UNIT–I : BASICS OF SOFTWARE ENGINEERING

Software and Software Engineering – simple definition, need, SDLC Phases: Requirements, Design, Coding, Testing, Deployment, Maintenance .Software Myths: few examples of management, customer, developer myths.Very small intro: how AI can help in SDLC

Software and Software Engineering :

Software is a set of instructions, data, and documentation that tells a computer or other device what to do and how to do it. It is a logical, intangible component of a computer system, distinct from the physical hardware.

Software Engineering is the application of a systematic, disciplined, and quantifiable approach to the design, development, operation, and maintenance of software. It is an engineering discipline focused on creating reliable, efficient, and scalable software products to solve real-world problems.

Need for Software Engineering:

Why Do We Need Software Engineering?

We need Software Engineering because **simple coding is not enough** for today's software. Modern software is:

- **Big**
- **Complex**
- Used by **many users**
- Must be **reliable** and **safe**

So we need **proper methods, tools, and processes** → that is Software Engineering.

Main Needs / Reasons for Software Engineering.

1) To handle complexity

- Modern software = very large (millions of lines of code).
- One person cannot manage everything just by memory.
- SE gives **structured methods** (modules, layers, design, architecture) to **reduce complexity**.

Example:
Banking system, railway reservation, hospital management – all have many modules (accounts, payments, reports…). SE helps to **organize** this.

2) To improve quality of software

- We want software that is:
  - **Correct** (fewer errors)
  - **Reliable** (works properly for a long time)
  - **Secure** (protects data)
  - **User-friendly**

Software Engineering gives:

- **Standards**, **design principles**, **testing methods**, **reviews**
  → to ensure **good quality**.

Example:
ATM software must **always** give correct balance and not crash. SE practices make it safe and reliable.

3) To reduce cost and save time

- If we directly jump into coding without SE,
  - Many **errors** later
  - We must **rewrite** code
  - This increases **cost** and **time**

Using SE:

- We do **proper planning, design, and testing**
- This reduces **rework**
- Total **cost and time** become less in the long run.

Example:
Spending 2 days to design properly can save many days of rework later.

4) To deliver on time (schedule control)

- Software projects have **deadlines** (exam portals before exam, result website before result date, etc.).
- Without SE, project may **get delayed**.

SE uses:

- **Project planning**
- **Scheduling**
- **Tracking progress**

This helps to **complete the project on time**.

Example:
College ERP system must be ready before the academic year starts. SE helps in proper scheduling.

5) To make software maintainable

- After delivery, software needs:
  - **Bug fixes**
  - **New features**
  - **Changes** due to new rules, laws, or user needs

SE practices (clean code, modular design, documentation) make software **easy to modify** later.

Example:
If exam pattern changes, college exam portal must be updated. Good SE makes this update easier.

6) To support team work

- Most software is built by a **team**, not a single person.
- SE defines:
  - **Roles** (analyst, designer, tester, etc.)
  - **Processes** (how to work together)
  - **Version control**, **documents**

This helps team members to **coordinate** and work smoothly.

Example:
One developer handles login module, another handles payment module. SE practices help them integrate their work correctly.

7) To manage changing requirements

- Customers often change their mind:
  - "Add this report"
  - "Change this screen"
  - "Add mobile app also"

Without SE, changes cause **big problems**.
With SE:

- We have **requirement management**, **change control**, **proper documentation** → changes are handled **systematically**.

Example:
E-commerce website may later add wallet, coupons, offers. SE helps to add such features without breaking old ones.

8) To ensure reliability and safety

For some systems (healthcare, aviation, banking, defense):

- **Failures can be dangerous** or very costly.

SE includes:

- Careful **design**
- Strict **testing**
- **Quality standards**

to make software **safe and reliable**.

Example:
Software that controls a heart monitor in a hospital must not fail. SE helps achieve that reliability.

