Software Engineering

Unit - 1

Definition of Software Engineering

Software engineering is a systematic and disciplined approach to designing, developing, testing, deploying, and maintaining software. It helps developers build reliable, efficient, and maintainable software in an organized way.

Software engineering is a systematic and disciplined approach to designing, developing, testing, deploying, and maintaining software. It helps developers build reliable, efficient, and maintainable software in an organized way. It involves managing the entire **Software Development Life Cycle (SDLC).**

Differentiation between Computer Science, Software Engineering and Systems Engineering

Computer Science studies the theoretical foundations of computation and information. Software Engineering applies engineering principles to design, build, and maintain software products. Systems Engineering focuses on the overall lifecycle and integration of complex, multidisciplinary systems (combining hardware, software, and physical components)

Aspect	Computer Science (CS)	Software Engineering (SE)	Systems Engineering (SysE)
Focus	Theory, algorithms, data structures, and how computers work	Practical application of developing high-quality software products.	The integration and management of multiple interacting subsystems (hardware, software, networking, and physical components).
Scope	Broad and foundational, covering everything from artificial intelligence and machine learning to networks and cybersecurity.	Operates entirely within the software realm, from coding and testing to deployment and scaling.	Large-scale and multidisciplinary; looks at the "big picture" to manage requirements and external interfaces.
Objective	Discover new ways to compute and solve complex computational problems.	Deliver reliable, maintainable software efficiently using methodologies like Agile.	Ensure that all parts of a complex system work together properly to meet user and stakeholder requirements.

PRAJYOTI DEKA

Program V/s software product

The core difference is that a program is a single set of instructions written by a developer to perform a specific task, while a software product is a complete, fully tested, and documented solution built for external users or the commercial market.

Aspect	Program	Software Product
Definition	A simple set of coded instructions.	A complete software solution developed for users or customers.
Target Audience	Usually built for personal use or a single developer.	Built for mass distribution or commercial clients.
Development Style	Informal; rarely follows strict guidelines.	Strict adherence to the Software Development Life Cycle (SDLC).
Components	Source code and object code.	Code, user interface (UI), documentation, and manuals.
Scale & Size	Small, ranging from a few lines to a few megabytes.	Can range from small applications to very large systems.
Maintenance	Maintained solely by the original programmer.	Maintained by dedicated teams via updates and patches.

Exploratory style and modern style of software development

Exploratory development is an unstructured "build-and-fix" approach where developers jump straight into coding. In contrast, modern software development is an engineered discipline driven by formal planning, error prevention, strict testing, and version control, which drastically reduce long-term costs.

Feature	Exploratory Style	Modern Style
Philosophy	**Error correction:** Write code quickly and fix bugs as they emerge.	**Error prevention:** Use structured planning, requirements, and architecture.
Process	Ad-hoc and unstructured; essentially a cycle of "code → test → modify".	Iterative but highly structured (e.g., Agile methodologies).
Role of Coding	Coding is considered synonymous with development.	Coding is just one phase; emphasis is on design, testing, and system architecture.
Scalability	Not scalable. Costs and development problems increase as the project becomes larger.	Built for scalability, robustness, and long-term maintainability.
Tooling & Standards	Rare upfront design; minimal or no testing and documentation.	Standardized deployment via CI/CD, Git version control, and automated testing.

AKJYOTI EKA

Need of Software Engineering

Software engineering is a systematic and disciplined way of developing, testing, and maintaining software. It helps developers build software in an organized way and makes it easier to manage large and complex projects.



Why Software Engineering is Needed

1. **Handling Big Projects**
   Software engineering helps manage large software projects in a proper and organized way.
2. **Managing Cost**
   Proper planning helps reduce unnecessary work and control the cost of software development.
3. **Saving Time**
   A planned development process helps complete the software on time and avoids unnecessary delays.
4. **Developing Reliable Software**
   Software engineering uses proper development and testing methods to make software more reliable and reduce errors.

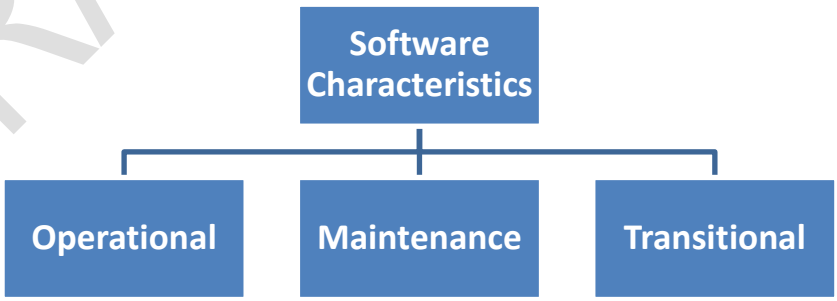
HIRAKJYOTI DEKA

5. **Improving Effectiveness**
   Following proper methods and standards helps developers create software that works as expected.
6. **Reducing Complexity**
   Large problems are divided into smaller parts. Each part can then be developed and solved separately.
7. **Improving Productivity**
   Proper planning, development, and testing help developers work more efficiently and produce better software.
8. **Easy Maintenance**
   Software engineering makes software easier to update, fix, and improve when requirements change.
9. **Better Teamwork**
   It provides a proper process for team members to work together and manage tasks, time, and resources.
10. **Better Quality**
    Testing and quality checks help find problems early and improve the overall quality of the software.

Characteristics of Good Software Product

Good software meets the needs of users and performs its required tasks properly. It should be easy to use, reliable, secure, and easy to maintain and improve. The characteristics of good software can be grouped into three categories: Operational, Maintenance, and Transitional.

The characteristics of good software can be grouped into three categories:



1. Operational Characteristics

These characteristics describe how the software works during normal use.

- **Functionality:** The software performs the tasks it was designed to do.
- **Reliability:** The software works correctly and consistently without frequent failures.
- **Usability:** The software is easy to understand and use.

HIRAKJYOTI DEKA

- **Efficiency:** The software uses system resources such as memory and processing power properly and gives good performance.
- **Security and Safety:** The software protects data from unauthorized access and helps prevent harmful system failures.

2. Maintenance Characteristics

These characteristics describe how easily the software can be changed, fixed, or improved.

- **Maintainability:** The software is easy to understand, fix, and update.
- **Scalability:** The software can handle more users, data, or workload when needed.
- **Flexibility:** The software can be changed or adapted when requirements change.
- **Modularity:** The software is divided into separate parts or modules, making it easier to develop, test, and modify.

3. Transitional Characteristics

These characteristics describe how easily the software can work in different environments and with other systems.

- **Portability:** The software can run on different operating systems, devices, or platforms.
- **Interoperability:** The software can communicate and exchange data with other systems.
- **Reusability:** Parts of the software can be used again in other projects.

Software Crisis

Software crisis refers to the problems faced in developing large and complex software. These problems include high development cost, delays, poor quality, difficulty in maintenance, and failure to meet user requirements.

Main reasons

1. Increasing software complexity
2. Increasing size of software projects
3. Changing user requirements
4. Poor planning
5. Difficulty in estimating cost and time
6. Lack of proper development methods

HIRAKJYOTI DEKA

Software Engineering

Unit – II

Software Development Life Cycle models

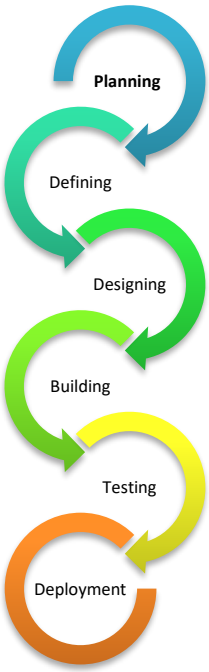
Definition of software development Life cycle (SDLC) models

An SDLC (Software Development Life Cycle) model is a structured framework that software engineering teams use to plan, design, build, test, and maintain software. It provides a step-by-step roadmap to ensure projects are delivered efficiently, on budget, and meet user requirements.

The Software Development Life Cycle (SDLC) is a structured process used to plan, design, develop, test, deploy, and maintain software. It ensures a systematic workflow and helps align software development with business goals and user requirements.

- Provides a clear and organized framework for managing development phases
- Helps in early detection of defects, reducing overall cost and time
- Ensures high-quality software delivery that meets user expectations

Stages of the Software Development Life Cycle

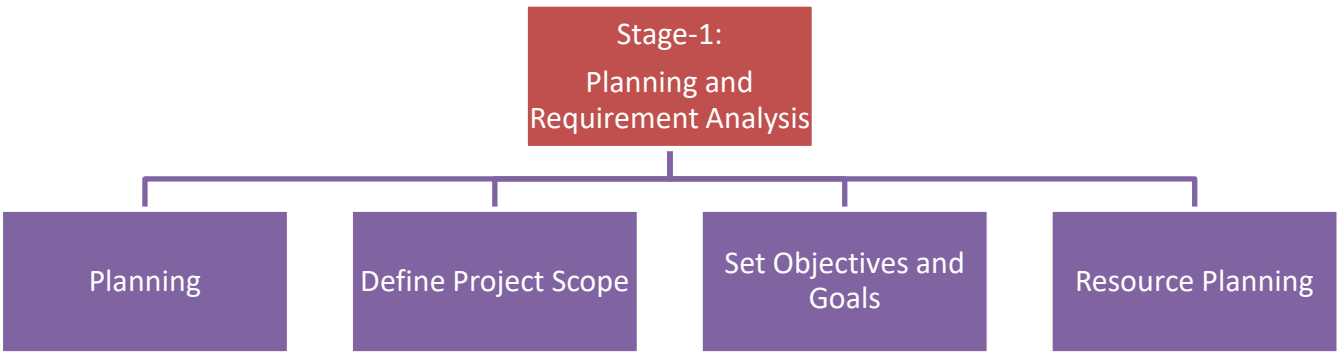


The Software Development Life Cycle (SDLC) typically consists of six or seven stages, depending on the development model used.

Stage 1: Planning & Feasibility Analysis

This stage determines whether the project is technically, financially, and operationally feasible.

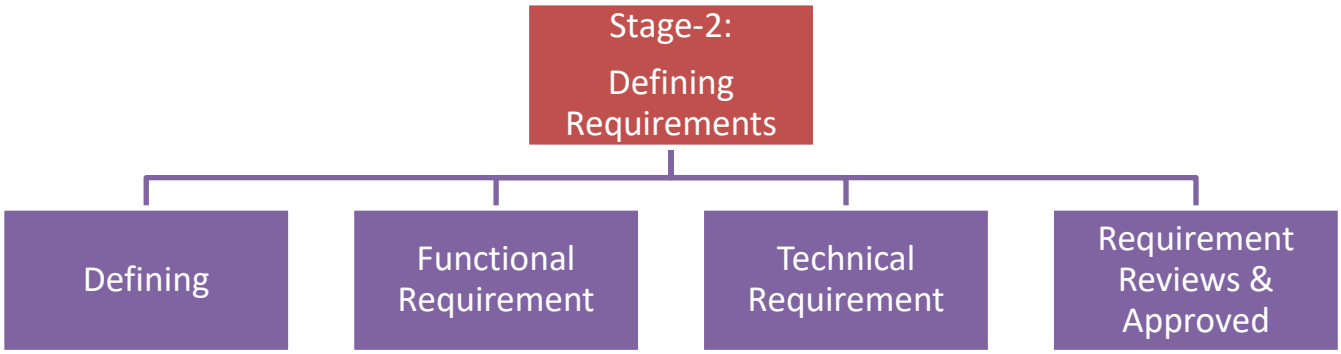
- **Activities:** Feasibility analysis, cost estimation, scheduling, resource planning
- **Output:** Project Plan, Feasibility Report
- **Key Roles:** Project Managers, Senior Engineers, Stakeholders



Stage 2: Requirement Specification (SRS)

In this stage, detailed functional and non-functional requirements are documented clearly and approved by stakeholders.

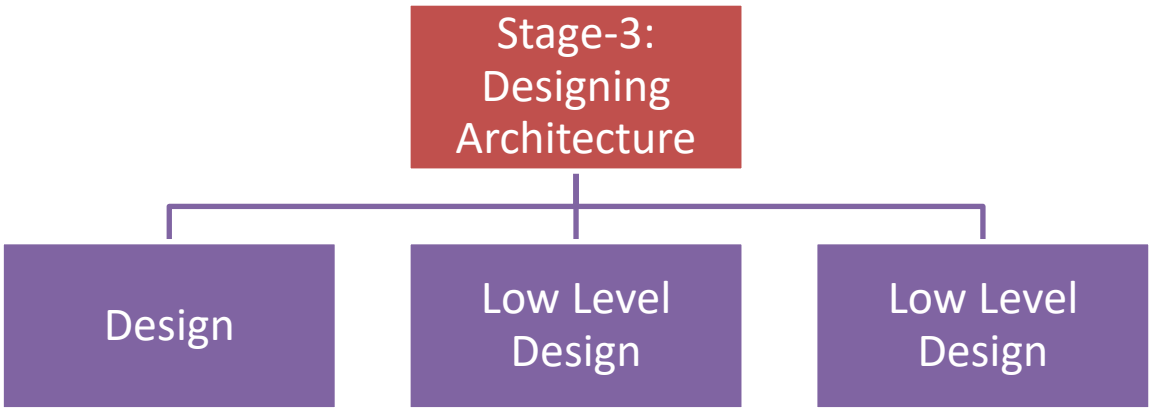
- **Activities:** Requirement gathering, validation, documentation
- **Output:** Software Requirement Specification (SRS)
- **Key Roles:** Business Analysts, Product Owners



Stage 3: System Design

In this stage, the approved requirements are transformed into a technical blueprint for implementation.

- **High-Level Design (HLD):** Defines system architecture, technology stack, database design, and major modules**.**
- **Low-Level Design (LLD):** Specifies component logic, APIs, data structures, and workflows.
- **Output:** Design Document Specification (DDS)

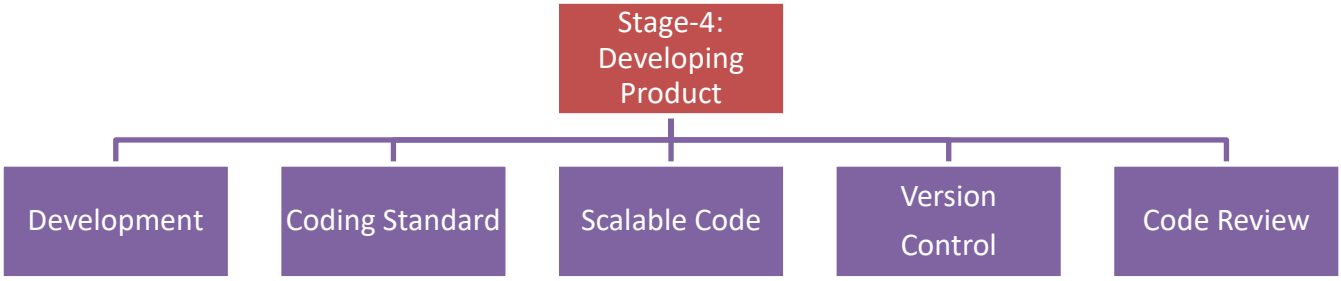


Stage 4: Development (Coding)

Developers build the software based on the approved design.

