Answers to be specific and within the preview of subject matter.

1. (a) Explain how both the waterfall model and the protyping model can be accommodated in the spiral process model. (4 Marks)

   (b) Mention the six specific design process activities. Give explanation for two of them. (6 Marks)

   (c) Table 1.C gives the task duration for software project activities. Draw an activity chart. (10 Marks)

Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}
Duration in days	10	15	10	20	10	15	20	35	15	05	10	20	35	10	20
Dependencies	-	T_1	T_1, T_2	-	-	T_3, T_4	T_3	T_7	T_3, T_6	T_5, T_9	T_9	T_{10}	T_3, T_4	T_8, T_9	T_9, T_{14}

Table 1.C

2. (a) Give the IEEE standard format for requirement document. (7 Marks)

   (b) Indicate the principle stages of VORD. (7 Marks)

   (c) Highlight the importance of DFD in software engineering life cycle. (6 Marks)

3. (a) Mention four weaknesses of structured analysis methods. (4 Marks)

   (b) Draw evaluationary prototyping flow diagram and mention its two main advantages. (6 Marks)

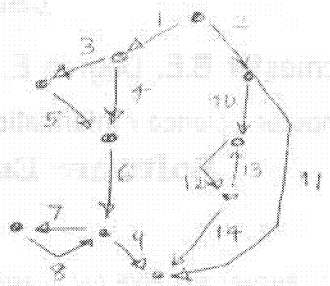
   (c) Bring out the advantages and disadvantages of a shared repository. (10 Marks)

4. (a) What are the four parameters of a system which affects the system architecture? Explain. (6 Marks)

   (b) Compare functional points and line of code with respect to a software life cycle. (14 Marks)

5. (a) Give the characteristics of GUI with description. (4 Marks)

(c) For the Figure5(c) shows a simple flow graph of a program. Indicate the minimal set of paths that satisfies white-box strategies. (10 Marks)



6. (a) Mention five levels in P-CMM model. Explain each of them. (8 Marks)

(b) Suggest meaningful names for the variables used in the program shown below and construct data dictionary entries for these names. (12 Marks)

```
routine BS(K,T,S,L)
T:=1
N×T : if S >= T go to CON
L= -1
go to STP
CON : L : =INTEGER (T/S)
L:=INTEGER(T+S)/2)
N×L : if T(L)=K then go to RTN
if T(L)>K then go to GRT
S:=L+1
go to N×T
GRT : S : = L -1
go to N×T
STP : end.
```


7. (a) For different forms of COCOMO give project complexity, formula and effort estimation plots. (10 Marks)

(b) Suppose that you are developing the software for a nuclear power plant control system. Select the most appropriate mode for the project and use the COCOMO model to give a crude estimate of the total number of person months required for the development, assuming that the estimated software size is 10,000 delivered source instructions. (10 Marks)

8. Write explanatory notes on :

(a) Ethnography

(b) Group cohesiveness

(c) Software equation

(d) Case Work benches. (4×5=20 Marks)

CS/IS6T2

USN | | | | | | | | | | |

Sixth Semester B.E. Degree Examination, January/February 2005

Computer Science/Information Science Engineering

Software Engineering

Time: 3 hrs.] [Max.Marks : 100

Note: *1. Answer any FIVE full questions.*
2. All questions carry equal marks.

1. (a) What is software engineering ? Explain the various process characteristics. (6 Marks)

   (b) With the help of a diagram explain Boehm's spiral model of the software process. What are its advantages over water fall method? (8 Marks)

   (c) Describe five different types of functional components that might be part of large - scale software system (6 Marks)

2. (a) What is requirement definition and specification ? With the help of a diagram explain the requirement engineering process. (6 Marks)

   (b) A software system is to be developed to automate a library catalogue. This system will contain information about all the books in a library and will be usable by library staff and by book borrowers and readers. The system should support catalogue browsing, querying, and should provide facilities allowing users to send messages to library staff reserving a book that is on loan. Identify the principal viewpoints which might be taken into account in the specification of this system. Show their relationships using a view point hierarchy diagram. (8 Marks)

   (c) Develop an object model including a class hierarchy diagram and an aggregation diagram showing the principal components of a personal computer system and its system software. (6 Marks)

3. (a) Describe three different types of non-functional requirements which may be placed on a system. Give examples of each of these different types of requirement. (6 Marks)

   (b) Explain why, for large system development, it is recommended that prototypes should be "throw-away" prototypes. (8 Marks)

   (c) Explain why it is important to use different notations to describe software design. (6 Marks)

4. (a) Explain why maximizing cohesion and minimizing coupling leads to more maintainable systems. What other attributes of a design might influence system maintainbility ? (6 Marks)

   (b) What is system structuring ? Explain different models in system structuring (8 Marks)

   (c) Design an architecture for an automated ticket issuing system used by passengers at a railway station, based on your choice of model. (6 Marks)

5. (a) Develop the design of the weather station design in detail by writing interface descriptions of the identified objects. Express it in $C++$ programming language. (10 Marks)

(c) Suggest situations in which it is unwise or impossible to provide a consistent user interface. (5 Marks)

6. (a) Suggest six reasons why software reliability is important. Using an example explain the difficulties of describing what software reliability means. (10 Marks)

(b) Write a set of guidelines for *C++* programmers which give advice on how to make functions resuable. (5 Marks)

(c) Explain fault tolerance. (5 Marks)

7. (a) Explain how back-to-back testing may be used to test their own programs in an objective way. (6 Marks)

(b) Discuss the differences between black-box and structural testing and suggest how they can be used together in the defect testing process. (8 Marks)

(c) Using your knowledge of *C++* programming language, derive a checklist of common errors (not syntax errors) which could not be detected by a compiler but which might be detected in a program inspection. (6 Marks)

8. (a) Briefly describe each activity in a software project plan. (6 Marks)

(b) What factors should be taken into account when selecting staff to work on a software development project? (8 Marks)

(c) In the development of large, embedded real time systems, suggest five factors which are likely to have a significant effect on the productivity of the software development team. (6 Marks)

NEW SCHEME

Reg. No.

Sixth Semester B.E. Degree Examination, January/February 2

Computer Science Information Science & Engineering

Software Engineering

Time: 3 hrs.) (Max.Mark

Note: *Answer any FIVE full questions.*

1. (a) Software is a product. Justify this statement. (4
   (b) Explain the different attributes of a good software. (6
   (c) Describe the salient features of spiral model of software process with an illu diagram. (10
2. (a) Explain the different stages in the testing process with a neat block diagrar (8
   (b) What are various metrics for specifying non-functional requirements? (4
   (c) Write the structure of a requirements document. (8

3. (a) Describe the requirements elicitation and analysis process with a neat figur (8
   (b) Explain the various types of checks to be carried out during requirements vali (7
   (c) What are different types of volatile requirements? (5
4. (a) Mention the several rapid prototyping techniques. Describe any one of the (6
   (b) Describe the suitability of interrupt - driven models for architectural design. (6
   (c) Explain the guidelines to be observed while designing user interface. (8
5. (a) How effectively could a colour be exploited in user interface design? (8
   (b) Describe the user interface evaluation process. (8
   (c) Define critical systems. Enumerate three types of critical systems. (4
6. (a) Explain the various reliability metrics. (8
   (b) Describe the characteristics of cleanroom software development. (8
   (c) Briefly explain the top-down and bottom-up testing processes. (4
7. (a) What types of plan are envisaged for project planning by management? (8
   (b) Describe the cost estimation techniques for software development. (8
   (c) Mention the various software product metrics. (4
8. (a) Describe the components of legacy systems with a block diagram. (10
   (b) Explain the activities involved in re-engineering process with an illustrative fi (10

** * **

CS62

USN | | | | | | | | | | |

NEW SCHEME

Sixth Semester B.E. Degree Examination, July 2007
CS / IS
Software Engineering

Time: 3 hrs.] [Max. Marks:100

Note : *Answer any FIVE full questions.*

1 a. Explain key challenges facing software engineering. **(04 Marks)**

b. What is process iteration? Describe the hybrid models of software development. **(10 Marks)**

c. Describe the general model of design process. **(06 Marks)**

2 a. Explain the structure of software requirements document. **(10 Marks)**

b. Why elicitation and analysis is a difficult process? Explain giving reasons. **(05 Marks)**

c. What problems might arise if requirements are not clearly understood and out on requirements in requirements document? **(05 Marks)**

3 a. What are the benefits of developing a system prototype? Explain. **(06 Marks)**

b. Describe a software process with throw away prototyping. What are the problems with this approach? **(08 Marks)**

c. What is a CASE workbench? Describe the tools included in an analysis and design workbench. **(06 Marks)**

4 a. What is modular decomposition? Explain dataflow model of an invoice processing system. **(05 Marks)**

b. Draw and explain sequence diagram and state diagram for a typical weather station. **(10 Marks)**

c. What are the guidelines that should be followed while using colour in a user interface? **(05 Marks)**

5 a. Describe the general inspection process. Also discuss possible inspection checks. **(08 Marks)**

b. Describe the metrics for specifying software reliability and availability. **(06 Marks)**

c. What is integration testing? Compare top down and bottom up testing. **(06 Marks)**

6 a. Explain the COCOMO2 costing model. **(10 Marks)**

b. Describe the project planning process, give pseudocode. **(05 Marks)**

c. Describe the factors affecting software engineering productivity. **(05 Marks)**

7 a. Which is the widely used method of validating the quality of process or product? Explain. **(06 Marks)**

b. Describe the static product metrics for assessing the quality attributes. **(08 Marks)**

c. Why assessment of legacy systems is required? Describe the strategies used for evolving these systems. **(06 Marks)**

8 Write short notes on :

CS62

USN

NEW SCHEME

Sixth Semester B.E. Degree Examination, Dec. 06 / Jan. 07
CS / IS
Software Engineering

Time: 3 hrs.] [Max. Marks:100

Note : *Answer any FIVE full questions.*

1 a. Describe the professional responsibilities of a software engineer. (05 Marks)
b. Describe the functional classification of CASE tools. (06 Marks)
c. Briefly discuss the Bohem's spiral model. Compare it with prototyping. (09 Marks)

2 a. Describe the functional and nonfunctional requirements with examples. (10 Marks)
b. With an example, explain the use of view point template and service template and service template in the VORD method. (06 Marks)
c. Identify four different matrix for specifying nonfunctional requirements. (04 Marks)

3 a. With an example describe the repository model and discuss its advantages and disadvantages. (08 Marks)
b. What are the advantages and disadvantages of evolutionary and throwaway prototyping? (06 Marks)
c. Explain [illegible] (06 Marks)

4 a. What are the advantages and disadvantages of client server model?
b. Discuss in detail both centralized and event based control models with examples.
c. What are the five different types of user and system documents supplied with any software systems?

5 a. Describe the characteristics of an OOAD, its advantages and explain the typical activities performed during this process. (10 Marks)
b. For each of the following three interaction styles, identify advantages, disadvantages and the application examples where they are used.
i) Direct manipulation
ii) Menu selection and
iii) Forms fill in. (06 Marks)
c. Briefly discuss four usability metrics. (04 Marks)

6 a. What is verification and validation? (02 Marks)
b. Explain static and dynamic testing technologies. (06 Marks)
c. Briefly explain with a diagram clean-room software development. (04 Marks)
d. Explain the differences between white box and black box testing. (08 Marks)

7 a. What is dependability? Precisely define the four factors. (06 Marks)
b. Explain POFOD, ROCOF with examples. (06 Marks)
c. Explain reverse engineering process. What are the advantages and disadvantages? (08 Marks)

8 a. What are the areas covered by ISO 9001 model for quality assurance? (10 Marks)
b. Explain software quality attributes. (05 Marks)
c. Identify the risks and risk types. (05 Marks)

*****

USN | | | | | | | | | | |

CS62

Sixth Semester B.E. Degree Examination, June/July 08
Software Engineering

Time: 3 hrs. Max. Marks:100

Note : ***Answer any FIVE full questions.***

1 a. What is Software? What are the attributes of a good software? **(05 Marks)**
b. Explain the requirements engineering process with a neat block diagram. **(06 Marks)**
c. Explain Bohem's spiral model with a neat diagram and compare it with prototyping. **(09 Marks)**

2 a. Explain briefly the metrics for specifying non-functional requirements. **(04 Marks)**
b. Explain the form – based approach for specifying system requirements. **(06 Marks)**
c. Explain the use of view – point, service template and event scenarios in VORD method with suitable example. **(10 Marks)**

3 a. Explain the state machine model of a simple microwave oven. **(08 Marks)**
b. What is a CASE Workbench? Describe the tools used in analysis and design of workbench. **(06 Marks)**
c. Explain any two rapid – prototyping techniques. **(06 Marks)**

4 a. Write an object and DFD model for an invoice processing system. **(08 Marks)**
b. Draw and explain the sequence diagram and state diagram for a typical weather station. **(12 Marks)**

5 a. Describe the principles of user interface design. **(06 Marks)**
b. How is safety achieved in a safety – critical software? Explain. **(06 Marks)**
c. Explain the iterative process of hazard and risk analysis. **(08 Marks)**

6 a. Explain the structure of a software test plan **(07 Marks)**
b. Explain the clean – room software development process with a neat diagram. **(07 Marks)**
c. Explain interface testing. **(06 Marks)**

7 a. What are the possible software risks? Explain briefly the risk management process. **(10 Marks)**
b. Explain briefly the factors affecting software pricing and software engineering productivity. **(10 Marks)**

8 a. Explain software product metrics **(06 Marks)**

USN | | | | | | | | | | |

CS62

Sixth Semester B.E. Degree Examination, Dec. 07 / Jan. 08
Software Engineering

Time: 3 hrs. Max. Marks:100

Note : *Answer any FIVE full questions.*

1 a. Explain the term software engineering. What are the key challenges that a software engineering is facing? **(05 Marks)**

b. What is software process model? Why is incremental model called hybrid model? Explain it with a neat diagram quoting its merits and demerits. **(10 Marks)**

c. Write a block diagram that illustrates classification of CASE from integration perspective. **(05 Marks)**

2 a. What is the objective of requirements engineering? Illustrate the various activities of requirements engineering with a neat diagram. **(06 Marks)**

b. Why is project planning an iterative activity? Briefly explain the purpose of each section in a project plan. **(08 Marks)**

c. From the evolution perspective classify the requirements of a software product **(06 Marks)**

3 a. Write the importance of requirements validation. List the various validation techniques and explain any one in detail. **(08 Marks)**

b. What are the problems in using natural language for specifying system requirements? Explain how structured natural language overcomes these problems with an example. **(10 Marks)**

c. What is [illegible] **(02 Marks)**

4 a. Based on your experience with a bank ATM draw a DFD modeling the processing involved when a customer withdraws cash from the machine. **(05 Marks)**

b. What are the benefits of developing a system prototype? Compare evolutionary prototyping with throwaway prototyping. **(10 Marks)**

c. What are control models? Write a brief note on call return control model. **(05 Marks)**

5 a. What are user interface design principles? **(06 Marks)**

b. Briefly outline the techniques for user interface evaluation. **(08 Marks)**

c. Define dependability of a computer system. What are the four principal dimensions of dependability? **(06 Marks)**

6 a. Which are the metrics available for specifying the reliability requirements quantitatively? **(04 Marks)**

b. What are the types of errors discovered through program inspection? **(06 Marks)**

c. Write the difference between black box testing and structural testing. With a suitable example explain black box testing approach. **(10 Marks)**

7 a. Explain the approach used by COCOMO model to estimate the person months for a software project. **(10 Marks)**

b. With a neat diagram explain the logical parts of a legacy system. **(10 Marks)**

8 Write short notes on :

a. Ethnography

b. Metrics for nonfunctional requirement

c. Stress testing

d. Clean room software development. **(20 Marks)**

USN | | | | | | | | | | |

06IS51

Fifth Semester B.E. Degree Examination, June-July 2009
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions choosing at least two from each part.***

Part A

1 a. What are the attributes of good software and what are the key challenges facing software engineering? **(10 Marks)**

b. Explain how both the waterfall model of the software process and the prototyping model can be accommodated in the spiral process model. **(05 Marks)**

c. Name the components of a design method. **(05 Marks)**

2 a. Describe four different types of non-functional requirement, which may be placed, on the system. Give examples of each of these types of requirements. **(10 Marks)**

b. During the requirement validation process, what are the different types of checks to be carried out on the requirements in the requirements document. Also what are the requirements validation techniques, which can be used in conjunction or individually? **(10 Marks)**

3 a. Based on your experience with a bank ATM, draw a data flow diagram modeling the data processing involved when a customer withdraws cash from a machine. **(10 Marks)**

b. Draw and explain the state machine model of a simple microwave oven. **(10 Marks)**

4 a. Table below sets out a number of activities, durations and dependencies. Draw an activity chart and a bar chart showing the project schedule. **(16 Marks)**

Task	Duration (days)	Dependencies
T_1	10	
T_2	15	T_1
T_3	10	T_1, T_2
T_4	20	
T_5	10	
T_6	15	T_3, T_4
T_7	20	T_3
T_8	35	T_7
T_9	15	T_6
T_{10}	5	T_5, T_9
T_{11}	10	T_9
T_{12}	20	T_{10}
T_{13}	35	T_3, T_4
T_{14}	10	T_8, T_9
T_{15}	20	T_{12}, T_{14}
T_{16}	10	T_{15}

Table Q4 (a)

b. What is the critical distinction between a milestone and a deliverable? **(04 Marks)**

Part B

5 a. What are the advantages and disadvantages of a shared repository model? **(10 Marks)**

b. Explain with figure the data-flow model of an invoice processing system. **(10 Marks)**

6	a.	List the various steps that need to be followed for object oriented design process.	(05 Marks)
	b.	What are the advantages and disadvantages of object oriented systems?	(05 Marks)
	c.	What are advantages and drawbacks of inheritance?	(05 Marks)
	d.	Explain about concurrent objects.	(05 Marks)
7	a.	What are the characteristics of rapid software development?	(05 Marks)
	b.	What is software prototyping? Give the benefits of software prototyping.	(05 Marks)
	c.	What is the objective of evolutionary prototyping? Give its advantages and the problems that are encountered.	(05 Marks)
	d.	What is an Agile method? Discuss the various principles used in Agile method.	(05 Marks)
8	a.	What are the various types of software maintenance?	(04 Marks)
	b.	Differentiate between Black Box testing and White Box testing.	(06 Marks)
	c.	Name and explain the factors affecting the productivity of software.	(05 Marks)
	d.	Name the various estimation techniques in software systems.	(05 Marks)

* * * * *

USN | | | | | | | | | | |

06IS51

Fifth Semester B.E. Degree Examination, June-July 2009
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions choosing at least two from each part.***

Part A

1 a. What are the attributes of good software and what are the key challenges facing software engineering? **(10 Marks)**

b. Explain how both the waterfall model of the software process and the prototyping model can be accommodated in the spiral process model. **(05 Marks)**

c. Name the components of a design method. **(05 Marks)**

2 a. Describe four different types of non-functional requirement, which may be placed, on the system. Give examples of each of these types of requirements. **(10 Marks)**

b. During the requirement validation process, what are the different types of checks to be carried out on the requirements in the requirements document. Also what are the requirements validation techniques, which can be used in conjunction or individually? **(10 Marks)**

3 a. Based on your experience with a bank ATM, draw a data flow diagram modeling the data processing involved when a customer withdraws cash from a machine. **(10 Marks)**

b. Draw and explain the state machine model of a simple microwave oven. **(10 Marks)**

4 a. Table below sets out a number of activities, durations and dependencies. Draw an activity chart and a bar chart showing the project schedule. **(16 Marks)**

Task	Duration (days)	Dependencies
T_1	10	
T_2	15	T_1
T_3	10	T_1, T_2
T_4	20	
T_5	10	
T_6	15	T_3, T_4
T_7	20	T_3
T_8	35	T_7
T_9	15	T_6
T_{10}	5	T_5, T_9
T_{11}	10	T_9
T_{12}	20	T_{10}
T_{13}	35	T_3, T_4
T_{14}	10	T_8, T_9
T_{15}	20	T_{12}, T_{14}
T_{16}	10	T_{15}

Table Q4 (a)

b. What is the critical distinction between a milestone and a deliverable? **(04 Marks)**

Part B

5 a. What are the advantages and disadvantages of a shared repository model? **(10 Marks)**

b. Explain with figure the data-flow model of an invoice processing system. **(10 Marks)**

6	a.	List the various steps that need to be followed for object oriented design process.	(05 Marks)
	b.	What are the advantages and disadvantages of object oriented systems?	(05 Marks)
	c.	What are advantages and drawbacks of inheritance?	(05 Marks)
	d.	Explain about concurrent objects.	(05 Marks)
7	a.	What are the characteristics of rapid software development?	(05 Marks)
	b.	What is software prototyping? Give the benefits of software prototyping.	(05 Marks)
	c.	What is the objective of evolutionary prototyping? Give its advantages and the problems that are encountered.	(05 Marks)
	d.	What is an Agile method? Discuss the various principles used in Agile method.	(05 Marks)
8	a.	What are the various types of software maintenance?	(04 Marks)
	b.	Differentiate between Black Box testing and White Box testing.	(06 Marks)
	c.	Name and explain the factors affecting the productivity of software.	(05 Marks)
	d.	Name the various estimation techniques in software systems.	(05 Marks)

* * * * *

2002 SCHEME

USN | | | | | | | | | | |

CS6

Sixth Semester B.E. Degree Examination, Dec.08 / Jan.09
Software Engineering

Time: 3 hrs. Max. Marks:100

Note : ***Answer any FIVE full questions.***

1 a. What is Software Engineering? What are the key challenges faced by software engineering? Explain. **(06 Marks)**

b. State and explain different professional and ethical responsibility of the software engineer. **(06 Marks)**

c. Describe the waterfall model with neat block diagram. Give its limitations. **(08 Marks)**

2 a. Explain the IEEE standard structure for requirement documents. **(06 Marks)**

b. Discuss the sequence of activities involved in requirements engineering process. **(08 Marks)**

c. What is software prototyping? What are its advantages and disadvantages? **(06 Marks)**

3 a. Differentiate between functional and non-functional requirements. **(04 Marks)**

b. What are behavioral models? Draw a data flow diagram for order processing. **(08 Marks)**

c. Explain different types of domain specific architectural models. **(08 Marks)**

4 a. With a neat block diagram, explain the user interface design process. **(08 Marks)**

b. What are the common activities involved in architectural design process? Explain. **(06 Marks)**

c. Explain different usability attributes of user interface evaluation. **(06 Marks)**

5 a. Discuss the role of flow graph in software testing with example. **(08 Marks)**

b. Describe the clean room software development with block diagram. **(07 Marks)**

c. Differentiate between software inspection process and software testing process. **(05 Marks)**

6 a. Explain the various approaches to improve the reliability of a critical system. **(06 Marks)**

b. State and explain reliability metrics in brief. **(06 Marks)**

c. With a neat block diagram explain risk management process. **(08 Marks)**

7 a. Explain the factors affecting the software pricing. **(07 Marks)**

b. Why software standards are important? Explain. **(06 Marks)**

c. Explain the different components of legacy systems. **(07 Marks)**

8 Write short notes on :

a. Object oriented design process

b. Reverse engineering

c. Project planning

d. Case workbenches. **(20 Marks)**

USN | | | | | | | | | | |

06IS51

Fifth Semester B.E. Degree Examination, June-July 2009
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions choosing at least two from each part.***

Part A

1 a. What are the attributes of good software and what are the key challenges facing software engineering? **(10 Marks)**

b. Explain how both the waterfall model of the software process and the prototyping model can be accommodated in the spiral process model. **(05 Marks)**

c. Name the components of a design method. **(05 Marks)**

2 a. Describe four different types of non-functional requirement, which may be placed, on the system. Give examples of each of these types of requirements. **(10 Marks)**

b. During the requirement validation process, what are the different types of checks to be carried out on the requirements in the requirements document. Also what are the requirements validation techniques, which can be used in conjunction or individually? **(10 Marks)**

3 a. Based on your experience with a bank ATM, draw a data flow diagram modeling the data processing involved when a customer withdraws cash from a machine. **(10 Marks)**

b. Draw and explain the state machine model of a simple microwave oven. **(10 Marks)**

4 a. Table below sets out a number of activities, durations and dependencies. Draw an activity chart and a bar chart showing the project schedule. **(16 Marks)**

Task	Duration (days)	Dependencies
T_1	10	
T_2	15	T_1
T_3	10	T_1,T_2
T_4	20	
T_5	10	
T_6	15	T_3,T_4
T_7	20	T_3
T_8	35	T_7
T_9	15	T_6
T_{10}	5	T_5,T_9
T_{11}	10	T_9
T_{12}	20	T_{10}
T_{13}	35	T_3,T_4
T_{14}	10	T_8,T_9
T_{15}	20	T_{12},T_{14}
T_{16}	10	T_{15}

Table Q4 (a)

b. What is the critical distinction between a milestone and a deliverable? **(04 Marks)**

Part B

5 a. What are the advantages and disadvantages of a shared repository model? **(10 Marks)**

b. Explain with figure the data-flow model of an invoice processing system. **(10 Marks)**

6 a. List the various steps that need to be followed for object oriented design process. (05 Marks)
b. What are the advantages and disadvantages of object oriented systems? (05 Marks)
c. What are advantages and drawbacks of inheritance? (05 Marks)
d. Explain about concurrent objects. (05 Marks)

7 a. What are the characteristics of rapid software development? (05 Marks)
b. What is software prototyping? Give the benefits of software prototyping. (05 Marks)
c. What is the objective of evolutionary prototyping? Give its advantages and the problems that are encountered. (05 Marks)
d. What is an Agile method? Discuss the various principles used in Agile method. (05 Marks)

8 a. What are the various types of software maintenance? (04 Marks)
b. Differentiate between Black Box testing and White Box testing. (06 Marks)
c. Name and explain the factors affecting the productivity of software. (05 Marks)
d. Name the various estimation techniques in software systems. (05 Marks)

* * * * *

USN | | | | | | | | | | |

06IS51

Fifth Semester B.E. Degree Examination, June-July 2009
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions choosing at least two from each part.***

Part A

1 a. What are the attributes of good software and what are the key challenges facing software engineering? **(10 Marks)**

b. Explain how both the waterfall model of the software process and the prototyping model can be accommodated in the spiral process model. **(05 Marks)**

c. Name the components of a design method. **(05 Marks)**

2 a. Describe four different types of non-functional requirement, which may be placed, on the system. Give examples of each of these types of requirements. **(10 Marks)**

b. During the requirement validation process, what are the different types of checks to be carried out on the requirements in the requirements document. Also what are the requirements validation techniques, which can be used in conjunction or individually? **(10 Marks)**

3 a. Based on your experience with a bank ATM, draw a data flow diagram modeling the data processing involved when a customer withdraws cash from a machine. **(10 Marks)**

b. Draw and explain the state machine model of a simple microwave oven. **(10 Marks)**

4 a. Table below sets out a number of activities, durations and dependencies. Draw an activity chart and a bar chart showing the project schedule. **(16 Marks)**

Task	Duration (days)	Dependencies
T_1	10	
T_2	15	T_1
T_3	10	T_1,T_2
T_4	20	
T_5	10	
T_6	15	T_3,T_4
T_7	20	T_3
T_8	35	T_7
T_9	15	T_6
T_{10}	5	T_5,T_9
T_{11}	10	T_9
T_{12}	20	T_{10}
T_{13}	35	T_3,T_4
T_{14}	10	T_8,T_9
T_{15}	20	T_{12},T_{14}
T_{16}	10	T_{15}

Table Q4 (a)

b. What is the critical distinction between a milestone and a deliverable? **(04 Marks)**

Part B

5 a. What are the advantages and disadvantages of a shared repository model? **(10 Marks)**

b. Explain with figure the data-flow model of an invoice processing system. **(10 Marks)**

6 a. List the various steps that need to be followed for object oriented design process. (05 Marks)
b. What are the advantages and disadvantages of object oriented systems? (05 Marks)
c. What are advantages and drawbacks of inheritance? (05 Marks)
d. Explain about concurrent objects. (05 Marks)

7 a. What are the characteristics of rapid software development? (05 Marks)
b. What is software prototyping? Give the benefits of software prototyping. (05 Marks)
c. What is the objective of evolutionary prototyping? Give its advantages and the problems that are encountered. (05 Marks)
d. What is an Agile method? Discuss the various principles used in Agile method. (05 Marks)

8 a. What are the various types of software maintenance? (04 Marks)
b. Differentiate between Black Box testing and White Box testing. (06 Marks)
c. Name and explain the factors affecting the productivity of software. (05 Marks)
d. Name the various estimation techniques in software systems. (05 Marks)

* * * * *

2002 SCHEME

USN | | | | | | | | | | |

Sixth Semester B.E. Degree Examination, Dec.08 / Jan.09
Software Engineering

Time: 3 hrs. Max. Marks:

Note : ***Answer any FIVE full questions.***

1 a. What is Software Engineering? What are the key challenges faced by soft engineering? Explain. (06 M
b. State and explain different professional and ethical responsibility of the software engi (06 M
c. Describe the waterfall model with neat block diagram. Give its limitations. (08 M

2 a. Explain the IEEE standard structure for requirement documents. (06 M
b. Discuss the sequence of activities involved in requirements engineering process. (08 M
c. What is software prototyping? What are its advantages and disadvantages? (06 M

3 a. Differentiate between functional and non-functional requirements. (04 M
b. What are behavioral models? Draw a data flow diagram for order processing. (08 M
c. Explain different types of domain specific architectural models. (08 M

4 a. With a neat block diagram, explain the user interface design process. (08 M
b. What are the common activities involved in architectural design process? Explain. (06 M
c. Explain different usability attributes of user interface evaluation. (06 M

5 a. Discuss the role of flow graph in software testing with example. (08 M
b. Describe the clean room software development with block diagram. (07 M
c. Differentiate between software inspection process and software testing process. (05 M

6 a. Explain the various approaches to improve the reliability of a critical system. (06 M
b. State and explain reliability metrics in brief. (06 M
c. With a neat block diagram explain risk management process. (08 M

7 a. Explain the factors affecting the software pricing. (07 M
b. Why software standards are important? Explain. (06 M
c. Explain the different components of legacy systems. (07 M

8 Write short notes on :
a. Object oriented design process
b. Reverse engineering
c. Project planning
d. Case workbenches. (20 M

USN | 1 | S | B | 0 | 7 | C | S | 1 | 2 | 4 |

06CS/IS51

Fifth Semester B.E. Degree Examination, December 2010

Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions, selecting at least TWO questions from each part.***

PART – A

1 a. What are the attributes of good software? What are the key challenges facing software engineering? **(10 Marks)**

b. Describe the general model of design process. **(06 Marks)**

c. Explain the requirements engineering process, with a neat block diagram. **(04 Marks)**

2 a. Describe four different types of non-functional requirement, which may be placed, on the systems. Give examples of each of these types of requirements. **(10 Marks)**

b. Describe the salient features of spiral model of software process, with an illustration diagram. **(10 Marks)**

3 a. With a neat block diagram, explain components of a CASE TOOLS for structured method support. **(10 Marks)**

b. What are the most important dimensions of system dependability? **(06 Marks)**

c. What is requirement elicitation and analysis? Explain. **(04 Marks)**

4 a. Explain state machine model for a simple microwave oven. **(10 Marks)**

b. Write the structure of a requirement document suggest by IEEE standard. **(05 Marks)**

c. What is object aggregation? Explain with an example. **(05 Marks)**

PART – B

5 a. Explain with a figure, the data flow model of an invoice processing system. **(10 Marks)**

b. Draw and explain the sequence and state diagram for a typical weather station. **(10 Marks)**