9) To provide proper documentation

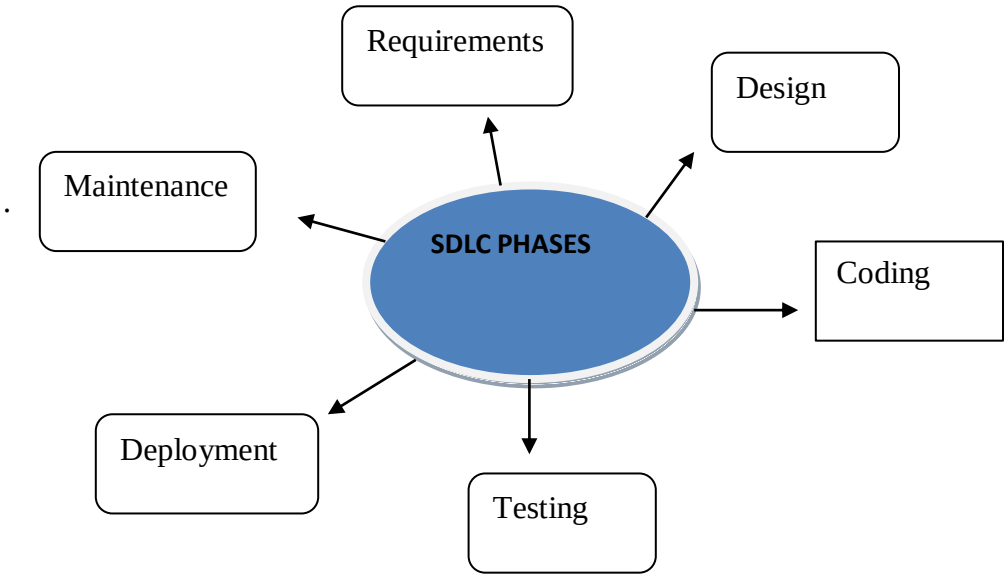
- Documentation = SRS, design docs, user manuals, etc.
- Without SE, code is there but **no explanations**.
- With SE, we create documents that help:
    - New developers
    - Support team
    - Users

Example:
If a new programmer joins the project, documents help them understand the system quickly.

SDLC Phases:

SDLC Phases: Requirements, Design, Coding, Testing, Deployment, Maintenance

.



The Software Development Life Cycle (SDLC) is a structured process used to design, develop, and test high-quality software. The key phases you mentioned are:

Requirements (Requirement Analysis)

Definition

The **Requirements phase** is the phase where we **understand, collect and document** what the customer or user wants from the software.

Explanation

- In this phase, the **development team** talks with:
  - Customers
  - End users
  - Stakeholders (management, domain experts)
- They identify:
  - **What functions** the software should perform
  - **Who will use** the system
  - **What data** will be stored and processed
  - **Constraints** (budget, time, technology, legal rules)
- All these details are written in a document called **SRS – Software Requirements Specification**.

Types of Requirements (you can mention):

- **Functional Requirements** – what the system should do
  - Example: "System shall allow student to login."
- **Non-functional Requirements** – quality constraints
  - Example: "System must respond within 2 seconds", "Data must be secure"

Example

For an **Online Result Portal**:

- Functional:
  - Student can login and view marks
  - Admin can upload marks
  - Student can download grade sheet
- Non-functional:
  - System should handle 500 students at a time
  - Only authenticated users can see results

Design

Definition

The **Design phase** converts the **requirements** into a **technical blueprint** of the system.
It describes **how the system will be built**.

