Unit 1

- what is Software Engineering

  Software engineering is a systematic, disciplined approach to developing software. It involves applying engineering principles to the entire software development lifecycle, from requirements gathering and design to coding, testing, deployment, and maintenance.[1] The goal is to create reliable, efficient, and maintainable software solutions that meet user needs and perform well. It's about building quality software using structured methods and best practices.
  IEEE standard Definition of Software Engineering :
  Software engineering is application of systematic, disciplined, quantifiable approach to development, operation and maintenance of software i.e. application of engineering to software.

- Importance of software engineering
  Software engineering is crucial for several reasons:

1. **Quality and Reliability:** It ensures software is built systematically, reducing errors, crashes, and security vulnerabilities. This means the software you use daily, from banking apps to car navigation, is dependable and safe.
2. **Cost-Effectiveness:** By using structured processes and best practices, software engineering helps manage projects efficiently, preventing costly rework, delays, and budget overruns throughout the software's lifespan.
3. **Scalability and Maintainability:** Well-engineered software is designed to grow and adapt as needs change. It's easier to update, add new features, and fix issues, extending its usefulness and reducing future expenses.
4. **Meeting User Needs:** It emphasizes understanding and fulfilling user requirements, leading to software that is intuitive, efficient, and truly solves real-world problems.
5. **Innovation and Progress:** Software engineering is the bedrock of technological advancement, enabling the creation of new applications, systems, and solutions that drive economic growth and improve various aspects of modern life.

- what are the software standards in software engineering

Software standards in software engineering are a set of rules, guidelines, and best practices that developers follow during the entire software development process. They are established by organizations like ISO and IEEE to ensure consistency, quality, and interoperability.
These standards cover various aspects, including:

- **Coding Standards:** Dictate consistent formatting, naming conventions, and code structure for readability and maintainability.
- **Documentation Standards:** Guide how software documentation (e.g., user manuals, technical specifications) should be written and organized.
- **Process Standards:** Define the stages and activities within the software development lifecycle (e.g., requirements, design, testing, deployment).
- **Security Standards:** Provide guidelines for secure coding practices and vulnerability management to protect against cyber threats.
- **Testing Standards:** Outline methods and procedures for effective software testing to identify and fix defects.

By adhering to these standards, teams can work more efficiently, produce higher-quality software, reduce errors, and ensure compatibility between different software components.

- what are the different Phases In The Software Development
  The Software Development Life Cycle (SDLC) typically involves several key phases, providing a structured approach to building software:
  1. **Requirements Gathering and Analysis:** This initial phase focuses on understanding what the software needs to do. Business analysts and stakeholders define the problem the software will solve, its features, and user expectations. This results in a detailed requirements document.
  2. **Design:** Based on the requirements, architects and designers create a blueprint for the software. This includes the overall system architecture, user interface (UI) and user experience (UX) design, database structure, and technical specifications.
  3. **Implementation/Coding:** This is where the actual software is built. Developers write code according to the design specifications, using chosen programming languages and tools.
  4. **Testing:** The software is rigorously tested to identify and fix defects. Various testing types, like unit testing, integration testing, system testing, and user acceptance testing (UAT), ensure the software meets requirements, performs correctly, and is reliable.
  5. **Deployment:** Once the software passes all tests, it's released to the end-users. This involves installing and configuring the software in a live environment, often with a pilot or beta launch.
  6. **Maintenance:** After deployment, ongoing support is provided. This includes fixing bugs, releasing updates, enhancing features, and adapting the software to new environments or user needs.