- **Activities:** Coding, code reviews, unit testing, version control management
- **Tools:** IDEs, version control systems, debuggers

- **Output:** Source code, executable application
- **Key Roles:** Frontend, Backend, Full Stack Developers



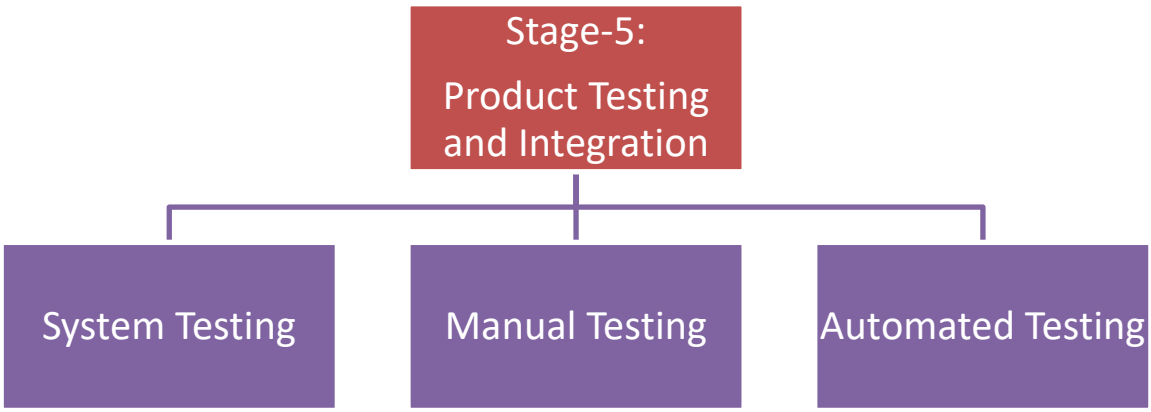
Stage 5: Testing

Testing ensures the software meets requirements and is free from defects before release.

Types of Testing include:

- **Unit Testing**: Verifies individual components
- **Integration Testing**: Ensures modules work together
- **System Testing:** Validates the complete system
- **User Acceptance Testing (UAT)**: Confirms business requirements are met

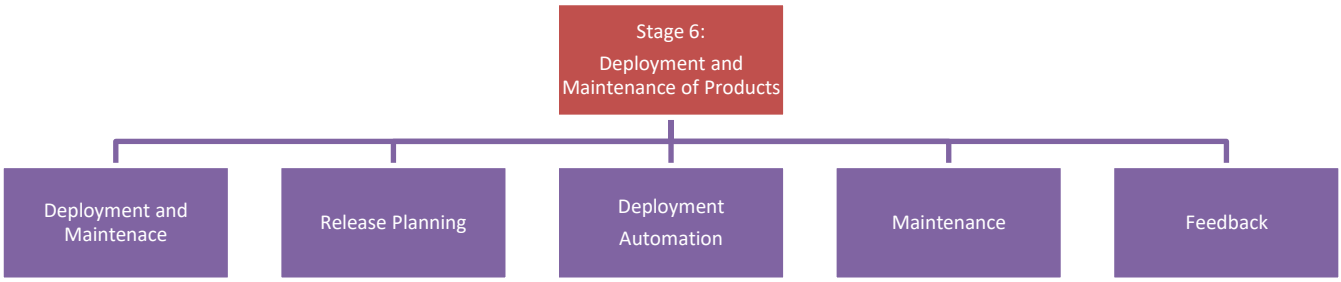
Output: Test cases, defect reports, and quality metrics.



Stage 6: Deployment

The tested software is released to users.

- **Activities:** Production setup, deployment, smoke testing
- **Modern Approach:** Continuous Integration and Continuous Deployment (CI/CD) pipelines for faster and reliable releases
- **Output:** Live application
- **Key Roles:** DevOps Engineers, Release Managers



Stage 7: Maintenance

Post-deployment support ensures long-term usability.

- **Activities:** Bug fixes, performance tuning, updates, feature enhancements
- **Output:** Patches, updates, new versions
- **Key Roles:** Support Engineers, Developers

Software Development Life Cycle Models

Software Development Models are structured frameworks that guide the planning, execution, and delivery of software projects. They define the sequence of development stages, such as requirements, design, coding, testing, and deployment.

Common Models

- Waterfall Model
- Agile Model
- V-Model
- Spiral Model
- Incremental Model
- RAD Model

Importance of SDLC

- Provides a clear and organized development framework.
- Improves planning, cost control, and project management.
- Ensures better quality through defined testing phases.
- Helps deliver software that meets user and business needs.

Embedding Security into the SDLC

Security is integrated throughout the Software Development Life Cycle using a DevSecOps approach. It is built into every stage, from design to deployment, ensuring continuous protection.

- Vulnerabilities are identified and fixed early in the development process.

- Automated security checks are integrated into build and CI/CD pipelines.
- Security becomes a shared responsibility across development, testing, and operations teams.

Embedding security into the SDLC reduces risks, improves software resilience, and enables the delivery of safer applications.

Common SDLC Mistakes Teams Make

Even experienced teams misuse SDLC by focusing on form over substance. Some frequent mistakes include:

- **Treating documentation as the goal:** Producing SRS and design documents without validating assumptions with real users.
- **Late or inadequate testing:** Testing is rushed or reduced due to time pressure, increasing post-release defects.
- **Ignoring non-functional requirements:** Performance, security, and scalability are considered too late.
- **Poor communication between teams:** Gaps between business, development, and QA lead to misaligned outcomes.
- **Skipping feedback loops:** Limited iteration and user feedback cause issues to surface only after deployment.
- **Overengineering early:** Designing overly complex solutions before validating core functionality.

Avoiding these mistakes requires continuous collaboration, early validation, and a mindset that values outcomes over processes.

Real Life Example of SDLC

Banking Application Development using SDLC:

- **Planning & Analysis:** Identify banking features such as account management, fund transfers, and security requirements**.**
- **Design**: Create UI designs, system architecture, databases, and technology stack.
- **Development**: Implement frontend interfaces, backend services, and APIs.
- **Testing:** Conduct functional, performance, and security testing.
- **Deployment & Maintenance**: Release the application and continuously monitor, fix defects, and add enhancements.

Reasons for Project Failure Despite Following SDLC

Following SDLC does not automatically guarantee project success. Many projects fail because teams treat SDLC as a checklist rather than a decision-making framework.

- **Poor requirement clarity:** Requirements are documented but not deeply understood, leading to incorrect assumptions and rework.

- **Weak stakeholder involvement:** Limited feedback from users or business teams results in solutions that don't solve real problems.
- **Rigid execution:** Teams follow the process mechanically and resist adapting to changing business or technical realities.
- **Underestimated complexity:** Risks related to scalability, integration, or performance are identified too late.
- **Lack of ownership:** Roles exist on paper, but accountability for outcomes is unclear.

SDLC provides structure, but success depends on how thoughtfully it is applied, not just whether it is followed.

Various life cycle modes: Classical Waterfall model, Iterative Waterfall model, Prototyping model, Evolutionary (Incremental) model, Spiral model, Agile Model

Software Development Life Cycle (SDLC) Models

Software development models are various processes or methods that are chosen for project development depending on the objectives and goals of the project. Many development life cycle models have been developed to achieve various essential objectives. Models specify the various steps of the process and the order in which they are executed.

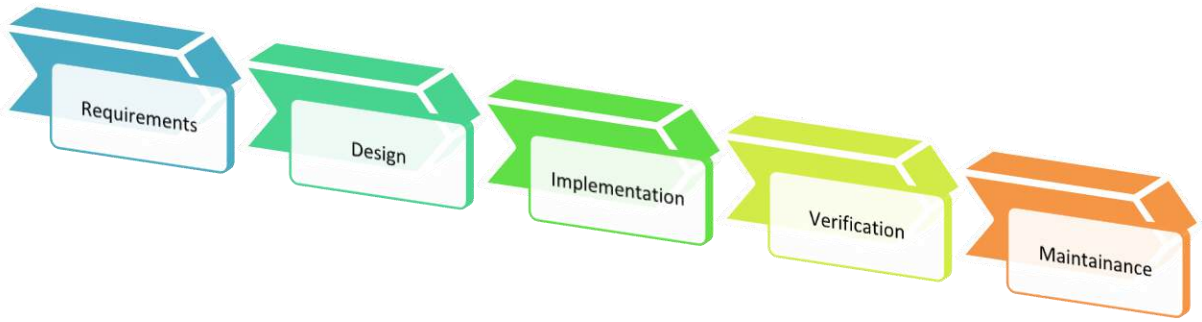
What is Software Modelling?

Software modelling is the process of creating abstract representations of a software system. These models serve as blueprints that guide developers, designers, and stakeholders through the system's structure, behaviour, and functionality.

By using diagrams and various modelling languages, software modelling helps in visualizing and understanding the complex aspects of the software, making it easier to plan, develop, and manage the system.

Classical Waterfall Model

Waterfall Model is a famous and good version of **SDLC (System Development Life Cycle)** for software engineering. The waterfall model is a linear and sequential model, which means that a development phase cannot begin until the previous phase is completed. We cannot overlap phases in waterfall model.

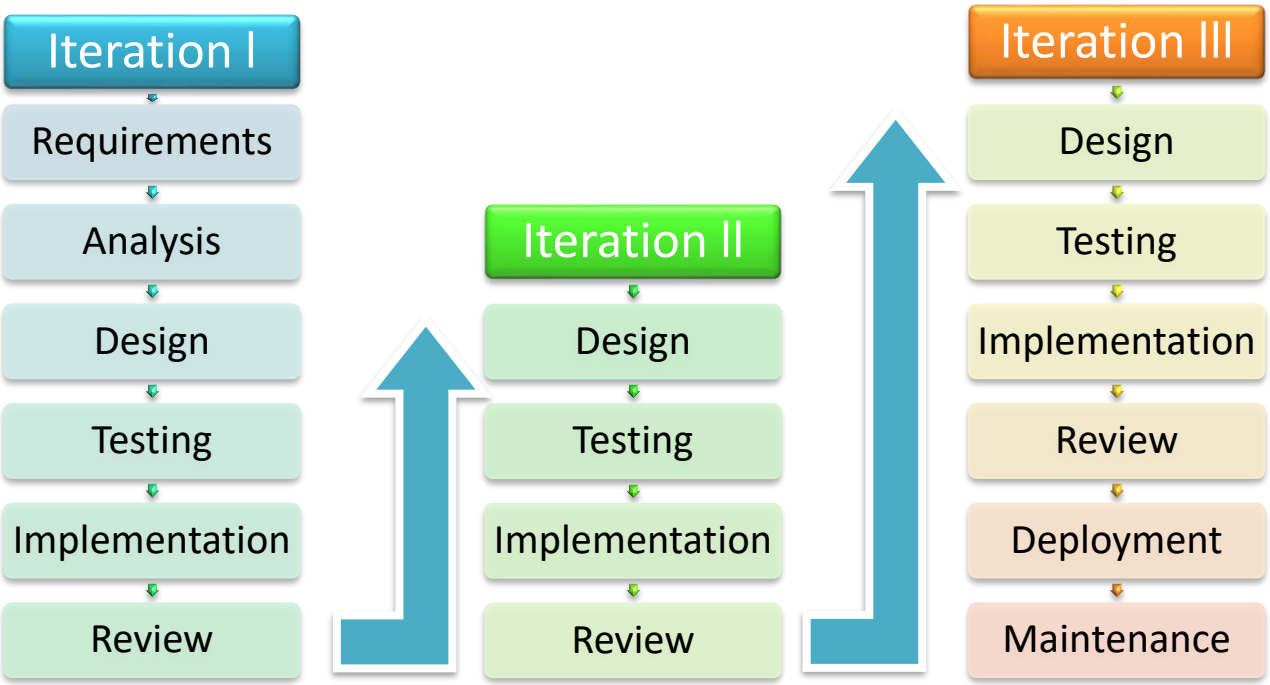


Similarly, waterfall model also works, once one phase of development is completed then we move to the next phase but cannot go back to the previous phase. In the waterfall model, the output of one phase serves as the input for the other phase.

Iterative Waterfall model

In **Iterative Model** we start developing the software with some requirements and when it is developed, it is reviewed. If there are requirements for changes in it, then we develop a new version of the software based on those requirements. This process repeats itself many times until we get our final product.

Through this diagram we can understand the Interactive model.

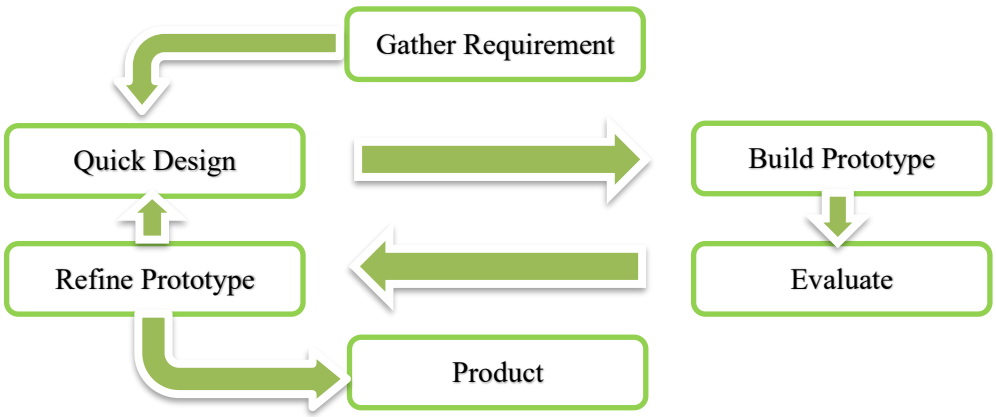


- So, in Iterative model a software is developed by following several iterations. Iteration means that we are repeating the Development Process again and again. For example, we develop the first version of the software following the SDLC process with some software requirements.
- After the first version is developed, if there is a need to change the software, then a new version is developed with the second iteration. Now again we will see if the new version is enough, if not then we will make changes in it with the third iteration. The iteration will be repeated until the complete software is ready.
- The basic concept of Iterative model is that the software should be developed through repeated cycles or what we also call iteration and only a small part of it should be developed at a time. This model was developed to overcome the drawbacks of the classical waterfall model.