Explanation

- It focuses on **structure** and **architecture** of the system.
- Main tasks:
    - Decide **overall architecture** (e.g., client-server, layered architecture)
    - Identify **modules / components**
    - Design **database** (tables, relationships)
    - Design **user interfaces** (forms, screens)
    - Design **data flow** and **control flow**

Types of Design

1. **High-Level Design (HLD)**
    - Describes the **overall structure**
    - Major modules and their relationships
    - Example: Login Module, Student Module, Result Module, Admin Module
2. **Low-Level Design (LLD)**
    - Describes **detailed logic** for each module
    - Class diagrams, function details, algorithms

Simple Design Diagram (Block view)

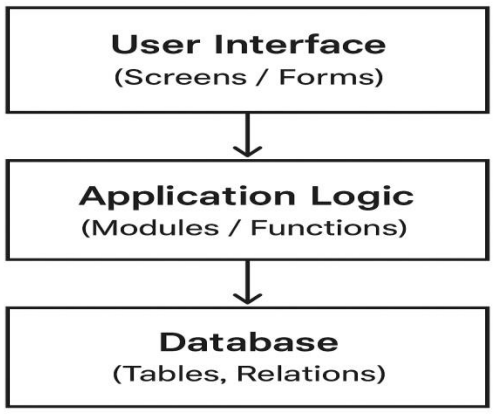


Figure: Simple Software Design (3-tier)

Coding / Implementation

Definition

The **Coding phase** is where the **design is translated into actual programs** using a programming language.

Explanation

- Developers write the code based on the **design documents**.
- Each module is implemented and then integrated.
- Good practices in this phase:
    - Follow **coding standards** (naming rules, formatting)
    - Add **comments** for better understanding
    - Use **version control** (like Git) for managing changes

What happens here?

- Write code for:
    - User interfaces
    - Business logic
    - Database access
    - Error handling
- Ensure the code is:
    - **Readable**
    - **Modular**
    - **Efficient**
    - **Maintainable**

Example

For **Student Login Functionality**:

- Code to get username/password
- Code to check details from database
- Code to show “Login successful” or “Invalid credentials”

Testing

Definition

The **Testing phase** is used to **check the developed software** to find and fix errors (bugs) and to verify that it meets the specified requirements.

Explanation

- Testing is done **after coding**, but **planning for testing** starts earlier.
- The main aim is to:
  - Ensure the software is **working correctly**
  - Ensure it satisfies the **SRS requirements**
  - Find defects before users face them

Types of Testing (brief points you can list)

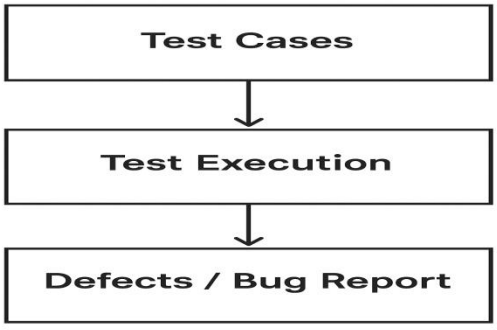
- **Unit Testing** – Test individual functions or modules.
- **Integration Testing** – Test modules together to see if they work as a group.
- **System Testing** – Test the complete system as a whole.
- **Acceptance Testing** – Customer checks if the software meets their needs.

Example

For an **Online Exam System**:

- Test login with correct and incorrect passwords
- Test if questions load properly
- Test timer working correctly
- Test result calculation and display

Simple Testing Diagram



Deployment

Definition

The **Deployment phase** is where the fully tested software is **installed, configured, and released** for actual use by the end users.

Explanation

- Software is moved from **development environment** to **production environment**.
- Steps may include:
  - Installing on servers or user machines
  - Configuring network, database, and security
  - Migrating old data (if required)
  - Training users or giving user manuals

Deployment Strategies (optional extra points)

- **Big Bang Deployment** – Entire system is deployed at once.
- **Phased Deployment** – System is deployed in parts/modules.
- **Pilot Deployment** – First deploy to a small group, then roll out to all users.

Example

For **College ERP System**:

- Install the ERP on the college server
- Create accounts for students and staff
- Share portal link and login instructions

Maintenance

Definition

The **Maintenance phase** is the phase where the software is **maintained and updated after deployment** to fix bugs, add new features, or adapt to changes.