6 a. Explain the structure of a software test plan. **(07 Marks)**

b. Give a brief description of five principles of agile methods. **(07 Marks)**

c. Discuss the advantages of pair programming. **(06 Marks)**

7 a. Explain the characteristics of clean room software development. **(07 Marks)**

b. What are the characteristics of rapid software development? **(07 Marks)**

c. What is software prototyping? Give benefits of software prototyping. **(06 Marks)**

8 a. Differentiate between black box testing and white box testing. **(07 Marks)**

b. List the factors governing staff selection. **(07 Marks)**

c. Name the various estimation techniques in software systems. **(06 Marks)**

* * * * *

2002 SCHEME

USN | | | | | | | | | | |

CS62

Sixth Semester B.E. Degree Examination, December 2010
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions.***

1 a. What are the four important attributes which all software products should have? Suggest four other attributes which may be significant. **(04 Marks)**

b. Explain the general model of the design process. **(06 Marks)**

c. With the help of a neat diagram, briefly explain Boehm's spiral model of a software process. What are its advantages over waterfall model? **(07 Marks)**

d. "Programs developed using evolutionary development are likely to be difficult to maintain". Justify. **(03 Marks)**

2 a. Explain rapid prototyping in software development. List the rapid development techniques that are practical for developing industrial strength prototypes. Explain how component and application assembly helps in rapid prototyping. **(08 Marks)**

b. Explain the structure of a requirement document. **(06 Marks)**

c. Explain the principal stages of VORD method. **(06 Marks)**

3 a. With an example, describe the client / server model and discuss its advantages and disadvantages. **(07 Marks)**

b. Design the state chart of a weather station object, that shows how it responds to request for various services. **(07 Marks)**

c. Give the user interface design principle. **(06 Marks)**

4 a. What do you mean by clean room software development? Explain the key characteristics the clean room approach is based on, with neat figure. Which are the three teams involved when the clean room process is used for a large system? **(09 Marks)**

b. Give the structure of the software test plan. **(06 Marks)**

c. List the different roles played by the team in the inspection process. **(05 Marks)**

5 a. Distinguish between black box testing and structural testing. **(07 Marks)**

b. Explain the general guidelines for interface testing. **(06 Marks)**

c. What is cyclomatic complexity? Figure shows the simple connected graph G of a program. Find the cyclomatic complexity and indicate the minimal set of paths. **(07 Marks)**

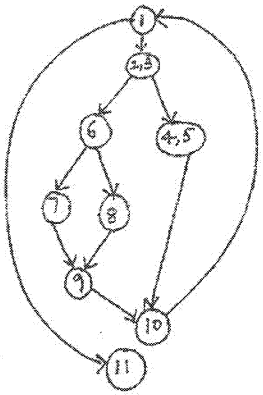


Fig. Q5(c)

6 a. Table below sets out the number of activities, duration and dependencies. Draw an activity chart showing the project schedule. Find the critical path. **(10 Marks)**

Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}
Duration (days)	8	15	15	10	10	5	20	25	15	15	7	10
Dependencies			T_1 (M_1)		T_2T_4 (M_2)	T_1T_2 (M_3)	T_1 (M_1)	T_4 (M_5)	T_3T_6 (M_4)	T_5T_7 (M_7)	T_9 (M_6)	T_{11} (M_8)

b. Project plans vary depending upon the type of project and the organization. Give the structure of a project plan that most plans must include. **(08 Marks)**

c. Define Milestone and Deliverable. **(02 Marks)**

7 a. List the factors affecting software pricing. **(06 Marks)**

b. Explain the approach used by basic COCOMO81 to estimate person month for a software development. Give the features of COCOMO model. **(08 Marks)**

c. List the areas covered by ISO 9001 model for quality assurance. **(06 Marks)**

8 Write short notes on :

a. Data dictionary. **(05 Marks)**

b. Use cases for library system. **(05 Marks)**

c. Legacy system. **(05 Marks)**

d. Reverse engineering. **(05 Marks)**

*****

(12) Q. papers

USN | | | | | | | | | | |

12/1

06IS51

Fifth Semester B.E. Degree Examination, May/June 2010
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions, selecting at least TWO questions from each part.***

PART – A

1 a. What are the key challenges facing software engineering? Explain. **(04 Marks)**

b. What are emergent system properties? Give examples for each. **(04 Marks)**

c. With a figure, explain the requirements of an engineering process. **(12 Marks)**

2 a. With a figure, explain the phases in the RUP. **(05 Marks)**

b. Explain the functional and non-functional requirements for any system. **(10 Marks)**

c. Give the number of possible metrics to specify non-functional system properties. **(05 Marks)**

3 a. What is an architectural design? Explain the architectural design decisions. **(06 Marks)**

b. Why requirements need to be validated? Explain the check made in requirement validation. **(06 Marks)**

c. Explain the requirement elicitation and analysis phase, with spiral diagram. Give reasons, why is it difficult phase in requirements engineering process. **(08 Marks)**

4 a. Explain the IEEE standard format for the requirement document in detail. **(06 Marks)**

b. Draw and explain the use-case diagram and sequence diagram for a library system or ATM withdraw system. **(06 Marks)**

c. Refer table below for task durations and interdependencies:

Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
Duration in days	9	16	11	15	7	20	26	15	15	16
Interdependencies	-	-	-	T_1 (M_1)	T_1T_2 (M_2)	T_2T_3 (M_3)	T_3 (M_5)	T_4T_5 (M_4)	T_5T_6 (M_6)	T_8 (M_7)

i) Draw activity network ii) Find and highlight critical path. **(08 Marks)**

PART – B

5 a. Name and explain the three organizational styles that are very widely used, with necessary figure. **(10 Marks)**

b. Explain with a figure, the central control and event based control system. **(10 Marks)**

6 a. What are agile methods? Discuss the principles of agile methods. **(07 Marks)**

b. What are the practices followed in extreme programming? **(06 Marks)**

c. With a figure, explain the process of prototype development. What are the benefits of using prototyping? **(07 Marks)**

7 a. What is verification and validation? Explain why validation is a particularly difficult process. **(05 Marks)**

b. Explain the software development process model, using V-model with figure. **(10 Marks)**

c. The clean room approach to software development is based on five key strategies. Explain them. **(05 Marks)**

8 a. Name and explain the factors governing staff selection. **(10 Marks)**

b. Explain with a figure, the people capability maturity model. **(10 Marks)**

* * * * *

USN | 1 | S | B | 0 | 7 | C | S | 0 | 1 | 2 |

06IS51

Fifth Semester B.E. Degree Examination, Dec.09/Jan.10

Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions, selecting at least TWO questions from each part.***

PART – A

1 a. What are the attributes of a good software? Explain. **(06 Marks)**

b. Mention the different stages in a system development. Explain any four phases. **(10 Marks)**

c. Define and distinguish between the system reliability and availability. **(04 Marks)**

2 a. What is process iteration? Explain Boehm's spiral model. **(10 Marks)**

b. With an example, explain the functional and non-functional requirements. **(10 Marks)**

3 a. Explain the need for requirements elicitation and analysis. Explain the different process activities involved. **(10 Marks)**

b. Why risk management is important in project management? Explain different stages in risk management. **(10 Marks)**

4 a. What is data flow model? With an example, show the notations used in data flow model. **(10 Marks)**

b. Explain the terms :

i) Domain requirements **(03 Marks)**

ii) User requirements **(03 Marks)**

iii) System requirements. **(04 Marks)**

PART – B

5 a. Explain why it is necessary to design the system architecture. What are the system factors affected by system architecture? Explain. **(10 Marks)**

b. Distinguish between an object and an object class. Give example. **(06 Marks)**

c. What are concurrent objects? Explain different kinds of concurrent object implementations. **(04 Marks)**

6 a. What is rapid delivery and deployment of new systems? Explain its importance. **(10 Marks)**

b. What are the different types of software maintenance? What are the key factors that distinguish development and maintenance? **(10 Marks)**

7 a. Distinguish between software inspection and testing. What are the advantages of inspection over testing? **(08 Marks)**

b. Explain with illustrations :

i) Integration testing **(06 Marks)**

ii) Release testing **(06 Marks)**

8 a. Explain Maslow's human-needs hierarchy of motivating people. **(10 Marks)**

b. What are the factors affecting software pricing? What are the two types of metrics used? Explain. **(10 Marks)**

* * * * *

2002 SCHEME

USN | | | | | | | | | | |

CS62

Sixth Semester B.E. Degree Examination, Dec.09-Jan.10
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions.***

1 a. Differentiate between program and product. Write the classification of software products. **(05 Marks)**

b. List the fundamental activities common to software processes. Explain any one hybrid process model. **(10 Marks)**

c. Define software engineering. What are the key challenges faced by software engineering? **(05 Marks)**

2 a. Differentiate between system requirement and user requirement. Explain the classification of system requirements giving suitable examples. **(08 Marks)**

b. With an example explain system requirement specification using PDL. **(05 Marks)**

c. With a neat process diagram explain requirements elicitation and analysis. **(07 Marks)**

3 a. What is risk management? Explain the risk management process with a neat process diagram. **(10 Marks)**

b. With suitable example explain the purpose of the following graphical tools used by software project manager :

i) Activity network diagram ii) Gantt charts

iii) Staff allocation Vs time chart **(10 Marks)**

4 a. What is the purpose of state machine model? Give the state machine model of a simple microwave oven. **(10 Marks)**

b. What are the benefits of proto typing? Compare Evolutionary proto typing with throwaway prototyping. **(10 Marks)**

5 a. Give atleast give examples of nonfunctional requirements that has impact on system architecture. Describe how they affect system architecture. **(10 Marks)**

b. Explain the repository model clearly indicating its advantages and disadvantages. **(06 Marks)**

c. Describe any four user interaction styles. **(04 Marks)**

6 a. What are the different techniques for evaluating the user interface? What are the usability attributes that are taken into consideration while evaluating interface? **(10 Marks)**

b. What is dependability? Explain four principal dimensions of dependability. **(05 Marks)**

c. Give a line of description for each of the following terminology :

i) System failure ii) System error iii) System fault

iv) Vulnerability v) Attack **(05 Marks)**

7 a. What are the objectives of program inspection? Explain inspection with a neat process diagram. **(08 Marks)**

b. Differentiate between functional testing and structural testing. Derive test cases for a simple Linear Search routine using functional testing approach. **(12 Marks)**

8 Write short notes on the following :

a. COCOMO model

b. Legacy system components

c) Software re-engineering process

d) Clean room software development **(20 Marks)**

* * * * *

USN | | | | | | | | | | |

06IS51

Fifth Semester B.E. Degree Examination, December 2011

Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions, selecting at least TWO questions from each part.***

PART – A

1 a. What is software? Explain the essential attributes of a good software. **(05 Marks)**

b. List and explain any five software engineering code of ethics and professional practices. **(05 Marks)**

c. Describe briefly the phases of the system engineering process, with a neat diagram. **(10 Marks)**

2 a. Define critical systems. Explain the four principle dimensions of system dependability. **(05 Marks)**

b. Briefly discuss the reliability terminologies and mention the approaches to system reliability enhancement. **(05 Marks)**

c. What is software process? With a neat diagram, explain the software design process activities, in detail. **(10 Marks)**

3 a. What are the different metrics for specifying non-functional requirements? Explain any two of them. **(05 Marks)**

b. Write the IEEE standard format for requirement document. **(05 Marks)**

c. Give reasons why requirement elicitation and analysis is a difficult phase in requirements engineering process. **(05 Marks)**

d. What are volatile requirements? Briefly discuss the classification of volatile requirements. **(05 Marks)**

4 a. Draw and explain the sequence diagram for ATM system. **(08 Marks)**

b. Mention the weaknesses of structured methods when used to produce system models. **(04 Marks)**

c. Explain the risk management process, with a neat diagram. **(08 Marks)**

PART – B

5 a. Define architectural design. With an example, describe the repository model and give its advantages and disadvantages. **(08 Marks)**

b. Briefly discuss the architectural design decisions. **(06 Marks)**

c. Draw and explain the state diagram for weather station system. **(06 Marks)**

6 a. What is pair programming? Highlight its advantages. **(04 Marks)**

b. Explain with a diagram, rapid application development environment. **(06 Marks)**

c. Explain the activities involved in reengineering process, with an illustrative figure. **(10 Marks)**

7 a. Briefly discuss some of the automated static analysis checks. **(05 Marks)**

b. Explain the five key strategies of clean room software development. **(05 Marks)**

c. What is test automation? Explain with figure the tools that might be included in a testing workbench. **(10 Marks)**

8 a. Name and explain any five factors governing staff selection. **(05 Marks)**

b. Briefly discuss the advantages and disadvantages of group cohesiveness that influence group working. **(05 Marks)**

c. Explain in detail algorithmic cost models in project planning. **(10 Marks)**

* * * * *

2002 SCHEME

USN | | | | | | | | | | |

CS62

Sixth Semester B.E. Degree Examination, December 2011
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: *Answer FIVE full questions*

1 a. "Software is a product". Justify this statement. **(04 Marks)**
b. Explain the key challenges facing software engineering. **(06 Marks)**
c. How spiral model is different from other process models? Explain with an illustrative diagram. **(10 Marks)**

2 a. Why requirement elicitation is a difficult task? Justify your answer. **(06 Marks)**
b. Write a note on CASE. **(04 Marks)**
c. Briefly explain the structure of software requirement specification. **(10 Marks)**

3 a. Explain the different techniques of prototyping in software development process? **(10 Marks)**
b. Describe the importance of DFD in software engineering life cycle. **(06 Marks)**
c. Why is detailed design required? Explain. **(04 Marks)**

4 a. Describe the suitability of interrupt – driven models for architectural design. **(06 Marks)**
b. Explain the guidelines to be observed while designing the user interface. **(08 Marks)**
c. Describe the repository model with an example. List out the merits and demerits. **(06 Marks)**

5 a. What is the difference between verification and validation? **(02 Marks)**
b. Differentiate between block box and white box testing. **(06 Marks)**
c. Briefly explain the topdown and bottom up integration testing. **(04 Marks)**
d. Explain the interface type and interface errors in interface testing. **(08 Marks)**

6 a. Explain the various reliability – metrics. **(08 Marks)**
b. Explain the characteristics of clean – room software – development. **(08 Marks)**
c. Define critical systems. Enumerate the three types of critical systems. **(04 Marks)**

7 a. Explain the cocomo model for cost estimation. **(10 Marks)**
b. List out the various project management activities. Explain the project scheduling process. **(10 Marks)**

8 Write short notes on :
a. Software evolution
b. Ethnography
c. Software reuse
d. CASE – workbenches. **(20 Marks)**

* * * * *

2002 SCHEME

USN | | | | | | | | | | |

CS62

~~Sixth~~ V Semester B.E. Degree Examination, June/July 2011
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions.***

1 a. What is the aim of software engineering? What are the essential product attributes of software engineering? **(05 Marks)**

b. What are the major shortcomings that we might face, if we use the classical waterfall model for developing a software? **(05 Marks)**

c. Describe a risk free iterative process of software development process model, with a suitable diagram. **(10 Marks)**

2 a. Discuss the various functional and non-functional requirements, with reference to the university library system. **(10 Marks)**

b. What are the various types of checks that should be carried out during the requirement validation process? **(05 Marks)**

c. What are the various requirements of validation techniques? Briefly discuss any one of them. **(05 Marks)**

3 a. Discuss the rapid prototyping technique using reusable component with an example. Suggest problems that may arise using this approach. **(10 Marks)**

b. Briefly explain various control models used in the design of software system. List the advantages and disadvantages of each model. **(10 Marks)**

4 a. What are the merits and demerits of object-oriented design? **(06 Marks)**

b. Discuss the role of UML in an object oriented design. **(04 Marks)**

c. What are the fundamentals of the design of the user support that is embedded in software, while dealing with user interface design? **(10 Marks)**

5 a. With a suitable example, list the difference between verification and validation. **(05 Marks)**

b. What do you understand by clean room software development? Explain the clean room software development process, with neat diagrams. **(10 Marks)**

c. With an illustration, discuss the integration testing. **(05 Marks)**

6 a. What are the three important classes of critical systems? Discuss briefly all of them. **(10 Marks)**

b. Discuss the principal tasks of software project managers. **(05 Marks)**

c. Explain the algorithmic cost estimation model. **(05 Marks)**

7 a. What is the importance of standards in the quality management process? **(05 Marks)**

b. What is quality planning? What are the software quality attributes? **(05 Marks)**

c. How are graphical representations like bar charts and activity charts used by project managers to represent project schedules? **(10 Marks)**

8 a. What is a "legacy system"? What are the different logical parts of a legacy system? Explain their relationships. **(10 Marks)**

b. Briefly explain the activities involved in re-engineering process, with a block diagram. **(08 Marks)**

c. Why re-engineering is an expensive process? **(02 Marks)**

* * * * *

USN | | | | | | | | | | |

06IS51

Fifth Semester B.E. Degree Examination, June/July 2011
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer FIVE full questions selecting at least TWO questions from each part.***

PART – A

1 a. What is software engineering? What are the differences between generic and customized software products? **(06 Marks)**

b. With a block diagram, briefly explain the different phases of systems engineering process. **(14 Marks)**

2 a. What are the three main types of critical systems? Explain. **(06 Marks)**

b. With a neat diagram, describe the waterfall model of software development process. **(10 Marks)**

c. Explain the most important dimensions of the system dependability? **(04 Marks)**

3 a. Explain the structure of software requirements document? **(08 Marks)**

b. Explain requirements elicitation and analysis process. **(06 Marks)**

c. What are the requirements validation techniques? Explain briefly. **(06 Marks)**

4 a. Briefly explain the purpose of each of the sections in a software project plan. **(10 Marks)**

b. With diagram, briefly explain the components of a CASE tool for structured method support. **(10 Marks)**

PART – B

5 a. With an example, describe the repository model and give its advantages and disadvantages. **(10 Marks)**

b. Draw and explain sequence diagram and state diagram for a typical weather station. **(10 Marks)**

6 a. Briefly explain the tools that are included in rapid application development (RAD) environment, with a diagram. **(10 Marks)**

b. With a block diagram, explain the activities of re – engineering process. **(10 Marks)**

7 a. Explain the roles of inspection process and discuss the possible inspection checks. **(10 Marks)**

b. Explain the different types of interfaces between program components. **(04 Marks)**

c. Briefly explain with a diagram, clean – room software development. **(06 Marks)**

8 a. Briefly explain the factors that may influence the decision of selecting staff for the project team. **(05 Marks)**

b. Explain the different cost estimation techniques briefly. **(05 Marks)**

c. Explain the basic COCOMO model with formula for different types of projects. **(10 Marks)**

* * * * *

USN | | | | | | | | | | |

10IS51

Fifth Semester B.E. Degree Examination, December 2012
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer FIVE full questions, selecting at least TWO questions from each part.***

PART – A

1 a. What are the attributes of a good software? Explain. Also list and explain the key challenges facing software engineering. **(10 Marks)**

b. Explain with block diagram, the system engineering process. **(10 Marks)**

2 a. Define the term dependability. List and explain the various dependability properties. **(10 Marks)**

b. What is a software process model? Explain with a block diagram the evolutionary development model. **(10 Marks)**

3 a. Distinguish between functional and non-functional requirements. With a block diagram, explain non-functional requirement types. **(10 Marks)**

b. List at least five stake holders for an automated university examination system. Classify the identified stake holders under different view points. **(10 Marks)**

4 Write short notes on:

a. Context models.

b. Object models.

c. Project scheduling.

d. Risk management. **(20 Marks)**

PART – B

5 Explain the terms:

a. Architectural design decisions.

b. The repository model.

c. Unified modeling language (UML).

d. Sequence models. **(20 Marks)**

6 a. List and explain the principles of agile methods. Also explain the problems with agile methods. **(10 Marks)**

b. Define "Program Evolution Dynamics". Discuss the Lehman laws for program evolution dynamics. **(10 Marks)**

7 a. Explain the various inspection roles and inspection checklists for software inspection process. **(10 Marks)**

b. What is partition testing? Identify equivalence class partitions for automated air conditioning system having at least four partitions. List also the boundary values for each class. **(10 Marks)**

8 a. Define people capability maturity model (PCMM). With a block diagram, explain various P-CMM levels. **(10 Marks)**

b. List and explain various COCOMO cost estimation models. **(10 Marks)**

* * * * *

USN | | | | | | | | | | |

06IS51

Fifth Semester B.E. Degree Examination, December 2012
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer FIVE full questions, selecting at least TWO questions from each part.***

PART – A

1 a. Explain the term software engineering. List and explain the key challenges that a software engineering is facing. **(06 Marks)**

b. With a neat diagram, explain the system engineering process. **(08 Marks)**

c. What are the professional responsibilities of a software engineer? **(06 Marks)**

2 a. What are critical systems? Explain different types of critical systems. **(06 Marks)**

b. Explain Boehm's spiral model of software process with a neat diagram. **(08 Marks)**

c. With a neat diagram, describe the component based software engineering. **(06 Marks)**

3 a. Differentiate between functional and non-functional requirements. Give examples for each. **(05 Marks)**

b. List and explain the metrics used for specifying non-functional requirements. **(06 Marks)**

c. Write a structure of requirement documents suggested by IEEE standards. **(04 Marks)**

d. Explain the requirements engineering process with a neat diagram. **(05 Marks)**

4 a. Describe different types of system models. **(05 Marks)**

b. Based on your experience with a bank ATM, draw a data flow diagram modeling the data processing involved when a customer withdraws cash from a machine. **(08 Marks)**

c. State and explain different management activities. **(07 Marks)**

PART – B

5 a. Write short note on:
i) The repository model.
ii) The client server model.
iii) The layered model. **(12 Marks)**

b. Explain object-oriented design process with example. **(08 Marks)**

6 a. Give brief descriptions of five principles of agile methods. **(06 Marks)**

b. Explain re-engineering process with a neat diagram. **(07 Marks)**

c. With a neat diagram describe the system evolution process. **(07 Marks)**

7 a. Explain the following:
i) Unit testing.
ii) Integration testing.
iii) Release testing. **(09 Marks)**

b. Explain the clean-room software development process with a neat diagram. **(07 Marks)**

c. Explain the stages involved in static analysis. **(04 Marks)**

8 a. Describe people's CMM levels. **(05 Marks)**

b. Explain any two cost estimation techniques with example. **(10 Marks)**

c. List and explain factors governing the staff selection. **(05 Marks)**

* * * * *

2002 SCHEME

USN | | | | | | | | | | |

CS62

Sixth Semester B.E. Degree Examination, December 2012
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions.***

1 a. What are the costs of software engineering? Explain. **(08 Marks)**
b. Describe the generic process model for reuse oriented development. **(06 Marks)**
c. Explain the phases of requirements engineering process. **(06 Marks)**

2 a. Discuss the problems of using natural language for defining user and system requirements. **(08 Marks)**
b. What is meant by viewpoint? Explain. **(04 Marks)**
c. Why event scenarios are used in VORD? Explain with an example. **(04 Marks)**
d. Describe the process of requirements review. **(04 Marks)**

3 a. Discuss the prototyping using dynamic high level language and database programming. **(10 Marks)**
b. Explain how CASE work benches support system modeling. **(06 Marks)**
c. Mention four weakness of structured analysis methods. **(04 Marks)**

4 a. What are the types of Domain specific architectural models? Explain. **(10 Marks)**
b. Discuss the types of proposals for identifying object classes. **(04 Marks)**
c. With a state chart, explain how weather station object responds to requests for different services. **(06 Marks)**

5 a. Explain different styles of interaction with their main advantages. **(10 Marks)**
b. Describe the principal characteristics of the graphical user interface. **(05 Marks)**
c. Discuss the types of checks that can be detected by automated static analysis. **(05 Marks)**

6 a. What is equivalent partitioning? Explain with search routine example. **(08 Marks)**
b. Describe the structure of a software test plan. **(07 Marks)**
c. Explain the metrics for specifying software reliability and availability. **(05 Marks)**

7 a. Explain the significant activities of software project managers. **(08 Marks)**
b. Describe briefly the different cost estimation techniques. **(06 Marks)**
c. Describe the structure of a quality plan. **(06 Marks)**

8 Write short notes on:
a. Reverse engineering.
b. Quality review.
c. Critical systems.
d. Factors affecting software pricing. **(20 Marks)**

* * * * *

USN | | | | | | | | | | |

06IS51

Fifth Semester B.E. Degree Examination, June 2012
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer FIVE full questions, selecting at least TWO questions from each part.***

PART – A

1 a. Explain the types of software process model. **(05 Marks)**
b. What are professional and ethical responsibility of software engineering? **(05 Marks)**
c. Differentiate between technical computer based system and socio–technical system. **(05 Marks)**
d. With a relevant figure, bring out different disparities involved in software engineering. **(05 Marks)**

2 a. Explain the concept of safety and security in system. **(08 Marks)**
b. Bring out the different factors of system dependability. **(06 Marks)**
c. Explain the software specification, with neat diagram. **(06 Marks)**

3 a. Briefly list out eh non–functional requirements of software system. **(10 Marks)**
b. Explain the requirement engineering process, with relevant figures. **(10 Marks)**

4 a. Explain the different system models which can be created during analysis process. **(10 Marks)**
b. Explain the management activities of a software manager. **(10 Marks)**

PART – B

5 a. What is need of architectural design decision? List out different models that can be developed. **(08 Marks)**
b. Describe the concept of client – server model with a neat diagram. **(06 Marks)**
c. Explain the centralized control model. **(06 Marks)**

6 a. Explain the principles of agile methods. **(08 Marks)**
b. Explain the different types of software maintenance. **(06 Marks)**
c. Explain the software evolution process. **(06 Marks)**

7 a. With a neat diagram, explain the inspection process. **(08 Marks)**
b. Explain the stages involved in automatic static analysis. **(06 Marks)**
c. Explain the performance testing. **(06 Marks)**

8 a. Describe the four factors of people management. **(05 Marks)**
b. Explain the Maslow concept of motivating people **(05 Marks)**
c. Describe the cost estimation techniques, with relevant example. **(10 Marks)**

* * * * *

USN | | | | | | | | | | |

10IS51

Fifth Semester B.E. Degree Examination, June/July 2013
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: *Answer FIVE full questions, selecting at least TWO questions from each part.*

PART – A

1 a. What are the attributes of a good software? **(04 Marks)**
b. Define software engineering. Explain the different types of software products. **(06 Marks)**
c. Explain emergent system properties with examples. **(10 Marks)**

2 a. Explain the different types of critical systems. **(06 Marks)**
b. Explain security terminologies. **(05 Marks)**
c. Describe rational unified process with block diagram. **(09 Marks)**

3 a. Explain the metrics for specifying non-functional requirements. **(06 Marks)**
b. Explain requirement engineering process. **(06 Marks)**
c. Explain the structure of the requirements document. **(08 Marks)**

4 a. List and explain different types of system models. **(10 Marks)**
b. What are project management activities? Explain. **(10 Marks)**

PART – B

5 a. With an example describe the repository model and give its advantages and disadvantages. **(10 Marks)**
b. Draw and explain state diagram for a typical weather station. **(10 Marks)**

6 a. Explain the principles of agile methods. **(06 Marks)**
b. What is pair programming? Explain its advantages. **(06 Marks)**
c. Explain Lehman's laws of program evolution dynamics. **(08 Marks)**

7 a. Briefly explain the roles in inspection process. **(06 Marks)**
b. Explain clean-room software development. **(06 Marks)**
c. Explain general model of testing with the help of block diagram. **(08 Marks)**

8 a. Explain any five factors governing staff selection. **(05 Marks)**
b. What are the factors that influence group working? **(05 Marks)**
c. Explain cost estimation techniques. **(10 Marks)**

* * * * *

USN | | | | | | | | | | |

06IS51

Fifth Semester B.E. Degree Examination, June/July 2013
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: *Answer FIVE full questions, selecting at least TWO questions from each part.*

PART – A

1 a. What are the key challenges faced by software engineering? Give an example for how delivery is challenge. **(10 Marks)**

b. Explain the emergent system properties. **(10 Marks)**

2 a. What are various types of critical systems? Give example to each and state at least one reason why your example is critical. **(10 Marks)**

b. With neat sketch, explain the spiral model of software development process. Give its advantages. **(10 Marks)**

3 a. Explain non-functional requirements giving examples to each types (product, organization, external). **(08 Marks)**

b. Discuss the problems with using natural languages for requirement specifications. **(08 Marks)**

c. Write the structure of standard requirement Document. **(04 Marks)**

4 a. Write the state chart for microwave oven model. Indicate all the possible states and their descriptions. **(10 Marks)**

b. Explain the software project planning activities. **(10 Marks)**

PART – B

5 a. Discuss the various architectural design decision made during software design. **(10 Marks)**

b. Distinguish between centralized control and event based control styles. Give one example to each. **(05 Marks)**

c. What are the advantages of adapting object models for modular decompositions? **(05 Marks)**

6 a. What are the principles of agile methods of software development? **(10 Marks)**

b. What are advantages of pair programming? **(05 Marks)**

c. What is software prototyping? What are its benefits? **(05 Marks)**

7 a. Differentiate between verification and validation. **(05 Marks)**

b. Explain the concept of clean room software development. **(05 Marks)**

c. What is equivalence partitioning? Explain with an example how equivalence partitioning helps in testing. **(10 Marks)**

8 a. Write a short note on P-CMM. **(05 Marks)**

b. How is COCOMO used to estimate the cost of a software? **(10 Marks)**

c. Write a note on project staffing. **(05 Marks)**

* * * * *

(2)

USN | | | | | | | | | | |

10IS51

Fifth Semester B.E. Degree Examination, June/July 2014

Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer FIVE full questions, selecting at least TWO questions from each part.***

PART – A

1 a. Answer the following frequently asked questions about software engineering:
i) Difference between software engineering and system engineering.
ii) What is a software process model?
iii) What are key challenges facing software engineering? **(06 Marks)**

b. What are emergent system properties? Give examples. Explain the types of emergent properties. **(08 Marks)**

c. Define legacy systems. Explain the layered model of a legacy system. **(06 Marks)**

2 a. What are the types of critical systems? Define. Write a simple safety critical system and explain. **(09 Marks)**

b. Explain the evolutionary development, and its problems. **(06 Marks)**

c. Write Boehm's spiral model of the software process and explain. **(05 Marks)**

3 a. List out the notations for requirement specification with description. **(06 Marks)**

b. Write the roles of the users of a requirement document. **(06 Marks)**

c. What is Ethnography? How ethonography is effective in discovering the types of requirements? **(08 Marks)**

4 a. Draw the state machine model of a microwave oven. **(06 Marks)**

b. What is object aggregation? Write an example showing aggregation, with notation. **(04 Marks)**

c. Following table shows number of activities, durations and dependencies and milestones. Draw an activity chart and a bar chart showing the critical path for the project schedule:

Tasks	Duration (days)	Dependencies
T_1	5	-
T_2	15	T_1 (M_1)
T_3	10	T_1 (M_1)
T_4	3	T_2 (M_2)
T_5	10	T_2, T_3 (M_2)
T_6	8	T_3 (M_2)
T_7	10	T_4,T_5,T_6 (M_3)
T_8	9	T_7
T_9	10	T_7
T_{10}	9	T_7
T_{11}	20	T_8,T_9,T_{10} (M_4)
T_{12}	10	T_{10} (M_4)
T_{13}	5	T_{11} (M_5)
T_{14}	10	T_{13}

(10 Marks)