- what are the different Types of Software Application?
  Let's simplify those software types:
  1. **System Software:** This is the core software (like your operating system) that manages your computer's hardware, letting other programs run and handling complex tasks.
  2. **Real-time Software:** This software reacts instantly to events as they happen, crucial for systems like nuclear reactor controls or car autopilots, where timing is critical.
  3. **Business Software:** Applications that help companies run smoothly, managing things like billing, payroll, and inventory using large databases.
  4. **Engineering and Scientific Software:** Used by professionals for complex tasks like simulations, data analysis, and product design, often relying on historical data.
  5. **Embedded Software:** This is specialized software built directly into devices like microwaves or car cruise controls, controlling their specific functions.
  6. **Personal Computer Software:** Applications commonly used on personal computers for everyday tasks, like Microsoft Office for writing and spreadsheets.
- Software Engineering Code of Ethics
  Developing complex software involves many people, creating an **unbalanced relationship** where developers might know more than managers. This can lead to **unethical behavior**, so organizations like IEEE and ACM created a **Code of Ethics** for software engineers.
  This code emphasizes eight key principles:
  - **Public Interest:** Engineers must prioritize public safety and well-being.
  - **Client & Employer:** Act in their best interest, aligning with public good.
  - **Product Quality:** Ensure software meets high professional standards.
  - **Independent Judgment:** Maintain integrity in expert opinions.
  - **Ethical Management:** Leaders should promote ethical development practices.
  - **Professional Integrity:** Uphold the reputation of the software engineering profession.
  - **Supportive Colleagues:** Be fair and supportive of fellow engineers.
  - **Self-Improvement:** Continuously learn and promote ethical practices.

  Ultimately, because software impacts the public's health, safety, and welfare, upholding these ethical guidelines is paramount.

- Software Engineering Approach
  The **Software Engineering Approach** has four layers:
  - **Quality Focus:** Emphasizes continuous improvement to build reliable, high-standard software.
  - **Process:** The guiding framework for all development steps, from planning to quality assurance.
  - **Methods:** Specific techniques used for tasks like design, coding, and testing.
  - **Tools:** Automated software that supports the process and methods, improving efficiency.

unit 2

- what is Requirements Analysis

  Requirements Analysis is the crucial first step in building software. It's about figuring out exactly what the software needs to do and what problems it should solve. This involves talking to everyone who will use or be affected by the software (stakeholders) to understand their needs, expectations, and any specific conditions the software must meet.
  During this phase, analysts gather information, clarify any ambiguities or conflicts in the requirements, and then document them clearly. This documentation often includes details about specific features (functional requirements) and how well the software should perform (non-functional requirements, like speed or security). The goal is to create a clear blueprint so that everyone involved — from developers to testers — has a shared understanding of the final product, minimizing costly changes later on.

  Software requirements analysis is divided into five areas of efforts

  - Problem recognition
  - Evaluation and synthesis
  - Modeling
  - Specification
  - Review

- what is the Need for SRS

An **SRS (Software Requirements Specification)** is a crucial document that acts as the blueprint for software.

Its **need** is simple:

- **Clarity:** It ensures everyone agrees on what the software will do, preventing misunderstandings.
- **Cost Savings:** Catches problems early, making fixes much cheaper than later in development.
- **Guidance:** Provides a clear roadmap for developers and a basis for thorough testing.
- **Control:** Helps manage changes effectively, keeping the project on track.

Essentially, an SRS ensures successful software by fostering clear communication and minimizing risks.

- what is Requirement Process

  The **Requirement Process**, also known as Requirements Engineering, is a structured set of activities to understand and define what a software system needs to do. It typically involves:

  - **Requirement gathering(Elicitation):** Gathering information from stakeholders (users, clients, etc.) about their needs and expectations through interviews, surveys, or workshops.
  - **Analysis:** Examining the gathered information to identify, categorize, and clarify requirements, ensuring they are complete, consistent, and feasible.
  - **Specification:** Documenting these requirements clearly and unambiguously, often in a Software Requirements Specification (SRS) document.
  - **Validation:** Reviewing the documented requirements with stakeholders to ensure they accurately reflect their needs and are achievable.
  - **Management:** Handling changes to requirements throughout the project lifecycle, tracking their status, and ensuring everyone stays updated.

  This iterative process ensures the final software truly meets user demands and project goals.