Explanation

- Software is not “finished” after deployment.
- During real-time use, users may:
  - Find **new bugs**
  - Request **new reports or features**
  - Need support for **new hardware / OS / rules**

Types of Maintenance

1. **Corrective Maintenance** – Fixing bugs and errors discovered during use.
2. **Adaptive Maintenance** – Modifying software to work in new environments (new OS, new DB, new devices).
3. **Perfective Maintenance** – Improving performance or adding new features to enhance the system.
4. **Preventive Maintenance** – Code cleanup and refactoring to prevent future problems.

Example

For a **Shopping Website**:

- Fix an error in discount calculation → **Corrective**
- Update code to support new UPI payment → **Adaptive**
- Add product recommendation feature → **Perfective**
- Refactor code to reduce complexity → **Preventive**

Software Myths: few examples of management, customer, developer myths

What are Software Myths?

- **Software myths** = Wrong beliefs / false ideas about software and software development.
- Many people (managers, customers, even developers) **believe these are true**, but actually **they are not**.
- Myths create **confusion, unrealistic expectations, stress, and project failure**.

We divide them into:

1. **Management myths**
2. **Customer myths**
3. **Developer myths**

1. Management Myths (Myths believed by managers)

Myth 1: "If we are behind schedule, just add more programmers."

- **Meaning**: Manager thinks: *"Project late? No problem, I will add more people and we will finish fast."*
- **Why it is wrong**:
  - New people **don't know the project** → Old team must **teach them** → This takes **time**.
  - More people = **more communication**, more confusion.
  - In the beginning, **project becomes even slower**.
- **Simple Example**:
  - One person is writing an exam. Can we add two more people to write the same answer sheet faster? **No**, only one person can write on that paper at a time.
- **Reality**:
  - Sometimes, adding people **late** in the project **delays** more instead of helping.

Myth 2: "Once we give requirements to the team, they will do everything."

- **Meaning**: Manager thinks: *"I told them the work one time. Now they will handle everything. No need to discuss again."*
- **Why it is wrong**:
  - Requirements **change** with time.
  - Team may **not fully understand** some requirements.
  - Without continuous communication, **misunderstandings happen**.
- **Simple Example**:
  - If a teacher explains project topic only once in June and never talks again, students may **do something different** from what teacher really wanted.
- **Reality**:
  - **Regular meetings** and feedback are needed to keep project correct.

Myth 3: "We must start coding immediately to show progress."

- **Meaning**: Manager thinks: *"If they are writing code, project is moving. Design and planning are a waste of time."*

- **Why it is wrong**:
  - Without proper **requirements** and **design**, code may be **wrong or messy**.
  - Later, we must **rewrite** many parts → **More time and cost**.
- **Simple Example**:
  - Building a house without a plan: workers start building rooms anywhere → Later they break walls and rebuild. Big waste.
- **Reality**:
  - Good **planning + design** saves time and cost in long term.

2. Customer Myths (Myths believed by clients/users)

Myth 1: "Software requirements can be changed anytime, no problem."

- **Meaning**: Customer thinks: *"I can ask for any new feature anytime, it is just software, they can easily add."*
- **Why it is wrong**:
  - Every change means **extra work**, more **time** and **money**.
  - Big changes late in project can **break** existing design and code.
- **Simple Example**:
  - You order a cake with vanilla flavor. When the cake is almost ready, you say: "Make it chocolate and add fruits also." The baker needs **more time and ingredients**.
- **Reality**:
  - Changes are possible, but they must be **controlled**, **planned**, and may require **extra cost and time**.

[

Myth 2: "A general idea is enough. Details can come later."

- **Meaning**: Customer says: *"Just make a simple app like Facebook for our college. Details we will tell later."*
- **Why it is wrong**:
  - Without **clear details**, developers **guess** what the customer wants.
  - This leads to **wrong features**, rework, and misunderstandings.
- **Simple Example**:
  - Telling a tailor: "Stitch some dress for me." Without size, style, length → The dress may **not fit** you.
- **Reality**:
  - Customer must give **clear, specific requirements** (what screens, what data, what reports, etc.) as much as possible.