PART – B

5 a. According to Bas et al, what are the advantages of designing and documenting software architecture? **(05 Marks)**

b. Explain event driven systems. **(07 Marks)**

c. What is a sequence model? Write the sequence model of operations in collecting the data from a weather station and explain. **(08 Marks)**

6 a. Explain the difficulties with iterative development and incremental delivery. **(06 Marks)**

b. Briefly discuss the extreme programming release cycle with a neat diagram. **(06 Marks)**

c. How software maintenance is carries out? Explain briefly. **(08 Marks)**

7 a. Explain V-model with a neat diagram for planning verification and validation process. **(07 Marks)**

b. Explain the characteristics of clean room software development. **(06 Marks)**

c. Explain any one of the approaches to test case design. **(07 Marks)**

8 a. Why people capability maturity model is used? Explain P-CMM model. **(08 Marks)**

b. List the factors that influence the effectiveness of communication. **(04 Marks)**

c. Write a note on project duration and staffing. **(06 Marks)**

d. Name the types of metrics used to estimate productivity. **(02 Marks)**

* * * * *

USN 1SB11CS018 **10IS51**

Fifth Semester B.E. Degree Examination, Dec. 2013/Jan. 2014
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer FIVE full questions, selecting atleast TWO questions from each part.***

PART – A

1 a. Define software, software engineering, software process. **(06 Marks)**
b. What are attributes of good software? **(08 Marks)**
c. Explain two types of emergent properties. **(06 Marks)**

2 a. Explain system dependability. **(10 Marks)**
b. Explain the process iteration. **(10 Marks)**

3 a. Give software requirement document (IEEE standard). **(10 Marks)**
b. Explain requirement validation. **(10 Marks)**

4 a. Explain structured methods. **(10 Marks)**
b. Explain risk management. **(10 Marks)**

PART – B

5 a. Explain system organization. **(10 Marks)**
b. Give state diagram for weather station and explain design evaluation. **(10 Marks)**

6 a. Explain extreme programming. **(10 Marks)**
b. Give Lehman's laws. **(10 Marks)**

7 a. Explain clean room software development. **(10 Marks)**
b. Explain component testing. **(10 Marks)**

8 a. Give factors governing staff selection. **(10 Marks)**
b. Name factors affecting software engineering productivity and cost estimation techniques. **(10 Marks)**

* * * * *

USN | | | | | | | | | |

10IS51

Fifth Semester B.E. Degree Examination, Dec.2014/Jan.2015
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer FIVE full questions, selecting at least TWO questions from each part.***

PART – A

1 a. What is a software process model? Explain the types of software process models. **(06 Marks)**
b. Explain the key challenges facing software engineering. **(06 Marks)**
c. With a neat block diagram explain the systems engineering process activities. **(08 Marks)**

2 a. With a neat block diagram, explain the spiral process model. **(08 Marks)**
b. Define dependability. Also explain briefly the four principle dimensions of dependability. **(06 Marks)**
c. With appropriate block diagram explain briefly the requirement engineering process or software specification activities. **(06 Marks)**

3 a. For the set of tasks shown below draw the project scheduling using,
i) Activity chart.
ii) Gantt / Bar chart.
iii) Staff allocation versus time chart.
Assuming start date of project as 01 Nov.2014. **(10 Marks)**

Task	Duration	Dependency
T_1	8	-
T_2	15	-
T_3	15	$T_1(m_1)$
T_4	10	-
T_5	10	$T_2, T_4\ (m_2)$
T_6	5	$T_1, T_2\ (m_3)$
T_7	20	$T_1\ (m_1)$
T_8	25	$T_4\ (m_4)$

b. Draw a state machine model of a simple microwave oven. **(05 Marks)**
c. Draw a sequence diagram for withdrawing money from ATM. **(05 Marks)**

4 a. Write the IEEE format of writing SRS. **(05 Marks)**
b. Differentiate between:
i) User requirements and system requirements.
ii) Functional requirements and non-functional requirements. **(05 Marks)**
c. Explain briefly the techniques of requirements discovery. **(10 Marks)**

PART – B

5 a. List the system structuring styles and explain the repository model with a block diagram. **(06 Marks)**
b. With a neat block diagram, explain the object oriented decomposition for invoice processing sub-system. **(06 Marks)**
c. Explain briefly:
i) Call-Return control model.
ii) Broadcast control model. **(08 Marks)**

6 a. With appropriate block diagram explain briefly extreme programming process model. **(10 Marks)**

b. With appropriate block diagram, explain the system evolution process. **(10 Marks)**

7 a. Explain briefly the software inspection process. **(06 Marks)**

b. With a neat block diagram explain the verification and validation process (V-model). **(06 Marks)**

c. Perform the path testing for the following program flow graph by computing Cyclomatic complexity. **(08 Marks)**

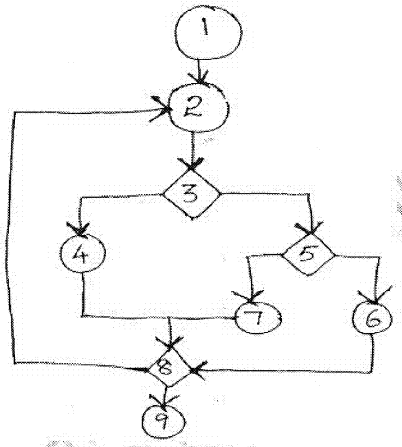


Fig. Q7 (c)

8 Write short notes on:

a. Legacy system.

b. Cocomo model.

c. Capability maturity model.

d. Software testing process. **(20 Marks)**

* * * * *

CBCS Scheme

USN | | | | | | | | | | 15CS42

Fourth Semester B.E. Degree Examination, June/July 2017
Software Engineering

Time: 3 hrs. Max. Marks: 80

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Module-1

1 a. What are the fundamental activities of software engineering? **(04 Marks)**
b. With neat diagram, explain the water-fall model of software development process. **(06 Marks)**
c. With a diagram, explain the rational unified process. **(06 Marks)**

OR

2 a. What is requirement specification? Explain various ways of writing system requirements. **(06 Marks)**
b. Why the understanding of requirements from stake holders is difficult task? Explain. **(05 Marks)**
c Explain the different checks to be carried out during requirement validation process. **(05 Marks)**

Module-2

3 a. Draw a context model for patient information system. How the interactions are modeled? **(06 Marks)**
b. Explain the terms class diagram, generalization and aggregation. **(06 Marks)**
c. What is model Driven engineering? State the three types of abstract system models produced. **(04 Marks)**

OR

4 a. What are the things to be done for a design of object oriented system? How the objects are identified? **(05 Marks)**
b. What is design pattern? Explain four elements of design pattern. **(06 Marks)**
c. What is software reuse? State the general models of open source licenses. **(05 Marks)**

Module-3

5 a. State the two goals and three levels of granularity of software testing process. **(05 Marks)**
b. What is test driven development? State the benefits of test driven developments. **(05 Marks)**
c. Explain the six stages of acceptance testing process. **(06 Marks)**

OR

6 a. With neat diagram, show the software evolution process and explain the 'Lehman's Law' concern to system change. **(10 Marks)**
b. What is software maintenance? State the activities of re-engineering process. **(06 Marks)**

Module-4

7 a. Explain the factors to be considered for approval of change. **(05 Marks)**
b. Explain the features provided by version management systems. **(05 Marks)**
c. What is configuration management? State the four activities of configuration management. **(06 Marks)**

OR

8 a. What is system building? State the features available in the system building tools. **(10 Marks)**
b. Explain the factors to be considered for release planning of system. **(06 Marks)**

Module-5

9 a. Explain the ways of coping with change and reduction of rework cost. **(06 Marks)**
b. Explain the practices involved in the extreme programming. **(10 Marks)**

OR

10 a. State the principles of agile methods. **(05 Marks)**
b. How the agile methods are scaled? State the coping of agile methods for large system engineering. **(05 Marks)**
c. Write a note on pair programming. **(06 Marks)**

* * * * *

USN | | | | | | | | | | |

10IS51

Fifth Semester B.E. Degree Examination, June/July 2013
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer FIVE full questions, selecting at least TWO questions from each part.***

PART – A

1 a. What are the attributes of a good software? **(04 Marks)**
b. Define software engineering. Explain the different types of software products. **(06 Marks)**
c. Explain emergent system properties with examples. **(10 Marks)**

2 a. Explain the different types of critical systems. **(06 Marks)**
b. Explain security terminologies. **(05 Marks)**
c. Describe rational unified process with block diagram. **(09 Marks)**

3 a. Explain the metrics for specifying non-functional requirements. **(06 Marks)**
b. Explain requirement engineering process. **(06 Marks)**
c. Explain the structure of the requirements document. **(08 Marks)**

4 a. List and explain different types of system models. **(10 Marks)**
b. What are project management activities? Explain. **(10 Marks)**

PART – B

5 a. With an example describe the repository model and give its advantages and disadvantages. **(10 Marks)**
b. Draw and explain state diagram for a typical weather station. **(10 Marks)**

6 a. Explain the principles of agile methods. **(06 Marks)**
b. What is pair programming? Explain its advantages. **(06 Marks)**
c. Explain Lehman's laws of program evolution dynamics. **(08 Marks)**

7 a. Briefly explain the roles in inspection process. **(06 Marks)**
b. Explain clean-room software development. **(06 Marks)**
c. Explain general model of testing with the help of block diagram. **(08 Marks)**

8 a. Explain any five factors governing staff selection. **(05 Marks)**
b. What are the factors that influence group working? **(05 Marks)**
c. Explain cost estimation techniques. **(10 Marks)**

* * * * *

USN | | | | | | | | | | |

10IS51

Fifth Semester B.E. Degree Examination, Dec. 2013/Jan. 2014
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer FIVE full questions, selecting atleast TWO questions from each part.***

PART – A

1 a. Define software, software engineering, software process. (06 Marks)
b. What are attributes of good software? (08 Marks)
c. Explain two types of emergent properties. (06 Marks)

2 a. Explain system dependability. (10 Marks)
b. Explain the process iteration. (10 Marks)

3 a. Give software requirement document (IEEE standard). (10 Marks)
b. Explain requirement validation. (10 Marks)

4 a. Explain structured methods. (10 Marks)
b. Explain risk management. (10 Marks)

PART – B

5 a. Explain system organization. (10 Marks)
b. Give state diagram for weather station and explain design evaluation. (10 Marks)

6 a. Explain extreme programming. (10 Marks)
b. Give Lehman's laws. (10 Marks)

7 a. Explain clean room software development. (10 Marks)
b. Explain component testing. (10 Marks)

8 a. Give factors governing staff selection. (10 Marks)
b. Name factors affecting software engineering productivity and cost estimation techniques. (10 Marks)

* * * * *

USN | | | | | | | | | | |

10IS51

Fifth Semester B.E. Degree Examination, Dec. 2013/Jan. 2014
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer FIVE full questions, selecting atleast TWO questions from each part.***

PART – A

1 a. Define software, software engineering, software process. (06 Marks)
b. What are attributes of good software? (08 Marks)
c. Explain two types of emergent properties. (06 Marks)

2 a. Explain system dependability. (10 Marks)
b. Explain the process iteration. (10 Marks)

3 a. Give software requirement document (IEEE standard). (10 Marks)
b. Explain requirement validation. (10 Marks)

4 a. Explain structured methods. (10 Marks)
b. Explain risk management. (10 Marks)

PART – B

5 a. Explain system organization. (10 Marks)
b. Give state diagram for weather station and explain design evaluation. (10 Marks)

6 a. Explain extreme programming. (10 Marks)
b. Give Lehman's laws. (10 Marks)

7 a. Explain clean room software development. (10 Marks)
b. Explain component testing. (10 Marks)

8 a. Give factors governing staff selection. (10 Marks)
b. Name factors affecting software engineering productivity and cost estimation techniques. (10 Marks)

* * * * *

USN | | | | | | | | | | |

10IS51

Fifth Semester B.E. Degree Examination, Dec.2016/Jan.2017
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer FIVE full questions, selecting at least TWO questions from each part.***

PART – A

1 a. Explain the term software engineering and system engineering; mention the important attributes of good software products. **(06 Marks)**

b. What is a software process model? Explain the types of software process models. **(06 Marks)**

c. What are legacy systems? Explain the components of legacy system with neat diagram. **(08 Marks)**

2 a. What are critical systems? Explain the different types of critical systems. **(06 Marks)**

b. With block diagram, explain water fall process model. Mention the advantages and disadvantages of waterfall model. **(08 Marks)**

c. Explain the requirement engineering process with diagram. **(06 Marks)**

3 a. Mention the differences between functional and non-functional requirements. Give example for each. **(06 Marks)**

b. Explain the structure of the requirements documents. **(08 Marks)**

c. Explain the following: i) Ethnography ii) Scenarios. **(06 Marks)**

4 a. Explain different types of system models. **(06 Marks)**

b. Differentiate between milestones and deliverables. **(02 Marks)**

c. List the activities of risk management with diagram. **(04 Marks)**

d. What are project management activities? Explain. **(08 Marks)**

PART – B

5 a. With an example describe the repository model and give its advantages and disadvantages. **(08 Marks)**

b. Draw and explain state diagram for a typical weather station. **(08 Marks)**

c. Define control styles. **(04 Marks)**

6 a. Explain the principle of agile methods. **(06 Marks)**

b. What is prototype? Explain the process prototype development with diagram. Mention the advantages of using prototype. **(08 Marks)**

c. With a neat diagram describe the system evolution process. **(06 Marks)**

7 a. Explain the following : i) Unit testing ii) Integration testing **(06 Marks)**

b. Explain clean Room software development. **(08 Marks)**

c. List classes of interface errors. **(06 Marks)**

8 Write short notes on the following :

a. IEEE/ACM code of Ethics

b. The client server model

c. Lehman's laws

d. Software cost estimation techniques. **(20 Marks)**

* * * * *

USN | | | | | | | | | | |

10MCA42

Fourth Semester MCA Degree Examination, June/July 2015
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions.***

1 a. Distinguish between : i) Software Engineering & System Engineering.
ii) Software process and Software process model. **(06 Marks)**
b. Explain any four professional & ethical responsibilities. **(04 Marks)**
c. Explain the steps involved in the design of a socio – technical system with the help of a neat diagram. **(10 Marks)**

2 a. Discuss the steps involved in the following software process models :
i) a) Evolutionary development b) Component based software development.
ii) Justify the cost distribution for these software development models. **(10 Marks)**
b. Explain RUP. Describe various phases of RUP. **(10 Marks)**

3 a. Differentiate between user requirements and system requirements. Explain with an example. **(10 Marks)**
b. Explain the steps involved in a requirements engineering process, with the help of a neat diagram. **(10 Marks)**

4 a. List out and discuss two types of behavioural models. **(10 Marks)**
b. What are object models? Discuss various types. Draw the object model for a course. **(10 Marks)**

5 a. Explain : i) Repository model ii) Client server model. **(10 Marks)**
b. Discuss in detail the various stages of object oriented design process, with the help of design models. **(10 Marks)**

6 a. What are agile methods? List out the principles of agile methods. **(10 Marks)**
b. Explain the steps of a re – engineering process, with the help of a diagram. **(10 Marks)**

7 a. Explain the structure of a software test plan. **(06 Marks)**
b. Explain static and dynamic V & V process. **(10 Marks)**
c. List out the roles in the inspection process. **(04 Marks)**

8 Write short notes on :
a. Software cost estimation models.
b. Project Management Activities.
c. People capability maturity model.
d. Component testing. **(20 Marks)**

*****

USN | | | | | | | | | | |

13MCA33

Third Semester MCA Degree Examination, June/July 2015
Software Engineering

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions.***

1 a. What is software? Explain the different attributes of a good software. **(05 Marks)**
b. Explain the various key challenges being faced in software engineering. **(05 Marks)**
c. What are the professional and ethical responsibilities a software engineer should have? **(05 Marks)**
d. Give a brief description of software engineering diversity. **(05 Marks)**

2 a. Explain with a neat diagram, the Boehm spiral model of software development process. What are merits and demerits of spiral model? **(10 Marks)**
b. What is software prototyping? What are its merits? **(04 Marks)**
c. Explain the various principles of agile methods. **(06 Marks)**

3 a. Explain the general structure of a software requirement document suggested by IEEE. **(06 Marks)**
b. Clearly distinguish between functional and non functional requirements. **(06 Marks)**
c. With a neat diagram, explain the different types of activities that are performed in the requirement engineering process. **(08 Marks)**

4 a. What is meant by system model? Explain in detail different behavioral model with examples. **(10 Marks)**
b. Explain different types of architectural styles for component and connector view. **(10 Marks)**

5 a. What is meant by component? Explain basic elements of component model. **(10 Marks)**
b. Explain the process of CBSE. **(10 Marks)**

6 a. Describe issues of distributed system. **(06 Marks)**
b. Explain client server computing. Discuss its advantages and disadvantages. **(10 Marks)**
c. What are the characteristics of software as a service? **(04 Marks)**

7 a. What are the factors to be considered during the process of staff selection? **(08 Marks)**
b. Explain the activities of software configuration management plan. **(08 Marks)**
c. What is project monitoring plan? **(04 Marks)**

8 a. What are the objectives of testing? **(06 Marks)**
b. Briefly explain the black box testing with advantages. **(10 Marks)**
c. Write a note on software testing process. **(04 Marks)**

* * * * *

Module-3 RS-20

Software Testing: Development testing, Test-driven development, Release testing, User testing, Test Automation.
Software Evolution: Evolution processes, Program evolution dynamics, Software maintenance, Legacy system management

Software Testing

- 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 (**test cases**). You check the results of the test run for errors, anomalies or information about the programs non-functional attributes.
- Can reveal the presence of errors NOT their absence.
- Testing is part of a more general verification and validation process, which also includes static validation techniques.

Programming test 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 behavior of the software is incorrect, undesirable or does not conform to its specification.
  - Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.
- The first goal leads to **validation testing**
  - You expect the system to perform correctly using a given set of test cases that reflect the systems expected use.
- The second goal leads to **defect testing**
  - The test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.
- **Validation testing**
  - To demonstrate to the developer and the system customer that the software meets its requirements
  - A successful test shows that the system operates as intended.
- **Defect testing**
  - To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

The below figure illustrates the difference between validation and defect testing.

- The system accepts inputs from some input set I and generates outputs in an output set O.

- Some of the outputs will be erroneous. These are the outputs in set O_e that are generated by the system in response to inputs in the set I_e.
- The priority in defect testing is to find those inputs in the set I_e because these reveal problems with the system.
- Validation testing involves testing with correct inputs that are outside I_e.

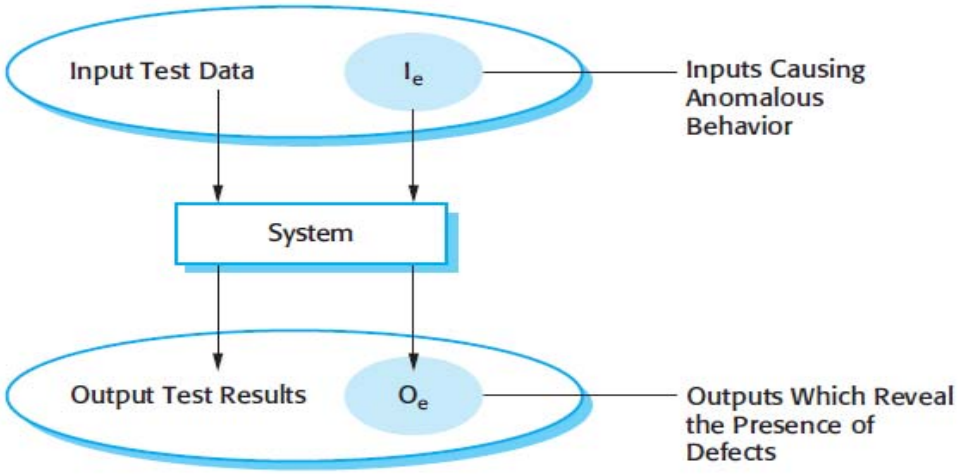


Figure 1.1: An input-output model of program testing

Table: Difference between validation and defect testing

Validation Testing	***Defect Testing***
Intended to show that the software meets its requirements.	Intended to discover defects in the software.
A successful test is one that shows that a requirement has been properly implemented.	A successful test is one which reveals the presence of defects in a system.
Statistical testing can be used → to test the program's performance and reliability → to check how it works under operational conditions	Goal is to find inconsistencies between → program and → specification

Verification	**Validation**
Are we building the product right, The software should **conform to its specification.**	Are we building the right product". The software **should do** what the user really requires
Verification is the process of evaluating products of a development phase to find out whether they meet the specified requirements.	**Validation** is the process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements.
Following activities are involved in **Verification**: Reviews, Meetings and Inspections.	Following activities are involved in **Validation**: Testing like black box testing, white box testing, gray box testing etc
Execution of code is not comes under	Execution of code is comes under

Verification.	**Validation.**
Cost of errors caught in **Verification** is less than errors found in Validation	Cost of errors caught in **Validation** is more than errors found in Verification.

- Depends on system's purpose, user expectations and marketing environment
  - **Software purpose**
    - The level of confidence depends on how critical the software is to an organisation.
  - **User expectations**
    - Users may have low expectations of certain kinds of software.
  - **Marketing environment**
    - Getting a product to market early may be more important than finding defects in the program.

Inspections and testing

- Software inspections Concerned with **analysis** of the static system representation to discover problems (static verification)
  - May be supplement by tool-based document and code analysis.
- Software testing Concerned with **exercising and observing product behaviour** (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed.

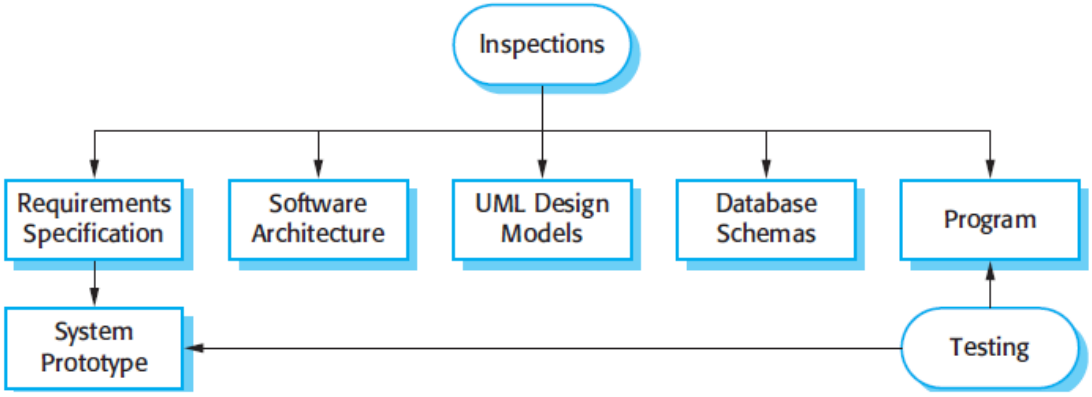


Figure 1.2: Inspections and Testing

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

Advantages of inspections

- During testing, errors can mask (hide) other errors. Because inspection is a static process, you don't have to be concerned with interactions between errors.

- Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

Inspections	***Testing***
Inspections can be done on any system-representation such as → requirements-document → design-diagrams and → program source-code	Involves running an implementation of the software with test-data.
This is a static V&V technique, as you don't need to run the software on a computer.	This is a dynamic V & V technique.
Static techniques can only check the correspondence between → program and → specification (verification).	You examine the outputs of the software and its operational behaviour to check that it is performing as required.
Static techniques cannot demonstrate that the software is operationally useful.	Dynamic techniques can demonstrate that the software is operationally useful.

Stages of Testing Process (Activities):

- The below figure is an abstract model of the 'traditional' testing process, as used in plan driven development.

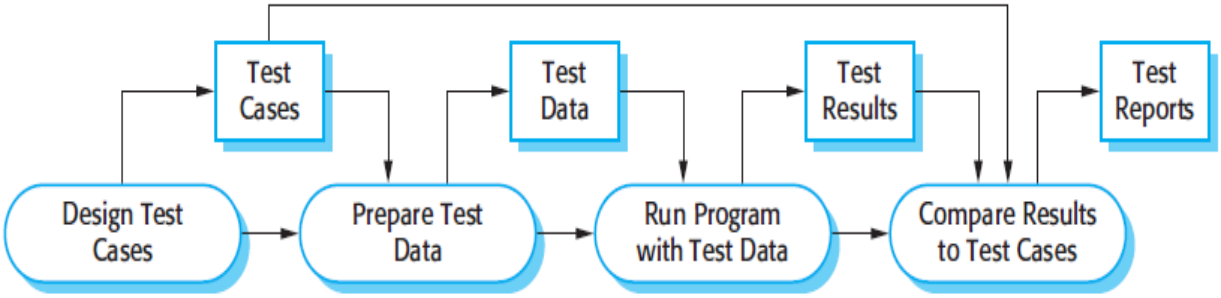


Figure 1.3: Stages Of Software Process

Types of Testing

1. **Development testing**, where the system is tested during development to discover bugs and defects.
2. **Release testing**, where a separate testing team tests a complete version of the system before it is released to users.
3. **User testing,** where users or potential users of a system test the system in their own environment.

1.1 Development testing

- Development testing includes all testing activities that are carried out by the team developing that particular software or system. **Which includes 3 stages,**
  1. **Unit testing**, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
  2. **Component testing**, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
  3. **System testing,** where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

1.1.1 Unit testing

- Unit testing is the process of **testing individual components** in isolation.
- It is a defect testing process.
- Units may include,
  - **Individual functions** or methods within an object
  - **Object classes** with several attributes and methods
  - **Composite components** with defined interfaces used to access their functionality.
- Complete test coverage of a class involves
  - **Testing all operations** associated with an object
  - **Setting and interrogating** all object attributes
  - **Exercising the object** in all possible states.

Problem in unit testing : **Inheritance** makes it more difficult to design object class tests as the information to be tested is not localised.

Example: The weather station object interface

- Weather station is class here, in that identifier is method with attributes and associated test cases.
- Need to define test cases for reportWeather, calibrate, test, startup and shutdown.
- Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions
- For example:
  - Shutdown -> Running-> Shutdown
  - Configuring-> Running-> Testing -> Transmitting -> Running
  - Running-> Collecting-> Running-> Summarizing -> Transmitting -> Running

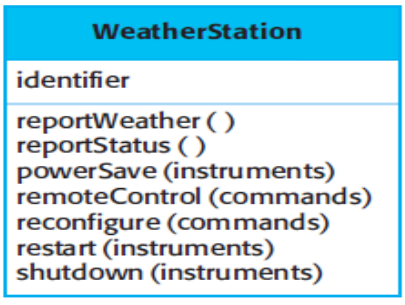


Figure 1.4: Weather station

Automation Testing:

- Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
- In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests.
- Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success of otherwise of the tests.

An automated test has three parts:

1. A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.
2. A call part, where you call the object or method to be tested.
3. An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful; if false, then it has failed.

1.1.2 Unit test cases:

- Testing is expensive and time consuming, so it is important that you choose effective unit test cases.
- Effectiveness, in this case, means two things:
    1. The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
    2. If there are defects in the component, these should be revealed by test cases.

2 types of unit test case:

- The first of these should reflect normal operation of a program and should show that the component works as expected.
- The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

Testing Strategies

1. **Partition testing:** where you identify groups of inputs that have common characteristics and should be processed in the same way.
    - You should choose tests from within each of these groups.
2. **Guideline-based testing:** where you use testing guidelines to choose test cases.
    - These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

Partition testing

- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.
- The below figure shows the Equivalence Partitioning.

- In the figure the large shaded ellipse on the left represents the set of all possible inputs to the program that is being tested.
- The smaller unshaded ellipses represent equivalence partitions. A program being tested should process all of the members of an input equivalence partitions in the same way.
- Output equivalence partitions are partitions within which all of the outputs have something in common. Sometimes there is a 1:1 mapping between input and output equivalence partitions.
- The shaded area in the left ellipse represents inputs that are invalid. The shaded area in the right ellipse represents exceptions that may occur (i.e., responses to invalid inputs).
- Once you have identified a set of partitions, you choose test cases from each of these partitions.
- You identify partitions by using the program specification or user documentation and from experience where you predict the classes of input value that are likely to detect errors.
- For example, say a program specification states that the program accepts 4 to 8 inputs which are five-digit integers greater than 10,000.
- You use this information to identify the input partitions and possible test input values. These are shown in Figure 1.6.

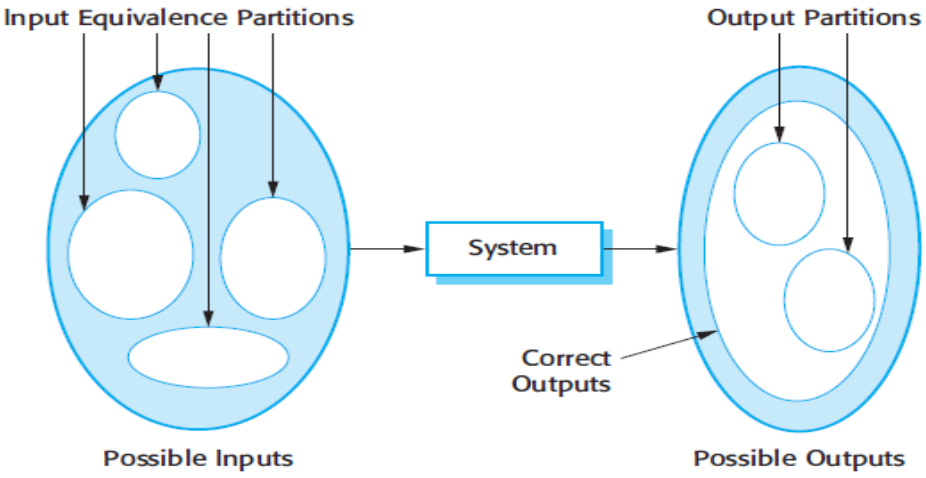


Figure 1.5: Equivalence partitioning

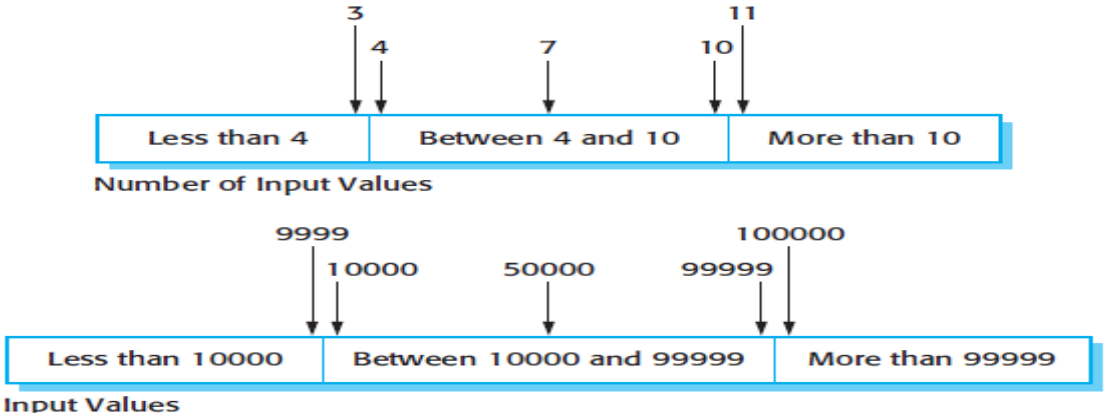


Figure 1.6: Equivalence Partitions

Testing guidelines that could help reveal defects include:

- Test software with sequences which have only a single value.
- Use sequences of different sizes in different tests.
- Derive tests so that the first, middle and last elements of the sequence are accessed.
- Test with sequences of zero length.
-

General testing guidelines

1. Choose inputs that force the system to **generate all error messages**
2. Design inputs that cause **input buffers** to overflow
3. **Repeat** the same input or series of inputs numerous times
4. Force **invalid outputs** to be generated
5. Force **computation results** to be too large or too small.