- what is Requirement gathering (Elicitation)/ Fact Finding

  Requirement Gathering (Elicitation) / Fact Finding is the critical initial stage where software developers uncover and understand what users and stakeholders truly need from a new software system. It's like being a detective, actively digging for information.

  This involves direct communication through various techniques:

- **Interviews:** One-on-one conversations to get detailed insights.
- **Workshops:** Group sessions to brainstorm ideas and resolve conflicts.
- **Surveys/Questionnaires:** To gather feedback from a large number of people.
- **Observation:** Watching users perform their tasks to identify real-world needs and pain points.
- **Document Analysis:** Reviewing existing documents (like manuals or process flows) for insights.

The goal is to collect all relevant "facts" about the desired software's features, functions, and limitations to ensure the development team builds the right solution.

- what is functional Requirements

  Functional Requirements define *what* a software system *must do*. They describe the specific actions, behaviors, and features that users expect from the software.

  Think of them as the "verbs" of the system. Examples include:

  - "The system **shall allow** users to log in with a username and password."
  - "The system **shall display** product details when clicked."
  - "The system **shall process** payments securely."

  These requirements directly relate to the user's interaction with the software and are crucial for meeting business needs. They are the core functionalities that make the software useful.

- what is non-functional Requirements

  Non-functional Requirements (NFRs) define *how well* a software system performs its functions, rather than what it does. They are quality attributes and constraints that impact the user experience and system operation.

  Think of them as the "adjectives" of the system. Examples include:

  - **Performance:** "The webpage **shall load** within 2 seconds."
  - **Security:** "User data **shall be encrypted** using AES-256."
  - **Usability:** "The interface **shall be intuitive** for first-time users."
  - **Reliability:** "The system **shall have** 99.9% uptime."

  NFRs are crucial for ensuring the software is not just functional, but also robust, efficient, secure, and user-friendly.

- what is Problem analysis

  Problem analysis in software engineering is the process of thoroughly understanding a specific issue or challenge that a software system needs to address. It's about going beyond the obvious symptoms to uncover the root causes of the problem.

This involves:

- Clearly defining what the problem is.
- Identifying who is affected and how (stakeholders).
- Understanding the current inefficiencies or pain points.
- Determining the boundaries and constraints of the problem.

The goal is to gain a deep, shared understanding of the actual problem before jumping into solutions. This ensures that the developed software truly solves the right issue, preventing wasted effort and resources.

- what is Structural analysis

In software engineering, **Structural Analysis** is a method used to understand a system by breaking it down into its logical components and showing how data flows between them. It focuses on the "what" of a system rather than the "how" it's physically implemented.

Key aspects include:

- **Decomposition:** Dividing complex processes into smaller, more manageable parts.
- **Graphical Tools:** Using diagrams like Data Flow Diagrams (DFDs) to visualize data movement and transformations.
- **Data Dictionary:** Defining all data elements consistently.

The goal is to create a clear, logical model of the system that is easy to understand, helping analysts and developers build the correct software based on the identified requirements.

- what are Prototyping methods and tools

Prototyping methods are different ways to create these early models. Common ones include:

- **Throwaway Prototyping:** A quick, basic prototype built just to understand requirements, then discarded.
- **Evolutionary Prototyping:** The prototype is gradually refined and built upon, eventually evolving into the final product.
- **Incremental Prototyping:** The system is broken into smaller parts, and separate prototypes are built and combined.

Tools provide the software support for creating prototypes. These range from simple to sophisticated:

- **Low-fidelity tools:** Like **Balsamiq** or **paper sketches**, focusing on basic layout and flow.
- **High-fidelity tools:** Like **Figma**, **Adobe XD**, or **Sketch**, which create interactive, visually detailed prototypes resembling the final product.

These methods and tools help teams efficiently gather feedback and validate designs before full development.

- Characteristics of SRS

  A **Software Requirements Specification (SRS)** is a detailed document created after gathering and analyzing what the software needs. It's written in plain language, often with diagrams, and covers all requirements, even obvious ones, for any project size.