Myth 3: "Once the software is delivered, the work is over."

- **Meaning**: Customer thinks: *"After delivery, no more cost. Software will run forever."*
- **Why it is wrong**:
  - Software needs:
    - **Bug fixing**
    - **Updates**
    - **New features**
    - **Security patches**
  - This is called **maintenance**, and it also costs time and money.
- **Simple Example**:
  - Buying a bike is not the end. You still need **service, repairs, petrol** regularly.
- **Reality**:
  - Software project includes **development + maintenance**.

3. Developer Myths (Myths believed by programmers)

Myth 1: "If the program works, it is good software."

- **Meaning**: Developer thinks: *"Code is running without crash, so my job is done."*
- **Why it is wrong**:
  - Good software needs:
    - **Readability**
    - **Maintainability**
    - **Performance**
    - **Security**
  - Just working once is not enough if:
    - Code is **hard to understand**
    - Difficult to **modify**
    - **Slow** or **insecure**
- **Simple Example**:
  - A car that only starts one time is not a "good car". It must be safe, efficient, easy to repair.
- **Reality**:
  - Developers must write **clean, well-structured, documented** code.

Myth 2: "Once we finish coding, testing is not very important."

- **Meaning**: Developer thinks: *"I wrote it carefully, so there will be no bugs. Testing is just formality."*
- **Why it is wrong**:

    - Humans **always make mistakes**.
    - Many bugs appear only with **real data** and **edge cases**.
- **Simple Example**:
    - Even a careful student can make mistakes in exam. That's why teachers **check** the answer sheets.
- **Reality**:
    - **Testing is essential** to find and fix defects before users face them.

Myth 3: "We can write documentation later, now only code is important."

- **Meaning**: Developer thinks: *"I will first code everything. Documentation can be done at the end if we have time."*
- **Why it is wrong**:
    - At the end of project, there is **time pressure**, so documentation is **ignored**.
    - Future developers **struggle** to understand the system.
- **Simple Example**:
    - Making a big project file but not writing index or explanation. Later, even you **forget** what each part means.
- **Reality**:
    - Documentation (comments, design docs, user manuals) should be written **along with development**, not only at the end.

Very Small Intro: How AI Can Help in SDLC

AI can support **almost every phase** of the Software Development Life Cycle and make work **faster, smarter, and less error-prone**.

1. Requirements Phase

- AI tools can **analyze emails, documents, and chats** to auto-suggest requirements.
- They can **highlight missing or conflicting requirements**.

Example: An AI tool reads customer emails and suggests: "Login with OTP" as a new requirement.

2. Design Phase

- AI can **recommend architectures or design patterns** based on similar past projects.
- It can **auto-generate UML diagrams** or database designs from text descriptions.

Example: You describe "library management system," and AI suggests modules like *Book, Student, Issue/Return*.

3. Coding Phase

- **AI code completion** helps developers write code faster (auto-suggestions, fix syntax).
- AI can **generate code** (forms, validation).
- It can also **find bugs or security issues** in code while typing.

Example: While writing a loop, AI warns: "This may cause an infinite loop" and suggests a fix.

4. Testing Phase

- AI can **generate test cases automatically** from requirements or code.
- It can **predict risky areas** where bugs are likely.
- AI-based tools run tests and **prioritize which bugs are most critical**.

Example: AI creates boundary value tests for a "marks" field: –1, 0, 39, 40, 100, 101.

5. Deployment Phase

- AI helps in **auto-scaling** applications (adding servers when load increases).
- It can **predict failures** in production and alert early.

Example: AI sees unusual CPU usage and warns that the server may crash soon.

6. Maintenance Phase

- AI analyzes **logs and user feedback** to detect recurring problems.
- It can **suggest code changes** to improve performance or fix defects.
- Chatbots can give **24×7 support** to users.