1.1.3 Component Testing

- Software components are often composite components that are made up of **several interacting objects.**
  - **For example**, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- You access the functionality of these objects through the defined component interface.
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
  - You can assume that unit tests on the individual objects within the component have been completed.
- The below figure illustrates the idea of component interface testing. Assume that components A, B, and C have been integrated to create a larger component or subsystem.

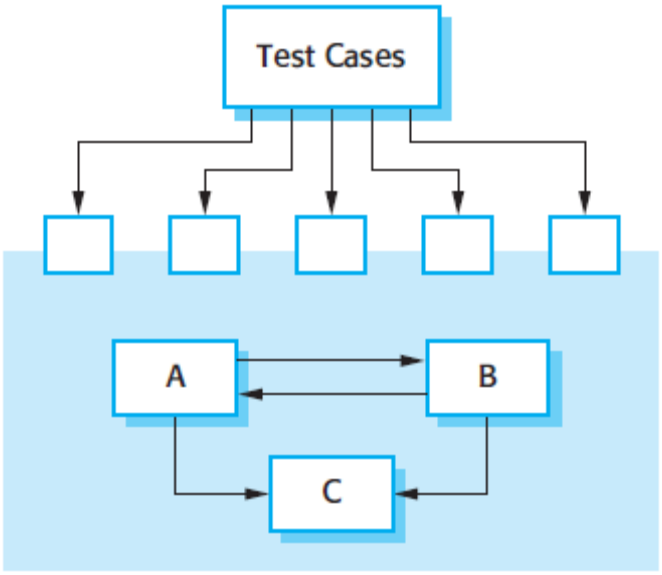


Figure 1.7: Interface Testing

Interface testing

- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.

Interface Types

1. **Parameter interfaces**:
   - Data passed from one method or procedure to another.
   - Methods in an object have a parameter interface.
2. **Shared memory interfaces**:
   - In this interface Block of memory is shared between procedures or functions.
   - Data is placed in the memory by one subsystem and retrieved from there by other sub-systems.
3. **Procedural interfaces**:
   - Sub-system encapsulates a set of procedures to be called by other sub-systems.
   - Objects and reusable components have this form of interface.
4. **Message passing interfaces:**
   - The interface in which one component requests a service from another component by passing a message to it.

Interface Errors

1. **Interface misuse**
   - A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.
2. **Interface misunderstanding**
   - A calling component embeds assumptions about the behaviour of the called component which are incorrect.
3. **Timing errors**
   - The called and the calling component operate at different speeds and out-of-date information is accessed.

Interface testing guidelines

1. Design tests so that **parameters** to a called procedure are at the extreme ends of their ranges.
2. Always test **pointer parameters** with null pointers.
3. Design tests which cause the **component to fail**.
4. Use **stress testing** in message passing systems.
5. In shared memory systems, **vary the order** in which components are activated.

1.1.4 <u>System Testing</u>

- ✧ System testing during development involves integrating components to create a version of the system and then **testing the integrated system.**
- ✧ The focus in system testing is testing the **interactions between components.**
- ✧ System testing checks that components are compatible interact correctly and transfer the right data at the right time across their interfaces.
- ✧ System testing tests the emergent behavior of a system. The main goal of the system testing is to test the interaction between the components.

Difference between system testing and component testing:

- During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
- Components developed by different team members or groups may be integrated at this stage. System testing is a collective rather than an individual process. In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

Component testing	***System testing***
Testing of individual components (Fig 23.1)	Testing of groups of components integrated to create a system(or subsystem).
Usually the responsibility of the component-developer.	Usually the responsibility of an independent testing team.
Tests are derived from the developer's experience.	Tests are based on a system-specification.

Use-case based testing

- The use-cases developed to identify system interactions can be used as a basis for system testing.
- Each use case usually involves several system components so testing the use case forces these interactions to occur.
- The sequence diagrams associated with the use case documents the components and interactions that are being tested.
- The below sequence diagram is used to identify operations that will be tested and to help design the test cases to execute the tests. Therefore, issuing a request for a report will result in the execution of the following thread of methods:

SatComms:request_ WeatherStation:reportWeather _ Commslink:Get(summary)_ WeatherData:summarize

- The sequence diagram helps you design the specific test cases that you need as it shows what inputs are required and what outputs are created:
    1. An input of a request for a report should have an associated acknowledgment. A report should ultimately be returned from the request. During testing, you should create summarized data that can be used to check that the report is correctly organized.
    2. An input request for a report to WeatherStation results in a summarized report being generated. You can test this in isolation by creating raw data corresponding to the summary that you have prepared for the test of SatComms and checking that the WeatherStation object correctly produces this summary. This raw data is also used to test the WeatherData object.

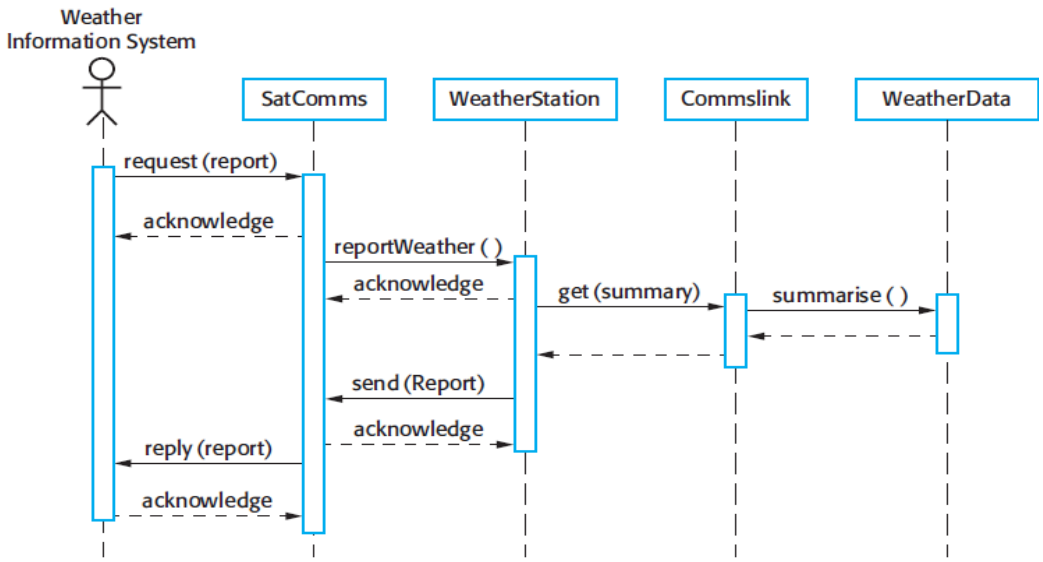


Figure 1.8: Collect weather data sequence chart

Testing policies

- Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.
- **Examples of testing policies:**
  - All system functions that are accessed through menus should be tested.
  - Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
  - Where user input is provided, all functions must be tested with both correct and incorrect input.

1.2 Test-driven development

- Test-driven development (TDD) is an approach to program development in which you **inter-leave testing and code development.**
- Tests are written before code and 'passing' the tests is the critical driver of development.
- You develop code incrementally, along with a test for that increment. You don't move on to the next increment until the code that you have developed passes its test.
- TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

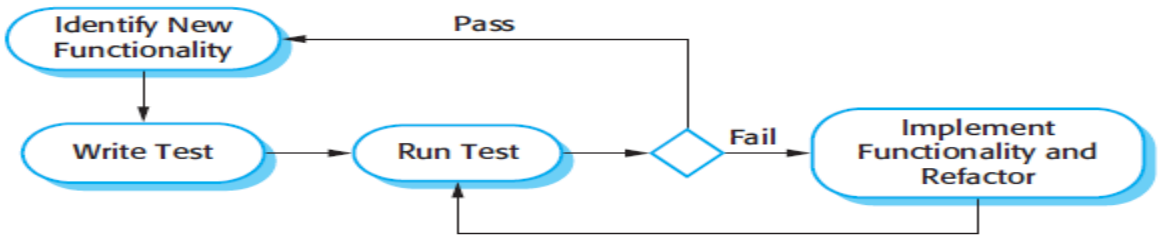


Figure 1.9: Test Driven Development

Steps in TDD process:

- Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- Write a test for this functionality and implement this as an automated test.
- Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
- Implement the functionality and re-run the test.
- Once all tests run successfully, you move on to implementing the next chunk of functionality.

Advantages of test-driven development

1. **Code coverage**
    - Every code segment that you write has at least one associated test so all code written has at least one test.
2. **Regression testing**
    - A regression test suite is developed incrementally as a program is developed.
3. **Simplified debugging**
    - When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.
4. **System documentation**
    - The tests themselves are a form of documentation that describe what the code should be doing

- One of the most important benefits of test-driven development is that it reduces the costs of regression testing.
- Regression testing involves running test sets that have successfully executed after changes have been made to a system.
- The regression test checks that these changes have not introduced new bugs into the system and that the new code interacts as expected with the existing code.
- Regression testing is very expensive and often impractical when a system is manually tested, as the costs in time and effort are very high.
- Automated testing, which is fundamental to test-first development, dramatically reduces the costs of regression testing. Existing tests may be re-run quickly and cheaply.
- Tests must run successfully before any further functionality is added.
- Test-driven development is of most use in new software development where the functionality is either implemented in new code or by using well-tested standard libraries.
- Test-driven development may also be ineffective with multi-threaded systems.
- Test-driven development has proved to be a successful approach for small and medium-sized projects.

1.3 Release testing

- Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.
- The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.
    - Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.
- Release testing is usually a black-box testing process where tests are only derived from the system specification.

Release Testing	System Testing
A separate team that has not been involved in the system development should be responsible for release testing.	System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

1.3.1 Requirements based testing

- Requirements-based testing involves examining each requirement and developing a test or tests for it.
- MHC-PMS requirements:
    - If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
    - If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.

Requirements tests in MHC-PMS

1	Set up a patient record with no known allergies, Prescribe medication for allergies that are known to exist	Check that a warning message is not issued by the system
2	Set up a patient record with a known allergy	Check that the warning is issued by the system.
3	Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately	Check that the correct warning for each drug is issued.
4	Prescribe two drugs that the patient is allergic too	Check that two warnings are correctly issued.
5	Prescribe a drug that issues a warning and overrule that warning	Check that the system requires the user to provide information explaining why the warning was overruled.

1.3.2 Scenario Testing:

- Scenario testing is an approach to release testing where you devise typical scenarios of use and use these to develop test cases for the system.
- A scenario is a story that describes one way in which the system might be used.
- Scenarios should be realistic and real system users should be able to relate to them.

Kate is a nurse who specializes in mental health care. One of her responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, Kate logs into the MHC-PMS and uses it to print her schedule of home visits for that day, along with summary information about the patients to be visited. She requests that the records for these patients be downloaded to her laptop. She is prompted for her key phrase to encrypt the records on the laptop.

One of the patients that she visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. Kate looks up Jim's record and is prompted for her key phrase to decrypt the record. She checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect so she notes the problem in Jim's record and suggests that he visits the clinic to have his medication changed. He agrees so Kate enters a prompt to call him when she gets back to the clinic to make an appointment with a physician. She ends the consultation and the system re-encrypts Jim's record.

After, finishing her consultations, Kate returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for Kate of those patients who she has to contact for follow-up information and make clinic appointments.

Figure 1.10: A usage scenario for the MHC-PMS

Example: It tests a number of features of the MHC-PMS:

1. Authentication by logging on to the system.
2. Downloading and uploading of specified patient records to a laptop.
3. Home visit scheduling.
4. Encryption and decryption of patient records on a mobile device.
5. Record retrieval and modification.
6. Links with the drugs database that maintains side-effect information. The system for call prompting.

1.3.3 Performance testing:

- Performance tests have to be designed to ensure that the system can process its intended load.
- Performance testing is concerned both with **demonstrating that the system meets its requirements and discovering problems and defects in the system.**
- To test whether performance requirements are being achieved, you may have to construct an operational profile. An operational profile is a set of tests that reflect the actual mix of work that will be handled by the system.
- Part of release testing may involve testing the emergent properties of a system, such as **performance and reliability**.
- Tests should reflect the **profile of use of the system**.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behavior.

Two functions:

1. It tests the failure behavior of the system. Circumstances may arise through an unexpected combination of events where the load placed on the system exceeds the maximum anticipated load. Stress testing checks that overloading the system causes it to 'fail-soft' rather than collapse under its load.
2. It stresses the system and may cause defects to come to light that would not normally be discovered. Although it can be argued that these defects are unlikely to cause system failures in normal usage, there may be unusual combinations of normal circumstances that the stress testing replicates.

1.4 User testing

- ✧ User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
- ✧ User testing is essential, even when comprehensive system and release testing have been carried out.
  - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

Types of user testing

1. **Alpha testing**
   - Users of the software work with the development team to test the software at the developer's site.
   - Users and developers work together to test a system as it is being developed. This means that the users can identify problems and issues that are not readily apparent to the development testing team.
2. **Beta testing**
   - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
   - Beta testing takes place when an early, sometimes unfinished, release of a software system is made available to customers and users for evaluation.
   - Beta testers may be a selected group of customers who are early adopters of the system.
   - Beta testing is mostly used for software products that are used in many different environments.
   - Beta testing is essential to discover interaction problems between the software and features of the environment where it is used.
3. **Acceptance testing**
   - Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.
   - Acceptance testing is an inherent part of custom systems development.
   - It takes place after release testing.

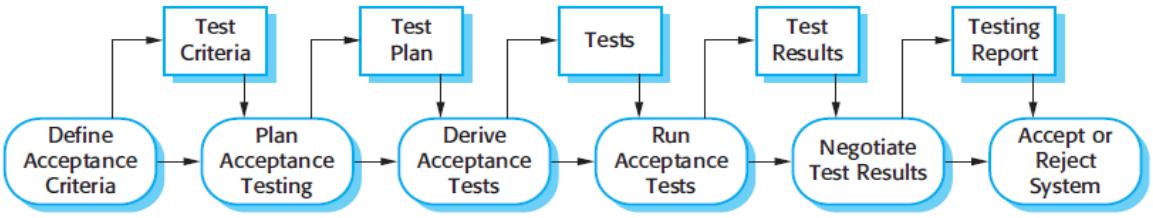


Figure 1.11: Acceptance Testing Process

Different stages includes,

1. **Define acceptance criteria:** should be defined at the early stage of process its normally between customer and software developer.
2. **Plan acceptance testing:** decide the time, budget for acceptance testing.
3. **Derive acceptance tests**: output of this system consisting of various test cases that are designed to meet the acceptable criteria.
4. **Run acceptance tests:** designed test cases are executed. Test results are documented.
5. **Negotiate test results:** if some minute errors occur then developers and customer should discuss about that and will make some decision about the results.
6. **Reject/accept system:** if the system tested is up to mark , customer expection is met without any error then it is subjected for acceptance. Else rejection

1.5 Test Automation

- Testing is an expensive and laborious phase of the software process. As a result, testing tools were among the first software tools to be developed. These tools now offer a range of facilities and their use can significantly reduce the costs of testing.
- A software testing workbench is an integrated set of tools to support the testing process. In addition to testing frameworks that support automated test execution, a workbench may include tools to simulate other parts of the system and to generate system test data.
- Figure shows some of the tools that might be included in such a testing workbench,
    1. **Test manager** Manages the running of program tests. The test manager keeps track of test data, expected results and program facilities tested.
    2. **Test data generator** Generates test data for the program to be tested. This may be accomplished by selecting data from a database or by using patterns to generate random data of the correct form.
    3. **Oracle** Generates predictions of expected test results. Oracles may either be previous program versions or prototype systems.
    4. **File comparator** Compares the results of program tests with previous test results and reports differences between them.
    5. **Report generator** Provides report definition and generation facilities for test results.
    6. **Dynamic analyser** Adds code to a program to count the number of times each statement has been executed.
    7. **Simulator** Different kinds of simulators may be provided. Target simulators simulate the machine on which the program is to execute

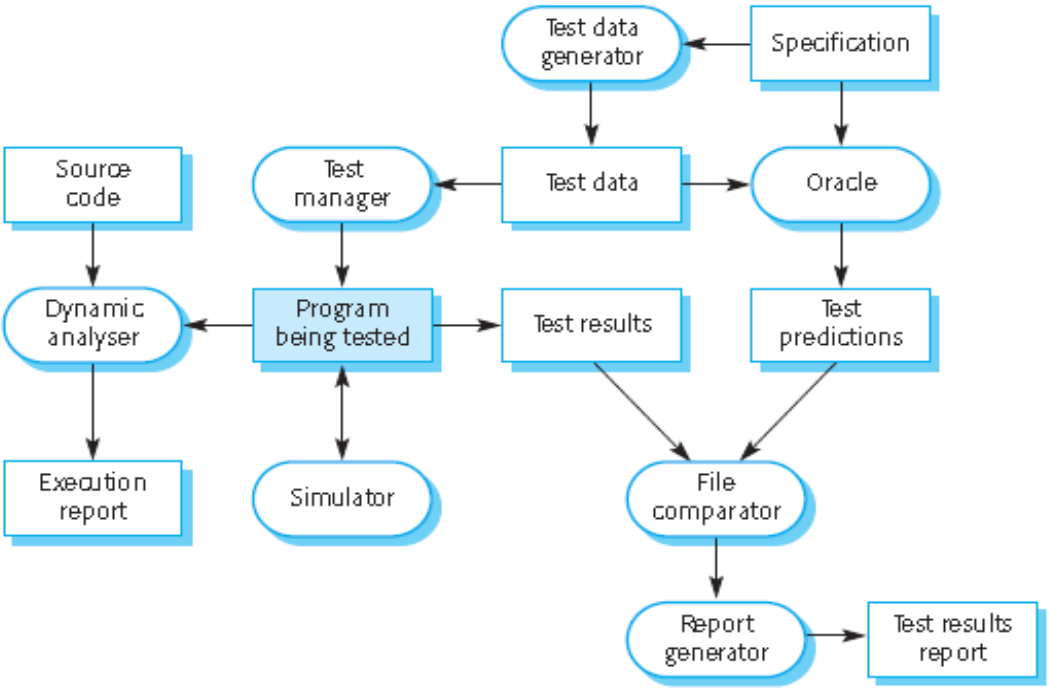


Figure 1.12: Test Automation

2. Software Evolution

- After systems have been deployed, they i**nevitably have to change if they are to remain useful.** Once software is put into use, new requirements emerge and existing requirements change. Business changes often generate new requirements for existing software.
- Software evolution is important because organizations have invested large amounts of money in their software and are now completely dependent on these systems.
- Their systems are critical business assets and they have to invest in system change to maintain the value of these assets.
- Consequently, most large companies spend more on maintaining existing systems than on new systems development.
- Software evolution may be triggered by changing business requirements, by reports of software defects, or by changes to other systems in a software system's environment.
- The evolution of a system can rarely be considered in isolation.
- Changes to the environment lead to system change that may then trigger further environmental changes.
- Useful software systems often have a very long lifetime. For example, large military or infrastructure systems, such as air traffic control systems, may have a lifetime of 30 years or more. Business systems are often more than 10 years old.
- The requirements of the installed systems change as the business and its environment change.
- The below figure illustrates software engineering as a spiral process with requirements, design, implementation, and testing going on throughout the lifetime of the system
- The majority of changes are a consequence of **new requirements** that are generated in response to changing business and user needs.
- Consequently, you can think of software engineering as a spiral process with requirements, design, implementation and testing going on throughout the lifetime of the system. This is illustrated in Figure 2.1.

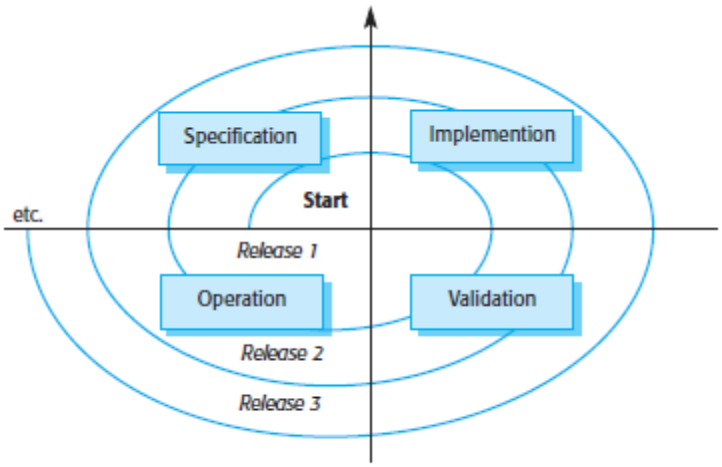


Figure 2.1: A spiral model of development and evolution

- During evolution, the software is used successfully and there is a constant stream of proposed requirements changes.
- At some stage in the life cycle, the software reaches a transition point where significant changes, implementing new requirements, become less and less cost effective.
- The below figure shows the evolution and servicing.

Evolution

- The stage in a software system's life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system.

Servicing

- At this stage, the software remains useful but the only changes made are those required to keep it operational i.e. bug fixes and changes to reflect changes in the software's environment. No new functionality is added.

Phase-out

- The software may still be used but no further changes are made to it.

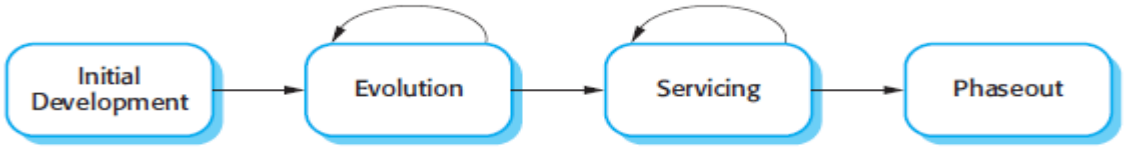


Figure 2.2: Evolution and Servicing

2.1 Evolution Processes

- Software evolution processes depend on
  - The type of software being maintained;
  - The development processes used;
  - The skills and experience of the people involved.
- Proposals for change are the driver for system evolution.
- Should be linked with components that are affected by the change, thus allowing the cost and impact of the change to be estimated.
- The below figure illustrates that change identification and evolution continues throughout the system lifetime.

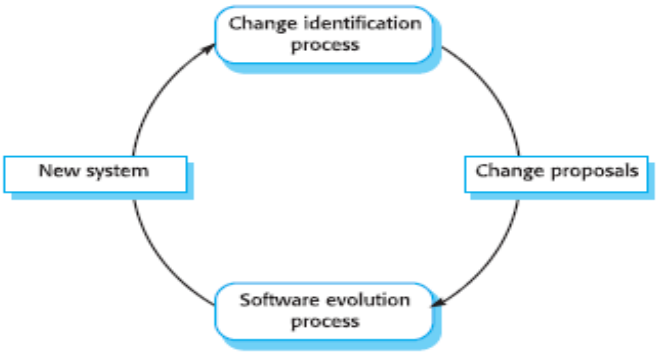


Figure 2.3: Change identification and evolution process

- System change proposals are the driver for system evolution in all organizations. These change proposals may involve existing requirements that have not been implemented in the released system, requests for new requirements and bug repairs from system stakeholders.
- The below figure shows an overview of the evolution process. The process includes the fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers.
- The cost and impact of these changes are assessed to see how much of the system is affected by the change and how much it might cost to implement the change.
- If the proposed changes are accepted, a new release of the system is planned. During release planning, all proposed changes (fault repair, adaptation and new functionality) are considered.

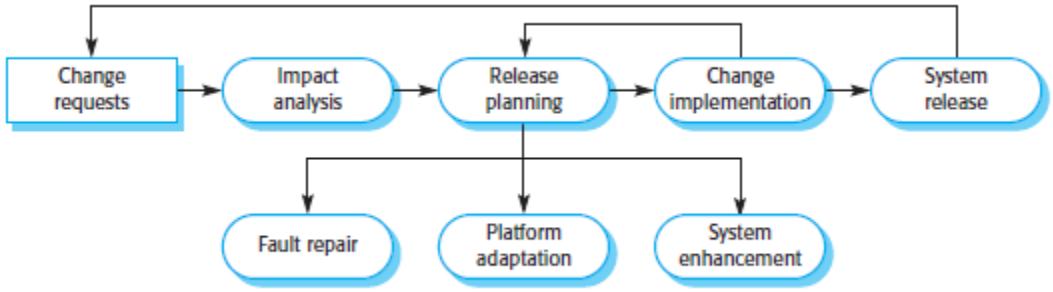


Figure 2.4: The system evolution process

Change Implementation:

- Iteration of the development process where the revisions to the system are designed implemented and tested.
- A critical difference is that the first stage of change implementation may involve program understanding, especially if the original system developers are not responsible for the change implementation.
- During the program understanding phase, you have to understand how the program is structured, how it delivers functionality and how the proposed change might affect the program.

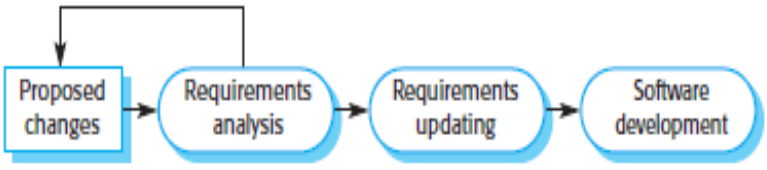


Figure 2.5: Change implementation

- During the evolution process, the requirements are analyzed in detail and implications of the changes emerge that were not apparent in the earlier change analysis process.
- This means that the proposed changes may be modified and further customer discussions may be required before they are implemented.

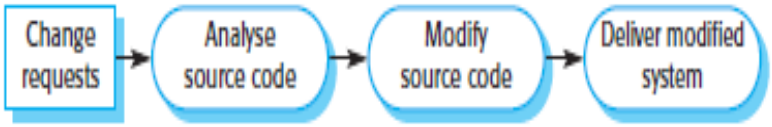


Figure 2.6: The emergency repair process

The urgent changes can arise for three reasons:

1. If a **serious system fault** occurs that has to be repaired to allow normal operation to continue.
2. If changes to the system's operating environment have unexpected effects that disrupt normal operation
3. If there are unanticipated changes to the business running the system, such as the emergence of new competitors or the introduction of new legislation?

- Rather than modify the requirements and design, you make an emergency fix to the program to solve the immediate problem. (Fig 2.6)
- When emergency code repairs are made, the change request should remain outstanding after the code faults have been fixed. It can then be re-implemented more carefully after further analysis.
- Agile methods are based on incremental development so the transition from development to evolution is a seamless one.
- Evolution is simply a continuation of the development process based on frequent system releases.
- Automated regression testing is particularly valuable when changes are made to a system.
- Changes may be expressed as additional user stories.

Handover problems

- Where the development teams have used an agile approach but the evolution team is unfamiliar with agile methods and prefers a plan-based approach.
  - The evolution team may expect detailed documentation to support evolution and this is not produced in agile processes.
- Where a plan-based approach has been used for development but the evolution team prefers to use agile methods.
  - The evolution team may have to start from scratch developing automated tests and the code in the system may not have been refactored and simplified as is expected in agile development.

2.2 Program Evolution Dynamics

- Program evolution dynamics is the study of system change. The majority of work in this area has been carried out by Lehman and Belady. From these studies, they proposed a set of laws (Lehman's laws) concerning system change. They claim these laws (hypotheses, really) are invariant and widely applicable.
- Lehman and Belady examined the growth and evolution of a number of large software systems. The proposed laws, shown in Figure 2.7.

❖ **The first law** states that system maintenance is an inevitable process. As the system's environment changes, new requirements emerge and the system must be modified. When the modified system is re-introduced to the environment, this promotes more environmental changes, so the evolution process recycles.

❖ **The second law** states that, as a system is changed, its structure is degraded. The only way to avoid this happening is to invest in preventative maintenance where you spend time improving the software structure without adding to its functionality. Obviously, this means additional costs, over and above those of implementing required system changes.

Law	Description
Continuing change	A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors is approximately invariant for each system release.
Organisational stability	Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.
Continuing growth	The functionality offered by systems has to continually increase to maintain user satisfaction.
Declining quality	The quality of systems will appear to be declining unless they are adapted to changes in their operational environment.
Feedback system	Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

Figure 2.7: Lehman's Laws

❖ **The third law** is, perhaps, the most interesting and the most contentious of Lehman's laws. It suggests that large systems have a dynamic of their own that is established at an early stage in the development process. This determines the gross trends of the system maintenance process and limits the number of possible system changes.

- **Lehman's fourth law** suggests that most large programming projects work in what he terms a *saturated* state. That is, a change to resources or staffing has imperceptible effects on the long-term evolution of the system.
- **Lehman's fifth** law is concerned with the change increments in each system release. Adding new functionality to a system inevitably introduces new system faults. The more functionality added in each release, the more faults there will be. Therefore, a large increment in functionality in one system release means that this will have to be followed by a further release where the new system faults are repaired.
- **The sixth** and seventh laws are similar and essentially say that users of software will become increasingly unhappy with it unless it is maintained and new functionality is added to it.
- **The final law** reflects the most recent work on feedback processes, although it is not yet clear how this can be applied in practical software development.

2.3 Software Maintenance

- Software maintenance is the general process of changing a system after it has been delivered.
- The term is usually applied to custom software where separate development groups are involved before and after delivery.
- Maintenance does not normally involve major changes to the system's architecture.
- Changes are implemented by modifying existing components and adding new components to the system.

There are **three different types** of software maintenance:

1. ***Maintenance to repair software faults*** Coding errors are usually relatively cheap to correct; design errors are more expensive as they may involve rewriting several program components. Requirements errors are the most expensive to repair because of the extensive system redesign that may be necessary.

2. ***Maintenance to adapt the software to a different operating environment*** This type of maintenance is required when some aspect of the system's environment such as the hardware, the platform operating system or other support software changes. The application system must be modified to adapt it to cope with these environmental changes.

3. ***Maintenance to add to or modify the system's functionality*** This type of maintenance is necessary when the system requirements change in response to organizational or business change. The scale of the changes required to the software is often much greater than for the other types of maintenance.

- These types of maintenance are generally recognized but different people sometimes give them different names.
- **'Corrective maintenance'** is universally used to refer to maintenance for fault repair.
- **'Adaptive maintenance'** sometimes means adapting to a new environment and sometimes means adapting the software to new requirements.
- **'Perfective maintenance'** sometimes means perfecting the software by implementing new requirements; in other cases it means maintaining the functionality of the system but improving its structure and its performance.
- The below figure shows an approximate distribution of maintenance costs.

- The specific percentages will obviously vary from one organization to another but, universally; repairing system faults is not the most expensive maintenance activity.
- Evolving the system to cope with new environments and new or changed requirements consumes most maintenance effort.
- The relative costs of maintenance and new development vary from one application domain to another.