Prototyping model

Prototype Model is an activity in which prototypes of software applications are created. First a prototype is created and then the final product is manufactured based on that prototype.

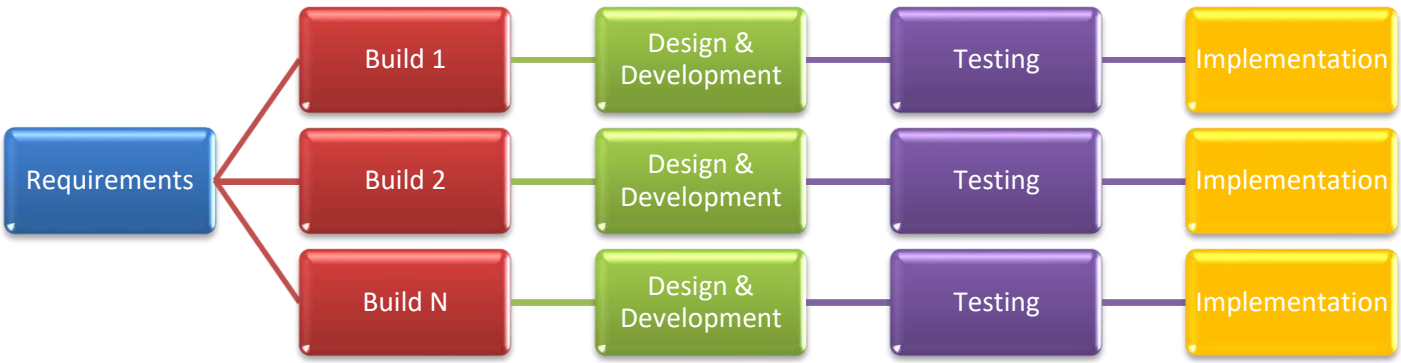
One problem in this model is that if the end users are not satisfied with the prototype model, then a new prototype model is created again, due to which this model consumes a lot of money and time



- The prototype model was developed to overcome the shortcomings of the waterfall model.
- This model is created when we do not know the requirements well.
- The specialty of this model is that this model can be used with other models as well as alone.

Evolutionary (Incremental) model

In **Incremental Model**, the **Software Development Process** is divided into several increments and the same phases are followed in each increment. In simple language, under this model a complex project is developed in many modules or builds.



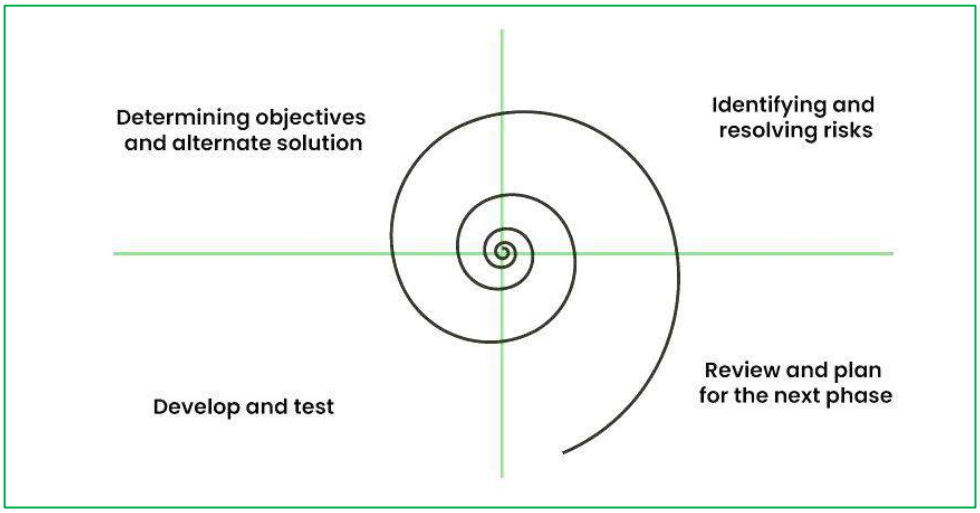
- For example, we collect the customer's requirements, now instead of making the entire software at once, we first take some requirements and based on them create a module or function of the software and deliver it to the customer. Then we take some more requirements and based on them add another module to that software.

- Similarly, modules are added to the software in each increment until the complete system is created. However, the requirements for making a complex project in multiple iterations/parts should be clear.
- If we understand the entire principle of Incremental methodology, then it starts by developing an initial implementation, then user feedback is taken on it, and it is developed through several versions until an accepted system is developed. Important functionalities of the software are developed in the initial iterations.

Each subsequent release of a software module adds functions to the previous release. This process continues until the final software is obtained.

Spiral model

Spiral Model is a software development process model. This model has characteristics of both iterative and waterfall models. This model is used in projects which are large and complex. This model was named spiral because if we look at its figure, it looks like a spiral, in which a long curved line starts from the center point and makes many loops around it. The number of loops in the spiral is not decided in advance but it depends on the size of the project and the changing requirements of the user. We also call each loop of the spiral a phase of the software development process.



In Spiral Model the entire process of software development is described in four phases which are repeated until the project is completed.

A software project goes through these loops again and again in iterations. After each iteration a more and more complete version of the software is developed. The most special thing about this model is that risks are identified in each phase and they are resolved through prototyping. This feature is also called Risk Handling.

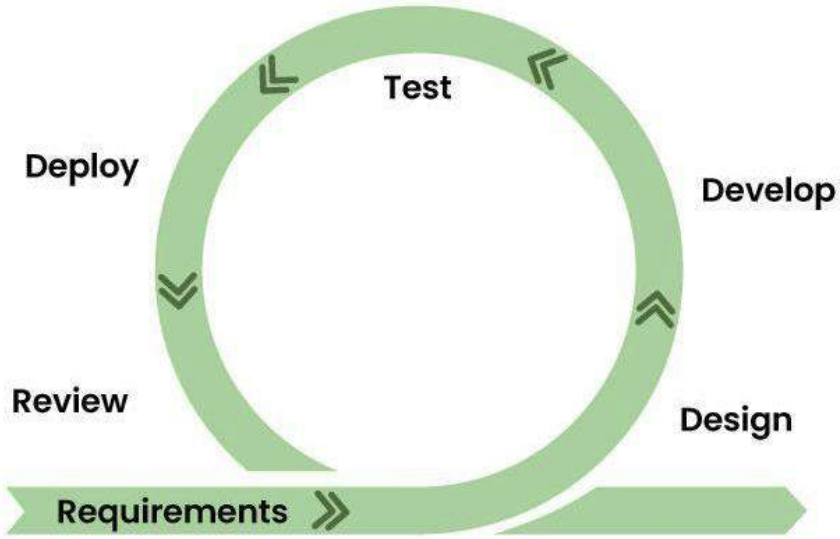
Since it also includes the approaches of other SDLC models, it is also called Meta Model. It was first developed by Barry Boehm in 1986.

Agile Model

Agile Development Model is a combination of iterative and incremental models, that is, it is made up of iterative and incremental models.

- In Agile model, focus is given to process adaptability and customer satisfaction.
- In earlier times, iterative waterfall model was used to create software. But in today's time developers have to face many problems. The biggest problem is that in the middle of software development, the customer asks to make changes in the software. It takes a lot of time and money to make these changes. So to overcome all these shortcomings, the agile model was proposed in the 1990s.

The Agile Model was created mainly to make changes in the middle of **software development** so that the software project can be completed quickly.



- In the agile model, the software product is divided into small incremental parts. In this, the smallest part is developed first and then the larger one.
- And each incremental part is developed over iteration.
- Each iteration is kept small so that it can be easily managed. And it can be completed in two-three weeks. Only one iteration is planned, developed and deployed at a time.

Agile V/s traditional SDLC Models

Aspect	AGILE	SDLC
Definition	Agile is a methodology following an iterative approach used for project management purposes.	SDLC is a process of design and development of a product or service.
Ease of Use	It is easier to use and implement.	Software Development Life Cycle is easy to understand and needs little systematic approach for implementation.

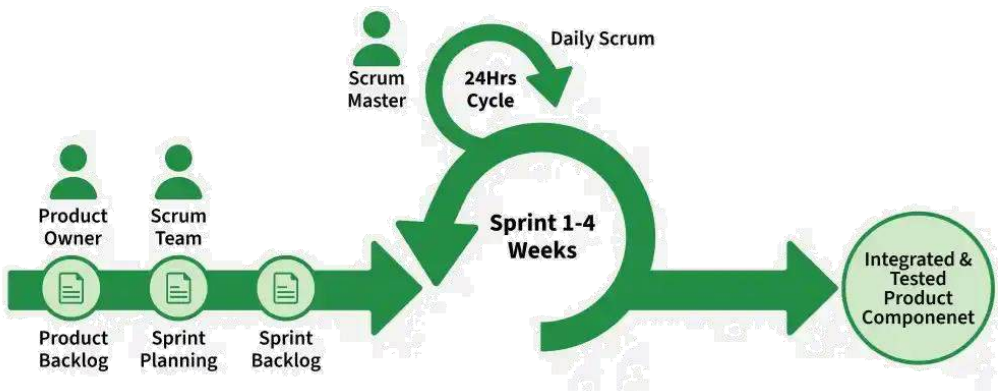
Aspect	AGILE	SDLC
Origin and History	It came out in 2001 and it was made by a joint collaboration of 17 software developers.	Software Development Life Cycle was found in 1960.
Structure	Agile constitutes of different phases.	SDLC constitutes of different stages
Approach Speed	Agile has a faster approach comparatively.	SDLC has a systematic approach.
Project Suitability	Agile is more suited for small scale projects.	SDLC can be used for any sized projects
Flexibility	Agile allows dynamic changes in requirements.	SDLC doesn't allow changes after initial stage.
Process Flow	It follows a continuous cycle.	It follows a sequential stage.
Stakeholder Involvement	It requires close customer involvement.	It requires close project manager involvement

SCRUM model

The Scrum model is an Agile project management framework that relies on the core roles of Product Owner, Scrum Master, and Developers, operating within structured time cycles called Sprints to build products incrementally. Defined officially in The Scrum Guide, it replaces heavy upfront planning with continuous adaptation, collaboration, and learning from experience.

Scrum is a widely used Agile framework for developing complex products, introduced by Ken Schwaber and Jeff Sutherland. It enables teams to self-organize, collaborate effectively, and deliver high-value products through an iterative and incremental approach, while adapting to changing requirements. Its key features are:

- Lightweight and simple to understand framework
- Emphasizes self-organization and teamwork
- Promotes collaboration and continuous improvement
- Supports iterative and incremental development



Key Components of the Scrum Lifecycle

Following points explain the key stages and components of the Scrum lifecycle as illustrated in the above diagram.

- **Sprint:** A time-boxed period of one month or less during which a product increment is developed is known as a Sprint, and a new sprint begins immediately after the previous one ends.
- **Release:** Usable product increment can be delivered based on requirements, which is referred to as a Release in Scrum.
- **Sprint Review:** Completed work is presented to stakeholders for feedback and future improvements during the Sprint Review.
- **Sprint Retrospective:** Continuous improvement in processes and team performance is achieved through the Sprint Retrospective, where the team reflects on the sprint.
- **Product Backlog:** All features, enhancements, and requirements are organized and prioritized in the Product Backlog.
- **Sprint Backlog:** Selected items from the Product Backlog, along with a plan to complete them during the sprint, make up the Sprint Backlog.

Advantage

- Fast-paced and cost-efficient development approach
- Breaks large projects into smaller, manageable parts (divide and conquer)
- Strong focus on customer satisfaction through continuous feedback
- Highly adaptive due to short and iterative sprints
- Improves product quality with regular reviews and feedback

Disadvantage

- Changes during a sprint are generally discouraged once it begins
- Not a fully defined methodology and may require integration with other practices (e.g., Extreme Programming (XP), Kanban, Dynamic Systems Development Method (DSDM)).
- Difficult to manage projects without clear requirements or vision
- Requires frequent meetings and active involvement, which can be time-consuming

ADVANTAGES AND DISADVANTAGES OF EACH SDLC MODEL

1. Classical Waterfall Model

Advantages

- Easy to understand and use.
- Each phase is completed one by one.
- Documentation is clear.
- Good for small projects with fixed requirements.
- Easy to manage because every step has a fixed order.

Disadvantages

- It is difficult to make changes after development starts.
- Customer cannot see the product until the end.
- If a mistake happens in the beginning, it is costly to fix later.
- Not suitable for large or changing projects.
- Testing is done only after development is finished.

2. Iterative Waterfall Model

Advantages

- Errors can be corrected by going back to the previous phase.
- Better than the classical waterfall model.
- Easy to understand.
- Suitable for projects with small changes.
- Improves software quality.

Disadvantages

- Going back to earlier phases takes extra time.
- Cost may increase because of repeated work.
- Not suitable for projects with many changing requirements.
- Customer involvement is still low.

3. Prototyping Model

Advantages

- Customer can see a sample (prototype) before the final product.
- Helps understand customer requirements clearly.
- Reduces misunderstanding between customer and developer.
- Changes can be made easily in the early stage.
- Improves customer satisfaction.

Disadvantages

- Building prototypes takes extra time and money.
- Customer may think the prototype is the final product.
- Frequent changes may delay the project.
- Not suitable for very large projects.

4. Evolutionary (Incremental) Model

Advantages

- Software is delivered in small parts.
- Customer gets useful features early.
- Easy to add new features later.
- Testing is easier for each part.
- Problems can be found and fixed early.

Disadvantages

- Good planning is needed.
- Total cost may become high.
- Managing many small releases can be difficult.
- Not suitable if the project cannot be divided into small parts.

5. Spiral Model

Advantages

- Best for large and risky projects.
- Risks are found and solved early.
- Customer can give feedback after every cycle.
- Changes can be made easily.
- Produces high-quality software.

Disadvantages

- Expensive model.
- Needs experienced developers.
- Takes more time.
- Difficult to manage.
- Not suitable for small projects.

6. Agile Model

Advantages

- Easy to make changes at any time.
- Customer is involved throughout the project.
- Software is delivered quickly.
- Problems are found and fixed early.
- Improves teamwork and communication.
- Gives high customer satisfaction.

Disadvantages

- Needs continuous customer involvement.
- Difficult to estimate total cost and time.
- Requires experienced team members.
- Documentation may be less.
- Not suitable if the customer is unavailable.

7. SCRUM Model

Advantages

- Work is completed in short time periods (Sprints).
- Customer gets regular updates.
- Easy to make changes.
- Team communication is better.
- Problems are solved quickly.
- Delivers working software faster.

Disadvantages

- Needs experienced Scrum team members.
- Requires active customer participation.
- Daily meetings take time.
- Difficult for very large teams.
- Success depends on good teamwork.