Example: AI chatbot answers common "login problem" queries instead of a human support person.

UNIT–II : SOFTWARE PROCESS MODELS

Need of software process model .Waterfall Model – simple diagram, uses, limits.Incremental Model – build in steps.Spiral Model – cycles + risk idea (only concept).Agile – short idea: sprints, customer feedback.One small point: AI tools for planning and estimation.

Need of Software Process Model

A **software process model** provides a structured framework to plan, develop, test, and maintain software systematically. It defines the sequence of activities and their relationships to ensure successful software development.

1. Provides a Systematic Approach

- Breaks complex software development into well-defined phases such as **requirements, design, implementation, testing, and maintenance**.
- Helps developers work in an organized and disciplined manner.

2. Improves Project Planning and Management

- Assists in **estimating cost, time, and resources** accurately.
- Enables project managers to track progress and control schedules effectively.

3. Ensures Quality and Reliability

- Integrates **verification and validation** activities at different stages.
- Early detection of errors reduces rework and improves software quality.

4. Reduces Risks

- Identifies potential risks early in the development lifecycle.
- Provides strategies to handle requirement changes and technical uncertainties.

5. Enhances Communication

- Acts as a common reference for **developers, managers, and clients**.
- Clearly defines roles, responsibilities, and deliverables at each phase.

6. Facilitates Requirement Management

- Helps in understanding, documenting, and managing user requirements systematically.
- Minimizes misunderstandings between stakeholders.

7. Supports Maintenance and Scalability

- Well-documented processes make future **modifications, updates, and scalability** easier.

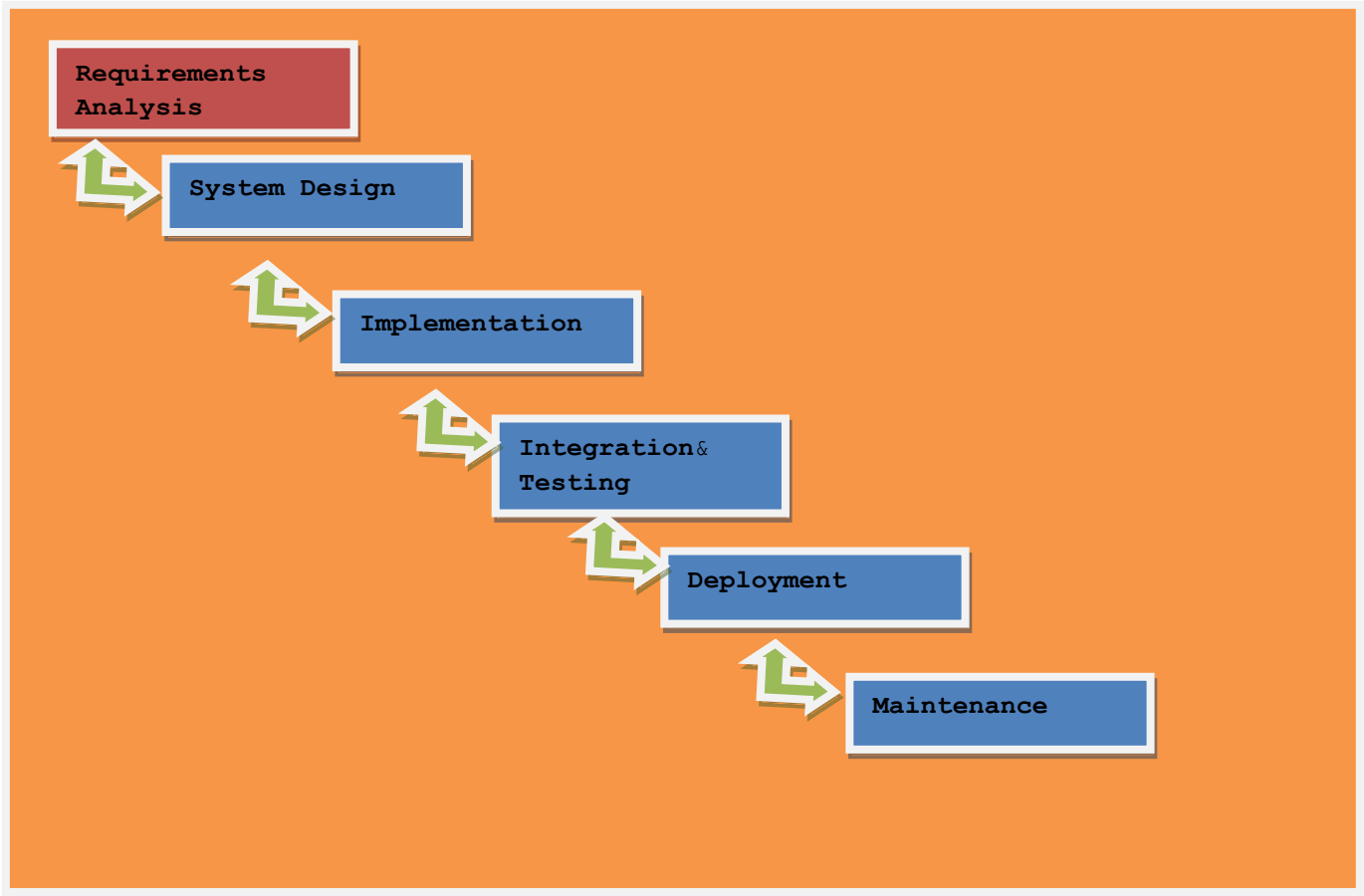
- Ensures long-term sustainability of the software.