  A good SRS has key characteristics:

1. **Clarity:** Everyone (developers, users, testers) interprets it the same way, enhanced by clear language and visuals.
2. **Completeness:** It includes all necessary details, covers all problems and goals, and clearly defines the software's scope.
3. **Traceability:** Each requirement can be tracked from its source to the final code and tests, helping understand impacts of changes or debug errors.
4. **Feasibility:** The requirements are technically and operationally realistic given technology, budget, and environment.
5. **Testability:** Requirements are written so that they can be effectively tested to confirm the software meets them.
6. **Correctness:** Requirements align with business goals and efficiently solve problems.
7. **Consistency:** There are no contradictions, using standard terms and rules throughout.

8. **Modifiability:** It's structured so that changes can be made easily and tracked without creating inconsistencies.

- 5 advantages & disadvantages of srs

  Here are 5 advantages and disadvantages of an SRS:

  Advantages of SRS:

1. **Clear Communication:** It establishes a common understanding between clients, developers, and testers, minimizing misunderstandings and misinterpretations.
2. **Reduces Rework:** By defining requirements upfront, it helps catch errors and inconsistencies early, which is significantly cheaper than fixing them during coding or testing.
3. **Better Project Planning:** Provides a solid basis for estimating project scope, resources, costs, and timelines more accurately.
4. **Guides Development & Testing:** Serves as a detailed blueprint for designers and developers, and a benchmark for testers to verify the software's functionality.
5. **Manages Change:** Offers a stable baseline against which all proposed changes can be evaluated, helping to control scope creep and project deviations.

  Disadvantages of SRS:

1. **Time-Consuming:** Creating a detailed SRS can be a lengthy process, especially for complex systems, potentially delaying the start of coding.
2. **Rigidity:** A very detailed SRS can sometimes be inflexible, making it difficult to adapt to evolving requirements or unforeseen changes during development.
3. **Requires Expertise:** Developing a high-quality SRS demands skilled analysts who can effectively elicit, analyze, and document requirements.
4. **Perceived Overhead:** Some stakeholders might view it as bureaucratic paperwork, especially in fast-paced or agile environments where rapid iteration is preferred.
5. **Information Overload:** For smaller or less complex projects, an overly detailed SRS might contain unnecessary information, leading to "analysis paralysis" and reduced efficiency.

- Importance of feasibility study

  The image you sent seems to be about the "validity of an instrument in research methodology" and doesn't directly relate to software engineering concepts.

  However, if you're asking about **Feasibility Study** in software engineering (which often includes factors like technical, economic, legal, operational, and scheduling – sometimes abbreviated as TELOS or TELOS + others), I can explain that.

  A **Feasibility Study** in software engineering determines if a proposed software project is practical and viable. It involves assessing various factors to decide if the project should proceed.

  Key aspects include:

- **Technical Feasibility:** Can we build it with existing technology and our team's skills? Do we have the necessary hardware and software?
- **Economic Feasibility:** Is the project financially worthwhile? Do the benefits outweigh the costs of development and maintenance?
- **Operational Feasibility:** Will the new system work smoothly within the organization's existing processes and culture? Will users accept it?
- **Legal Feasibility:** Does the project comply with all relevant laws, regulations, and intellectual property rights?
- **Scheduling Feasibility:** Can the project be completed within a realistic timeframe?

  This study helps identify risks and opportunities early, guiding decision-makers on whether to invest time and resources, or to abandon or adjust the project.

- what is dfd

  A **Data Flow Diagram (DFD)** is a visual way to show how information moves through a system. It doesn't focus on *how* data is processed (the technology or specific steps), but rather *what* data goes in, *what* comes out, and *where* it goes.

  It uses a few simple symbols:
  - **Processes (circles or ovals):** Where data is transformed or changed.
  - **Data Stores (parallel lines or open rectangles):** Where data is held or stored (like a database or file).
  - **External Entities (rectangles):** Sources or destinations of data outside the system (like customers or other systems).
  - **Data Flows (arrows):** The movement of data between processes, data stores, and external entities.