Quick Revision Table

SDLC Model	Advantages	Disadvantages
Classical Waterfall	Simple, easy to manage, clear documentation	No easy changes, testing at the end
Iterative Waterfall	Can fix mistakes, better quality	More time and cost, limited flexibility
Prototyping	Clear requirements, customer feedback	Extra time, extra cost, frequent changes
Evolutionary (Incremental)	Early delivery, easy testing, easy updates	Needs good planning, higher cost
Spiral	Handles risks, flexible, high quality	Expensive, complex, time-consuming
Agile	Fast delivery, flexible, customer involvement	Needs experienced team, less documentation
SCRUM	Quick updates, better teamwork, fast delivery	Needs skilled team, daily meetings, customer involvement

Software Engineering

Unit – III

Requirement Analysis and Specification

Requirements Gathering

Requirement gathering and analysis is the foundational phase in project management and software development where teams collect what stakeholders want, study those needs for clarity and feasibility, and document them to build the right product. It bridges the gap between vague business ideas and technical execution.

This step involves actively communicating with users, clients, and managers to discover what the system must accomplish.

- **Identify stakeholders**: Find out who owns, uses, or is affected by the system.
- **Conduct interviews**: Ask direct questions to key individuals.
- **Run workshops**: Host group brainstorming sessions.
- **Observe users**: Watch how people do their daily tasks.
- **Send surveys**: Collect broad feedback from large user groups.

Requirement Analysis

This step takes the raw data collected and refines it to ensure it makes sense, has no contradictions, and is realistic to build.

- **Resolve conflicts**: Fix disagreements between different stakeholder requests.
- **Check feasibility**: Ensure the tech, budget, and time allow for the request.
- **Prioritize needs**: Rank features by importance (such as using the MoSCoW method: Must have, Should have, Could have, Won't have).
- **Model systems**: Draw diagrams or create mockups to visualize data and workflows.

The process finishes with a clear record, usually a **Software Requirements Specification (SRS)** document or user stories, which guides the design and coding stages.

Feasibility Study

A feasibility study is an early evaluation of a proposed software project to determine whether the project is practical, achievable, and worth developing. It is conducted during the early planning stage of the Software Development Life Cycle (SDLC), before significant development work begins.

The main purpose of a feasibility study is to determine whether the proposed software can be developed successfully within the available technology, budget, resources, time, and legal requirements. It helps the organization make an informed decision about whether the project should be developed or not.

Objectives of Feasibility Study

The main objectives of conducting a feasibility study are:

- To determine whether the proposed software can be developed successfully.
- To check whether the project will provide sufficient benefits to the organization.
- To avoid unnecessary use of time, money, and effort.
- To identify major risks and problems before development begins.
- To determine the resources, technology, and time required for the project.
- To support the decision of whether to proceed with the project or not.

Types of Feasibility Study

1. Technical Feasibility

Meaning:
Technical feasibility checks whether the required technology, hardware, software, and technical skills are available to develop the proposed software system.

Questions to Consider:

- Do we have the required hardware?
- Do we have the required software and technology?
- Do developers have the necessary technical skills?
- Can the existing technology support the proposed system?
- Can the system be maintained and upgraded?

Simple Question:
Can we build the system with the available technology and skills?

Example:
A college wants to develop an online examination system. Technical feasibility checks whether the college has suitable computers, servers, software, network facilities, and people with the required programming skills.

2. Operational Feasibility

Meaning:
Operational feasibility checks whether the proposed software will work effectively in the real-world environment and whether users will be able to use and accept the system.

Questions to Consider:

- Will the software meet the needs of users?
- Will users find the system easy to use?
- Can the organization operate the system effectively?
- Can the system be maintained after implementation?
- Will users and the organization accept the new system?

Simple Question:
Can people use and operate the system effectively?

Example:
A college plans to introduce an online attendance system. Operational feasibility checks whether teachers and students can use the system easily and whether it fits the college's existing working process.

3. Economic Feasibility

Meaning:
Economic feasibility examines the cost of developing and operating the proposed software and compares it with the expected benefits.

Questions to Consider:

- How much will the project cost?
- What will be the hardware and software costs?
- What will be the development and operating costs?

- Will the expected benefits justify the cost?

Simple Question:
Is the project worth the money?

Example:
A college wants to develop a student management system. Economic feasibility checks the cost of development, computers, servers, software, maintenance, and training and compares these costs with the expected benefits.

4. Legal Feasibility

Meaning:
Legal feasibility checks whether the proposed software complies with applicable laws, regulations, licenses, copyright requirements, and other legal requirements.

Questions to Consider:
- Does the system follow applicable data protection and privacy requirements?
- Are required software licenses available?
- Does the project follow copyright rules?
- Does the system meet legal and ethical requirements?

Simple Question:
Is the project legally allowed?

Example:
A college develops a student information system that stores personal information. Legal feasibility checks whether the system handles and protects this information according to applicable requirements.

5. Schedule Feasibility

Meaning:
Schedule feasibility checks whether the proposed software can be completed within the available time and before the required deadline.

Questions to Consider:

- How much time will development require?
- Can the team complete the project before the deadline?
- Are the proposed deadlines realistic?
- What will happen if the project is delayed?

Simple Question:
Can we complete the project on time?

Example:
A college wants an online admission system to be ready before the next admission session. Schedule feasibility checks whether the development team has enough time to design, develop, test, and deploy the system before admissions begin.

6. Cultural and Political Feasibility

Meaning:
Cultural and political feasibility examines whether the proposed software fits the organization's culture and whether internal resistance or organizational issues may affect the project.

Questions to Consider:

- Will employees accept the new system?
- Does the system fit the organization's working culture?
- Is there any internal opposition to the project?
- Are there organizational or political issues that may affect implementation?

Simple Question:
Will people in the organization accept the new system?

Example:
A college replaces its manual attendance process with a computerized attendance system. Cultural and political feasibility considers whether teachers and staff will accept the new method and adapt to the change.

7. Market Feasibility

Meaning:
Market feasibility checks whether the proposed software or product is likely to be accepted and used by its intended users and whether there is sufficient demand for it.

Questions to Consider:

- Who are the target users?
- What do customers or users need?
- Are similar products already available?
- Is there sufficient demand for the proposed product?

Simple Question:
Will people use or buy the product?

Example:
A software company plans to develop a mobile application for online learning. Market feasibility checks who the target users are, what features they need, whether similar applications already exist, and whether there is sufficient demand.

8. Resource Feasibility

Meaning:
Resource feasibility checks whether sufficient human, financial, hardware, and software resources are available to complete the proposed software project.

Questions to Consider:

- Is sufficient funding available?
- Are the required hardware and software available?
- Are skilled employees available?
- Are enough people available to complete the project?

Simple Question:
Do we have enough resources to complete the project?

Example:
A college wants to develop a student portal. Resource feasibility checks whether sufficient funds, computers, software, developers, testers, and other required resources are available.

Importance of Feasibility Study

A feasibility study is important because it helps an organization evaluate a software project before investing significant resources in development.

The major benefits are:

- **Risk Reduction:** It helps identify major problems and risks at an early stage.
- **Better Budgeting:** It helps determine whether the project cost is reasonable.
- **Better Decision-Making:** It provides information for deciding whether to proceed with the project.
- **Better Planning:** It helps identify the required resources, technology, and time.
- **Avoids Waste:** It prevents unnecessary expenditure of time, money, and effort on impractical projects.

Easy Way to Remember

1. **Technical** → Can we build it?
2. **Operational** → Can people use it?
3. **Economic** → Is it worth the money?
4. **Legal** → Is it allowed by law?
5. **Schedule** → Can we finish it on time?
6. **Cultural & Political** → Will people accept it?
7. **Market** → Will people use or buy it?
8. **Resource** → Do we have enough resources?

Software Requirements Specification (SRS)

4.1 Definition of SRS

A Software Requirements Specification (SRS) is a detailed document that describes the requirements of a software system. It specifies what the software should do, how the system should behave, and the functional and non-functional requirements that the system must satisfy.

The SRS acts as a common reference between the client, users, project managers, developers, designers, and testers. It helps everyone understand the requirements of the proposed software before the design and coding stages begin.

A simple way to understand SRS is to think of it as a blueprint for software. Just as a building blueprint describes what needs to be constructed, an SRS describes what needs to be developed in a software system.

4.2 Purpose and Importance of SRS

The main purpose of an SRS is to clearly document the requirements of the software system so that the development team and stakeholders have a common understanding of the project.
An SRS is important because it:

- Prevents Misunderstandings: It provides a clear understanding of what the client and users expect from the software.
- Reduces Rework: Clearly documented requirements help identify missing or incorrect requirements before coding begins.
- Guides Development: Developers use the SRS to understand what features and functions need to be implemented.

- Supports Testing: Testers use the requirements to prepare test cases and verify whether the system works as expected.
- Helps Project Planning: Project managers can use the requirements to estimate resources, cost, time, and development activities.
- Provides a Reference: The SRS can be used as a reference during development, testing, maintenance, and future changes.

4.3 Contents of an SRS

An SRS generally contains the following major sections:

1. Introduction

The Introduction provides an overview of the software project. It describes the purpose and scope of the system, intended users, important definitions, and abbreviations used in the document.

2. Overall Description

The Overall Description gives a general view of the proposed system. It may describe the major functions of the product, types of users, operating environment, assumptions, dependencies, and constraints.

3. Functional Requirements

Functional requirements describe the specific functions and activities that the software must perform. They may include:

- Inputs accepted by the system.
- Outputs produced by the system.
- Processing performed by the system.
- Responses to user actions.
- Specific functions provided to users.

Example:

For a student management system, the software should allow an authorized user to add, update, search, and delete student records.

4. Non-Functional Requirements

Non-functional requirements describe the quality attributes and constraints of the software rather than specific functions.

They may include:

- Performance
- Security
- Reliability
- Usability
- Scalability
- Maintainability

Example:

The student management system should display a student's record within 2 seconds after a valid search request.

5. External Interfaces

External interfaces describe how the software interacts with users, hardware, other software, or external systems.

Examples include:

- User interfaces
- Hardware interfaces
- Software interfaces
- Communication interfaces

4.4 Characteristics of a Good SRS

A good SRS should have the following characteristics:

1. Correctness

The requirements stated in the SRS should accurately represent the actual needs of the client and users.

Example:

If the client requires online student registration, the SRS should correctly describe the required registration functions.

2. Completeness

The SRS should contain all important requirements needed to understand and develop the system.

Example:

If a system has a login function, the SRS should also describe what happens when the username or password is incorrect.

3. Unambiguity

Each requirement should have only one clear interpretation. It should not contain vague or confusing statements.

Poor requirement:

"The system should be fast."

Better requirement:

"The system should display the search result within 2 seconds."

4. Consistency

The requirements in the SRS should not contradict one another.

Example:

If one part of the SRS states that only administrators can delete records, another part should not state that every user can delete records.

5. Verifiability

Each requirement should be clear enough that it can be checked or tested.

Example:

"The login page should be displayed within 2 seconds" can be tested by measuring the response time.

6. Modifiability

The SRS should be organized in such a way that requirements can be changed or updated easily without affecting the entire document.

Clear headings, numbering, and sections help make an SRS easy to modify.

7. Traceability

Each requirement should be identifiable and traceable throughout the software development process.

Traceability helps connect a requirement with its design, implementation, and related test cases.

8. Feasibility

The requirements should be realistic and achievable within the available technology, resources, budget, and time.

Example:

A requirement should not demand a feature that cannot be developed with the available technology or resources.

4.5 Users of an SRS

Different people involved in a software project use the SRS for different purposes.

1. Customers and Stakeholders

They use the SRS to verify that the proposed software meets their business needs and expectations.

2. Project Managers

Project managers use the SRS to estimate project cost, plan schedules, allocate resources, and monitor development progress.

3. Developers and Programmers

Developers use the SRS to understand the functions and requirements that need to be implemented.

4. Designers

UI/UX designers use the SRS to understand interface requirements and design appropriate screens and user flows.

5. Testers and Quality Assurance Team

Testers use the SRS to prepare test cases and verify whether the developed software satisfies the specified requirements.

6. Maintenance and Support Teams

Maintenance teams use the SRS as a reference when fixing problems, modifying the system, or adding new features after implementation.

7. Documentation Writers

Documentation writers use the SRS to understand the features and functions of the software and prepare user manuals and help documents.

Decision Table and Decision Tree

1. Decision Table

A **Decision Table** is a tabular method used to represent different combinations of conditions and the actions that should be performed for each combination. It is useful when a software system has multiple conditions that can lead to different actions.

A decision table consists of:

- **Conditions:** The situations or inputs that are checked.
- **Rules:** Different combinations of conditions.
- **Actions:** The activities performed when a particular rule is satisfied.

Example: Online Login System

Consider an online student portal where a user enters a username and password.

The system needs to check whether the username and password are valid.

Conditions / Actions	Rule 1	Rule 2	Rule 3
Username valid?	Yes	Yes	No
Password valid?	Yes	No	-
Action	Login successful	Show "Invalid Password"	Show "Invalid Username"

Here:

- **Conditions:** Username validity and password validity.
- **Rules:** Different combinations of valid and invalid conditions.
- **Actions:** Login successful, show invalid password, or show invalid username.

Understanding the Decision Table

The decision table follows the sequence:

Condition → Rule → Action

For example:

Rule 1:

Username is valid + Password is valid → **Login successful**

Rule 2:

Username is valid + Password is invalid → **Show "Invalid Password"**

Rule 3:

Username is invalid → **Show "Invalid Username"**

The table makes it easy to see what action the software should take for each possible combination of conditions.

Advantages of Decision Table

- Represents complex decision logic in a simple tabular form.
- Helps identify different combinations of conditions.
- Makes business rules easier to understand.
- Helps identify missing or conflicting rules.
- Can be useful for designing and testing software requirements.

Limitations of Decision Table

- Large numbers of conditions can make the table very large.
- It may be difficult to understand when there are many conditions and actions.
- It does not show the sequence of decisions as clearly as a decision tree.

2. Decision Tree

A **Decision Tree** is a graphical method used to represent decisions and their possible outcomes. It shows conditions as decision points and uses branches to represent the possible results of each condition.

A decision tree is useful when decisions are made step by step.