8. Enables Process Improvement

- Helps organizations analyze past projects and improve future development processes.
- Encourages adoption of best practices and standards.

Waterfall Model

The **Waterfall Model** is a **linear and sequential software development process model** in which each phase must be completed before the next phase begins. It is called “waterfall” because progress flows steadily downward through the phases like a waterfall.



1. Requirements Analysis

- This is the **starting phase** of the Waterfall Model.
- The main aim is to **clearly understand the user's needs**.
- Developers interact with customers, users, and stakeholders.
- They study:
    - What functions the software should perform
    - What problems it should solve
    - What limitations or constraints exist
- All requirements must be **clear, complete, and fixed**.
- No technical work like design or coding is not done here.
- This phase is very important because **all next phases depend on it**.
- If mistakes happen here, they are costly to fix later.

2. System Design

- This phase explains **how the software will be built**.
- Requirements are converted into a **technical plan**.
- Developers decide:
    - Overall system structure
    - How different modules will interact
    - Data flow and control flow
- Both hardware and software decisions are made.
- The design acts like a **blueprint of the system**.
- A good design reduces confusion during coding.
- Once design is completed, changes are very difficult.

3. Implementation

- This phase is also known as the **coding phase**.
- Developers start writing the actual program code.
- Each module is developed according to the design.
- Programming languages and tools are selected.
- Developers follow coding standards and rules.
- Focus is on:
    - Correct logic

    - Proper structure
    - Clean and readable code
- No major changes are expected during this phase.

4. Integration & Testing

- After coding, all modules are **combined together**.
- The complete system is checked step-by-step.
- The aim is to **find and fix errors**.
- Testing ensures:
    - All modules work together
    - Software behaves as expected
    - User requirements are satisfied
- Errors found here may need changes in earlier phases.
- This phase improves **quality and reliability** of the software.

5. Deployment

- In this phase, the software is **delivered to users**.
- The system is installed in the real environment.
- Configuration and setup are completed.
- Users start using the software for actual work.
- Training or guidance may be provided.
- This phase marks the **end of development work**.