  DFDs help users and developers understand the system's logic and data movement without getting bogged down in technical details, making complex systems easier to grasp.
- 5 advantages & disadvantages

  Here are 5 advantages and 5 disadvantages of using Data Flow Diagrams (DFDs):

  Advantages of DFDs:

1. **Easy to Understand:** They are highly visual and use simple symbols, making them understandable even to non-technical stakeholders, promoting better communication.
2. **Focus on Data Flow:** They clearly illustrate how data moves through a system, highlighting inputs, outputs, and transformations without getting into implementation details.
3. **System Overview:** Provide a high-level overview of the entire system or specific processes, helping to identify major components and their interactions.
4. **Identify Missing Information:** By visualizing data flow, DFDs can reveal gaps or inconsistencies in requirements or processes.
5. **Aid in Design:** They serve as a good starting point for system design, helping to break down complex systems into manageable modules.

   Disadvantages of DFDs:

1. **No Control Information:** DFDs don't show control flow, timing, or decision logic (e.g., "if-then-else" statements), which are crucial for understanding system behavior.
2. **Can Become Complex:** For very large or intricate systems, DFDs can become cluttered and difficult to read, especially at lower levels of detail.
3. **Doesn't Show Hardware Details:** They are purely logical and don't illustrate physical aspects like hardware components, networks, or specific technologies used.
4. **Limited for Real-time Systems:** Because they lack control flow, DFDs are less effective for modeling real-time systems where timing and event sequencing are critical.
5. **Requires Iteration:** Developing a complete and accurate set of DFDs often requires multiple iterations and revisions as understanding of the system evolves.

- what is erd
  An **Entity-Relationship Diagram (ERD)** is a visual tool used to design databases. It maps out the structure of a database by showing how different pieces of information (entities) relate to each other.
  It uses three main components:
    1. **Entities (rectangles):** These represent real-world objects or concepts about which data is stored, like "Customer," "Product," or "Order."
    2. **Attributes (ovals/ellipses):** These are the characteristics or properties of an entity, such as a "Customer ID," "Product Name," or "Order Date."
    3. **Relationships (diamonds):** These show how entities are connected or interact with each other, like a "Customer places an Order" or a "Product belongs to a Category."

  ERDs are crucial for database designers as they provide a clear blueprint, ensuring that the database is well-organized, consistent, and efficiently stores and retrieves data.
- give 5 advantages & disadvantages
  Here are 5 advantages and 5 disadvantages of using Entity-Relationship Diagrams (ERDs):
  Advantages of ERDs:

1. **Clear Database Design:** ERDs provide a straightforward visual representation of the database structure, making it easy to understand the relationships between different data elements.
2. **Improved Communication:** They serve as an excellent communication tool between database designers, developers, and stakeholders, ensuring everyone has a shared understanding of the data model.
3. **Reduces Redundancy:** By clearly defining entities and relationships, ERDs help in identifying and minimizing redundant data, leading to a more efficient and consistent database.
4. **Foundation for Implementation:** An ERD acts as a direct blueprint for creating the actual database schema (tables, columns, keys), speeding up the implementation phase.
5. **Easy to Modify:** If business requirements change, ERDs can be easily updated to reflect new entities, attributes, or relationships, making the database design adaptable.
   Disadvantages of ERDs:
1. **Complexity for Large Systems:** For very large and complex databases with numerous entities and relationships, ERDs can become overwhelming and difficult to read or manage.
2. **Doesn't Show Operations:** ERDs primarily focus on data structure and relationships; they don't directly show how data is processed, manipulated, or the logic behind operations.

3. **Limited to Relational Databases:** While adaptable, ERDs are most effective for designing relational databases and may not fully capture the nuances of other database types (e.g., NoSQL).
4. **Abstraction Level:** They represent a logical view of the data, which means they don't show physical implementation details like data types, storage methods, or performance considerations.
5. **Requires Experience:** Creating a well-structured and optimized ERD requires a good understanding of database design principles and often some experience in data modeling.