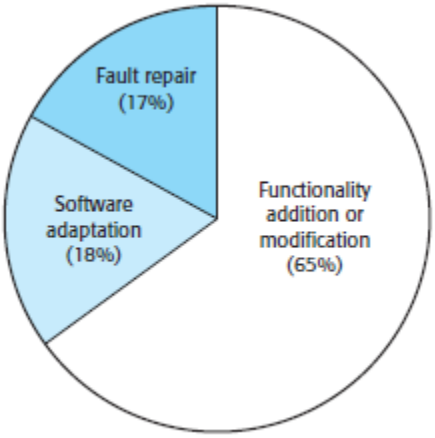


Figure 2.8: Maintenance effort distribution

- It is usually cost effective to invest effort in designing and implementing a system to reduce the costs of future changes.
- Adding new functionality after delivery is expensive because you have to spend time learning the system and analyzing the impact of the proposed changes.
- Figure 2.9 shows how overall lifetime costs may decrease as more effort is expended during system development to produce a maintainable system. Because of the potential reduction in costs of understanding, analysis, and testing, there is a significant multiplier effect when the system is developed for maintainability.
- For System 1, extra development costs of $25,000 are invested in making the system more maintainable. This results in a savings of $100,000 in maintenance costs over the lifetime of the system. This assumes that a percentage increase in development costs results in a comparable percentage decrease in overall system costs.

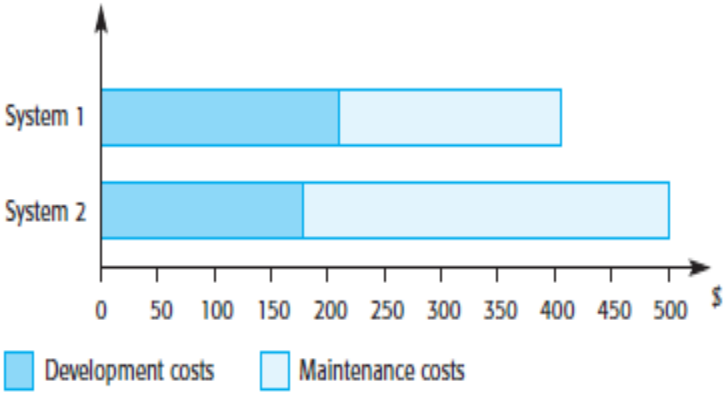


Figure 2.9: Development and maintenance costs

The key factors that distinguish development and maintenance, and which lead to higher maintenance costs, are:

1. ***Team stability****:* After a system has been delivered, it is normal for the development team to be broken up and people work on new projects. The new team or the individuals responsible for system maintenance do not understand the system or the background to system design decisions.
2. ***Contractual responsibility****:* The contract to maintain a system is usually separate from the system development contract. The maintenance contract may be given to a different company rather than the original system developer. This factor, along with the lack of team stability, means that there is no incentive for a development team to write the software so that it is easy to change.
3. ***Staff skills****:* Maintenance staff are often relatively inexperienced and unfamiliar with the application domain. Maintenance has a poor image among software engineers. It is seen as a less skilled process than system development and is often allocated to the most junior staff
4. ***Program age and structure****:* As programs age, their structure tends to be degraded by change, so they become harder to understand and modify. Some systems have been developed without modern software engineering techniques.

2.3.1 Maintenance prediction

Managers hate surprises, especially if these result in unexpectedly high costs. You should therefore try to predict what system changes are likely and what parts of the system are likely to be the most difficult to maintain. You should also try to estimate the overall maintenance costs for a system in a given time period. Figure 2.8 illustrates these predictions and associated questions.

These predictions are obviously closely related:

1. Whether a system change should be accepted depends, to some extent, on the maintainability of the system components affected by that change.
2. Implementing system changes tends to degrade the system structure and hence reduce its maintainability.
3. Maintenance costs depend on the number of changes, and the costs of change implementation depend on the maintainability of system components. Predicting the number of change requests for a system requires an understanding of the relationship between the system and its external environment. To evaluate the relationships between a system and its environment, you should assess:

1. *The number and complexity of system interfaces* The larger the number of interfaces and the more complex they are, the more likely it is that demands for change will be made.

2. *The number of inherently volatile system requirements* The requirements that reflect organizational policies and procedures are likely to be more volatile than requirements that are based on stable domain characteristics.

3. *The business processes in which the system is used* As business processes evolve, they generate system change requests. The more business processes that use a system, the more the demands for system change.

Process metrics that can be used for assessing maintainability are:

1. *Number of requests for corrective maintenance* An increase in the number of failure reports may indicate that more errors are being introduced into the program than are being repaired during the maintenance process. This may indicate a decline in maintainability.
2. *Average time required for impact analysis* This reflects the number of program components that are affected by the change request. If this time increases, it implies that more and more components are affected and maintainability is decreasing.

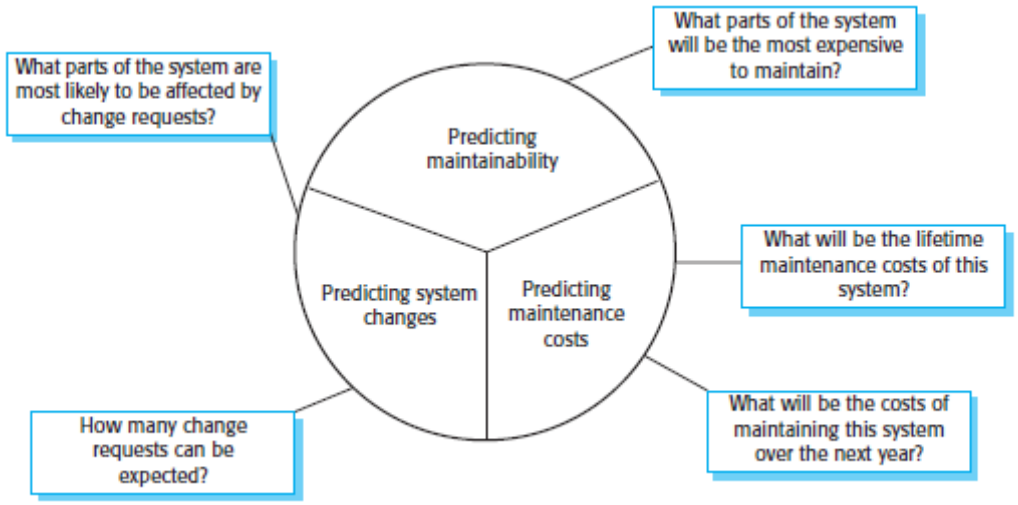


Figure 2.10: Maintenance prediction

3. *Average time taken to implement a change request* This is not the same as the time for impact analysis although it may correlate with it. This is the amount of time that you need to actually modify the system and its documentation, after you have assessed which components are affected. An increase in the time needed to implement a change may indicate a decline in maintainability.
4. *Number of outstanding change requests* An increase in this number over time may imply a decline in maintainability.

2.3.2 Software re-engineering

Re-engineering may involve re-documenting the system, organizing and restructuring the system, translating the system to a more modern programming language, and modifying and updating the structure and values of the system's data.

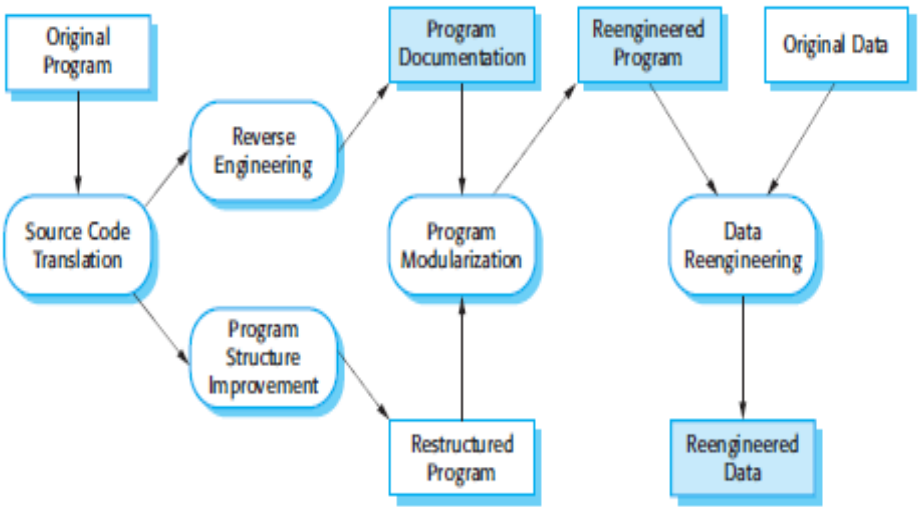


Figure 2.11: Re-engineering

The activities in this re-engineering process are:

1. *Source code translation*: The program is converted from an old programming language to a more modern version of the same language or to a different language.

2. *Reverse engineering*: The program is analyzed and information extracted from it. This helps to document its organization and functionality.

3. *Program structure improvement*: The control structure of the program is analyzed and modified to make it easier to read and understand.

4. *Program modularization*: Related parts of the program are grouped together and, where appropriate, redundancy is removed. In some cases, this stage may involve architectural transformation where a centralized system intended for a single computer is modified to run on a distributed platform.

5. *Data re-engineering*: The data processed by the program is changed to reflect program changes.

Re-engineering advantages are:

1. *Reduced risk* There is a high risk in re-developing business-critical software. Errors may be made in the system specification, or there may be development problems. Delays in introducing the new software may mean that business is lost and extra costs are incurred.

2. *Reduced cost* The cost of re-engineering is significantly less than the cost of developing new software.

The costs of re-engineering obviously depend on the extent of the work that is carried out. There is a spectrum of possible approaches to re-engineering as shown in below figure, Reengineering as part of architectural migration is the most expensive.

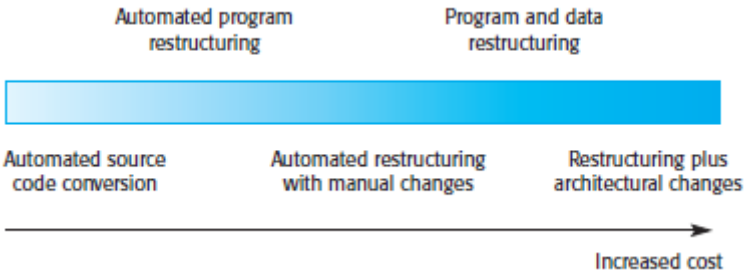


Figure 2.12: Reengineering Approaches

2.3.3 Preventative maintenance by refactoring

- Refactoring is the process of making improvements to a program to slow down degradation through change.
- You can think of refactoring as 'preventative maintenance' that reduces the problems of future change.
- Refactoring involves modifying a program to improve its structure, reduce its complexity or make it easier to understand.
- When you refactor a program, you should not add functionality but rather concentrate on program improvement.

Refactoring and reengineering:

- Re-engineering takes place after a system has been maintained for some time and maintenance costs are increasing. You use automated tools to process and re-engineer a legacy system to create a new system that is more maintainable.
- Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

Stereotypical situations ('Bad smells')in which code of program can be improved:

- **Duplicate code**
  - The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.
- **Long methods**
  - If a method is too long, it should be redesigned as a number of shorter methods.
- **Switch (case) statements**
  - These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.
- **Data clumping**
  - Data clumps occur when the same group of data items (fields in classes, parameters in methods) re-occurs in several places in a program. These can often be replaced with an object that encapsulates all of the data.
- **Speculative generality**

- This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.

2.4 Legacy system evolution

- Organizations that have a limited budget for maintaining and upgrading their legacy systems have to decide how to get the best return on their investment.
- This means that they have to make a realistic assessment of their legacy systems and then decide what is the most appropriate strategy for evolving these systems.

There are **four strategic options:**

1. ***Scrap the system completely*** This option should be chosen when the system is not making an effective contribution to business processes. This occurs when business processes have changed since the system was installed and are no longer completely dependent on the system.
2. ***Leave the system unchanged and continue with regular maintenance*** This option should be chosen when the system is still required but is fairly stable and the system users make relatively few change requests.
3. ***Re-engineer the system to improve its maintainability*** This option should be chosen when the system quality has been degraded by regular change and where regular change to the system is still required.
4. ***Replace all or part of the system with a new system*** This option should be chosen when other factors such as new hardware mean that the old system cannot continue in operation or where off-the-shelf systems would allow the new system to be developed at a reasonable cost.

Two Assessments (judgment) of legacy system.

1. Relative Business value
2. System quality

From a **business perspective,** you have to decide whether the business really needs the system. From a **technical perspective**, you have to assess the quality of the application software and the system's support software and hardware.
From below figure you can see that there **are four clusters of systems**:

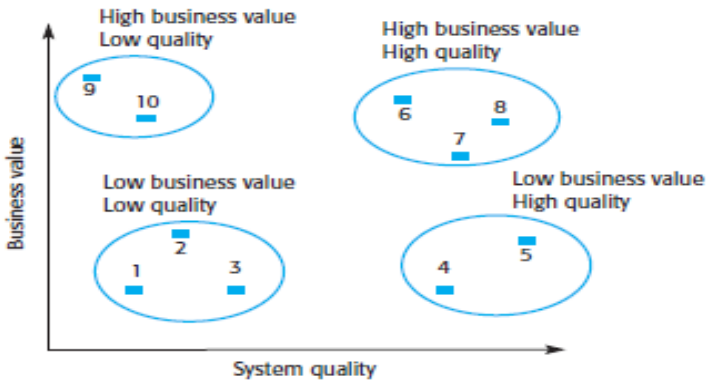


Figure 2.13: Legacy system assessments

1. *Low quality, low business value* Keeping these systems in operation will be expensive and the rate of the return to the business will be fairly small. These systems should be scrapped.
2. *Low quality, high business value* These systems are making an important business contribution so they cannot be scrapped. However, their low quality means that it is expensive to maintain them. These systems should be re-engineered to improve their quality or replaced, if a suitable off-the-shelf system is available.
3. *High quality, low business value* These are systems that don't contribute much to the business but that may not be very expensive to maintain. It is not worth replacing these systems so normal system maintenance may be continued so long as no expensive changes are required and the system hardware is operational. If expensive changes become necessary, they should be scrapped.
4. *High quality, high business value* These systems have to be kept in operation, but their high quality means that you don't have to invest in transformation or system replacement. Normal system maintenance should be continued.

There are **four basic issues for low business value**:

1. ***The use of the system*** If systems are only used occasionally or by a small number of people, they may have a low business value. A legacy system may have been developed to meet a business need that has either changed or that can now be met more effectively in other ways.
2. ***The business processes that are supported*** When a system is introduced, business processes to exploit that system may be designed. However, changing these processes may be impossible because the legacy system can't be adapted. Therefore, a system may have a low business value because new processes can't be introduced.
3. ***The system dependability*** System dependability is not only a technical problem but also a business problem. If a system is not dependable and the problems directly affect the business customers or mean that people in the business are diverted from other tasks to solve these problems, the system has a low business value.
4. ***The system outputs*** The key issue here is the importance of the system outputs to the successful functioning of the business. If the business depends on these outputs, then the system has a high business value.

- To assess a software system from a technical perspective, you need to consider both the application system itself and the environment in which the system operates.
- The environment includes the hardware and all associated support software (compilers, development environments, etc.) that are required to maintain the system.
- The environment is important because many system changes result from changes to the environment, such as upgrades to the hardware or operating system.

Factors that you should consider during the environment assessment are shown in below figure.

Factor	Questions
Supplier stability	Is the supplier is still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, does someone else maintain the systems?
Failure rate	Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts?
Age	How old is the hardware and software? The older the hardware and support software, the more obsolete it will be. It may still function correctly but there could be significant economic and business benefits to moving to more modern systems.
Performance	Is the performance of the system adequate? Do performance problems have a significant effect on system users?
Support requirements	What local support is required by the hardware and software? If there are high costs associated with this support, it may be worth considering system replacement.
Maintenance costs	What are the costs of hardware maintenance and support software licences? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs.
Interoperability	Are there problems interfacing the system to other systems? Can compilers, for example, be used with current versions of the operating system? Is hardware emulation required?

Figure 2.14: Factors used in environment assessment

To assess the technical quality of an application system, you have to assess a range of factors shown in below figure.

1. ***The number of system change requests*** System changes tend to corrupt the system structure and make further changes more difficult. The higher this value, the lower the quality of the system

2. ***The number of user interfaces*** This is an important factor in forms-based systems where each form can be considered as a separate user interface. The more interfaces, the more likely that there will be inconsistencies and redundancies in these interfaces.

3. ***The volume of data used by the system*** The higher the volume of data (number of files, size of database, etc.), the more complex the system.

Factor	Questions
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent and current?
Data	Is there an explicit data model for the system? To what extent is data duplicated across files? Is the data used by the system up-to-date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?
Programming language	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
Configuration management	Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?
Test data	Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
Personnel skills	Are there people available who have the skills to maintain the application? Are there only a limited number of people who understand the system?

Figure 2.15: Factors used in application assessment

QUESTION BANK
MODULE-3

Sl.No	Questions	Marks
1.	What is verification and validation Or Differentiate between verification and validation?	4M
2.	Distinguish between software inspection and testing. What are the advantages of inspection over testing?	8M
3.	Explain traditional model of software testing process with neat diagram?	5M
4.	Explain why testing can only detect the presence of errors, not their absence.	2M
5.	State the two goals and three levels of granularity of software testing process.	5M
6.	What is partition testing? Identify equivalence class partition for Date, Month and year in calendar. List also the boundary values for each class.	6M
7.	What is partition testing? Identify equivalence class partitions for automated air condition system having at least four partitions. List also the boundary values for each	8M
8.	With neat diagram explain partition testing/ Equivalence Partitioning?	6M
9.	Explain Component testing?	8M
10.	Explain different types of interface error and also specify the classes of interface errors	10M
11.	Write a scenario that could be used to help design tests for the wilderness weather station system.	3M
12.	Differentiate between white box and black box Testing.	5M
13.	Explain the following. A. Integration Testing B. Release Testing	8M
14.	Explain general guidelines for interface testing?	6M
15.	Explain performance testing?	5M
16.	What is test driven development? State the benefits of test driven development.	5M
17.	Explain the six stages of acceptance testing.	6M
18.	What is test automation? Explain with a figure the tools that might be included in a testing workbench?	8M
19.	What are legacy systems? Explain the components of legacy system with a neat diagram	10M
20.	Define “Program Evolution Dynamics”. Discuss lehman’s law for program evolution dynamics.	10M
21.	With appropriate block diagram, explain the system evolution process.	
22.	Explain program evolution dynamics.	8M
23.	Explain evolution process with example.	10M
24.	Explain the activities involved in reengineering process, with an illustrative figure.	10M
25.	Write notes on i) Legacy System ii) Software Reengineering Process.	10M
26.	With neat diagram, Show the software evolution process and explain the Lehman’s law concerned to system change.	10M
27.	What is software maintenance? Explain different types of software maintenance.	6M
28.	Explain the process metrics in assessing software maintenance?	6M

MODULE-4

Project Planning: Software pricing, Plan-driven development, Project scheduling, Estimation techniques.

Quality management: Software quality, Reviews and inspections, Software measurement and metrics ,Software standards.

PROJECT PLANNING

Project Planning

- Project planning involves **breaking down the work into parts and assign these to project team members**, anticipate problems that might arise and prepare tentative solutions to those problems.
- The project plan, which is created at the start of a project, is used to communicate how the work will be done to the project team and customers, and to help assess progress on the project.

Planning stages (3 stages)

1. At the **proposal stage**, when you are bidding for a contract to develop or provide a software system.
2. During the **project startup phase**, when you have to plan who will work on the project, how the project will be broken down into increments, how resources will be allocated across your company, etc.
3. **Periodically throughout the project**, when you modify your plan in the light of experience gained and information from monitoring the progress of the work.

Proposal planning

- Planning at the proposal stage is inevitably speculative, as you do not usually have a complete set of requirements for the software to be developed.
- A plan is often a required part of a proposal, so you have to produce a credible plan for carrying out the work. If you win the contract, you then usually have to replan the project, taking into account changes since the proposal was made.
- As a starting point for calculating price, you need to draw up an estimate of your costs for completing the project work.
- Estimation involves working out how much effort is required to complete each activity and, from this, calculating the total cost of activities.
- You should always calculate software costs objectively, with the aim of accurately predicting the cost of developing the software.
- There are **three main parameters** that you should use when computing the costs of a software development project:

Parameters for software development costs:

1. Effort costs-costs of paying for employees by company.
2. Hardware and software costs.
3. Travel costs- onsite project may be to clients location

- For most projects, the biggest cost is the effort cost. You have to estimate the total effort (in person-months) that is likely to be required to complete the work of a project.
- Development planning is intended to ensure that the project plan remains a useful document for staff to understand what is to be achieved and when it is to be delivered.
- If an agile method is used, there is still a need for a project startup plan, as regardless of the approach used, the company still needs to plan how resources will be allocated to a project.
- During development, an informal project plan and effort estimates are drawn up for each release of the software, with the whole team involved in the planning process.

1.1 Software pricing

- In principle, the price of a **software product to a customer** is simply the cost of **development plus profit** for the developer.
- The relationship between the project cost and **the price quoted to the customer** is not usually so simple. When calculating a price, you should take **broader organizational, economic, political, and business considerations into account**, such as **those shown in Figure 1.1.**
- As the cost of a project is only **loosely related to the price** quoted to a customer, **'pricing to win'** is a commonly used strategy. Pricing to win means that a company has some idea of the price that the customer expects to pay and makes a bid for the contract based on the customer's expected price.
- A project cost is agreed on the basis of an outline proposal. Negotiations then take place between client and customer to establish the detailed project specification.
- This specification is constrained by the agreed cost. The requirements may be changed so that the cost is not exceeded.
- **Training costs**-training for employee regarding the project and cost for that training

Factor	Description
Market opportunity	A development organization may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the organization the opportunity to make a greater profit later. The experience gained may also help it develop new products.
Cost estimate uncertainty	If an organization is unsure of its cost estimate, it may increase its price by a contingency over and above its normal profit.
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Requirements volatility	If the requirements are likely to change, an organization may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business. Cash flow is more important than profit in difficult economic times.

Figure 1.1 : Factors affecting software pricing

1.2 Plan-Driven Development

- Plan-driven or **plan-based development** is an approach to software engineering where the development process is planned in detail.
- **Plan-driven development** is based on **engineering project management techniques** and is the 'traditional' way of managing large software development projects.
- A **project plan** is created that records the work to be done, who will do it, the **development schedule** and the work products.
- **Managers use the plan to support project** decision making and as a way of measuring progress.

1.2.1 Project Plans

- In a plan-driven development project, a project plan sets out the resources available to the project, the work breakdown and a schedule for carrying out the work.
- Plans normally include the following sections.

1	**Introduction**	This briefly describes the objectives of the project and sets out the constraints (e.g., budget, time, etc.) that affect the management of the project
2	**Project organization**	This describes the way in which the development team is organized, the people involved, and their roles in the team
3	**Risk analysis**	This describes possible project risks, the likelihood of these risks arising, and the risk reduction strategies that are proposed
4	**Hardware and software resource requirements**	This specifies the hardware and support software required to carry out the development. If hardware has to be bought, estimates of the prices and the delivery schedule may be included
5	**Work breakdown**	This sets out the breakdown of the project into activities and identifies the milestones and deliverables associated with each activity Milestones are key stages in the project where progress can be assessed; deliverables are work products that are delivered to the customer.
6	**Project schedule**	This shows the dependencies between activities, the estimated time required to reach each milestone, and the allocation of people to activities.
7	**Monitoring and reporting mechanisms**	This defines the management reports that should be produced, when these should be produced, and the project monitoring mechanisms to be used.

- As well as the principal project plan, which should focus on the risks to the projects and the project schedule**, you may develop a number of supplementary plans to support** other process activities such as testing and configuration management. The below figure shows **the Examples of possible supplementary plans.**

Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources, and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements, costs, and effort.
Staff development plan	Describes how the skills and experience of the project team members will be developed.

Figure 1.2: Project Plan Supplements

Plan driven development Pros and Cons

- The arguments in favor of a plan-driven approach are that early planning allows organizational issues (availability of staff, other projects, etc.) to be closely taken into account, and that potential problems and dependencies are discovered before the project starts, rather than once the project is underway.

- The principal argument against plan-driven development is that many early decisions have to be revised because of changes to the environment in which the software is to be developed and used.

1.2.2 The Planning Process

- Project planning is an iterative process that starts when you create an initial project plan during the **project startup phase**.
- Plan changes are inevitable.
- As more information about the system and the project team becomes available during the project, you should regularly revise the plan to reflect requirements, schedule and risk changes.
- **Changing business goals also leads to changes in project plans**. As business goals change, this could affect all projects, which may then have to be re-planned**.**

Procedure:

Step 1: At the beginning of planning process, identify the constraints that are affecting the project. The constraints are may be resources like budget, time skilled staffs. At the same time identify the possible risks, define milestones and deliverables.

Step 2: Prepare estimated schedule and after some time review the progress identify the problems or minor problems, minor modifications in the plan might be required at this stage.

Step 3: If serious problem may cause delay in project, then replan the project.

Step 4: Perform the work, repeat step 2, and step 3 until project is finished.

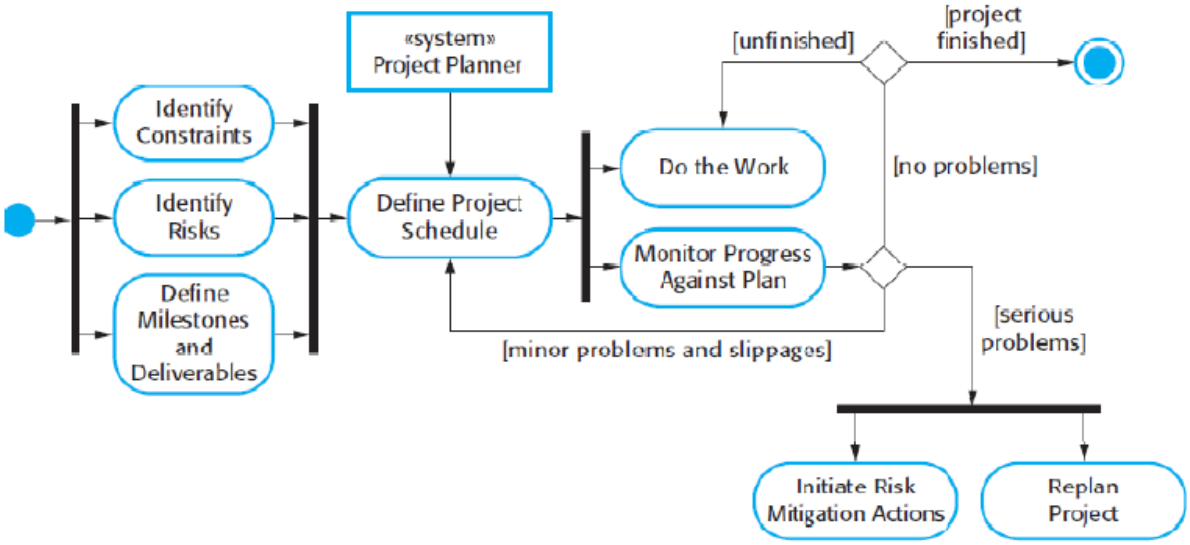


Figure 1.3: Project Planning Process

Advantages:

1. Initial assessment about the project is easier by this planning process.
2. Can get clear idea about resource requirement and allotment.
3. Finding a risks is very important w.r.t project quality, that could be overcome by making proper analysis in planning stage.

1.3 Project scheduling

- ✧ Project scheduling is **the process of deciding how the work in a project will be organized** as separate tasks, and when and how these tasks will be executed.
- ✧ You estimate the **calendar time needed to complete each task**, the effort required and who will work on the tasks that have been identified.
- ✧ You also have to **estimate the resources needed to complete each task**, such as the **disk space required on a server**, the **time required on specialized hardware**, such as a simulator, and what the travel budget will be.
- ✧ Both **plan-based and agile processes** need an initial project schedule, although the level of detail may be less in an agile project plan. This **initial schedule** is used to plan how people will be allocated to **projects and to check the progress of the project** against its **contractual commitments**. In **traditional development processes**, the complete schedule is initially developed and then modified as the project progresses.
- ✧ **In agile processes, there has to be an overall schedule that identifies** when the major Phases of the project will be completed. An iterative approach to scheduling is then used to plan each phase.

Project scheduling activities

- Split project into tasks and estimate time and resources required to complete each task.
- Tasks should normally last at least a week and no longer than months. Finer subdivision means that a disproportionate amount of time must be spent on replanning and updating the project plan.

- The maximum amount of time for any task should be around 8 to 10 weeks. If it takes longer than this, the task should be subdivided for project planning and scheduling.
- Organize tasks concurrently to make optimal use of workforce.
- Minimize task dependencies to avoid delays caused by one task waiting for another to complete.
- Dependent on project manager's intuition and experience.

Milestones and deliverables

- **Milestones** are points in the schedule against which you can assess progress, for example, the handover of the system for testing.
- **Deliverables** are work products that are delivered to the customer, e.g. a requirements document for the system.

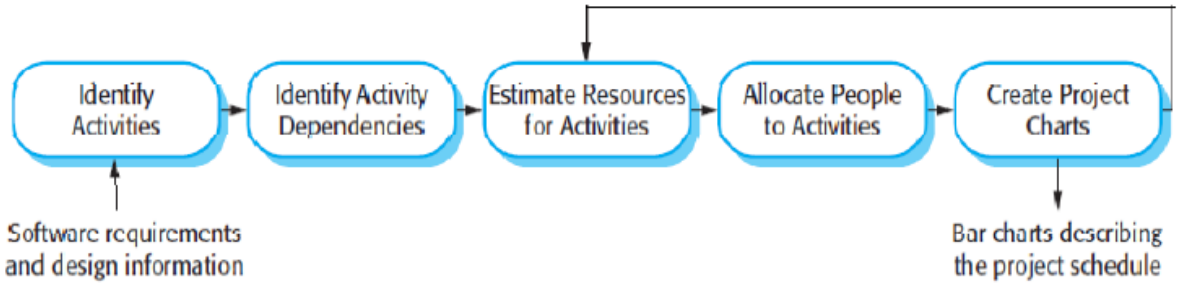


Figure 1.4: The project scheduling process

Scheduling problems

- Estimating the difficulty of problems and hence the cost of developing a solution is hard.
- Productivity is not proportional to the number of people working on a task.
- Adding people to a late project makes it later because of communication overheads.
- The unexpected always happens. Always allow contingency in planning.

1.3.1 Schedule representation

- Project schedules may simply be represented in a table or spreadsheet showing the tasks, effort, expected duration, and task dependencies (Figure 1.5).
- Gra**phical notations** are normally used to illustrate the project schedule.(gantt charts, barcharts.)
- These show the project **breakdown into tasks**. Tasks should not be too small. They should take about a week or two.

There **are two types of representation that are commonly** used:

1. **Bar charts, which are calendar-based**, show who is responsible for each activity, the expected elapsed time, and when the activity is scheduled to begin and end. **Bar charts are sometimes called 'Gantt charts'**, after their inventor, Henry Gantt.
2. **Activity networks, which are network diagrams**, show the dependencies between the different activities making up a project.

Project activities are the basic planning elements, each activity has,
1. A duration in calendar days or months.
2. An effort estimate, which reflects the number of person-days or person-months to complete the work.
3. A deadline by which the activity should be completed.
4. A defined endpoint.

Problems w.r.t scheduling
For the problem given below, draw,
i. Scheduling Activity bar chart/ gantt chart.
iii. Draw staff allocation graph.

Task	Effort (person-days)	Duration (days)	Dependencies
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

Solution:
1. Scheduling barchart/gantt chart. **T1, T2,… represents various tasks**, the milestones of various tasks is shown by **M1,M2…starting from week 0,** and every week is having 5 days of working days. Barchart is represented.