Components of a Decision Tree

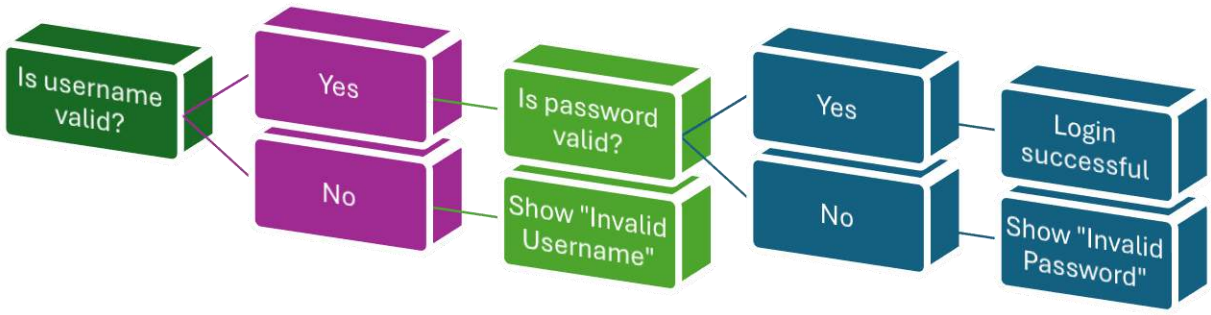
A decision tree generally contains:

- **Decision Node:** Represents a condition or decision that must be checked.
- **Branch:** Represents a possible result of a condition.
- **Terminal Node:** Represents the final action or outcome.

Example: Online Login System

Consider the same online student portal login system.

The system first checks whether the username is valid. If the username is valid, it then checks whether the password is valid.



Understanding the Decision Tree

The decision tree represents the decision process step by step.

Step 1: Check whether the username is valid.

- If **No** → Show "Invalid Username".
- If **Yes** → Check the password.

Step 2: Check whether the password is valid.

- If **Yes** → Login successful.
- If **No** → Show "Invalid Password".

Therefore, the decision tree follows the sequence:

Decision Node → Condition → Branch → Result

Advantages of Decision Tree

- Easy to understand because it represents decisions graphically.
- Shows the sequence of decisions clearly.
- Helps identify possible outcomes.
- Useful for representing step-by-step software logic.
- Can help developers and testers understand decision paths.

Limitations of Decision Tree

- Large numbers of conditions can make the tree complicated.
- A tree can become difficult to read when there are many branches.
- It may require more space than a simple decision table.

Decision Table vs Decision Tree

A Decision Table and a Decision Tree are both used to represent decision logic in software requirements. However, they represent the information in different ways.

Decision Table	Decision Tree
Uses rows and columns to represent conditions and actions.	Uses nodes and branches to represent decisions and outcomes.
Represents decision rules in tabular form.	Represents decisions in graphical form.
Suitable for handling many combinations of conditions.	Suitable for showing step-by-step decisions.
Makes it easy to check whether different condition combinations and rules are covered.	Makes it easy to visualize the different decision paths.
Best suited for complex combinations of conditions and actions.	Best suited for sequential or hierarchical decisions.
Can become large when many conditions and rules are involved.	Can become complicated when many branches are added.

Simple Way to Remember
Decision Table → Think in Rows and Columns
Decision Tree → Think in Nodes and Branches

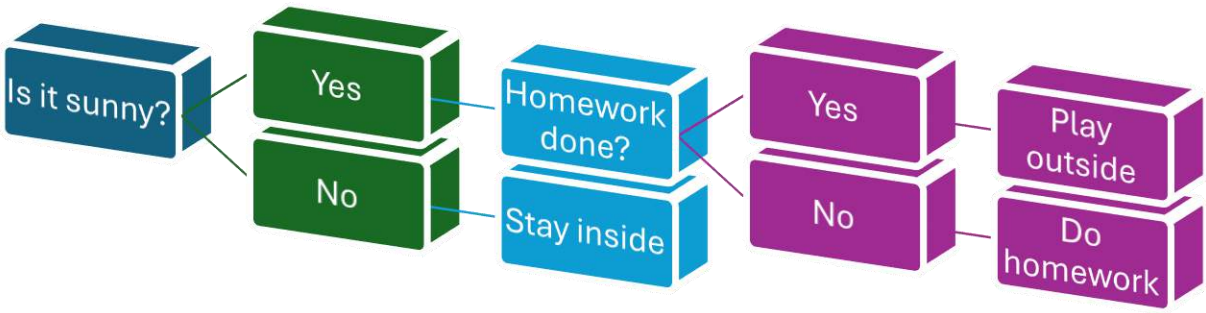
Example

If you are deciding whether to play outside:

Decision Table:

Weather	Homework Done?	Action
Sunny	Yes	Play outside
Sunny	No	Do homework
Rainy	Yes	Stay inside
Rainy	No	Do homework

Decision Tree:



So, a **Decision Table is like a list of rules**, while a **Decision Tree is like a path that helps us reach an answer**.

UNIT 5

Software Design Principles and Methodology

Detailed and briefly explained notes

1. Introduction to Software Design

Software design is the process of transforming software requirements into a blueprint that describes how a software system will be developed and implemented. After requirements are collected and analysed, the design stage explains the structure, components, modules, interfaces and data of the system.

Example: In a College Management System, software design describes how student details are stored, how attendance is recorded, how teachers manage records and how examination results are generated.

Objectives of Software Design

- To develop a clear structure for the software.
- To reduce software complexity and make the system easier to understand.
- To improve software quality, reliability and efficiency.
- To make the software easier to test, modify and maintain.
- To ensure that user requirements are satisfied.
- To reduce development time and cost.

Characteristics of Good Software Design

- **Correctness:** The design satisfies the specified requirements.
- **Simplicity:** The design is easy to understand.
- **Modularity:** The system is divided into manageable modules.
- **Efficiency:** Resources such as memory and processing time are used effectively.
- **Maintainability:** The software can be modified without unnecessary difficulty.
- **Reusability:** Components can be reused in other applications where suitable.
- **Reliability:** The design supports dependable system operation.

2. Top-Down Approach

The **Top-Down Approach** is a software design technique in which a large and complex problem is divided into smaller, manageable sub-problems. Design begins with the complete system and gradually moves towards its smaller modules. It is also known as **stepwise refinement**.

Example: A College Management System may first be divided into Student Management, Teacher Management, Examination Management and Attendance Management. Student Management can then be divided into Add Student, Update Student, Delete Student and Search Student.

Advantages

- Makes a complex system easier to understand and plan.
- Provides a clear view of the overall system.
- Helps identify major functions and responsibilities.
- Supports systematic development and project planning.

Disadvantages

- Detailed, lower-level requirements may be considered late.
- Changes to a high-level design can affect many lower-level modules.
- Reusable low-level components may not be identified early.

3. Bottom-Up Approach

The **Bottom-Up Approach** is a design technique in which development starts with small, basic components. These components are tested and combined to form larger modules, and finally the complete system.

Example: In a College Management System, Student Registration, Attendance Calculation, Marks Calculation and Fee Calculation components may be developed separately and then integrated into the complete application.

Advantages

- Small components are easier to develop and test.
- Reusable components can be identified early.
- Different components can be developed independently.
- Supports modular programming and component integration.

Disadvantages

- The overall system structure may not be clear at the beginning.
- Integration of independently developed components may be difficult.
- Higher-level requirements can be overlooked if planning is weak.

Difference Between Top-Down and Bottom-Up Approaches

Top-Down Approach	Bottom-Up Approach
Starts with the complete system.	Starts with small components.
Divides a large system into smaller modules.	Combines small modules into larger systems.
Overall structure is considered first.	Individual components are considered first.
Lower-level details are developed later.	Higher-level structure emerges through integration.

4. External Design

External Design defines how a software system interacts with users and external systems. It focuses on the visible features and behaviour of the software—what users see and how they interact with it.

Main Elements

- **User interface:** Screens, buttons, menus, forms and messages.
- **Input design:** How users enter data, such as through forms or search boxes.
- **Output design:** How information is displayed, printed or downloaded.
- **Navigation design:** How users move between screens and functions.
- **External interfaces:** How the software communicates with other systems or services.

Example: In an online banking application, the login page, account-balance screen, fund-transfer form and transaction-history page are parts of its external design.

Importance: Good external design makes software understandable, accessible and convenient to use.

5. Architectural Design

Architectural Design defines the overall structure of a software system. It identifies the major components or subsystems, their responsibilities and the relationships and communication between them.

Main Activities

1. Identify the major components of the system.
2. Divide the system into suitable modules or subsystems.
3. Define the responsibility of each component.
4. Specify how components communicate and exchange information.
5. Select an appropriate architectural style or structure.

Example: An Online Shopping System may contain User Interface, Product Management, Shopping Cart, Payment and Database modules.

Advantages

- Provides an overall structure for the system.
- Makes complex systems easier to manage.
- Supports scalability and future development.
- Helps developers understand component relationships.
- Improves maintainability and coordination.

6. Detailed Design

Detailed Design describes the internal working of each software module. It explains how individual modules perform their tasks, including their algorithms, data structures, functions, interfaces and error-handling logic.

Main Elements

- Algorithm and logic design.
- Data structure and variable design.
- Function and procedure specifications.
- Database and file-handling details.
- Input validation and error handling.
- Module-level interfaces.

Example: The detailed design of a student-registration module explains how details are accepted, validated, checked for duplicate student IDs, stored in the database and acknowledged to the user.

Difference Between External, Architectural and Detailed Design

External Design	Architectural Design	Detailed Design
Focuses on user interaction.	Focuses on overall system structure.	Focuses on internal module logic.
Defines screens and interfaces.	Defines components and relationships.	Defines algorithms and data structures.
User-oriented.	System-oriented.	Module-oriented.

7. Cohesion in Software Design

Cohesion is the degree to which the elements or functions within a single module are related to one another. A module has high cohesion when all its parts work together to perform one clear and closely related task.

Example: A Student Registration module that only handles registration activities has high cohesion.

Types of Cohesion: The commonly taught seven types are listed below, from weakest to strongest.

1. Coincidental Cohesion

This is the weakest type. Unrelated tasks are placed together in one module without a meaningful relationship. **Example:** A single module performs printing, salary calculation and attendance checking.

2. Logical Cohesion

Similar categories of activities are grouped together, and a control condition selects which activity is performed. **Example:** One input module handles keyboard input, file input or mouse input.

3. Temporal Cohesion

Activities are grouped because they are performed at the same time, usually during initialization or shutdown. **Example:** A startup module opens files, allocates memory and initializes variables.

4. Procedural Cohesion

Activities are grouped because they must follow a particular sequence of execution, although they may operate on different data. **Example:** A module accepts information, displays a message and then prints a report in a specified order.

5. Communicational Cohesion

Activities are grouped because they use or operate on the same data. **Example:** A student-record module displays, updates and prints information from the same student record.

6. Sequential Cohesion

Activities are grouped so that the output of one activity becomes the input of the next. **Example:** Accept marks, calculate total, calculate percentage and display the result.

7. Functional Cohesion

This is the strongest type. Every part of the module contributes to one specific function. **Example:** A module that only calculates a student's percentage.

Importance of Cohesion

- Makes modules easier to understand and test.
- Improves maintainability and reliability.
- Supports reuse of well-defined modules.
- Reduces unnecessary complexity.

Important: High cohesion is desirable in good software design.

8. Coupling in Software Design

Coupling is the degree of dependency or interconnection between different software modules. If one module depends heavily on the internal details of another module, the modules have high coupling.

Example: If the Student Management module directly accesses and changes the internal data of the Fee Management module, the modules are tightly coupled.

Types of Coupling: The following six types are commonly discussed, from strongest to weakest.

1. Content Coupling

This is the strongest and least desirable type. One module directly accesses or modifies the internal code or data of another module. **Example:** Module A changes a local variable inside Module B.

2. Common Coupling

Two or more modules share the same global data. **Example:** Several modules read and modify the same global variable containing student marks.

3. External Coupling

Modules depend on an externally imposed format, protocol, device or communication interface. **Example:** Two modules depend on the same external file format.

4. Control Coupling

One module passes control information to another module to determine what operation it should perform. **Example:** A module passes a flag telling another module whether to print a report or display it.

5. Stamp Coupling

A module passes a complete data structure to another module, although the receiving module uses only part of it. **Example:** A complete student record is passed when only the student's name is needed.

6. Data Coupling

Modules communicate by passing only the data required through parameters. It is the weakest and most desirable type among these six. **Example:** A module passes only a student's roll number to another module to search for the record.

Importance of Low Coupling

- Reduces dependency between modules.
- Makes individual modules easier to modify and test.
- Improves maintainability and flexibility.
- Reduces the chance that a change in one module will affect others.

Important: Good software design aims for high cohesion and low coupling.

Difference Between Cohesion and Coupling

Cohesion	Coupling
Measures relationships within a module.	Measures dependency between modules.
Focuses on one module.	Focuses on connections between modules.
High cohesion is desirable.	Low coupling is desirable.
Helps a module perform one clear task.	Helps keep modules independent.

9. Structured Analysis and Structured Design (SA/SD)

SA/SD stands for Structured Analysis and Structured Design. It is a systematic approach used to analyse software requirements and convert them into a structured design.

A. Structured Analysis (SA)

Structured Analysis focuses on **what the system should do**. It studies user requirements and represents the system's processes and data flows using tools such as Data Flow Diagrams (DFDs).

- Collect and study requirements.
- Identify system inputs, outputs and processes.
- Identify data flows and data stores.
- Represent system functions using DFDs.

B. Structured Design (SD)

Structured Design focuses on **how the system should be organized and developed**. It converts analysis results into a modular design.

- Identify software modules.
- Define module responsibilities and relationships.
- Design module interfaces and data structures.
- Prepare detailed specifications for implementation.

Structured Analysis	Structured Design
Focuses on what the system does.	Focuses on how the system is designed.
Analyses user requirements.	Converts requirements into a design.
Identifies processes and data flows.	Identifies modules and their relationships.
Common tool: DFD.	Uses modular design and structure charts.

10. Data Flow Diagram (DFD)

A **Data Flow Diagram (DFD)** is a graphical representation of how data moves through a system. It shows how data enters the system, how it is processed, where it is stored and how it leaves the system.

Example: In a Student Registration System, a student submits registration details. The system processes the details, stores them in a database and produces a confirmation.

Four Basic Symbols Used in DFD

1. External Entity

An external entity is a person, organization or another system that sends data to or receives data from the system. It is generally represented by a rectangle. **Examples:** Student, Teacher, Bank.

2. Process

A process represents an activity that transforms input data into output data. It is commonly represented by a circle or rounded rectangle, depending on the notation. **Examples:** Calculate Result, Process Payment.

3. Data Flow

Data flow represents the movement of data from one part of the system to another. It is represented by an arrow. **Examples:** Student Details, Payment Information.

4. Data Store

A data store represents a place where information is stored for future use. It is represented by two parallel lines or another notation-specific symbol. **Examples:** Student Database, Employee Records.

DFD Symbol	Meaning	Example
Rectangle	External Entity	Student
Circle / rounded rectangle	Process	Calculate Result
Arrow	Data Flow	Student Details
Parallel lines	Data Store	Student Database

11. Context Diagram in DFD

A **Context Diagram** is the highest-level representation of a DFD. It shows the entire software system as a single process and represents its interaction with external entities. It gives an overall view without showing the system's internal processes.