- what are Data Model in erd
  In an ERD, **Data Models** are the conceptual frameworks that define how data is organized and related within a database. Essentially, they represent the structure of the data.
  There are different levels of data models:
    1. **Conceptual Data Model:** This is the highest-level view, showing the main entities and their relationships, without focusing on technical details. It represents the business view of the data.
    2. **Logical Data Model:** This adds more detail, defining attributes for each entity and specifying primary/foreign keys, but still remains independent of a specific database system.
    3. **Physical Data Model:** This is the most detailed, showing how the data is actually stored in a specific database system, including data types, table names, and constraints.

  These models guide the transformation of real-world concepts into a structured database.
- what are Entity Types in erd
  In an ERD, **Entity Types** represent a class or category of real-world objects or concepts about which you want to store data. Think of them as the "nouns" in your database.
  For example:

1. "Customer" is an entity type, representing all individual customers.
2. "Product" is an entity type, representing all different products.
3. "Order" is an entity type, representing all orders placed.
   Each entity type has instances (individual occurrences), like a specific customer named "John Doe" or a particular product "Laptop X." In an ERD, entity types are usually drawn as **rectangles**. They form the fundamental building blocks around which the entire database structure is organized.

- what are attributes in erd
  In an ERD, **Attributes** are the characteristics or properties that describe an Entity Type. Think of them as the specific pieces of information you want to store about each entity.
  For example, if "Customer" is an entity type, its attributes might include:

1. Customer ID (a unique identifier)
2. Customer Name
3. Address
4. Email
5. Phone Number

Attributes are typically represented as **ovals or ellipses** connected to their respective entity type. They define the data points that make up each instance of an entity, providing the granular detail for your database.

- classification of relationships based on:No of Entity Types,Cardinality,Optionality;in erd
  In ERDs, relationships between entities are classified in several ways:

1. **Based on No. of Entity Types (Degree):**
    - **Unary (Recursive):** A relationship involving a single entity type, where instances of that entity relate to other instances of the *same* entity (e.g., an "Employee manages another Employee").
    - **Binary:** A relationship involving two different entity types (e.g., "Customer places Order"). This is the most common type.
    - **Ternary:** A relationship involving three different entity types (e.g., "Student registers for Course with Professor").
    - **N-ary:** A general term for a relationship involving *n* number of entity types (where n > 2). So, ternary is a type of N-ary relationship.
2. **Based on Cardinality:** This defines *how many* instances of one entity relate to instances of another.
    - **One-to-One (1:1):** One instance of Entity A relates to exactly one instance of Entity B (e.g., "Employee is assigned one Parking Space").
    - **One-to-Many (1:M):** One instance of Entity A relates to many instances of Entity B, but an instance of B relates to only one of A (e.g., "Department has many Employees").
    - **Many-to-Many (M:N):** Many instances of Entity A relate to many instances of Entity B (e.g., "Student enrolls in many Courses, and a Course has many Students").
3. **Based on Optionality (Participation/Modality):** This indicates whether an entity *must* participate in a relationship or *can optionally* participate.
    - **Mandatory (or Total):** An instance of an entity *must* participate in the relationship (e.g., "Every Order must have a Customer"). Represented by a double line.
    - **Optional (or Partial):** An instance of an entity *may or may not* participate in the relationship (e.g., "An Employee may be assigned a Project"). Represented by a single line.

- what is Data Dictionary
  A **Data Dictionary** is a centralized repository of information about data, essentially a "metadata" store. It defines and describes every piece of data used in a system or database.
  For each data element (like an attribute in an ERD or a field in a database), a data dictionary typically includes:

1. **Name:** The official name of the data element.
2. **Description:** A clear explanation of what it represents.
3. **Data Type:** Whether it's text, numbers, dates, etc.
4. **Format/Length:** How the data is structured (e.g., maximum characters).
5. **Validation Rules:** Any constraints or rules for its values.
6. **Source:** Where the data originates.
7. **Relationships:** How it connects to other data elements.