2. Diamond symbol indicates milestone

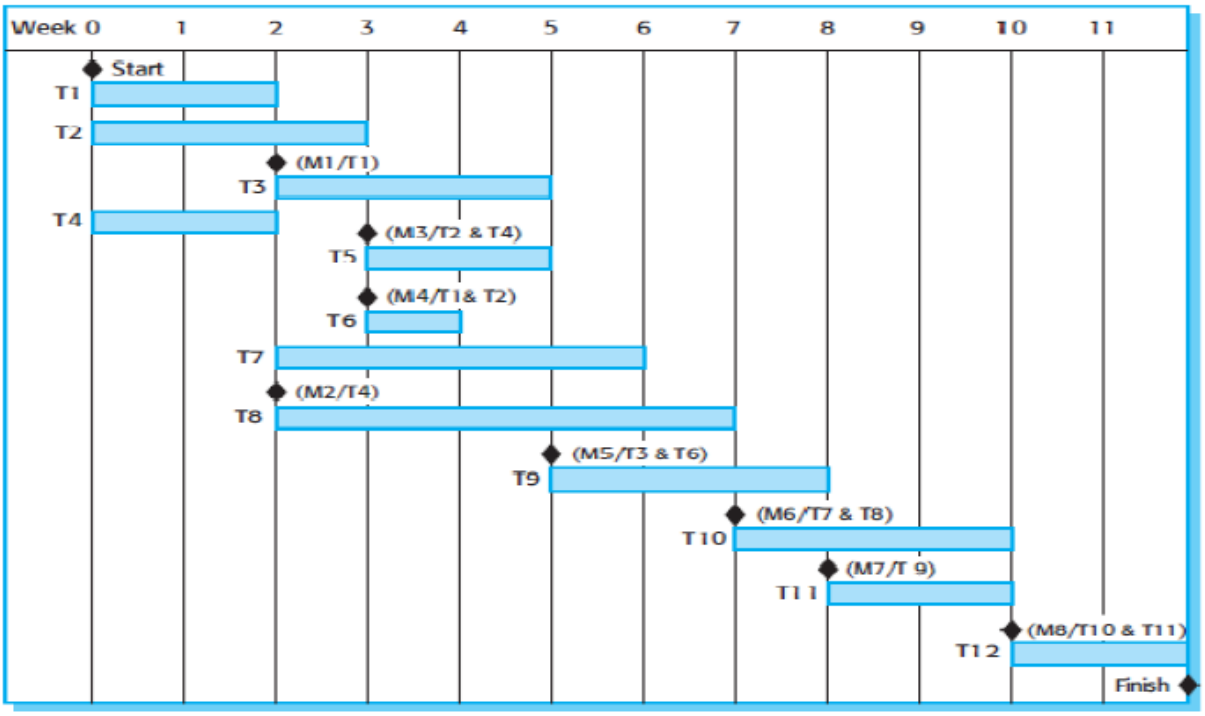


Figure 1.6: Activity Bar chart

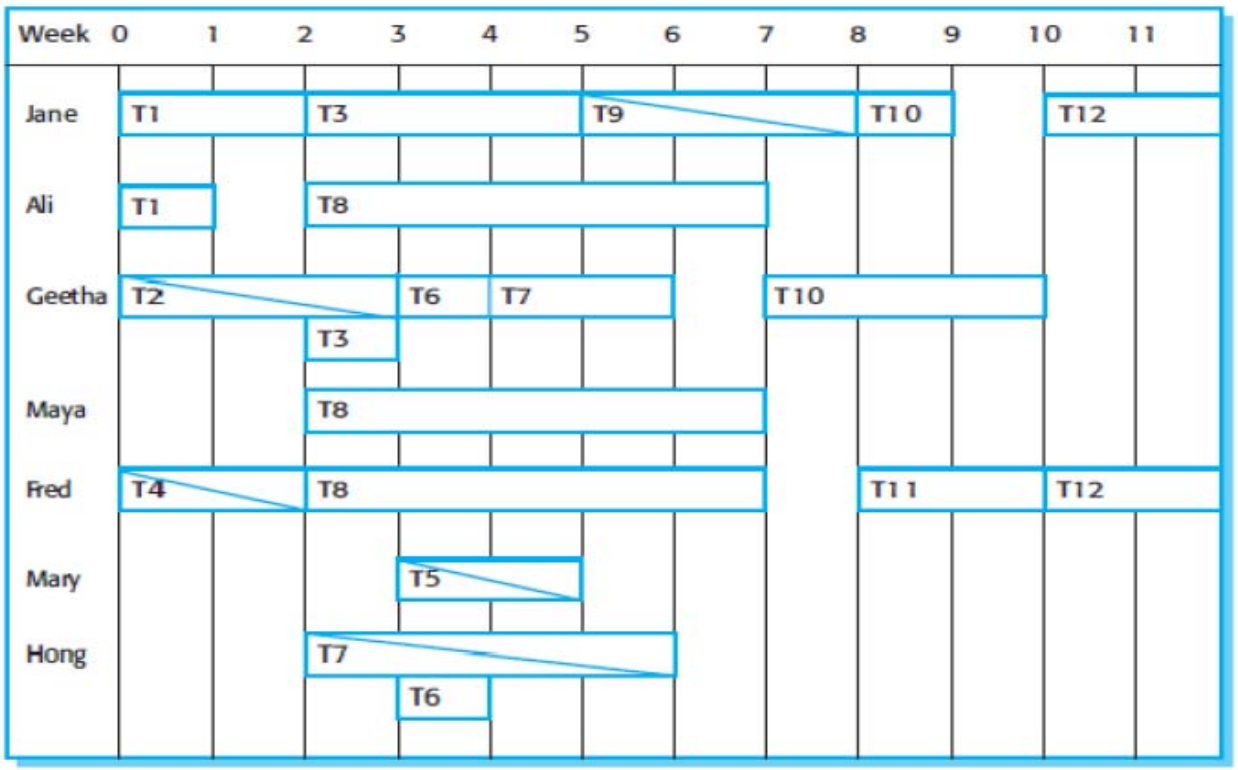


Figure 1.7: Staff allocation chart.

1.4 Estimation techniques

Organizations need to make software effort and cost estimates. There are two types of technique that can be used to do this:

Experience-based techniques: The estimate of future effort requirements is based on the manager's experience of past projects and the application domain. Essentially, the manager makes an informed judgment of what the effort requirements are likely to be.

***Algorithmic cost modeling*:** In this approach, a formulaic approach is used to compute the project effort based on estimates of product attributes, such as size, and process characteristics, such as experience of staff involved.

- Based on data collected from a large number of projects, **Boehm, et al. (1995)** discovered that startup estimates vary significantly.
- If the initial estimate of effort required is *x* months of effort, they found that the range may **be from 0.25*x* to 4*x* of the actual** effort as measured when the system was delivered. During development planning, estimates become more and more accurate as the project progresses **(Figure 1.8).**

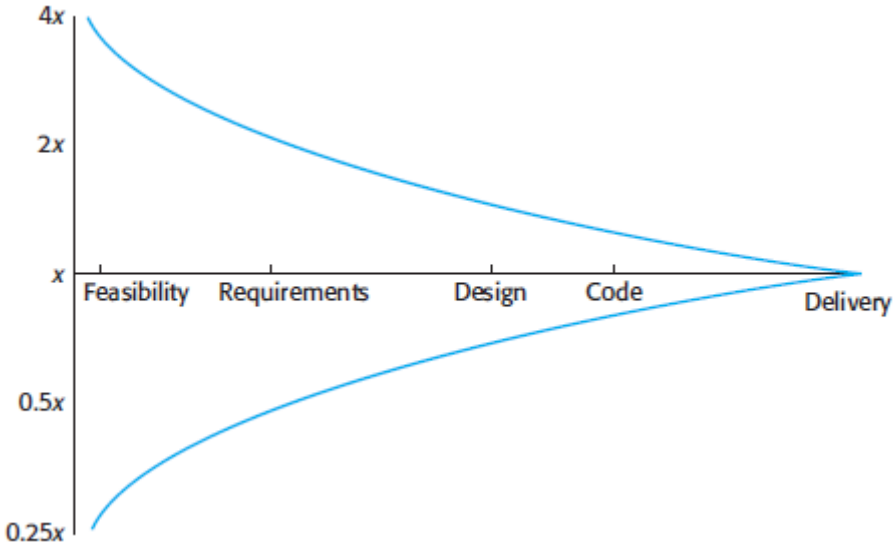


Figure 1.8: Estimate uncertainty

- **Experience-based techniques** rely on the **manager's experience of past projects** and the actual effort expended in these projects on activities that are related to software development.
- **Identify the deliverables** to be produced in a project and the different software components or systems that are to be developed. Document these in a spreadsheet, estimate them individually, and compute the total effort required.
- The **difficulty with experience-based techniques** is that a new software project may not have much in common with previous projects. Software development changes very quickly and a project will often use unfamiliar techniques such as **web services, COTS-based development, or AJAX.**

1.4.1 Algorithmic Modeling

- **Algorithmic cost modeling** uses a **mathematical formula** to predict project costs based on **estimates of the project size; the type of software being developed**; and other team, process, and product factors.
- An algorithmic cost model can be built by **analyzing the costs and attributes of completed projects**, and finding the closest-fit formula to actual experience.
- Algorithmic cost models are primarily used to make estimates of software development costs.
- **Algorithmic models for estimating effort** in a **software project** are mostly based on a **simple formula:**

$$\text{Effort} = A \times \text{Size}^B \times M$$

A is a constant factor which depends on local organizational practices and the type of software that is developed.

Size may be either an assessment of the code size of the software or a functionality estimate expressed in function or application points.

The value of **exponent B usually lies between 1 and 1.5**.

M is a multiplier made by combining process, product, and development attributes, such as the dependability requirements for the software and the experience of the development team.

- **The number of lines of source code (SLOC) in** the delivered system is the fundamental size metric that is used in many algorithmic cost models.
- Most **algorithmic estimation models** have an **exponential component** (B in the above equation) that is related to the size and complexity of the system. This reflects the fact that costs do not usually increase linearly with project size.
- The **more complex the system**, the **more these factors affect the cost**.
- Therefore, the **value of B usually increases** with the **size and complexity** of the system.

All algorithmic models have similar problems:

  - It is often **difficult to estimate Size at** an early stage in a project, when only the specification is available. Function-point and application-point estimates (see later) are easier to produce than estimates of code size but are still often inaccurate.
  - The **estimates of the factors contributing to B and M are subjective**. Estimates vary from one person to another, depending on their background and experience of the type of system that is being developed.

- Accurate code size estimation is difficult at an early stage in a project because the size of the final program depends on design decisions that may not have been made when the estimate is required.
- **The programming language used for system development also** affects the number of lines of code to be developed. A language like Java might mean that more lines of code are necessary than if C (say) was used.
- Algorithmic cost models are a **systematic way to estimate the effort required to** develop a system.
- Algorithmic cost modeling has been limited to a **small number of companies.**

- Practical use of algorithmic models, therefore, has to start with the published values for the model parameters. It is practically impossible for a modeler to know how closely these relate to their own organization.

1.4.2 The COCOMO 2 model

- An **empirical model** based on project experience.
- **Well-documented, 'independent'** model which is not tied to a specific software vendor.
- Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2.
- COCOMO 2 takes into account different approaches to software development, reuse, etc.
- COCOMO 2 incorporates a range of sub-models that produce increasingly detailed software estimates.
- The **sub-models in COCOMO 2(Figure 1.9)**
  - **Application composition model**. Used when software is composed from existing parts.
  - **Early design model.** Used when requirements are available but design has not yet started.
  - **Reuse model.** Used to compute the effort of integrating reusable components.
  - **Post-architecture model.** Used once the system architecture has been designed and more information about the system is available.

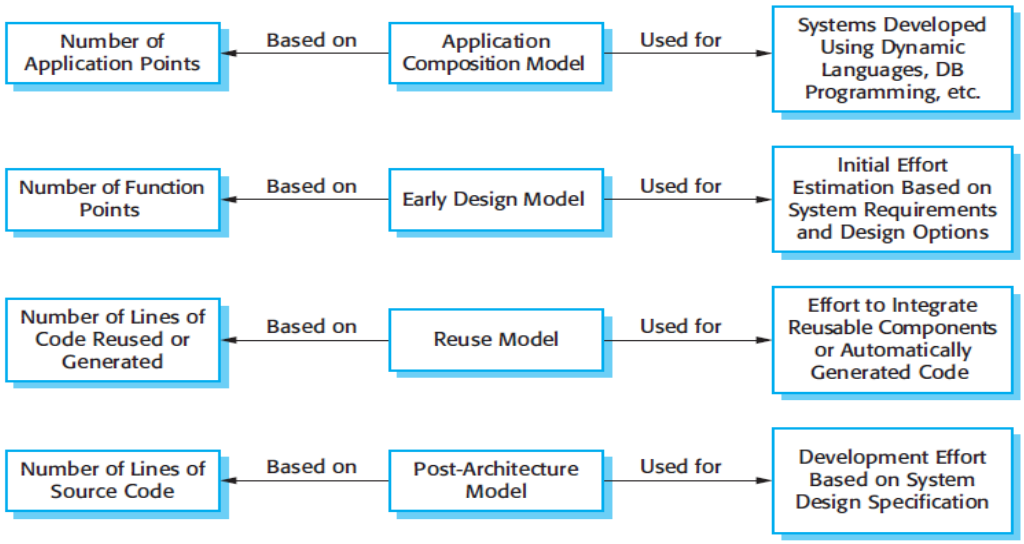


Figure 1.9: COCOMO estimation models

Application composition model

- For **estimating the efforts required for the prototyping** and the projects in which the existing software components are used application-composition model is introduced.
- **Supports prototyping projects** and projects where there is extensive reuse.
- Based on **standard estimates of developer productivity in application (object) points/month.**
- Takes CASE tool use into account.

Formula is

$$PM = (NAP \times (1 - \%reuse/100))/PROD$$

PM is the effort estimate in **person-months.**
NAP is the **total number of application points** in the delivered system.
"%reuse" is an estimate of the amount of reused code in the development.
PROD is the application-point productivity, as shown in Figure 1.10. The model produces an approximate estimate as it does not take into account the additional effort involved in reuse.

Developer's experience and capability	Very low	Low	Nominal	High	Very high
ICASE maturity capability	Very low	Low	Nominal	High	Very high
PROD (NAP/month)	4	7	13	25	50

Figure 1.10: Application-point productivity

Early design model

- This model is used in the early stage of the project development. That is after gathering the user requirements and before project development actually starts, this model is used. Hence approximate cost estimation can be made in this model.
- Estimates can be made after the requirements have been agreed.
- Based on a standard formula for algorithmic models

$$Effort = A \times Size^{B} \times M$$

A = 2.94 in initial calibration.
Size in KLOC,
B varies from 1.1 to 1.24 depending on novelty **of the project, development flexibility**, risk management approaches and the process maturity.
M=Multipliers.

$$M = PERS \times RCPX \times RUSE \times PDIF \times PREX \times FCIL \times SCED$$

- Multipliers reflect the capability of the developers, the non-functional requirements, the familiarity with the development platform, etc.
  - RCPX - product reliability and complexity;
  - RUSE - the reuse required;
  - PDIF - platform difficulty;
  - PREX - personnel experience;
  - PERS - personnel capability;
  - SCED - required schedule;
  - FCIL - the team support facilities.

This results in an effort computation as follows:

$$PM = 2.94 \times Size^{(1.1 - 1.24)} \times M$$

The reuse model

- ✧ Takes into account black-box code that is reused without change and code that has to be adapted to integrate it with new code.
- ✧ **There are two versions:**
  - **Black-box** reuse where code is not modified. An effort estimate (PM) is computed.
  - **White-box** reuse where code is modified. A size estimate equivalent to the number of lines of new source code is computed. This then adjusts the size estimate for new code.

➢ **Reuse model estimates 1**

For generated code:

$$PM_{Auto} = (ASLOC \times AT/100) / ATPROD \text{ // Estimate for generated code}$$

AT is percentage of automatically generated code.
ASLOC is the number of lines of generated code
ATPROD is the productivity of engineers in integrating this code.

- Boehm, et al. (2000) has measured ATPROD to be about 2,400 source statements per month. Therefore, if there are a total of 20,000 lines of reused source code in a system and 30% of this is automatically generated, then the effort required to integrate the generated code is:

$$(20{,}000 \times 30/100) / 2400 = 2.5 \text{ person-months // Generated code}$$

➢ **Reuse model estimates 2**

The following formula is used to calculate the number of equivalent lines of source code:

$$ESLOC = ASLOC \times AAM$$

ESLOC is the equivalent number of lines of new source code.
ASLOC is the number of lines of code in the components that have to be changed.
AAM is an Adaptation Adjustment Multiplier, as discussed below.

- The Adaptation Adjustment Multiplier (AAM) adjusts the estimate to reflect the additional effort required to reuse code. Simplistically, **AAM is the sum of three components:**
  1. **An adaptation component (referred to as AAF) t**hat represents the costs of making changes to the reused code. The adaptation component includes subcomponents that take into account design, code, and integration changes.
  2. **An understanding component (referred to as SU)** that represents the costs of understanding the code to be reused and the familiarity of the engineer with the code. SU ranges from 50 for complex unstructured code to 10 for well-written, object-oriented code.
  3. **An assessment factor (referred to as AA)** that represents the costs of reuse decision making. That is, some analysis is always required to decide whether or not code can be reused, and this is included in the cost as AA. AA varies from 0 to 8 depending on the amount of analysis effort required.

➢ **Reuse model estimates 3**

When code has to be understood and integrated:

$$ESLOC = ASLOC \times (1 - AT/100) \times AAM$$

ESLOC= no of lines of new source code
ASLOC and AT as before.
AAM is the adaptation adjustment multiplier computed from the costs of changing the reused code, the costs of understanding how to integrate the code and the costs of reuse decision making.

Post-architecture level

- **The code size is estimated as:**
  - **Number of lines of new code** to be developed;
  - **Estimate of equivalent** number of lines of new code computed using the reuse model;
  - An estimate of the number of lines of code that have to be modified according to requirements changes.

  Effort is calculated by,

$$PM = A \times Size^{B} \times M$$

  - In this model efforts should be estimated accurately.

Code size is having 3 components.

- The estimate about new lines of code that is added in the program.
- Equivalent number of source lines of code (ESLOC) is used in reuse model.
- Due to changes in requirements the lines of code get modified. The estimate of amount of code being modified.

- The values of B depends on **5 scale factors(figure 1.11)**
- **The exponent term (B) in** the effort computation formula is related to the levels of **project complexity.** As projects become more complex, the effects of increasing system size become more significant.
- These factors are rated on a six-point scale from 0 to 5,
- where 0 means 'extra high' and 5 means 'very low'.
- To calculate B, you add the ratings, divide them by 100, and add the result to 1.01 to get the exponent that should be used.
- For example, imagine that an organization is taking on a project in a domain in which it has little previous experience. The project client has not defined the process to be used or allowed time in the project schedule for significant risk analysis. A new development team must be put together to implement this system.
- Possible values for the ratings used in exponent calculation are therefore:
  ***Precedentedness*, rated low (4).** This is a new project for the organization.
  ***Development flexibility*, rated very high (1).** No client involvement in the development process so there are few externally imposed changes.
  ***Architecture/risk resolution*, rated very low (5).** There has been no risk analysis carried out.
  ***Team cohesion*, rated nominal (3).** This is a new team so there is no information available on cohesion.
  ***Process maturity*, rated nominal (3).** Some process control is in place.
- The sum of these values is 16. You then calculate the exponent by dividing this by 100 and adding the result to 0.01. The adjusted **value of B is therefore 1.17.**

- The overall effort estimate is refined **using an extensive set of 17 product**, process, and organizational attributes (cost drivers), rather **than the seven attributes used in the early design model.**

Scale factor	Explanation
Precedentedness	Reflects the previous experience of the organization with this type of project. Very low means no previous experience; extra-high means that the organization is completely familiar with this application domain.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; extra-high means that the dient sets only general goals.
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis; extra-high means a complete and thorough risk analysis.
Team cohesion	Reflects how well the development team knows each other and work together. Very low means very difficult interactions; extra-high means an integrated and effective team with no communication problems.
Process maturity	Reflects the process maturity of the organization. The computation of this value depends on the CMM Maturity Questionnaire, but an estimate can be achieved by subtracting the CMM process maturity level from 5.

Figure 1.11: Scale factors used in the exponent computation in the post-architecture model

Multipliers

- ✧ **Product attributes**
  - Concerned with required characteristics of the software product being developed.
- ✧ **Computer attributes**
  - Constraints imposed on the software by the hardware platform.
- ✧ **Personnel attributes**
  - Multipliers that take the experience and capabilities of the people working on the project into account.
- ✧ **Project attributes**
  - Concerned with the particular characteristics of the software development project.
- ✧ Figure 1.12 shows how the cost driver attributes can influence effort estimates.
- ✧ All of the other cost drivers have a nominal value of 1, so they do not affect the computation of the effort.

Exponent value	1.17
System size (including factors for reuse and requirements volatility)	128,000 DSI
Initial COCOMO estimate without cost drivers	**730 person-months**
Reliability	Very high, multiplier = 1.39
Complexity	Very high, multiplier = 1.3
Memory constraint	High, multiplier = 1.21
Tool use	Low, multiplier = 1.12
Schedule	Accelerated, multiplier = 1.29
Adjusted COCOMO estimate	**2,306 person-months**
Reliability	Very low, multiplier = 0.75
Complexity	Very low, multiplier = 0.75
Memory constraint	None, multiplier = 1
Tool use	Very high, multiplier = 0.72
Schedule	Normal, multiplier = 1
Adjusted COCOMO estimate	**295 person-months**

Figure 1.12: The effect of cost drivers on effort estimates

1.4.3 Project duration and staffing

- As well as effort estimation, managers must estimate the calendar time required to complete a project and when staff will be required.
- **Calendar time can be estimated using a COCOMO 2 formula**

$$TDEV = 3 \times (PM)^{(0.33 + 0.2*(B - 1.01))}$$

TDEV is the nominal schedule for the project, in calendar months, ignoring any multiplier that is related to the project schedule.
PM is the effort computed by the COCOMO model.
B is the complexity-related exponent.

- if B = 1.17 and PM = 60 then

$$TDEV = 3 \times (60)^{0.36} = 13 \text{ months}$$

Staffing requirements

- Staff required can't be computed by diving the development time by the required schedule.
- The number of people working on a project varies depending on the phase of the project.
- The more people who work on the project, the more total effort is usually required.
- A very rapid build-up of people often correlates with schedule slippage.

QUALITY MANAGEMENT

Quality Management

Quality management ensures that an organization, product or service is consistent. It has **four** main components: **quality planning, quality assurance, quality control and quality improvement.** Quality management is focused not only on product and service quality, but also on the means to achieve it. Quality management, therefore, uses quality assurance and control of processes as well as products to achieve more consistent quality.

- Three principal concerns:

1. **At the organizational** level, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high-quality software.
2. **At the project level,** quality management involves the application of specific quality processes and checking that these planned processes have been followed.
3. **At the project level**, quality management is also concerned with establishing a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.

- The terms **'quality assurance'** and **'quality control'** are widely used in manufacturing industry.
- **Quality assurance (QA) i**s the **definition of processes** and **standards that should lead to high-quality products** and the **introduction of quality processes** into the manufacturing process.
- **Quality control is the application of these quality processes to** weed out products that are not of the required level of quality.
- Quality management provides an independent check on the software development process.
- The quality management process checks the project deliverables to ensure that they are consistent with organizational standards and goals(Figure 2.1)

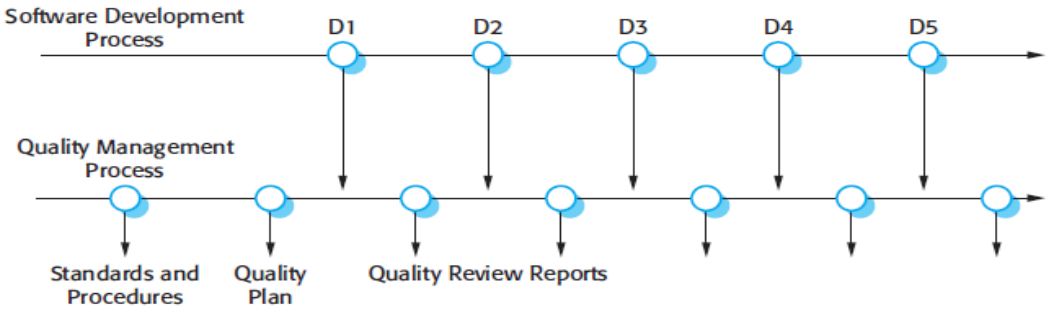


Figure 2.1: Quality management and software development

Quality planning

Definition: Quality planning can be defined as process in which the **quality plan for the project is prepared** .It must include the desired software qualities and description on how to access these qualities. The quality plan should define the quality assessment process.

- ✧ It should set out which organisational standards should be applied and, where necessary, define new standards to be used.

List/ Structure of Quality plans

1. **Product Introduction:** A description of the product, its intended market, and the quality expectations for the product
2. **Product Plans**: The critical release dates and responsibilities for the product, along with plans for distribution and product servicing.
3. **Process Descriptions**: The development and service processes and standards that should be used for product development and management.
4. **Quality Goals**: The quality goals and plans for the product, including an identification and justification of critical product quality attributes.
5. **Risks and Risk Management**: The key risks that might affect product quality and the actions to be taken to address these risks.

2.1 Software Quality

Software Quality: simplistically, means that a product should meet its specification. Software quality is not directly comparable with quality in manufacturing.

- **This is problematical for software systems**
  - There is a tension between customer quality requirements (efficiency, reliability, etc.) and developer quality requirements (maintainability, reusability, etc.);
  - Some quality requirements are difficult to specify in an unambiguous way.
  - Software specifications are usually incomplete and often inconsistent.
- The focus may be 'fitness for purpose' rather than specification conformance.
- Therefore software quality is not just about whether the software functionality has been correctly implemented, but also depends on non-functional system attributes.
- Boehm suggested that there were 15 important software quality attributes, as shown in Figure 2.2.
- These attributes relate to the software dependability, usability, efficiency, and maintainability.

Safety	Understand ability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learn ability

Figure 2.2: Software quality attributes

Software fitness for purpose

- Have programming and documentation standards been followed in the development process?
- Has the software been properly tested?
- Is the software sufficiently dependable to be put into use?
- Is the performance of the software acceptable for normal use?
- Is the software usable?
- Is the software well-structured and understandable?

2.1.1 Quality conflicts

- It is not possible for any system to be optimized for all of these attributes – **for example, improving robustness may lead to loss of performance.**
- The quality plan should therefore define the most important quality attributes for the software that is being developed.
- The plan should also include a definition of the quality assessment process, an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.

2.1.2 Process and product quality

- The quality of a developed product is influenced by the quality of the production process.
- This is important in software development as some product quality attributes are hard to assess.
- However, there is a very complex and poorly understood relationship between software processes and product quality.
    - **The application of individual skills and experience is particularly important in software development;**
    - **External factors such as the novelty of an application or the need for an accelerated development schedule may impair product quality.**
- Software quality management is that the **quality of software** is directly related to the quality of the software development process.
- This again comes from manufacturing systems where product quality is intimately related to the production process.
- A manufacturing process involves **configuring, setting up** and **operating the machines** involved in the process. Once the machines are operating correctly, product quality naturally follows.
- You measure the **quality of the product** and **change the process until** you achieve the quality level that you need.
- Figure 2.3 illustrates this process-based approach to achieving product quality.

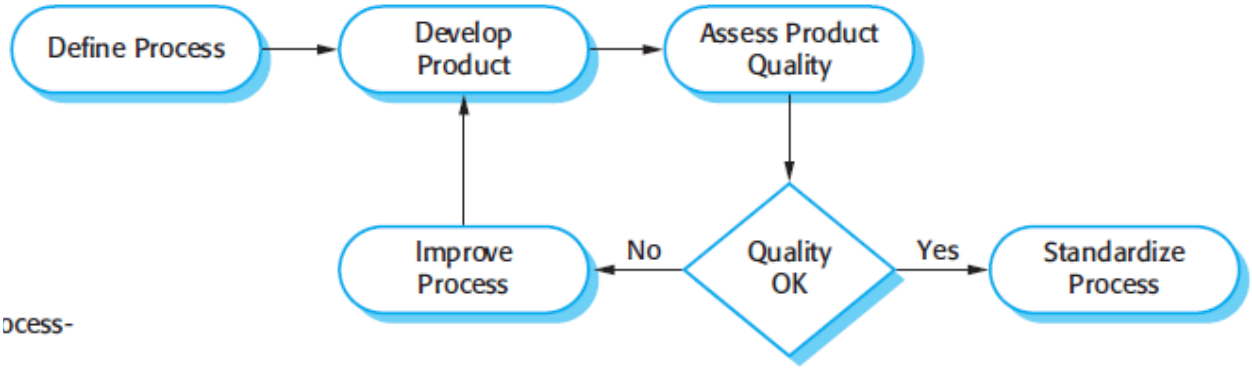


Figure 2.3: Process Based Quality

2.2 Software standards

- **Standards define the required attributes of a product or process.** They play an important role in quality management.
- Standards may be international, national and organizational or project standards.

There is **2 type of standards** they are, product standards and process standards.

1. ***Product standards:*** These apply to the software product being developed. They include document standards, such as the structure of requirements documents, documentation standards, such as a standard comment header for an object class definition, and coding standards, which define how a programming language should be used.
2. ***Process standards:*** These define the processes that should be followed during software development. They should encapsulate good development practice. Process standards may include definitions of specification, design and validation processes, process support tools, and a description of the documents that should be written during these processes.

Product standards	Process standards
Design review form	Design review conduct
Requirements document structure	Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Figure 2.4: Product and process standards

Importance of standards

- **Encapsulation** of best practice- avoids repetition of past mistakes.
- They are a **framework for defining what quality** means in a particular setting i.e. that organization's view of quality.

- They provide continuity - new staff can understand the organisation by understanding the standards that are used.

Problems with standards

- Software engineers sometimes think that the product **standard which is set by the company is irrelevant**. Hence they become reluctant in adopting these standards,
- They may not be seen as relevant and up-to-date by software engineers.
- They often involve too much bureaucratic form filling.
- If they are unsupported by software tools, tedious form filling work is often involved to maintain the documentation associated with the standards.
- **To avoid these problems. some possible solutions are (importance of standard)**
    1. Involve software engineers in developing the product standard so that they can understand the **need of particular standard.**
    2. **Review and modify** the standard periodically in order to adopt the new changes.
    3. Make use **of automated software tool** to support the standard whenever possible.

Standards development

- **Involve software engineers in the selection of product standards** If developers understand why standards have been selected, they are more likely to be committed to these standards.
- **Review and modify standards regularly to reflect changing technologies** Standards are expensive to develop and they tend to be enshrined in a company standards handbook. Because of the costs and discussion required, there is often a reluctance to change them. A standards handbook is essential but it should evolve to reflect changing circumstances and technology.
- **Provide software tools to support standards** Developers often find standards to be a bugbear when conformance to them involves tedious manual work that could be done by a software tool. If tool support is available, very little effort is required to follow the software development standards. For example, document standards can be implemented using word processor styles.

2.3.1 ISO 9001 standards framework

- An international set of standards that can be used as a basis for developing quality management systems.
- ISO 9001, the most general of these standards, applies to organizations that design, develop and maintain products, including software.
- The ISO 9001 standard is a framework for developing software standards.
    - It sets out general quality principles, describes quality processes in general and lays out the organizational standards and procedures that should be defined. These should be documented in an organizational quality manual.