Example: In a College Management System, Student, Teacher and Administrator are external entities. The College Management System is represented as one main process. These entities exchange information with the system.

Characteristics

- Represents the complete system as one process.
- Shows external entities and data entering or leaving the system.
- Does not show internal processes or data stores in detail.
- Provides a simple overview of system boundaries and interactions.

12. Advantages of DFD

1. **Easy to understand:** It presents system operations graphically.
2. **Better communication:** It helps users, analysts and developers discuss requirements.
3. **System analysis:** It helps identify inputs, outputs, processes and data stores.
4. **Documentation:** It provides a visual record of system requirements.
5. **Error identification:** It can reveal missing, duplicated or unnecessary data flows.
6. **Design support:** It provides a foundation for structured software design.
7. **Modularity:** It helps break a complex system into smaller processes.

13. Disadvantages of DFD

1. A DFD does not show the exact sequence or timing of operations.
2. It does not explain detailed program logic or algorithms.
3. It cannot represent decision-making logic in detail.
4. Large systems may require many levels of diagrams, which can become difficult to manage.
5. It does not describe the internal structure of data in sufficient detail.

14. Balanced DFD

Balanced DFD means maintaining consistency of input and output data flows between different levels of Data Flow Diagrams. When a process in a higher-level DFD is expanded into a lower-level diagram, the data entering and leaving that process must remain consistent.

Example: Suppose a College Management System receives Student Registration Details and produces Registration Confirmation. When the system is expanded into a Level 1 DFD, these external input and output flows must still be represented.

Importance

- Maintains consistency between DFD levels.
- Prevents important data flows from being omitted.
- Improves accuracy and documentation.
- Makes system analysis easier to verify.

15. Structured Design: Transaction Analysis and Transform Analysis

Structured Design is a systematic approach to designing software by dividing a large system into smaller, manageable modules. It aims to achieve modularity, high cohesion and low coupling. Two important techniques are Transaction Analysis and Transform Analysis.

A. Transaction Analysis

Transaction Analysis is used when a system receives different types of input transactions and performs different activities depending on the type of transaction. A transaction is a request or input that causes a particular operation.

Example: Banking System. A banking system may receive Deposit Money, Withdraw Money and Check Balance transactions. The system identifies the selected transaction and directs it to the appropriate operation.

Steps of Transaction Analysis

1. Identify all possible input transactions.
2. Identify the transaction centre or point where the transaction type is determined.
3. Identify the different action paths.
4. Create separate modules for the different transactions.
5. Connect the modules to form the complete design.

B. Transform Analysis

Transform Analysis is used when input data passes through a series of processing steps and is transformed into output. It is suitable for systems with a clear input-processing-output flow.

Example: Student Result Processing System. The system accepts marks, calculates total marks, calculates percentage, determines the grade and displays the result.

Steps of Transform Analysis

1. Identify the input data flows.
2. Identify the main transformation or processing steps.
3. Identify the output data flows.
4. Divide the system into input, transformation and output modules.
5. Develop the structured design for these modules.

Difference Between Transaction Analysis and Transform Analysis

Transaction Analysis	Transform Analysis
Handles different types of transactions.	Handles data transformed through processing stages.
Selects an action path based on transaction type.	Organizes input, transformation and output stages.
Different transactions lead to different operations.	Data passes through a processing sequence.
Example: Banking system.	Example: Student result processing.

16. Object-Oriented Analysis and Design (OOAD)

Object-Oriented Analysis and Design (OOAD) is an approach to analysing and designing software systems using objects and classes. It represents real-world entities as objects that contain data and behaviour.

Need for OOAD

- **Reusability:** Classes and components can be reused where appropriate.
- **Modularity:** Software is divided into classes and objects.
- **Maintainability:** Changes can be made more easily to well-organized components.
- **Flexibility:** New features can be added through extension and composition.
- **Data protection:** Encapsulation controls access to internal data.
- **Real-world modelling:** Real entities and their relationships can be represented naturally.

Basic Concepts of Object-Oriented Design

1. Class

A **class** is a blueprint or template used to create objects. It defines the attributes (data) and methods (functions) that its objects can have. **Example:** Student is a class with attributes Student_ID, Name and Department, and methods such as register() and display().

2. Object

An **object** is an instance of a class. It represents a particular entity with its own data and behaviour. **Example:** A student with ID 101 and name Rahul is an object of the Student class.

3. Encapsulation

Encapsulation is the process of combining data and the methods that operate on it into a single unit, usually a class, while controlling access to the data. **Example:** Student details are stored inside a Student class and accessed through suitable methods.

4. Abstraction

Abstraction means hiding unnecessary implementation details and showing only the essential features to the user. **Example:** An ATM user can withdraw money without knowing the internal banking operations.

5. Inheritance

Inheritance is a mechanism through which one class acquires properties and methods of another class. **Example:** A Student class may inherit common properties such as name and age from a Person class.

6. Polymorphism

Polymorphism means that one interface or method name can represent different behaviours. **Example:** A calculate() method may calculate the area of a circle, rectangle or triangle depending on the object.

17. UML (Unified Modeling Language)

UML stands for **Unified Modeling Language**. It is a standard visual modelling language used to specify, visualize, construct and document the structure and behaviour of software systems. UML is widely used in object-oriented software development.

Objectives of UML

- To visualize the design of a software system.
- To represent system structure and behaviour.
- To simplify complex systems through diagrams.
- To improve communication among developers and stakeholders.
- To provide software design documentation.
- To help identify design problems before implementation.

18. Different Views of UML

UML provides different views of a system so that its structure, behaviour, development and physical deployment can be understood.

1. Use Case View

Describes system functionality from the user's perspective. It shows how actors interact with the system. **Example:** A student logs in, registers for a course and views examination results.

2. Logical View

Describes the static structure of the system, including classes, objects, attributes, methods and relationships. Class and object diagrams are commonly used.

3. Process View

Describes the dynamic behaviour of the system and how processes interact during execution. It considers concurrency, communication and synchronization.

4. Implementation View

Describes how software components, modules and files are organized during development.

5. Deployment View

Describes how software is physically deployed on hardware devices or computers. **Example:** A web application deployed on a server and accessed through client computers.

19. Various UML Diagrams

The syllabus includes five important UML diagrams: Use Case Diagram, Class Diagram, Object Diagram, Sequence Diagram and Collaboration Diagram.

19.1 Use Case Diagram

A **Use Case Diagram** is a UML behavioural diagram that represents interactions between external users or systems (called actors) and the functions provided by a software system. It describes what the system does from the user's point of view.

Main Components

- **Actor:** A person or external system that interacts with the software. Examples: Student, Teacher and Administrator.
- **Use case:** A function or service provided by the system. Examples: Login, Register and View Result.
- **System boundary:** The boundary that shows which functions belong to the system.
- **Association:** A connection showing communication between an actor and a use case.

Example: In a Student Management System, a Student actor may interact with Login, Register and View Result use cases.

Advantages

- Easy to understand and communicate.
- Shows user-system interaction.
- Helps identify functional requirements.
- Useful during requirement analysis.

19.2 Class Diagram

A **Class Diagram** is a UML structural diagram that represents classes, their attributes, methods and relationships. It describes the static structure of a software system.

Main Components

- **Class name:** The name of the class.
- **Attributes:** Data members or properties of the class.
- **Methods:** Functions or operations performed by the class.
- **Relationships:** Connections between classes.

Example: A Student class may contain attributes student_id, name and department, and methods register(), display() and update().

Common Relationships in Class Diagrams

- **Association:** A general connection between classes.
- **Aggregation:** A whole-part relationship in which the part can exist independently of the whole.
- **Composition:** A strong whole-part relationship in which the part's lifecycle depends on the whole.
- **Inheritance:** One class inherits properties and methods from another class.
- **Dependency:** One class uses or depends on another class.

Advantages

- Represents the static structure of the system.
- Helps in object-oriented programming.
- Shows relationships between classes.
- Makes system design easier to understand.

19.3 Object Diagram

An **Object Diagram** is a UML structural diagram that represents objects and their relationships at a particular point in time. It shows actual instances of classes with their attribute values.

Example: If Student is a class, student1 may have ID = 101 and Name = Rahul, while student2 may have ID = 102 and Name = Priya.

Advantages

- Shows actual object instances and their values.
- Helps understand relationships between objects.
- Can be useful in testing and debugging.
- Provides a snapshot of the system at a particular time.

Class Diagram vs Object Diagram

Class Diagram	Object Diagram
Represents classes.	Represents objects or instances.
Shows general structure.	Shows a snapshot of actual instances.
Shows attributes and operations.	Shows actual attribute values.
Describes the class-level design.	Describes an example object configuration.

19.4 Sequence Diagram

A **Sequence Diagram** is a UML behavioural diagram that shows how objects communicate through messages in a particular time sequence. It emphasizes the order in which messages are exchanged.

Main Components

- **Actor:** An external user or system participating in the interaction.
- **Object:** An object involved in the process.
- **Lifeline:** A vertical dashed line showing an object's existence during the interaction.
- **Activation bar:** Shows the period during which an object performs an operation.
- **Message:** Communication sent from one participant to another.

Example: Student Login. The student enters a username and password. The login page sends the details to the authentication system. The system checks the database, receives the result and displays a success or failure message.

Advantages

- Shows the order of interactions.
- Helps understand communication between objects.
- Supports the design of system operations.
- Can reveal missing or unnecessary interactions.

19.5 Collaboration Diagram

A **Collaboration Diagram**, called a **Communication Diagram** in UML 2.x, is a behavioural diagram that shows how objects communicate with one another to perform a task. It focuses on the organization and relationships of objects and the messages exchanged between them. Messages are numbered to indicate their order.

Main Components

- **Objects:** Participants in the interaction.
- **Links:** Connections between objects.
- **Messages:** Information or requests exchanged between objects.
- **Sequence numbers:** Indicate the order in which messages occur.

Example: Student Login. The Student sends login details to the Login Page. The Login Page forwards them to the Authentication System. The Authentication System requests verification from the Database. The result is returned and displayed to the student.

Advantages

- Shows communication and relationships between objects.
- Helps understand how objects work together.
- Useful for analysing interactions in a system.

20. Important Differences for Revision

Sequence Diagram vs Collaboration Diagram

Sequence Diagram	Collaboration Diagram
Focuses on the time order of messages.	Focuses on object relationships and communication.
Uses lifelines and activation bars.	Uses object links and numbered messages.
Emphasizes when messages occur.	Emphasizes how objects are connected.

Quick Revision Points

- Top-Down: Start with the complete system and divide it into smaller modules.
- Bottom-Up: Build small components and combine them into a complete system.
- Cohesion: Relationship among elements within one module; high cohesion is preferred.
- Coupling: Dependency between modules; low coupling is preferred.
- Structured Analysis: Describes what the system should do.
- Structured Design: Describes how the system should be organized.
- DFD: Shows the movement and processing of data.
- Context Diagram: Shows the entire system as one process and its external interactions.
- Balanced DFD: Maintains input/output consistency between DFD levels.
- Transaction Analysis: Used when different transaction types lead to different action paths.
- Transform Analysis: Used when input data is transformed through processing stages into output.
- OOAD: Analyses and designs software using objects and classes.
- UML: Standard visual language for modelling software systems.
- Use Case Diagram: Shows actors and system functions.
- Class Diagram: Shows classes, attributes, methods and relationships.
- Object Diagram: Shows object instances and their values.
- Sequence Diagram: Shows messages in time order.

• Collaboration Diagram: Shows object communication and relationships.

Coding and Testing

BCA – Software Engineering | Unit Notes | Student-Friendly, Exam-Oriented

Study focus: Understand the goals of coding, review techniques, test cases and test suites, testing levels, verification and validation, black-box and white-box testing, cyclomatic complexity, and common system testing approaches.

1. Goals of Coding

Definition

Coding is the process of translating a software design into instructions written in a programming language that a computer can execute.

Main goals

1. Correctness: The code should implement the specified requirements and produce the expected results.
2. Readability: Code should be clear and easy for other developers to understand.
3. Maintainability: The program should be easy to correct, update, and extend.
4. Efficiency: Code should use time, memory, and other resources appropriately.
5. Reliability and robustness: The program should handle expected inputs and errors properly.
6. Reusability: Common functions or components should be designed so they can be reused where suitable.
7. Testability: The code should be structured so that it can be tested effectively.

Example: In a library management system, a book-issue function should check whether the book is available, update the records correctly, and be written clearly enough for another developer to maintain.

Key takeaway: Good coding is not only about making a program run. It is also about correctness, clarity, efficiency, reliability, and future maintenance.

2. Code Review Techniques

A code review is the examination of source code by someone other than its author, or by a team, to find defects, improve quality, and check whether coding standards are followed. Two traditional review techniques are code walkthrough and code inspection.

2.1 Code Walkthrough

Definition: A code walkthrough is an informal or semi-formal review in which the code author guides reviewers through the program logic, often using examples or test data.

Typical steps:

1. The author selects the code and prepares an explanation.
2. Reviewers examine the code or follow the explanation.
3. The team discusses logic, assumptions, and possible defects.
4. Issues are recorded and the author makes necessary corrections.

Advantages: Encourages discussion, shares knowledge, and is relatively easy to organize.

Limitation: Its effectiveness depends on preparation and reviewer attention; it may be less systematic than a formal inspection.

2.2 Code Inspection

Definition: Code inspection is a formal, structured review of source code using defined procedures, roles, checklists, and defect recording.

Typical steps:

1. Planning: Select the code, participants, and review materials.
2. Preparation: Reviewers study the code individually and use checklists.
3. Inspection meeting: The team identifies and records defects rather than debating solutions at length.
4. Rework: The author corrects the identified defects.
5. Follow-up: The moderator or responsible person checks that issues have been addressed.

Advantages: Systematic defect detection, clear records, and repeatable process.

Limitation: Requires preparation, trained participants, and more time than an informal review.

Walkthrough vs. Inspection

Basis	Code walkthrough	Code inspection
Formality	Usually informal or semi-formal	Formal and structured
Leadership	Often led by the author	Usually managed by a moderator
Preparation	Varies by team	Defined preparation is expected
Main emphasis	Understanding and discussion of code	Systematic detection and recording of defects
Records	May be less formal	Defects and follow-up are documented

3. Test Cases and Test Suites

3.1 Test Case

Definition: A test case is a set of inputs, preconditions, execution steps, and expected results used to check whether a particular requirement or behaviour of software works correctly.

A test case commonly includes:

- Test case ID and objective.
- Preconditions and required test data.
- Input values and steps to execute.
- Expected result.
- Actual result and pass/fail status.