It ensures consistency, clarity, and control over data, helping developers and users understand and manage information effectively.

Unit 3

- what are the Importance of Design
  In software engineering, **Design** is the crucial phase where the "how" of the software is determined. It's like creating detailed architectural blueprints before building a house.
  Its importance lies in:

1. **Translating Requirements:** It transforms abstract user needs (from the requirements phase) into a concrete plan for developers to follow.
2. **Ensuring Quality:** A good design leads to software that is robust, efficient, secure, and easy to maintain and extend in the future.
3. **Minimizing Errors:** Identifying potential problems and complexities at this stage is much cheaper and easier than fixing them during coding or after deployment.
4. **Guiding Implementation:** Provides clear instructions for coders, reducing ambiguities and speeding up the development process.
5. **Facilitating Collaboration:** Offers a shared vision that helps different team members work together effectively.

Preliminary Design

- **What is Preliminary Design?**
  The **preliminary design** phase is the early stage in software development where the planning and structure of the software begin. It is like creating a rough sketch before building a house.
    - **Preliminary Design**, also known as High-Level Design or Architectural Design, is the initial stage of software design. It focuses on creating the overall structure and components of the software system without getting into minute details.

  Think of it as drawing the main rooms and walls of a house:

1. It defines the major modules or subsystems.
2. It outlines how these modules will interact with each other.
3. It identifies key data structures and external interfaces.
4. It establishes the system's architecture (e.g., client-server, layered).
   The goal is to establish a solid foundation and ensure that the chosen architecture can meet the system's functional and non-functional requirements before delving into detailed coding.

Main Activities in Preliminary Design:

- **Create an algorithm**: Write clear step-by-step instructions to solve the problem.
- **Define system modes/states**: Decide how the system behaves in different situations.
- **Design data and control flow**: Plan how data moves in the system.
- **Set data structures**: Choose how data is organized (like arrays, lists).
- **Allocate memory**: Plan how much memory the system needs.
- **User interface design**: Sketch how the software will look to users.

- **Make a timeline**: Estimate how long tasks will take.
- **Review the design**: Check everything to ensure the design is correct.

👷 **Activities of the Development Team:**

- **Draw diagrams**: Make visual representations of how the system works.
- **Evaluate design options**: Choose the best design for speed, reliability, and ease of use.
- **Write prologs/PDL**: Create simple code-like structures to explain how the program works internally.
- **Document decisions**: Write down why certain designs were chosen.
- **Fix design issues (RIDs)**: Track and fix any problems found during reviews.

🤵 **Activities of the Management Team:**

- **Check schedules and staffing**: Make sure the right people are doing the right tasks on time.
- **Handle changes in requirements**: Keep track of any changes from the client and update the plan accordingly.
- **Monitor quality**: Attend design reviews and walkthroughs to ensure quality.
- **Prepare for next phase**: Get ready to move into detailed design by organizing everything clearly.

📋 **Activities of the Requirements Definition Team:**

- **Clear up unclear requirements**: Talk to developers if any requirement is confusing or missing.
- **Join design reviews**: Help developers understand the requirements better during meetings.
- **Update documents**: Make changes in the requirement documents when needed.

- what are the Design Principles?

  Design Principles are fundamental rules or guidelines that help software engineers create good, effective software designs. They are like a set of best practices to follow during the design phase.

  These principles aim to achieve qualities like:

1. **Modularity:** Breaking down a system into smaller, independent, and interchangeable parts.
2. **Cohesion:** Ensuring that the elements within a module work closely together to achieve a single purpose.
3. **Coupling:** Minimizing the dependencies between different modules to reduce ripple effects from changes.
4. **Abstraction:** Hiding complex details and showing only the essential information to the user.
5. **Encapsulation:** Bundling data and the methods that operate on the data into a single unit, and restricting direct access to some of the component's.

   Following these principles leads to software that is easier to understand, build, test, and maintain, ultimately making it more robust and adaptable.