- To be conformant with ISO 9001, a company must have defined the types of process shown in Figure 2.5 and have procedures in place that demonstrate that its quality processes are being followed.
- This allows flexibility across industrial sectors and company sizes. Quality standards can be defined that are appropriate for the type of software being developed.
- The relationships between ISO 9001, organizational quality manuals, and individual project quality plans are shown in Figure 2.6
- Some people think that ISO 9001 certification means that the quality of the software produced by certified companies will be better than that from uncertified companies. This is not necessarily true.
- The ISO 9001 standard focuses on ensuring that the organization has quality management procedures in place and it follows these procedures. There is no guarantee that ISO 9001 certified companies use the best software development practices or that their processes lead to high-quality software.

ISO 9001 certification

- Quality standards and procedures should be documented in an organisational quality manual.
- An external body may certify that an organisation's quality manual conforms to ISO 9000 standards.
- Some customers require suppliers to be ISO 9000 certified although the need for flexibility here is increasingly recognised.

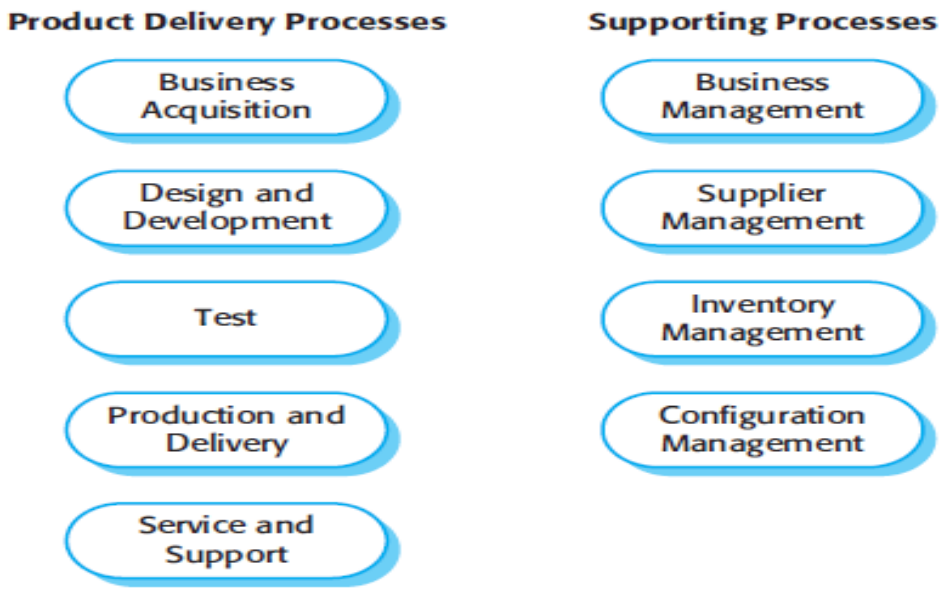


Figure 2.5: ISO 9001 core processes

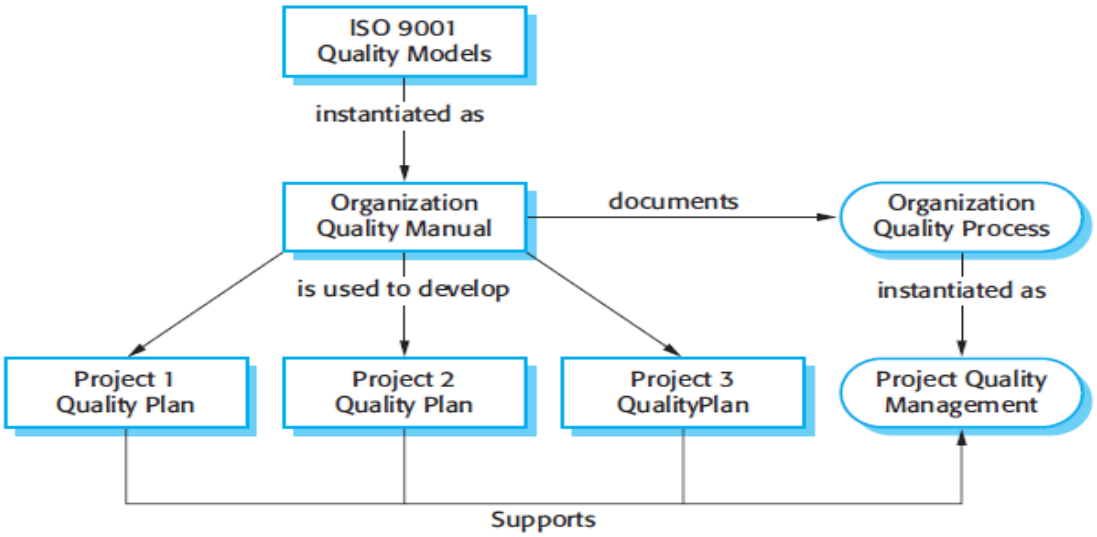


Figure 2.6: ISO 9001 and quality management

2.3 Reviews and inspections

Reviews	Examining the software products.
Inspections	Finding the bugs or errors in software product.

2.3.1 Quality reviews

In order to check the **quality of product** deliverables reviews and inspections are conducted

- In review process A group of people carefully examine part or all of a software system and its associated documents. **Project managers** may then use these assessments to make **planning decisions** and allocate resources to the development Process
- Quality reviews are based on documents that have been produced **during the software development process**
- They are also used to help discover problems and omissions in the software or project documentation.
- The purpose of reviews and inspections is to improve software quality, not to assess the performance of people in the development team.
- Reviewing is a public process of error detection, compared with the more private component-testing process.
- The conclusions of the review should be **formally recorded as part of the quality management process.**
- Code, designs, specifications, test plans, standards, etc. can all be reviewed.
- Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.

- Review teams should normally have a core of three to four people who are selected as principal reviewers. One member should be a senior designer who will take the responsibility for making significant technical decisions.

2.3.2 Review Process

Majorly there are **3 phases,**

Pre-review activities
The review meeting
Post review meeting

1. ***Pre-review activities*** These are preparatory activities that are essential for the review to be effective. Typically, **pre-review activities are concerned with review planning** and **review preparation**. Review planning involves setting up a review team, arranging a **time and place for the review**, and distributing the documents to be reviewed. Individual review team members read and understand the software or **documents** and **relevant standards**. They work independently to find errors, omissions, and departures from standards. Reviewers may supply written comments on the software if they cannot attend the review meeting.
2. ***The review meeting*** During the review meeting, an author of the document or program being reviewed should 'walk through' the document with the review team. The review itself should be relatively short two hours at most. **One team member should chair the review** and **another should formally record all review decisions** and **actions to be taken.** During the review, the chair is responsible for ensuring that all written comments are considered. The review chair should sign a record of comments and actions agreed during the review.

3. ***Post-review activities*** After a **review meeting has finishe**d, **the issues** and **problem raised during the review must be addressed**. This may involve **fixing software bugs, refactoring software so that it conforms to quality standards, or rewriting documents.** Sometimes, the problems discovered in a quality review are such that a management review is also necessary to decide if more resources should be made available to correct them. After changes have been made, the review chair may check that the review comments have all been taken into account.

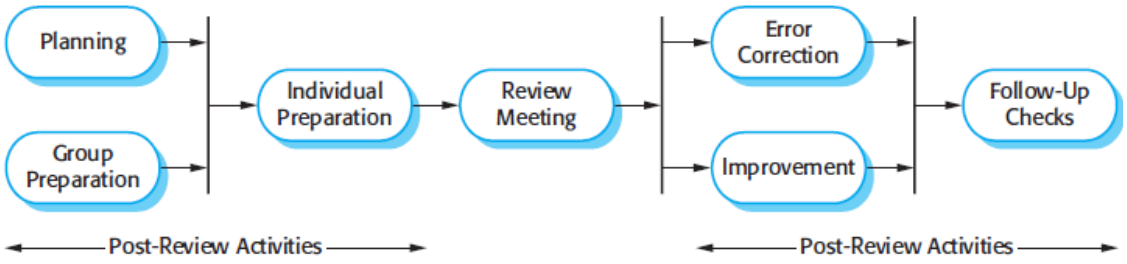


Figure 2.7: The software review process

2.3.3 Program Inspections

- Program inspection can be defined as the **process in which system is reviewed to find errors, anomalies**. During this inspection process the requirement of the system, a design model or source code can be reviewed.
- Inspections do not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design,configuration data, test data, etc.).
- They have been shown to be an effective technique for **discovering program errors**
- Program inspections involve **team members from different backgrounds who make a careful, line-by-line review of the program source code.**
- They look for defects and problems and describe these at an inspection meeting. Defects may be logical errors, anomalies in the code that might indicate an erroneous condition or features that have been omitted from the code During an inspection, a checklist of common programming errors is often used to focus the search for bugs.

Advantages of inspection:

1. Single inspection can discover multiple errors.
2. Incomplete version also detected by this inspection method.
3. Can find program defects, poor programming styles.

Agile methods and inspections

- Agile processes rarely use formal inspection or peer review processes.
- Rather, theyrely on team members cooperating to check each other's code, and informal guidelines, such as 'check before check-in', which suggest that programmers should check their own code.
- Extreme programming practitioners argue that pair programming is an effective substitute for inspection as this is, in effect, a continual inspection process.
- Two people look at every line of code and check it before it is accepted.

Inspection checklists

- Checklist of common errors should be used to drive the inspection.
- Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- **Examples:** Initialisation, Constant naming, loop termination, array bounds, etc.

Fault class	Inspection check
Data faults	• Are all program variables initialized before their values are used? • Have all constants been named? • Should the upper bound of arrays be equal to the size of the array or Size -1? • If character strings are used, is a delimiter explicitly assigned? • Is there any possibility of buffer overflow?
Control faults	• For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for? • If a break is required after each case in case statements, has it been included?
Input/output faults	• Are all input variables used? • Are all output variables assigned a value before they are output? • Can unexpected inputs cause corruption?
Interface faults	• Do all function and method calls have the correct number of parameters? • Do formal and actual parameter types match? • Are the parameters in the right order? • If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	• If a linked structure is modified, have all links been correctly reassigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly deallocated after it is no longer required?
Exception management faults	• Have all possible error conditions been taken into account?

Figure 2.8: Inspection Checklists

2.4 Software Measurement and Metrics

- Software measurement is **concerned with deriving a numeric value for an attribute of a software product or process.**
- By comparing these values to each other and to the standards that apply across an organization, from this we can **draw conclusions about the quality of software** or assess the effectiveness of software processes, tools, and methods
- The long-term goal of software measurement is to use measurement **in place of reviews to make judgments about software quality**. Using software measurement, a system could ideally be assessed using a range of metrics and, from these measurements, a value for the quality of the system could be decided.
- Although some companies have introduced measurement programmes, most organisations still don't make systematic use of software measurement.

2.4.1 Software metric

- **Definition**: The software metrics is a characteristic of software system or system development process.
- **Example:** Lines of code in a program, number of person-days required to develop a component.

- Allow the software and the software process to be quantified.
- May be **used to predict product** attributes or to control the software process.
- Product metrics can be used for general predictions or to identify anomalous components. (normal or expected)

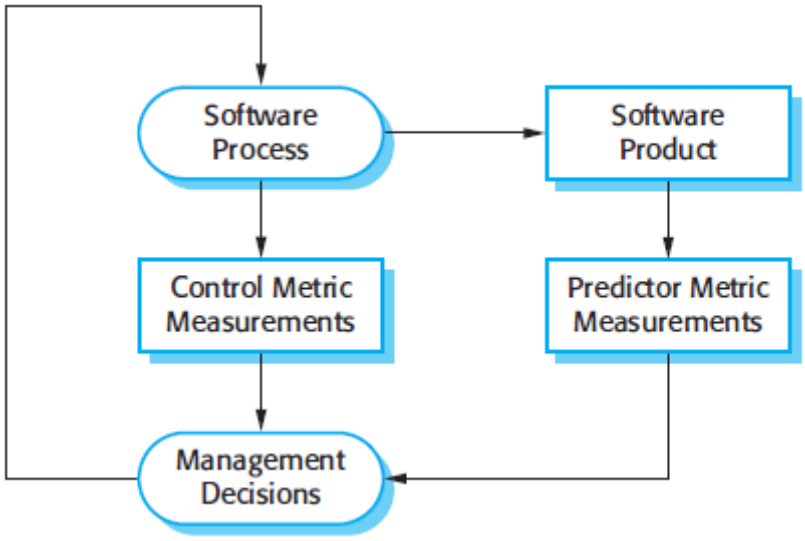


Figure 2.9: Predictor and Control Measurements

There are 2 ways in which software measurement of a software system may be used,

1. **To assign a value to system quality attributes**
   By measuring the characteristics of system components, such as their cyclomatic complexity, and then aggregating these measurements, you can assess system quality attributes, such as maintainability.
2. **To identify the system components whose quality is sub-standard**
   Measurements can identify individual components with characteristics that deviate from the norm. For example, you can measure components to discover those with the highest complexity. These are most likely to contain bugs because the complexity makes them harder to understand.

2.4.2 Relationship between internal and external software quality attributes

- By the Metrics A software property can be measured.
- **External attributes**: Are those attributes can be seen by the **user as well as by developer** who has designed that code, these attributes shows the characteristics of software quality .**example**: maintainability, reliability etc
- **Internal attributes**: Are those attributes that can be measured directly **mostly by developer side only**, **example**: size of code
- The relationship exists between what we can measure and what we want to know. We can only measure internal attributes but are often more interested in external software attributes as shown in figure 2.10

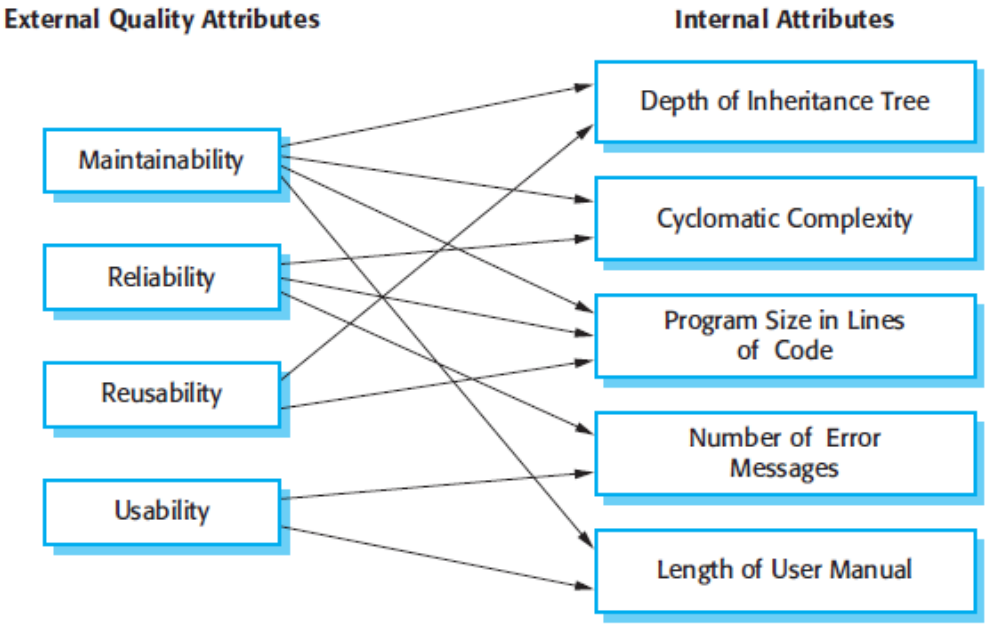


Figure 2.10: Relationship between Internal And External Software.

Problems with measurement in industry

- It is impossible to quantify the return on investment of introducing an organizational metrics program.
- There are no standards for software metrics or standardized processes for measurement and analysis.
- In many companies, software processes are not standardized and are poorly defined and controlled.
- Most work on software measurement has focused on code-based metrics and plan-driven development processes. However, more and more software is now developed by configuring ERP systems or COTS.
- Introducing measurement adds additional overhead to processes.

2.4.3 Product metrics

- Product metrics are predictor metrics that are used to measure internal attributes of a software system. Examples of product metrics include the system size, measured in lines of code, or the number of methods associated with each object class.

- A quality metric should be a predictor of product quality. List of static metrics as shown in figure 2.11

There are 2 types of metrics , Dynamic and static metrics

1. **Dynamic metrics: are closely related to software quality attributes**: It is relatively easy to measure the response time of a system (performance attribute) or the number of failures (reliability attribute).

2. **Static metrics: Have an indirect relationship with quality attributes**: You need to try and derive a relationship between these metrics and properties such as complexity, understandability and maintainability.

Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.
Length of identifiers	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.

Figure 2.11: Static Software Product Metrics

Object-oriented metric	Description
Weighted methods per class (WMC)	This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so cannot be reused effectively as superclasses in an inheritance tree.
Depth of inheritance tree (DIT)	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.
Coupling between object classes (CBO)	Classes are coupled when methods in one class use methods or instance variables defined in a different class. CBO is a measure of how much coupling exists. A high value for CBO means that classes are highly dependent, and therefore it is more likely that changing one class will affect other classes in the program.
Response for a class (RFC)	RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class. Again, RFC is related to complexity. The higher the value for RFC, the more complex a class and hence the more likely it is that it will include errors.
Lack of cohesion in methods (LCOM)	LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes. The value of this metric has been widely debated and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics.

Figure 2.12: Object Oriented Metrics Lists

2.4.4 Software component analysis

- Software measure is a process in which the **software components are analyzed for obtaining the values of software metrics**. These values are compared with each other and with previous projects metrics. If any **anomalous components are** obtained then quality assurance efforts are focused.
- System **component can be analyzed separately** using a range of metrics.

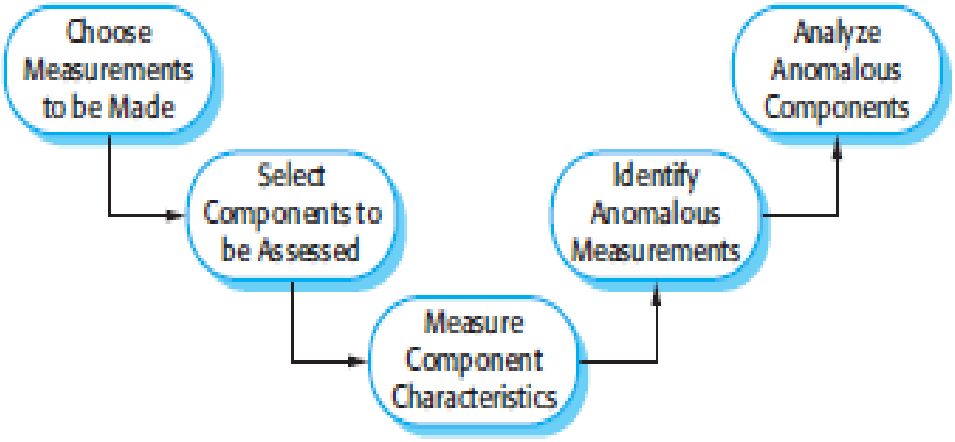


Figure 2.13: Product measurement process (component analysis)

The key stages in this component measurement process are:

1. Choose measurements to be made: The questions that the measurement is intended to answer should be formulated and the measurements required to answer these questions defined. Measurements that are not directly relevant to these questions need not be collected.

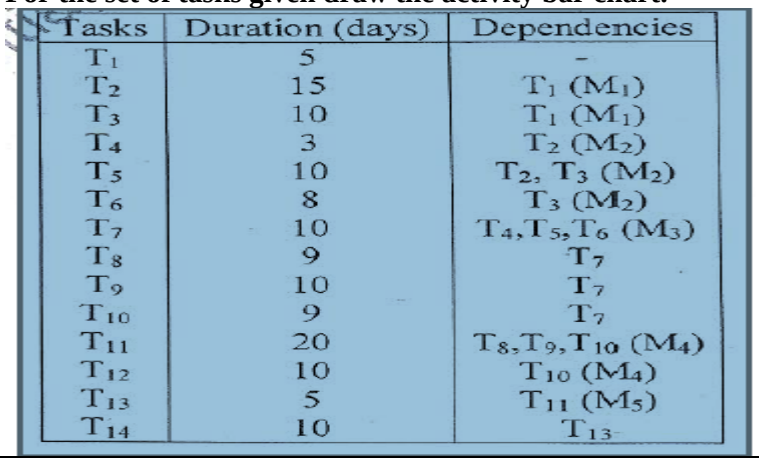
2. Select components to be assessed: You may not need to assess metric values for all of the components in a software system. Sometimes, you can select a representative selection of components for measurement, allowing you to make an overall assessment of system quality. At other times, you may wish to focus on the core components of the system that are in almost constant use. The quality of these components is more important than the quality of components that are only rarely used.

3. Measure component characteristics The selected components are measured and the associated metric values computed. This normally involves processing the component representation (design, code, etc.) using an automated data collection tool. This tool may be specially written or may be a feature of design tools that are already in use.

4. Identify anomalous measurements After the component measurements have been made, you then compare them with each other and to previous measurements that have been recorded in a measurement database. You should look for unusually high or low values for each metric, as these suggest that there could be problems with the component exhibiting these values.

5. Analyze anomalous components When you have identified components that have anomalous values for your chosen metrics, you should examine them to decide whether or not these anomalous metric values mean that the quality of the component is compromised. An anomalous metric value for complexity (say) does not necessarily mean a poor quality component. There may be some other reason for the high value, so may not mean that there are component quality problems

QUESTION BANK
MODULE 4

Sl.No	Questions	Marks
1.	**Explain the factors affecting software pricing.**	**6M**
2.	**Describe project plan sections with the different plans.**	**12M**
3.	**Explain planning process.**	**6M**
4.	**For the set of tasks given draw the activity bar chart.** 	**10M**
5.	**Discuss Algorithmic cost modelling.**	**5M**
6.	Explain with a neat diagram COCOMO II model or Explain sub-models of COCOMO II model	**10M**
7.	Explain i) Application composition model ii) An early design model iii) Reuse model iv) Post architecture model	**10M**
8.	Write a note on project duration and staffing requirement.	**6M**
9.	Explain structure of plan.	**6M**
10.	Define quality plan, software quality and list Quality attribute.	**5M**
11.	List out the importance of software standards.	**5M**
12.	What is a Software standard? Explain the Product and Process standard	
13.	Explain ISO 9001 Standards.	
14.	Define reviews and Explain review process explain the inspection process	**7M**
15.	With neat diagram explain software component analysis	
16.	Explain inspection roles and inspection checklist for software inspection process.	**10M**
17.	Explain Product Metrics.	**10M**

Tasks	Duration (days)	Dependencies
T_1	5	-
T_2	15	T_1 (M_1)
T_3	10	T_1 (M_1)
T_4	3	T_2 (M_2)
T_5	10	T_2, T_3 (M_2)
T_6	8	T_3 (M_2)
T_7	10	T_4,T_5,T_6 (M_3)
T_8	9	T_7
T_9	10	T_7
T_{10}	9	T_7
T_{11}	20	T_8,T_9,T_{10} (M_4)
T_{12}	10	T_{10} (M_4)
T_{13}	5	T_{11} (M_5)
T_{14}	10	T_{13}

Module-5

Agile Software Development: Coping with Change, The Agile Manifesto: Values and Principles. Agile methods: SCRUM (Ref "The SCRUM Primer, Ver 2.0") and Extreme Programming, Plan-driven and agile development, Agile project management, Scaling agile methods.

1.1Coping with Change

- Change is inevitable in all large software projects.
  - Business changes lead to new and changed system requirements
  - New technologies open up new possibilities for improving implementations
  - Changing platforms require application changes
- Change leads to rework so the costs of change include both rework (e.g. re-analysing requirements) as well as the costs of implementing new functionality

There are **2 related approaches** that may be used to reduce the costs of rework:

1. **Change avoidance**, where the software process includes activities that can anticipate possible changes before significant rework is required.
   For example, a prototype system may be developed to show some key features of the system to customers.
2. **Change tolerance**, where the process is designed so that changes can be accommodated at relatively low cost.
   - This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have be altered to incorporate the change.

Two(2) ways of coping with changes

1. **System prototyping** : where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of some design decisions.
2. **Incremental delivery :** where system increments are delivered to the customer for comment and experimentation

1.1.1 Prototyping

- A prototype is an **initial version** of a software system that is used to demonstrate concepts, try out design options, and find out more about the problem and its possible solutions.
- In the **system design process, a prototype** can be used to explore particular software solutions and to support user interface design.

A prototype can be used in:

- The requirements engineering process to help with requirements elicitation and validation;
- In design processes to explore options and develop a UI design;
- In the testing process to run back-to-back tests.

- System prototypes allow users to see how well the system supports their work. They may get new ideas for requirements, and find areas of strength and weakness in the software.
- For example, a **database design may be prototyped and tested** to check that it supports efficient data access for the most common user queries. Prototyping is also an essential part of the user interface design process. Because of the dynamic nature of user interfaces, textual descriptions and diagrams are not good enough for expressing the user interface requirements.
- A process model for prototype development is shown in Figure 1.1

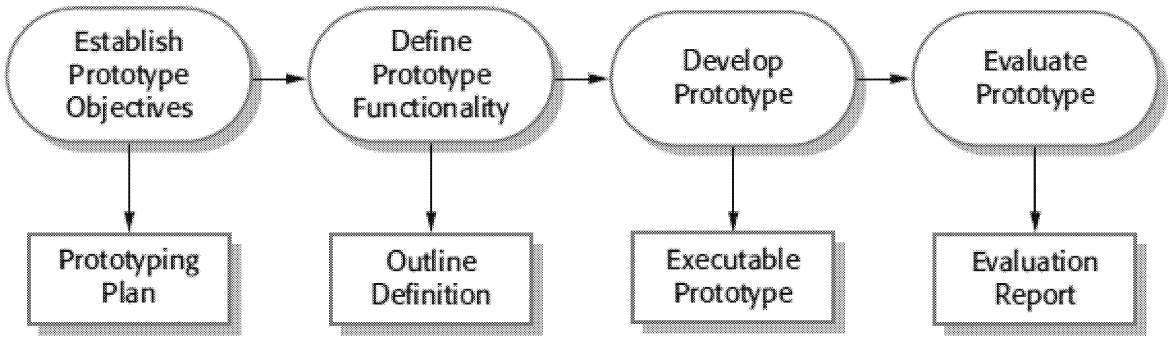


Figure 1.1: Process of prototyping development

Procedure is as follows
Step1: initially object(goal) of the prototype should be decided, like These may be to develop a system to prototype the user interface, to develop a system to validate functional system requirements
Step 2: functionalites are expected in current prototype. Some minor functionalities can be emitted from the prototype reduce the cost and accelerate the process of prototyping
Step 3: develop prototype so that executable prototype can be set ready.
Step 4: The final stage of the process is prototype evaluation. Provision must be made during this stage for user training and the prototype objectives should be used to derive a plan for evaluation.

Prototype development
- May be based on rapid prototyping languages or tools
- May involve leaving out functionality
  - Prototype should focus on areas of the product that are not well-understood;
  - Error checking and recovery may not be included in the prototype;
  - Focus on functional rather than non-functional requirements such as reliability and security

Benefits of prototyping
- Improved system usability.
- A closer match to users' real needs.
- Improved design quality.

- Improved maintainability.
- Reduced development effort.

Drawbacks:

1. A general problem with prototyping is that the prototype may not necessarily be used in the same way as the final system.
2. The tester of the prototype may not be typical of system users.
3. The training time during prototype evaluation may be insufficient.
4. If the prototype is slow, the evaluators may adjust their way of working and avoid those system features that have slow response times.

Throw-away prototypes

- Prototypes should be discarded after development as they are not a good basis for a production system:
  - It may be impossible to tune the system to meet non-functional requirements;
  - Prototypes are normally undocumented;
  - The prototype structure is usually degraded through rapid change;
  - The prototype probably will not meet normal organizational quality standards.

1.1.2 Incremental Delivery

- Incremental delivery is an approach **to software development** where some of the **developed increments** are delivered to the customer and deployed for use in an operational environment.
- In an incremental delivery process, **customers identify, in outline**, the services to be provided by the system
- Incremental deliver is an approach in which instead of single delivery the development is broken into increments and each increment is responsible for delivering some functionality of the system
- The user requirements are prioritized and highest priority requirements are included in the requirements.
- The incremental delivery process can be shown in Figure 1.2.

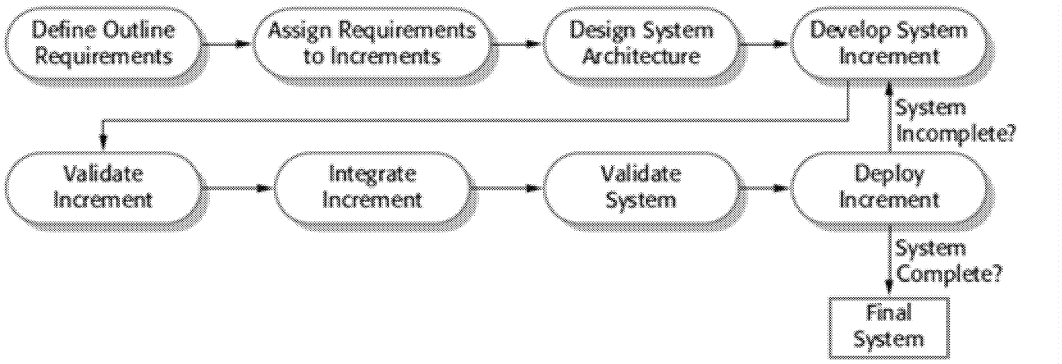


Figure 1.2 :Incremental Delivery

Advantages:

- Customer can understand and analyze the functionalities of the system in each increment.
- Each increment helps the customer to elicit the requirement for next increment.
- Chances to risks are reduced.
- The highest priority requirements are tested fully.

Drawbacks:

- Most systems require a set of basic facilities that are used by different parts of the system. until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
- Iterative development can also be difficult when a replacement system is being developed. Therefore, getting useful customer feedback is difficult.
- The essence of iterative processes is that the specification is developed in conjunction with the software.
- In the incremental approach, there is no complete system specification until the final increment is specified. This requires a new form of contract, which large customers such as government agencies may find difficult to accommodate.

Agile Software Development

- The software may then be out of date when it is delivered.
- Software development processes that plan on completely specifying the requirements and then designing, building, and testing the system are not geared to rapid software development.
- **Rapid development** and **delivery** is now often the most important requirement for software systems
  - **Businesses operate in a fast –changing requirement** and it is practically impossible to produce a set of stable software requirements
  - Software has to evolve quickly to reflect changing business needs.
- **Rapid software development**
  - **Specification, design** and **implementation are inter-leaved.**
  - System is developed as a series of versions with stakeholders involved in version evaluation.
  - **User interfaces** are often developed using an **IDE** and **graphical toolset**.

2.1 Agile Methods

- Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
  - Focus on the code rather than the design
  - Are based on an iterative approach to software development
  - Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.

- The **aim of agile methods is to reduce overheads in the software process** (e.g. by limiting documentation) and to be able to respond quickly to changing requirements without excessive rework.
- Agile (quick) methods universally rely on an **iterative approach** to software specification, development and were designed primarily to support business application development where the **system requirements** usually **changed rapidly** during the development process .they are **intended to deliver working software quickly** to customers, who can then probably the best know agile method is **extreme programming** .

2.1.2 Manifesto for Agile

Agile manifesto includes 4 values. They are,

1. Individuals and interactions over processes and tools

- People make biggest impact on success
  - Process and environment help, but will not create success
- Strong individuals not enough without good team interaction.
  - Individuals may be stronger based on their ability to work on a team
- Tools can help, but bigger and better tools can hinder more than help
  - Simpler tools can be better

2. Working software over comprehensive documentation

Documentation is important, but too much is worse than too little

- Long time to produce, keep in sync with code
- Keep documents short and salient

- Focus effort on producing code, not descriptions of it
  - Code should document itself
  - Knowledge of code kept within the team
- Produce no document unless its need is immediate and significant.