Example: For a college portal login page, enter a registered username and correct password. The expected result is that the user is logged in and the dashboard is displayed.

3.2 Test Suite

Definition: A test suite is a collection of related test cases organized to test a feature, module, or complete system.

Example: A login test suite may contain test cases for valid credentials, incorrect password, empty fields, locked accounts, and password reset.

Remember: A test case checks one defined situation or behaviour; a test suite groups multiple test cases together.

4. Positive Testing and Negative Testing

Positive Testing

Definition: Positive testing checks whether software works correctly when given valid inputs and used according to its requirements.

Example: Entering a valid student ID and a correctly formatted date of birth should retrieve the student's record.

Negative Testing

Definition: Negative testing checks how software behaves when it receives invalid, unexpected, incomplete, or inappropriate inputs.

Example: Entering letters in a field that accepts only numeric marks should produce a clear validation message and should not save invalid data.

Basis	Positive testing	Negative testing
Input	Valid and expected	Invalid or unexpected
Purpose	Confirm intended behaviour	Check error handling and protection
Example	Correct login credentials	Incorrect password or blank username
Expected behaviour	Operation succeeds as specified	Invalid operation is safely rejected or handled

Key takeaway: Positive testing checks that things work; negative testing checks that invalid situations are handled safely and correctly.

5. Different Levels of Software Testing

Testing is often performed at several levels. Each level focuses on a different scope of the software.

5.1 Unit Testing

Definition: Unit testing checks an individual unit or small component of software, such as a function, method, or class.

It is commonly performed by developers and helps find defects early, before components are combined.

Example: Test the function that calculates a student's grade using several marks and expected grade results.

5.2 Integration Testing

Definition: Integration testing checks whether two or more modules work correctly together and exchange data as intended.

It helps identify interface errors, data-transfer problems, and incorrect interactions between components.

Example: Test whether the student registration module correctly sends student information to the fee-management module.

5.3 System Testing

Definition: System testing tests the complete integrated software system against its specified requirements.

It examines the overall behaviour of the application, including functional and relevant non-functional requirements.

Example: Test the complete college management system, from student registration to fee payment and report generation.

5.4 Acceptance Testing

Definition: Acceptance testing evaluates whether the software meets user or customer needs and is suitable for acceptance or release.

It is often carried out by customers, users, or their representatives. User Acceptance Testing (UAT) is a common form.

Example: College administrators check whether the completed system supports their real examination and admission workflows before approving it.

Testing Levels: Summary

Level	Main focus	Example
Unit testing	Individual component	Grade-calculation function
Integration testing	Interactions between components	Registration module communicating with fees module
System testing	Complete integrated system	End-to-end college management application
Acceptance testing	User needs and acceptance criteria	Administrators verify real workflows before approval

Typical sequence: Unit testing → Integration testing → System testing → Acceptance testing. Actual project workflows may vary, and some testing activities overlap.

6. Verification and Validation

Verification

Definition: Verification checks whether software work products conform to specified requirements and design. It asks: “Are we building the product right?”

It can include reviews, inspections, walkthroughs, and other checks of requirements, design, and code. Verification activities may find defects without executing the software.

Validation

Definition: Validation checks whether the software satisfies user needs and its intended use. It asks: “Are we building the right product?”

It commonly involves executing the software and evaluating its behaviour against user needs, though validation can also include other forms of evaluation.

Basis	Verification	Validation
Main question	Are we building the product right?	Are we building the right product?
Focus	Conformance to specifications and design	Fitness for intended use and user needs
Typical activities	Reviews, inspections, walkthroughs	System tests, acceptance tests, user evaluation
Software execution	May not require execution	Often involves executing the software
Example	Reviewing a fee formula against the specification	Checking that the fee system supports the college's actual payment process

Exam memory aid: Verification = product built right. Validation = right product built.

7. Black-Box Testing Approaches

Black-box testing tests software behaviour using inputs and expected outputs without relying on knowledge of the internal source-code structure.

7.1 Equivalence Class Partitioning

Definition: Equivalence Class Partitioning (ECP) divides the input domain into groups, or equivalence classes, whose values are expected to be processed similarly. A representative value from each class is selected for testing.

Example: A form accepts marks from 0 to 100 inclusive.

- Invalid class: Marks less than 0.
- Valid class: Marks from 0 through 100.

- Invalid class: Marks greater than 100.

Representative tests might use −1, 50, and 101. Additional tests may be needed for other requirements such as empty values or non-numeric input.

Benefit: Reduces the number of test cases while providing coverage of different categories of input.

7.2 Boundary Value Analysis

Definition: Boundary Value Analysis (BVA) tests values at and near the boundaries of input ranges because defects often occur around minimum and maximum limits.

Example: If marks must be from 0 to 100 inclusive, useful boundary tests are −1, 0, 1, 99, 100, and 101.

Technique	Main idea	Example for range 0–100
Equivalence Class Partitioning	Choose representatives from valid and invalid groups	−1, 50, 101
Boundary Value Analysis	Test limits and values just below or above them	−1, 0, 1, 99, 100, 101

Key distinction: ECP divides inputs into classes; BVA concentrates on the edges of those classes.

8. White-Box Testing Approaches

White-box testing designs tests using knowledge of the program's internal structure, such as statements, branches, conditions, and execution paths.

8.1 Statement Coverage

Definition: Statement coverage measures the percentage of executable statements that are executed by a test suite.

Statement coverage = (Executed statements ÷ Total executable statements) × 100%

If a test suite executes 90 out of 100 executable statements, statement coverage is 90%.

Statement coverage helps identify code that has not been executed, but it does not guarantee that every decision outcome has been tested.

8.2 Branch Coverage

Definition: Branch coverage measures the percentage of decision outcomes or branches that are executed by the tests. For a simple if statement, the outcomes are commonly True and False.

Branch coverage = (Executed branches ÷ Total branches) × 100%

If 6 out of 8 branches are executed, branch coverage is 75%. Branch coverage is usually stronger than statement coverage because it requires testing decision outcomes.

8.3 Condition Coverage

Definition: Condition coverage checks whether each individual Boolean condition in a decision has evaluated to both True and False across the test suite.

Example: In the decision (age ≥ 18 AND has_ID = true), tests should make each condition evaluate to True at least once and False at least once.

Condition coverage does not necessarily guarantee that every possible branch outcome has occurred, especially when decisions contain multiple conditions.

8.4 Path Coverage

Definition: Path coverage measures whether the possible execution paths through a program have been exercised by tests.

A path is a sequence of statements and branches followed during execution. Full path coverage can be impractical or impossible in programs with loops, because loops can create very many or infinitely many paths.

Example: For a function with an if-else decision, tests should exercise the path through the if branch and the path through the else branch.

Comparison of White-Box Coverage

Coverage type	What must be exercised?	Main limitation
Statement coverage	Executable statements	Can miss untested decision outcomes
Branch coverage	Decision outcomes/branches	Does not ensure every individual condition has both outcomes
Condition coverage	Each Boolean condition as True and False	Does not necessarily cover all branches
Path coverage	Execution paths through the code	Full coverage may be impractical with loops or many paths

9. McCabe's Cyclomatic Complexity

Definition

McCabe's Cyclomatic Complexity is a software metric that measures the number of linearly independent paths through a program's control-flow graph. It helps estimate logical complexity and the minimum number of basis-path tests for a connected, single-entry/single-exit control-flow graph.

Formula

$$V(G) = E - N + 2P$$

Where:

- V(G) = cyclomatic complexity.
- E = number of edges in the control-flow graph.
- N = number of nodes in the graph.
- P = number of connected components; for one connected program graph, P = 1.

For a single connected graph, the formula becomes:

$$V(G) = E - N + 2$$

A common shortcut for structured code is: cyclomatic complexity = number of decision points + 1.

Example

Suppose a simple function contains two decision points, such as two if statements, with no other branching constructs.

$$V(G) = 2 + 1 = 3$$

The function has a cyclomatic complexity of 3, meaning that three linearly independent basis paths can be identified for basis-path testing.

Importance

- Helps estimate the logical complexity of a program.
- Supports the design of basis-path tests.
- Highlights code that may be difficult to understand, test, or maintain.

Important note: Cyclomatic complexity is a measure of control-flow complexity, not a direct count of all possible execution paths or a guarantee of software quality.

10. System Testing Approaches

10.1 Smoke Testing

Definition: Smoke testing is a preliminary set of tests used to check whether the major functions of a new build work well enough for further testing.

Example: After a new college portal build is installed, testers check whether it launches, users can log in, and key pages open.

Purpose: Identify major build problems early before spending time on detailed testing.

10.2 Stress Testing

Definition: Stress testing evaluates how a system behaves under unusually high load or resource pressure, beyond normal operating conditions.

Example: Test a college result portal with an exceptionally large number of simultaneous students attempting to view results.

Purpose: Discover performance limits, failure behaviour, and the system's ability to recover.

10.3 Volume Testing

Definition: Volume testing checks the system's behaviour when it processes or stores a very large quantity of data.

Example: Test whether a library management system can search and update records when its database contains several million book and transaction records.

Purpose: Identify problems related to data volume, storage, querying, and processing.

10.4 Compatibility Testing

Definition: Compatibility testing checks whether software operates correctly across specified environments, such as operating systems, browsers, devices, databases, or hardware configurations.

Example: Test a college website in Chrome, Firefox, and Edge, and on desktop and mobile screen sizes.

Purpose: Ensure that the software works as expected in the environments it is intended to support.

Comparison of System Testing Approaches

Approach	Main focus	Example
Smoke testing	Basic stability of a new build	Check login and key pages after installation
Stress testing	Behaviour under extreme load or pressure	Very high simultaneous result requests
Volume testing	Large amounts of data	Database containing millions of records
Compatibility testing	Different environments and configurations	Test across browsers and operating systems

11. Important Exam Points

Topic	Remember
Goals of coding	Correctness, readability, maintainability, efficiency, reliability, reusability, and testability.
Code walkthrough	Review where the author guides participants through the code and its logic.
Code inspection	Formal, structured review using defined roles, procedures, and defect records.
Test case	Inputs, preconditions, steps, and expected results for a particular check.
Test suite	A collection of related test cases.
Positive testing	Uses valid inputs to check expected behaviour.
Negative testing	Uses invalid or unexpected inputs to check safe error handling.
Testing levels	Unit → Integration → System → Acceptance (typical sequence).
Verification	Are we building the product right?
Validation	Are we building the right product?
ECP	Select representative values from equivalence classes.
BVA	Test boundaries and nearby values.
Statement coverage	Measures executed statements.
Branch coverage	Measures executed decision outcomes.
Condition coverage	Checks each Boolean condition as True and False.
Path coverage	Checks execution paths; full coverage may be impractical.
Cyclomatic complexity	$V(G) = E - N + 2P$; for a connected graph, $E - N + 2$.
Smoke testing	Checks whether a build is stable enough for further testing.
Stress testing	Tests behaviour under extreme load or resource pressure.
Volume testing	Tests handling of large amounts of data.
Compatibility testing	Checks operation across specified environments.

12. Practice Questions for University Examinations

Short-answer questions

1. State the main goals of coding.
2. Define code walkthrough and code inspection.
3. What is a test case? What is a test suite?
4. Differentiate between positive and negative testing.
5. Define unit testing and integration testing.
6. What is the difference between verification and validation?
7. Define equivalence class partitioning and boundary value analysis.
8. What are statement coverage and branch coverage?
9. Define McCabe's cyclomatic complexity.
10. What is smoke testing?

Descriptive questions

1. Explain the goals of coding and discuss code walkthrough and code inspection.

2. Explain the four levels of software testing with suitable examples.
3. Differentiate between verification and validation.
4. Explain equivalence class partitioning and boundary value analysis with examples.
5. Discuss statement, branch, condition, and path coverage.
6. Explain McCabe's cyclomatic complexity, its formula, and its importance.
7. Describe smoke testing, stress testing, volume testing, and compatibility testing.

Practice exercise

A form accepts an age from 18 to 60 inclusive.

a. Use equivalence class partitioning to identify valid and invalid classes.

b. Suggest boundary values for boundary value analysis.

Suggested answer: Valid class: 18–60. Invalid classes: below 18 and above 60. Boundary values: 17, 18, 19, 59, 60, and 61.

Final Revision Summary

Coding aims to produce correct, readable, efficient, maintainable, and testable software. Code reviews help identify defects before execution, while test cases and test suites organize testing. Unit, integration, system, and acceptance testing examine software at different levels. Verification checks conformance to specifications, while validation checks whether the software meets user needs. Black-box techniques include equivalence class partitioning and boundary value analysis; white-box techniques include statement, branch, condition, and path coverage. McCabe's cyclomatic complexity measures independent control-flow paths. Smoke, stress, volume, and compatibility testing examine different aspects of system readiness and behaviour.

Software Reliability and Maintenance

BCA – Software Engineering | Unit 5 | Student-Friendly, Exam-Oriented Notes

Study focus: Understand reliability metrics and their formulas, the purpose of ISO 9001 certification, the three syllabus maintenance types, and methods of estimating maintenance cost.

1. Software Reliability

Definition

Software reliability is the probability that a software system will perform its intended functions without failure for a specified period of time under specified operating conditions.

In simple words, software reliability tells us how likely a software application is to work correctly without crashing, producing errors, or failing to perform its required functions.

Explanation

Reliability is an important attribute of software quality. A reliable system provides consistent and dependable services to its users.

For example, a college examination management system correctly stores marks, calculates results, and generates mark sheets whenever required. It demonstrates good reliability. If it frequently crashes or calculates incorrect results, its reliability is poor.

Software reliability depends on several factors:

- Correctness: The software performs its intended functions accurately.
- Fault-free operation: The software contains fewer defects that can cause failures.
- Testing: Proper testing helps identify and remove defects.
- Operating conditions: Reliability may vary depending on workload, input data, and the environment in which the software operates.

Failure and fault

- Fault (defect or bug): A mistake in the software, such as an incorrect formula in a program.
- Failure: The software's inability to perform a required function during operation, such as displaying an incorrect examination result.

A fault may cause a failure when the relevant part of the software is executed.

Key takeaway: Reliability measures the ability of software to operate correctly over time. It is especially important in banking, healthcare, aviation, and examination systems, where failures can have serious consequences.

2. Reliability Metrics of Software Products

Definition

Software reliability metrics are quantitative measures used to assess and evaluate the reliability, failure behaviour, recovery performance, and availability of software systems.

These metrics help developers identify reliability problems, compare software products, and estimate how well a system will perform during operation.

The main metrics in the syllabus are ROCOF, MTTF, MTTR, MTBF, POFOD, and availability.

2.1 ROCOF — Rate of Occurrence of Failures

Definition: ROCOF measures how frequently failures occur in a software system during a specified period or amount of operation. It is also commonly described as the failure rate in operational reliability analysis.

ROCOF = Number of failures ÷ Total operating time

Example: If a college management system experiences 6 failures during 300 hours of operation:

ROCOF = 6 ÷ 300 = 0.02 failures per operating hour

Therefore, the system experiences an average of 0.02 failures per operating hour, or 2 failures per 100 operating hours.

Importance: A lower ROCOF generally indicates fewer failures per unit of operating time.

2.2 MTTF — Mean Time to Failure

Definition: MTTF is the average operating time before a system or component fails. It is particularly useful for systems or components treated as non-repairable during the measurement period.

MTTF = Total operating time ÷ Number of failures

Example: If a software component operates for a total of 1,000 hours across 5 failure observations:

MTTF = 1,000 ÷ 5 = 200 hours

The average operating time before failure is 200 hours.

Importance: MTTF helps estimate how long software may operate before a failure occurs.

2.3 MTTR — Mean Time to Repair (or Restore)

Definition: MTTR is the average time required to repair a failed system or restore it to working condition. In software operations, MTTR often includes diagnosis, correction, testing, and restoration, depending on the measurement policy.

MTTR = Total repair or restoration time ÷ Number of repairs

Example: If a software team takes a total of 12 hours to restore a system after 4 failures:

MTTR = 12 ÷ 4 = 3 hours

The average restoration time is 3 hours per failure.

Importance: A lower MTTR generally means that the system can be restored more quickly after a failure.

2.4 MTBF — Mean Time Between Failures

Definition: MTBF is the average operating time between successive failures of a repairable system. It is commonly used to describe the reliability of systems that are restored and returned to service after failures.

In a simplified model:

MTBF = MTTF + MTTR

This relationship applies when MTBF is measured as the complete cycle from the start of one operating period to the start of the next, including repair time. If only operating time is counted, the convention may differ.

Example: Suppose a system has an MTTF of 100 hours and an MTTR of 5 hours.

MTBF = 100 + 5 = 105 hours

The average complete failure-to-failure cycle is 105 hours under this measurement convention.

Importance: A higher MTBF generally indicates that failures occur less frequently.

2.5 POFOD — Probability of Failure on Demand

Definition: POFOD is the probability that a software system will fail when a particular service or function is requested. It is especially relevant to systems that remain idle or on standby until a demand occurs.

POFOD = Failed demands ÷ Total demands

Example: If an emergency backup system fails to respond to 2 out of 1,000 demands:

POFOD = 2 ÷ 1,000 = 0.002

POFOD percentage = 0.002 × 100 = 0.2%

Therefore, the probability of failure on demand is 0.2%.

Importance: A lower POFOD is desirable for systems such as emergency alarms, backup protection systems, and safety-critical applications.

2.6 Availability

Definition: Availability is the proportion of time a system is operational and ready to provide its required services. It considers both how frequently failures occur and how quickly the system is restored.

Availability = Uptime ÷ (Uptime + Downtime)

For a simplified repairable system under steady-state conditions:

Availability = MTTF ÷ (MTTF + MTTR)

Example: If a library management system operates for an average of 990 hours and requires 10 hours of repair or restoration time per cycle:

Availability = 990 ÷ (990 + 10) = 0.99 = 99%

The system is available for approximately 99% of the measured time.

Importance: High availability is important for systems that users need to access regularly, such as online banking and college portals.

Comparison of Software Reliability Metrics

Metric	What it measures	Desirable direction
ROCOF	Frequency of failures	Lower
MTTF	Average operating time before failure	Higher
MTTR	Average repair or restoration time	Lower
MTBF	Average time between failures	Higher
POFOD	Probability of failure when demanded	Lower
Availability	Proportion of time the system is operational	Higher

Important distinction: Reliability and availability are related but not identical. Reliability concerns failure-free operation over a specified period, whereas availability concerns whether the system is ready for use at a particular time or over a period.

3. ISO 9000 Certification

Definition

ISO 9000 is a family of international standards concerned with quality management systems (QMS). These standards provide principles and guidance for organizations to manage quality and improve their processes.

In practice, organizations are commonly certified against ISO 9001, the standard that specifies requirements for a quality management system. ISO 9000 itself provides fundamentals and vocabulary rather than the certification requirements.

Explanation

In software engineering, quality management involves establishing defined procedures for activities such as:

- Software development and testing.
- Documentation and version control.
- Requirements management.
- Handling defects and customer complaints.
- Reviewing and improving development processes.

An organization that implements a quality management system establishes consistent procedures for performing and checking its work.

ISO 9001 certification indicates that an independent certification body has assessed the organization's quality management system against the applicable requirements. It does not guarantee that every software product will be defect-free.

Need for ISO Certification

1. Consistent quality: It encourages the organization to follow documented processes.
2. Customer confidence: Customers gain greater confidence in the organization's quality management practices.
3. Improved documentation: Responsibilities, procedures, and records become clearer.
4. Process improvement: Organizations identify problems and improve their working methods.
5. Market opportunities: Some clients and tenders require or prefer certified suppliers.
6. Better accountability: Responsibilities for managing and checking work are more clearly defined.

How to Obtain ISO 9001 Certification

The following are the general steps for obtaining certification:

Step 1 — Understand the requirements: Study the applicable ISO 9001 requirements.

Step 2 — Establish a Quality Management System: Define processes, responsibilities, procedures, and necessary records.

Step 3 — Implement the system: Train employees and follow the documented processes.

Step 4 — Conduct an internal audit and management review: Check compliance and correct identified problems.

Step 5 — Apply for external certification: A certification body assesses the organization's QMS.

Step 6 — Correct nonconformities and maintain certification: Resolve audit findings and continue monitoring and improving the QMS.

Example

Suppose a software company develops a college management system. It defines procedures for collecting requirements, reviewing code, testing modules, recording defects, and approving releases. It then implements these procedures and undergoes an independent audit.

If the organization satisfies the applicable requirements, it may receive ISO 9001 certification for the defined scope of its quality management system.

Key takeaway: ISO 9001 certification focuses on the organization's quality management processes, not on guaranteeing that its software is perfect.

4. Software Maintenance

Definition

Software maintenance is the process of modifying and updating software after its delivery to correct faults, adapt it to changes in its environment, or improve its performance and maintainability.

Explanation

Software maintenance is necessary because software does not remain unchanged after delivery. Users may discover defects, operating systems may change, security requirements may evolve, and new features may be needed.

Maintenance may involve modifying program code, updating databases, changing documentation, testing the modified software, and releasing a new version.

Real-life analogy

Consider a college building after construction. Repairs may be needed when a pipe breaks, modifications may be required when new safety rules are introduced, and additional rooms may be built when the college expands.

Similarly, software requires correction, adaptation, and improvement throughout its useful life.

Example

A college management system has already been installed and is in use:

- A defect in the result calculation is corrected.
- The system is updated to work with a new operating system.
- A new attendance-reporting feature is added.

All three are examples of software maintenance, but they belong to different types.

5. Types of Software Maintenance

The syllabus identifies three main types: corrective, adaptive, and perfective maintenance.

5.1 Corrective Maintenance

Definition: Corrective maintenance involves modifying software to correct faults discovered after delivery.

It is performed when the software does not behave as expected.

Examples:

- Correcting an error in calculating students' examination results.
- Fixing a login failure.
- Correcting a program that stores the wrong book accession number.

Purpose: To restore correct operation by removing defects.

5.2 Adaptive Maintenance

Definition: Adaptive maintenance involves modifying software to keep it usable when its external environment changes.

These changes may include new operating systems, hardware, database versions, regulations, or external services.

Examples:

- Updating a college application to support a new operating system.
- Modifying a library system to work with an upgraded database.
- Updating a payment module to comply with a changed payment-service interface.

Purpose: To maintain compatibility with the changing environment.

5.3 Perfective Maintenance

Definition: Perfective maintenance involves modifying software to improve its performance, maintainability, usability, or functionality according to user needs.

It generally focuses on enhancements rather than correcting an existing defect.

Examples:

- Adding an online fee-payment facility to a college portal.
- Improving search speed in a library management system.
- Adding new reports requested by college administrators.
- Simplifying the user interface to make the application easier to use.

Purpose: To improve the software and meet changing user requirements.

Comparison of Maintenance Types

Basis	Corrective	Adaptive	Perfective
Main purpose	Correct faults	Adjust to environmental changes	Improve or enhance software
Main reason	Defects are discovered	External conditions change	User needs or improvement opportunities arise
Example	Fixing incorrect marks calculation	Supporting a new database version	Adding attendance analytics
Main focus	Correctness	Compatibility	Improvement

Memory aid: Remember C–A–P: Correct faults, Adapt to changes, Perfect or improve the software.

Note: Some textbooks also identify preventive maintenance as a fourth category. It involves changes intended to reduce the likelihood of future failures or make future maintenance easier. For the syllabus given here, focus primarily on the three specified types.

6. Estimation of Software Maintenance Cost

Definition

Software maintenance cost estimation is the process of predicting the effort, time, and money required to maintain software after its delivery.

Maintenance costs can be significant because software may need continuous correction, adaptation, testing, and enhancement throughout its useful life.

Factors Affecting Maintenance Cost

1. Software size and complexity: Large and complex programs usually require more effort to understand and modify.
2. Software age: Older software may use outdated technologies and have limited documentation.
3. Code quality: Poorly structured code is harder to understand, test, and modify.
4. Documentation quality: Complete and updated documentation helps developers understand the system.
5. Staff experience: Experienced developers can often identify and resolve problems more efficiently.
6. Frequency of changes: Software requiring frequent modifications generally costs more to maintain.
7. Testing requirements: Changes must be tested to ensure they do not introduce new defects.
8. Environmental changes: Operating-system, hardware, database, or regulatory changes may require additional work.

Methods of Estimating Maintenance Cost

1. Expert judgment

Experienced software engineers estimate the required effort based on their knowledge of the system, previous maintenance work, and the expected changes.

2. Historical data

The organization uses records from previous maintenance projects to estimate the cost of similar future work.

For example, if fixing similar defects previously required an average of 8 developer-hours, that information can help estimate the effort for comparable defects.

3. Estimation based on expected changes

The estimated amount of code to be modified or added is used to predict maintenance effort. The estimate may also account for testing, documentation, and integration.

4. Parametric models

Mathematical models use measurable factors such as software size, complexity, and historical effort. Some models extend software development cost estimation techniques to maintenance.

Annual Change Traffic (ACT)

Annual Change Traffic (ACT) is one useful measure for estimating maintenance effort. It represents the proportion of the software that is added or modified during one year.

ACT = KLOC added or modified during a year ÷ Total KLOC

Here, KLOC means thousand lines of code.

Example: Suppose a software system contains 100 KLOC, and 15 KLOC are modified or added during a year.

ACT = 15 ÷ 100 = 0.15 = 15%

This means that 15% of the system's size was modified or added during the year. ACT is an indicator of maintenance activity, not a direct measure of monetary cost.

Estimating Maintenance Cost

Maintenance Cost = Estimated Maintenance Effort × Cost per Unit of Effort

For example, if estimated maintenance effort is 200 person-hours and the cost is ₹500 per person-hour:

Cost = 200 × ₹500 = ₹1,00,000

Therefore, the estimated maintenance cost is ₹1,00,000, excluding any additional expenses not included in the hourly rate.

Steps in Maintenance Cost Estimation

1. Study the existing software and its documentation.
2. Identify the required corrections, adaptations, and enhancements.
3. Estimate the size and complexity of the changes.
4. Calculate the required developer, testing, and documentation effort.
5. Determine staff costs and other expenses.
6. Add an appropriate contingency allowance for uncertainty.
7. Review the estimate and update it as new information becomes available.

Key takeaway: Maintenance cost depends not only on the size of the software but also on its complexity, quality, documentation, frequency of changes, and the effort needed to test and release modifications.

7. Important Exam Points

Revise these facts before the examination.

Topic	Remember
Software reliability	Probability of failure-free operation for a specified time under specified conditions.
ROCOF	Number of failures divided by total operating time.
MTTF	Average operating time before failure.
MTTR	Average repair or restoration time.
MTBF	Average time between failures in a repairable system; measurement conventions matter.
POFOD	Probability that the system fails when a service is demanded.
Availability	Proportion of time the system is operational and ready for use.
ISO 9000 family	International standards for quality management; ISO 9001 specifies QMS requirements for certification.
Software maintenance	Modification of delivered software to correct faults, adapt to changes, or improve it.
Corrective maintenance	Corrects faults discovered after delivery.
Adaptive maintenance	Adjusts software to changes in its environment.
Perfective maintenance	Improves software or adds enhancements to meet user needs.
ACT	Annual proportion of software added or modified, commonly expressed as a percentage.
Maintenance cost	Depends on effort, staff rates, complexity, documentation, testing, and required changes.

8. Practice Questions for University Examinations

Short-answer questions

1. Define software reliability.
2. What is ROCOF?
3. Define MTTF and MTTR.
4. What is POFOD?
5. Define software availability.

6. What is ISO 9001 certification?
7. Define software maintenance.
8. What is Annual Change Traffic (ACT)?

Descriptive questions

1. Explain the important reliability metrics of software products with formulas and examples.
2. Explain the need for ISO 9001 certification and describe the steps for obtaining it.
3. Define software maintenance and explain corrective, adaptive, and perfective maintenance with examples.
4. Discuss the factors affecting software maintenance cost and explain the methods used to estimate it.
5. Differentiate between reliability and availability.
6. Differentiate between MTTF, MTTR, and MTBF.

Numerical practice

Question 1

A software application experiences 8 failures in 400 operating hours. Calculate ROCOF and MTTF.

Answer:

$$\textbf{ROCOF} = 8 \div 400 = 0.02$$

$$\textbf{MTTF} = 400 \div 8 = 50 \text{ hours}$$

Question 2

A system has an average uptime of 480 hours and downtime of 20 hours. Calculate its availability.

Answer:

$$\textbf{Availability} = 480 \div (480 + 20) = 480 \div 500 = 0.96$$

Availability = 96%.

Question 3

A backup system fails on 3 out of 1,500 demands. Calculate POFOD.

Answer:

$$\textbf{POFOD} = 3 \div 1{,}500 = 0.002$$

POFOD = 0.2%.

Final Revision Summary

Software reliability measures how dependably software performs without failure. ROCOF, MTTF, MTTR, MTBF, POFOD, and availability measure different aspects of failure behaviour and operational readiness. ISO 9001 certification evaluates an organization's quality management system, while software maintenance keeps delivered software correct, compatible, and useful through corrective, adaptive, and perfective changes. Maintenance cost estimation helps organizations plan the resources needed to keep software operational and improve it over time.