3. Customer collaboration over contract negotiation

- Not reasonable to specify what's needed and then have no more contact until final product delivered
- Get regular customer feedback
- Use contracts to specify customer interaction rather than requirements, schedule, and cost

4. Responding to change over following a plan

- Environment, requirements, and estimates of work required will change over course of large project.
- Planning out a whole project doesn't hold up
- Changes in shape, not just in time
- Keep planning realistic
- Know tasks for next couple of weeks
- Rough idea of requirements to work on next few months
- Vague sense of what needs to be done over year

Agile principles

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

2.1.3 Agile method Principle/ Principles of agile method.

1. **Customer involvement**: Customers should be closely involved in the development process. Their role is provide & prioritize new system requirements & to evaluate the iteration of the system.
2. **Incremental delivery**: Software is developed in increments with the customer specifying requirements to be included in each increment.
3. **People not process**: Team members should be left to develop their own ways of working without prescribed processor.
4. **Embrace change:** Expect the system requirements to change so design the system to accommodate these changes.
5. **Maintain simplicity**: Focus simplicity in both software & development process avoids complexity.

Difficulties in agile methods

1. customer involvement in the development process is an attractive one, its success depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Frequently, the customer representatives are subject to other pressures and **cannot take full part** in the software development.
2. Individual team members may **not have suitable personalities** for the intense involvement that is typical of agile methods. They may **therefore not interact well with other team members.**
3. Prioritising changes can be extremely difficult, especially in systems where there are **many stakeholders**. Typically, each stakeholder gives different priorities to different changes.

4. Maintaining simplicity requires extra work. Under pressure from delivery schedules, the team members may not have time to carry out desirable system simplifications.

Agile methods and software maintenance

- Most organizations spend more on maintaining existing software than they do on new software development. So, if agile methods are to be successful, they have to support maintenance as well as original development.
- Two key issues:
  - Are systems that are **developed using an agile approach maintainable**, given the emphasis in the development process of minimizing formal documentation?
  - Can agile methods be used effectively for evolving a system in response to customer change requests?
- Problems may arise if original development team cannot be maintained.

2.2 Scrum

- **Scrum is a development framework** in which **cross-functional teams develop products or projects in an iterative, incremental manner**.
- It structures development in cycles of work called **Sprints**.
- Scrum is an agile process model which is used for **developing complex software systems.**
- It is a **lightweight process framework** that can be used to manage and control the software development using iterative and incremental approach.
- Here the term lightweight means the overhead of the process is kept as small as possible in order to maximize productive time.

2.1.1 Scrum Principles

1. There are **small working teams on the software development projects**. Due to this there is a maximum communication and minimum overhead.
2. The task of people, must be **partitioned into small** and **clean packets or partitions**.
3. The **process must accommodate the technical** or **business changes** if they occur.
4. The process should **produce software increments**. These increments must be inspected, tested, documented and built on.
5. During the product building the **constant testing** and **documentation must be conducted**.
6. The scrum process must produce the **working model of the product** whenever demanded or required.

2.1.2 Scrum roles

In Scrum, **there are three roles:**

1. **Product Owner.**
2. **Team.**
3. **ScrumMaster.**

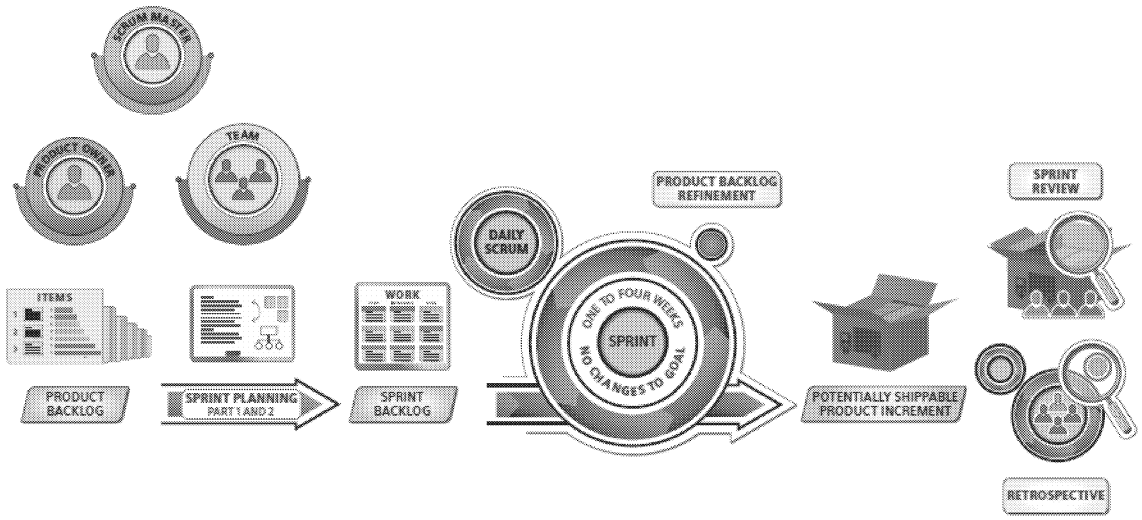


Figure 2.1: Scrum Overview

Product owner

- Scrum **product owner is a central role**, having responsibilities of a classical product manager.
- In some cases, the Product Owner and the customer are the same person; this is common for internal applications.
- Product Owner role is similar to the **Product Manager or Product Marketing Manager** position in many product organizations.
- It is important to note that in Scrum there is one and only one person who serves as – and has the final authority of – Product Owner, and he or she is responsible for the value of the work; though that person doesn't have to work alone.

Responsibilities are:

- Manages the scrum product backlog
- Creates and maintains the release plan and decides about deliveries, functionalities and costs of a project.
- Performs stakeholders' management.
- Works closely with the scrum team.
- For maximizing **return on investment (ROI) by identifying product features**, translating these into a prioritized list, deciding which should be at the top of the list for the next Sprint, and continually re-prioritizing and refining the list.

Scrum master

- Helps the **product groups learn** and **apply** Scrum to achieve business value.
- The **Scrum master is not the manager of the Team members**, nor is they a project **manager, team lead, or team representative**.
- Instead, the **Scrum master serves the Team**; he or she helps to remove impediments, protects the Team from outside interference, and helps the Team to adopt modern development practices.
- The **Scrum master is a coach and teacher.**

- Scrum master leads the meeting and analyses the response of each team member. Problems are discussed in meetings.

Responsibilities are:

- Acts as a change agent and adopt processes to **maximize productivity of a team**
- Guide the scrum team.
- Solve the problems faced by scrum team.
- Ensure efficient communication between scrum team and the scrum product owner.
- Facilitate the various scrum events.
- Teaching
- Implementing
- Ensuring

Scrum Team

- Builds the product that the Product Owner indicates: the application or website, for example.
- The Team in Scrum is **"cross-functional"** – it includes all the expertise necessary to deliver the potentially shippable product each Sprint – and it is **"selforganizing" (self-managing),** with a very high degree of autonomy and accountability.
- Each member of the Team is just a **team member**. The Team in Scrum is seven plus or minus two people, and for a software product the Team might include people with skills in analysis, development, testing, interface design, database design, architecture, documentation, and so on.
- The Team develops the product and provides ideas to the Product Owner about how to make the product great.
- Team avoids multi-tasking across multiple products or projects, to avoid the costly drain of divided attentions and context-switching.
- Teams are also known as **feature teams.**
- Many people are working in a team constituting a scrum team.

Responsibilities are:

- They have to breakdown the requirements, create task, estimate and distribute these tasks among themselves.
- They have to perform the short daily sprint meeting
- They have to ensure that at the end of the sprint proper functionality is delivered.
- They have to update the status and remaining efforts for their tasks.
- In addition to these three roles, there are other stakeholders who contribute to the success of the
- product, including managers, customers and end-users.

2.1.3 Scrum cycle:

- Scrum software development progress via a series of iterations called sprints, which last from one to four hour weeks.
- The scrum model suggests each sprint begins with a brief planning meeting and concludes a review, includes.

Product backlog

- A list of the features to be implemented in the project (subdivided to next release), ordered by priority.
- Can adjust over time as needed, based on feedback.
- A product manager is responsible for maintaining.
- Ever changing
- Prioritized list
- Owned by product owner
- Spread sheet example
- Product Backlog items are articulated in any way that is clear and sustainable. Contrary to popular misunderstanding, the Product Backlog does not contain "user stories"; it simply contains items.

A good Product Backlog is DEEP

Detailed appropriately: The top priority items are more fine-grained and detailed than the lower priority items, since the former will be worked on sooner than the latter.

Estimated: The items for the current release need to have estimates, and furthermore, should be considered for re-estimation each Sprint as every ones learns and new information arises.

Emergent: In response to learning and variability, the Product Backlog is regularly refined. Each Sprint, items may be added, removed, modified, split, and changed in priority.

Prioritized: The items at the top of the Product Backlog are prioritized or ordered in a 1-N order. In general, the highest-priority items should deliver the most bang for your buck: lots of bang (business value) for low buck (cost).

Sprint planning meeting

Summary: A meeting to prepare for the Sprint, typically divided into two parts (part one is "what" and part two is "how").

Participants: Part One: Product Owner, Team, ScrumMaster. Part Two: Team, ScrumMaster, Product Owner (optional but should be reachable for questions)

Duration: Each part is timeboxed to one hour per week of Sprint.

- In **Sprint Planning Part One**, the Product Owner and Team review the high-priority items in the Product Backlog that the Product Owner is interested in implementing this Sprint.
- Part One focuses on understanding what the Product Owner wants and why they are needed.
- **Sprint Planning Part Two** focuses on how to implement the items that the Team decides to take on.
- The Team forecasts the amount of items they can complete by the end of the Sprint, starting at the top of the **Product Backlog** (in others words, starting with the items that are the highest priority for the Product Owner) and working down the list in order.
- This is a **key practice in Scrum:** The Team decides how much work it will complete, rather than having it assigned to them **by the Product Owner.**
- Stake-holders to refine and re-prioritize the **Product Backlog and Release Backlog** and to choose the goals for the next iteration, usually droved by the highest business value and risk

- Scrum team and Product Owner meet to consider how to achieve the requests, and to create a sprint backlog of tasks to meet the goals.

Daily Scrum

Summary: Update and coordination between the Team members.
Participants: Team is required; Product Owner is optional; Scrum master is usually present but ensures
Team holds one.
Duration: Maximum length of 15 minutes.

- Once the Sprint has started, the Team engages in another of the key Scrum practices: The **Daily Scrum**. This is a short (15 minutes or less) meeting that happens every workday at an appointed time.
- Everyone on the Team attends. There is little or no in-depth discussion during the Daily Scrum, the theme is
- reporting answers to the three questions; if discussion is required it takes place immediately after the
- Daily Scrum in one or more parallel follow-up meetings, although in Scrum no one is required to
- attend these.

Product Backlog Refinement

Summary: Split big items, analyze items, re-estimate, and re-prioritize, for future Sprints.
Participants: Team; Product Owner will attend the entire activity if they are the expert who can help
with the detailed refinement, otherwise they may attend only a subset to set direction or re-prioritize;
others who understand the requirements and can help the Team; ScrumMaster will attend during initial
sessions to coach the group to be effective, otherwise may not attend.
Duration: Usually, no more than 10% of the capacity of the Team for the Sprint, though it may be
longer for "analysis heavy" items. For example, in a two-week Sprint, perhaps one day is spent on refinement.

- This refinement activity is not for items selected for the current Sprint; it is for items for the future, most likely in the next one or two Sprints.

Sprint Review

Summary: Inspection and adaption related to the product increment of functionality.
Participants: Team, Product Owner, Scrum master. Other stakeholders as appropriate, invited by the Product Owner.
Duration: Time boxed to one hour per week of Sprint.

- After the Sprint ends, there is the **Sprint Review**, where people review the Sprint.
- Present at this meeting are the Product Owner, Team members, and Scrum master, plus customers, users, stakeholders, experts, executives, and anyone else who is interested.

Sprint Retrospective

Summary: Inspection and adaption related to the process and environment.

Participants: Team, ScrumMaster, Product Owner (optional). Other stakeholders may be invited by the
Team, but are not otherwise allowed to attend.

Duration: Timeboxed to 45 minutes per week of Sprint.

- The Sprint Review involves inspect and adapt regarding the product.
- The **Sprint Retrospective**, which follows the Review, involves inspect and adapt regarding the process and environment.
- Sprint retrospective is usually the last thing done in a sprint. Many teams will do it immediately after the sprint review.
- Sprint retrospective meeting is conducted upto 1-3 hours.
- The sprint retrospective is an integral part of scrum cycle and without this meeting the team will never be able to improve their overall output.
- In this meeting all team members reflect on the past sprint and check three things,
    - What went well during the sprint?
    - What didn't?
    - What improvements could be made in the next sprint?
- Who can attend meeting?
    - Team, product owner, scrum master

Sprint backlog

- Team selects the first item on the Product Backlog – the Product Owner's highest priority item – and work their way down until they are 'full'.
- For each item they create a list of work which consists of either decomposed Product Backlog items into tasks or, when the Product Backlog item are so small they would only take a couple hours to implement, simply the Product Backlog item.
- This list of work to be done during the Sprint is called the **Sprint Backlog.**
- Sprint backlog defines the work, or tasks, that a team defines for turning the Product backlog it selects for that Spring into an increment of potentially shippable product functionality.
- Task should be 4-16 hours each
- Highly visible, real-time picture of the work
- Owned by the team
- Maintained as spread sheet daily by a tracker or responsible individuals.

Burn-down chart

- Visualize the correlation between the amount of work remaining and the progress in reducing the work
- X: date
- Y: hours of work remaining
- Updated according the Sprint backlog

2.3 Extreme programming

- Extreme programming **is one of the agile methods.**
- Extreme Programming (XP) takes an 'extreme' approach to iterative development.
    - New versions may be built several times per day.
    - Increments are delivered to customers every 2 weeks.
    - All tests must be run for every build and the build is only accepted if tests run successfully.
- In extreme programming all requirements expressed as **scenarios (user stories)** which are implemented as services of tasks.

XP and agile principles

- Incremental development is supported through small, frequent system releases.
- Customer involvement means full-time customer engagement with the team.
- People not process through pair programming, collective ownership and a process that avoids long working hours.
- Change supported through regular system releases.
- Maintaining simplicity through constant refactoring of code.
- Programmers develop tests for each task before writing code (test first development)
- All tests must be successfully executed when new code is integrated into the system. Figure 2.2 illustrates the XP process to produce an increment of the system that is being developed.

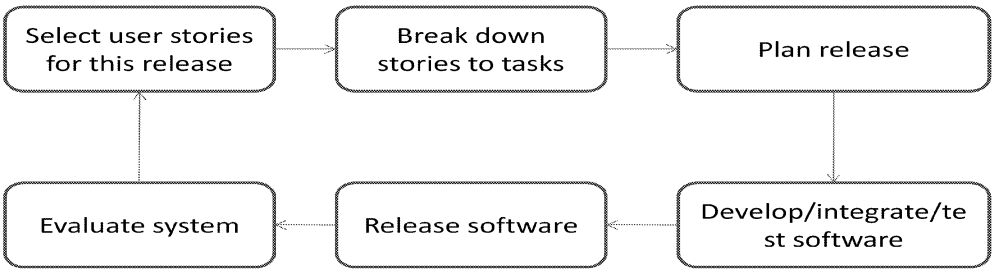


Figure 2.2: The Extreme Programming release cycle

Extreme Programming practices:

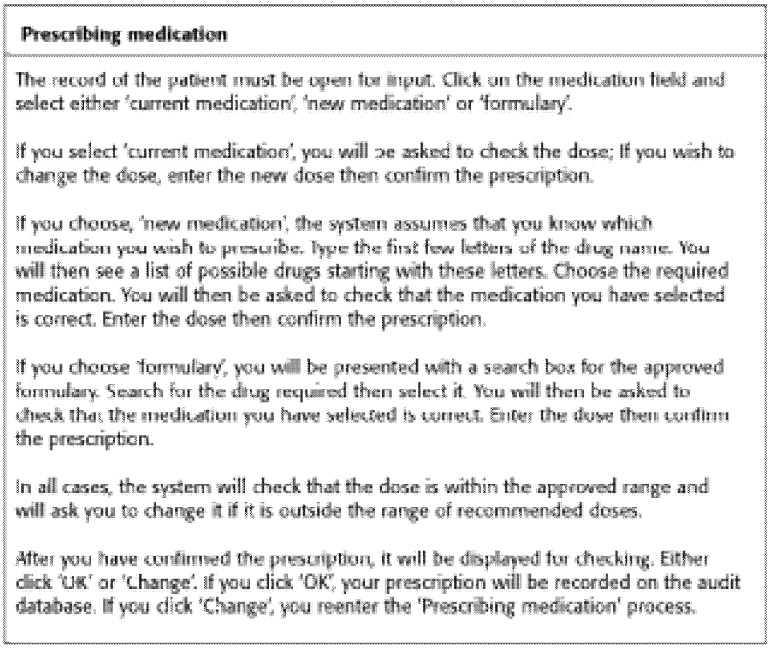
Principle or practice	Description
Incremental planning	Requirements are recorded on Story Cards and the Stories to be included in a release are determined by the time available and their relative priority. The developers break these Stories into development 'Tasks'. See Figures 3.5 and 3.6.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

Figure 2.3: Extreme programming practices (principles of XP)

- In an XP process, customers are intimately involved in specifying and prioritizing system requirements.
- The system customer is part of the development team and discusses scenarios with other team members. Together, they develop a 'story card' that encapsulates the customer needs.
- The development team then aims to implement that scenario in a future release of the software.
- Once the story cards have been developed, the development team breaks these down into tasks and estimates the effort and resources required for implementation. As shown in fig 1.7
- The customer then priorities the stories for implementation, choosing those stories that can be used immediately to deliver useful business support.
- Requirements are recorded on Story Cards and the Stories to be included in a release are determined by the time available and their relative priority.
- The developers break these Stories into development 'Tasks'.

Requirements scenarios

- In XP, a customer or user is part of the XP team and is responsible for making decisions on requirements.
- User requirements are expressed as scenarios or user stories.
- These are written on cards and the development tea break them down into implementation tasks. These tasks are the basis of schedule and cost estimates.
- The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.



Prescribing medication

The record of the patient must be open for input. Click on the medication field and select either 'current medication', 'new medication' or 'formulary'.

If you select 'current medication', you will be asked to check the dose; If you wish to change the dose, enter the new dose then confirm the prescription.

If you choose 'new medication', the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

If you choose 'formulary', you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

In all cases, the system will check that the dose is within the approved range and will ask you to change it if it is outside the range of recommended doses.

After you have confirmed the prescription, it will be displayed for checking. Either click 'OK' or 'Change'. If you click 'OK', your prescription will be recorded on the audit database. If you click 'Change', you reenter the 'Prescribing medication' process.

Figure 2.4: A 'prescribing medication' story

- The story cards are the main inputs to the XP planning process or the 'planning game'.
- Once the story cards have been developed, the development team breaks these down into tasks (Figure 2.5) and estimates the effort and resources required for implementing each task.
- This usually involves discussions with the customer to refine the requirements.
- The customer then prioritizes the stories for implementation, choosing those stories that can be used immediately to deliver useful business support.
- The intention is to identify useful functionality that can be implemented in about two weeks, when the next release of the system is made available to the customer.

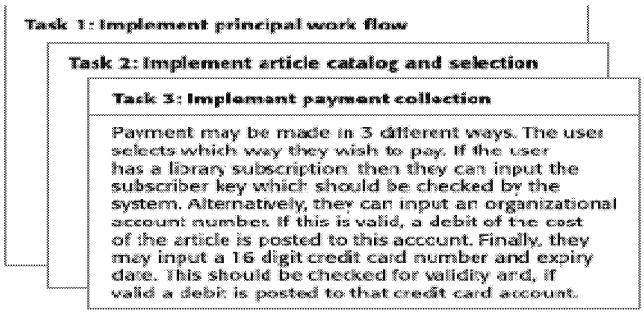


Figure 2.5: Task cards for document downloading

2.3.1 Testing in XP

- XP places more emphasis than other agile methods on the testing process.
- System testing is central to XP where an approach has been developed that reduces the likelihood that producing new system increments will introduce errors into the existing software.

The key features of testing in XP are:

1. Test-first development
2. Incremental test development from scenarios
3. User involvement in the test development and validation
4. The use of automated test harnesses

- **Test-first development** is one of the most important innovations in XP. Writing tests **first implicitly** defines both an interface and a specification of behavior for the functionality being developed.
  - Writing tests before code clarifies the requirements to be implemented.
  - Tests are written as programs rather than data so that they can be executed automatically. The test includes a check that it has executed correctly.
  - Usually relies on a testing framework such as Junit.
  - All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has not introduced errors.

Customer involvement

- The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system.
- The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs.
- However, people adopting the customer role have limited time available and so cannot work full-time with the development team. They may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.

Test automation

- Test automation means that tests are written as executable components before the task is implemented
  - These testing components should be stand-alone, should simulate the submission of input to be tested and should check that the result meets the output

specification. An automated test framework (e.g. Junit) is a system that makes it easy to write executable tests and submit a set of tests for execution.

- As testing is automated, there is always a set of tests that can be quickly and easily executed

Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

- **Problems of requirements** and interface misunderstandings are reduced. This approach can be adopted in any process where there is a clear relationship between a system requirement and the code implementing that requirement.
- User requirements in XP are expressed as scenarios or stories and the user prioritises these for development. The development team assesses each scenario and breaks it down into tasks.
- Each task generates one or more unit tests that check the implementation described in that task. For example, Figure 1.8 is a shortened description of a test case that has been developed to check that a valid credit card number has been implemented.
- With test-first development, there is always a set of tests that can be quickly and easily executed. This means that whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

Test 4: Test credit card validity

Input:
A string representing the credit card number and two integers representing the month and year when the card expires
Tests:
Check that all bytes in the string are digits
Check that the month lies between 1 and 12 and the year is greater than or equal to the current year.
Using the first 4 digits of the credit card number, check that the card issuer is valid by looking up the card issuer table. Check credit card validity by submitting the card number and expire date information to the card issuer
Output:
OK or error message indicating that the card is invalid

Figure 2.6: Test case description for credit card validity

- However, test-first development does not always work as intended. Programmers prefer programming to testing and sometimes write incomplete tests that do not check for exceptional situations.
- Furthermore, some tests can be very difficult to write. For example, in a complex user interface, it is often difficult to write unit tests.

XP testing difficulties

- Programmers prefer programming to testing and sometimes they take short cuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.

- Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the 'display logic' and workflow between screens.
- It difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage.

2.4 Pair programming

Another innovative practice that has been introduced is that programmers work in pairs to develop the software. They actually sit together at the same work station to develop the software The use of pair programming has a number of advantages:

1. It supports the idea of **common ownership and responsibility** for the system.
2. This reflects **egoless programming** where the software is owned by the team as a whole and individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.
3. It acts as an **informal review process** because each line of code is looked at by at least two people.
4. Code inspections and reviews are very successful in discovering a high percentage of software errors
5. Pair programming is a less formal process that probably doesn't find so many errors, it s a much **cheaper inspection process** than formal program inspections.
6. It helps **support refactoring** , which is a process of software improvement. A principle of XP is that the software should be constantly refactored.

2.5 Plan and agile development

- **Agile approaches** to software development consider design and implementation to be the central activities in the software process. They incorporate other activities, such as requirements elicitation and testing, into design and implementation.

- By contrast, **a plan-driven approach** to software engineering identifies separate stages in the software process with outputs associated with each stage. The outputs from one stage are used as a basis for planning the following process activity. Figure 3.2 shows the distinctions between plan-driven and agile approaches to system specification.

- In a plan-driven approach, iteration occurs within activities with formal documents used to communicate between stages of the process. For example, the requirements will evolve and, ultimately, a requirements specification will be produced.

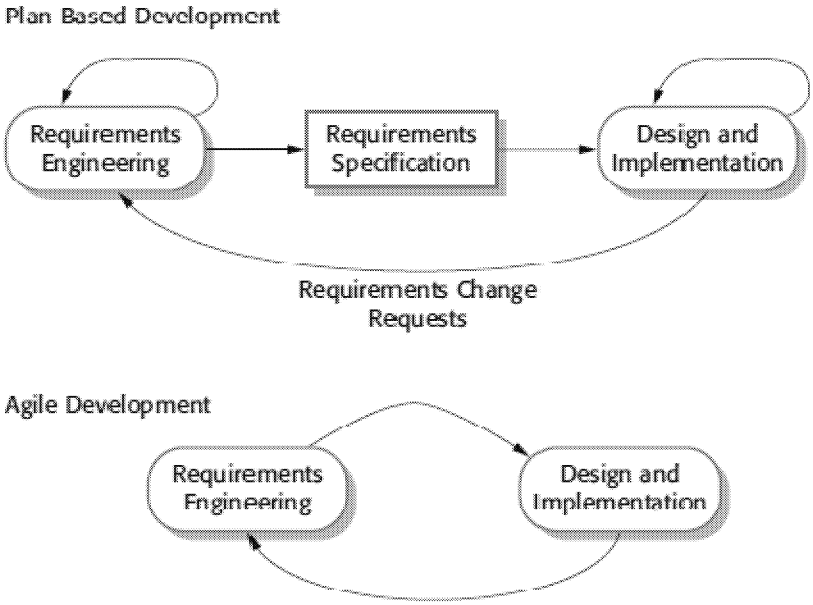


Figure 2.7: Plan and agile development

Technical, human, organizational issues

- **Most projects include elements of plan-driven and agile processes. Deciding on the balance depends on:**
  - Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
  - Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.
  - How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.
  - What type of system is being developed?
    - Plan-driven approaches may be required for systems that require a lot of analysis before implementation (e.g. real-time system with complex timing requirements).
  - What is the expected system lifetime?
    - Long-lifetime systems may require more design documentation to communicate the original intentions of the system developers to the support team.
  - What technologies are available to support system development?
    - Agile methods rely on good tools to keep track of an evolving design
  - How is the development team organized?
    - If the development team is distributed or if part of the development is being outsourced, then you may need to develop design documents to communicate across the development teams.
  - Are there cultural or organizational issues that may affect the system development?

- ❖ Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.
- How good are the designers and programmers in the development team?
  - ❖ It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code

2.6 Agile Project Management

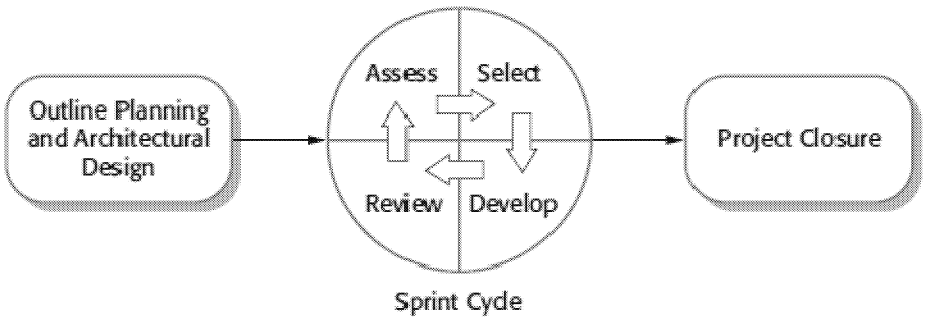


Figure 2.8: Agile project management

- The principal responsibility **of software project managers is to manage the project** so that the software is delivered on time and within the planned budget for the project.
- Like every other professional **software development process, agile development has to be managed so that the best use is made of the time and resources available to the team**. This requires a different approach to project management, which is adapted to **incremental development** and the particular strengths of agile methods
- **The Scrum approach is a general agile method** but its focus is on managing iterative development rather than specific technical approaches to agile software engineering.

Three phases in Scrum.

- **The first** is an outline planning phase where you establish the general objectives for the project and design the software architecture.
- **This is followed by a series of sprint cycles**, where each cycle develops an increment of the system.

- **Finally, the project closure phase wraps up the project**, completes required documentation such as system help frames and user manuals, and assesses the lessons learned from the project

Key characteristics

1. **Sprints are fixed length,** normally 2–4 weeks. They correspond to the development of a release of the system in XP.
2. The starting point for planning is the product backlog, which is the list of work to be done on the project. During the assessment phase of the sprint, this is reviewed, and priorities and risks are assigned. The **customer is closely involved in this process and can introduce new requirements or tasks at the beginning of each sp**rint.
3. The **selection phase** involves all of the project team who work with the customer to select the features and functionality to be developed during the sprint.
4. Once these are agreed, the team organizes themselves to develop the software. **Short daily meetings** involving all team members are held to review progress and if necessary, reprioritize work. During this stage the team is isolated from the customer and the organization, with all communications channeled through the so-called 'Scrum master'.
5. At the end of the sprint, the **work done is reviewed** and presented to stakeholders. The next sprint cycle then begins.

Advantages

1. The **product is broken down** into a set of manageable and understandable chunks.
2. **Unstable requirements** do not hold up progress.
3. The whole team has visibility of everything and consequently **team communication** is improved.
4. Customers see **on-time delivery of increments and gain feedback** on how the product works.
5. **Trust between customers and developers** is established and a positive culture is created in which everyone expects the project to succeed.

2.7 Scaling agile methods

- Agile methods have proved to be successful for small and medium sized projects that can be developed by a small co-located team.
- It is sometimes argued that the success of these methods comes because of improved communications which is possible when everyone is working together.
- Scaling up agile methods involves changing these to cope with larger, longer projects where there are multiple development teams, perhaps working in different locations.

Large systems development

- Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones.
- Large systems are 'brownfield systems, that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development.

- ❖ Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development.
- ❖ Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed.
- ❖ Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
- ❖ Large systems usually have a diverse set of stakeholders. It is practically impossible to involve all of these different stakeholders in the development process.

There are **two perspectives** on the scaling of agile methods:

1. **A 'scaling up'** perspective, which is concerned with using these methods for developing large software systems that cannot be developed by a small team.
2. **A 'scaling out'** perspective, which is concerned with how agile methods can be introduced across a large organization with many years of software development experience.

critical adaptations that have to be introduced are as follows:

1. **For large systems development, it is not possible to focus only on the code of the system**. The software architecture has to be designed and there has to be documentation produced to describe critical aspects of the system, such as database schemas, the work breakdown across teams, etc.

2. **Cross-team communication mechanisms have to be designed and used.** This should involve regular phone and video conferences between team members and frequent, short electronic meetings where teams update each other on progress

3. **Continuous integration**, where the whole system is built every time any developer checks in a change.

It is difficult to introduce agile methods into large companies for a number of reasons:

1. **Project managers who do not have experience** of agile methods may be reluctant to accept the risk of a new approach, as they do not know how this will affect their particular projects.

2. Large organizations often have quality procedures and standards that all projects are expected to follow and, **because of their bureaucratic nature, these are likely to be incompatible with agile methods.**

3. Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities, and **people with lower skill levels may not be effective team members in agile processes.**

4. There may be **cultural resistance** to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

QUESTION BANK
MODULE 5

Sl.No	Questions	Marks
1.	Explain 2 ways in coping with change.	8M
2.	What is agile method? Discuss agile method principles.	8M
3.	What is pair programming? write its advantages	6M
4.	Explain extreme programming practices.	10M
5.	Describe extreme programming release cycle with a neat diagram.	6M
6.	Discuss key features in testing in XP.	5M
7.	Discuss the characteristics of Scrum Process.	