6. Maintenance

- This is the **longest phase** of the Waterfall Model.
- Software needs support after it is released.
- Activities include:
    - Fixing bugs reported by users
    - Improving performance
    - Making small updates
- Software must adapt to new environments or needs.
- Maintenance continues as long as the software is in use.

Important Characteristics

- Linear and sequential process
- One phase at a time
- Easy to understand
- Less flexible to change
- Heavy planning at the beginning

Advantages of Waterfall Model

- Simple and easy to understand.
- Clear documentation at every stage.
- Easy to manage due to well-defined phases.
- Suitable for **small projects** with **stable requirements**.

Disadvantages of Waterfall Model

- Changes in requirements are difficult to handle.
- No working software until late stages.
- High risk for complex and long-term projects.
- Not suitable when requirements are unclear or frequently changing.

Applications / When to Use

- Projects with **well-defined and fixed requirements**.
- Short-term projects.
- Government and defense projects where documentation is critical.

Example of Waterfall Model: Library Management System

1. Requirements Analysis

- The college wants a **Library Management System**.
- Developers talk to the librarian and staff.
- They understand requirements such as:
    - Add and remove books

    - Issue and return books
    - Search books
    - Maintain student records
- All requirements are clearly fixed before moving to the next phase.

2. System Design

- Developers plan how the library system will work.
- They design:
    - Database for books, students, and transactions
    - Screens for login, book search, issue/return
    - Flow of information between modules
- The complete structure of the system is prepared.

3. Implementation

- Programmers start coding the system.
- Different modules are developed:
    - Book management module
    - Student management module
    - Issue/return module
- Code is written according to the design plan.

4. Integration & Testing

- All modules are combined into one system.
- The system is tested:
    - Check if book issue and return work correctly
    - Verify student records update properly
- Errors are found and corrected.

5. Deployment

- The software is installed in the college library.
- Librarians start using the system.
- Old manual records are replaced with the new system.

6. Maintenance

- If bugs are found, they are fixed.
- New books or students are added.
- Small improvements are made when needed.

USES (APPLICATIONS) OF WATERFALL MODEL

- Used when **requirements are clear and fixed** from the beginning.
- Suitable for **small and simple software projects**.
- Used in **academic and learning projects** to understand SDLC.
- Applied in **government and defense projects** where rules are strict.
- Used when **technology is well known** and stable.
- Helpful when **strong documentation** is required.
- Suitable for projects with **no frequent changes**.

LIMITATIONS OF WATERFALL MODEL

- Difficult to make changes once a phase is completed.
- Not suitable for **large and complex projects**.
- Errors are usually found **late** during testing.
- Users get to see the software **only at the end**.
- Less customer involvement during development.
- High risk if requirements are not correct initially.
- Not suitable for projects with **changing requirements**.

Incremental Model – build in steps

- The **Incremental Model** is a software development model in which the system is **developed in small parts called increments**.

- Each increment adds **new functionality** to the existing system.

- The software is **not built all at once**; instead, it is delivered **step by step**.

- After every increment, a **working version of the software** is available.

- User feedback is taken after each increment and used to improve the next one.

- This model reduces risk and allows early use of the software.

Phases of Incremental Model

Phase 1: Requirements Analysis

- All **basic system requirements** are identified at the beginning.
- Requirements are divided into **small parts called increments**.
- Each increment contains a set of related features.
- Priority is given to important features first.
- Requirements can be refined in later increments.

Phase 2: System Design

- A **general system design** is prepared initially.
- Design supports future increments.
- For each increment:
  - Detailed design is done
  - Interfaces are planned
- Proper design ensures easy integration of increments.

Phase 3: Implementation (Coding)

- Each increment is coded separately.
- Developers write code for the selected features.
- Previous increments remain unchanged.
- Coding follows design and standards.
- After each increment, a working version is produced.

Phase 4: Testing

- Each increment is tested individually.
- Testing ensures new features work correctly.
- Integration testing checks compatibility with earlier increments.
- Errors are identified and corrected early.
- Quality improves with each increment.

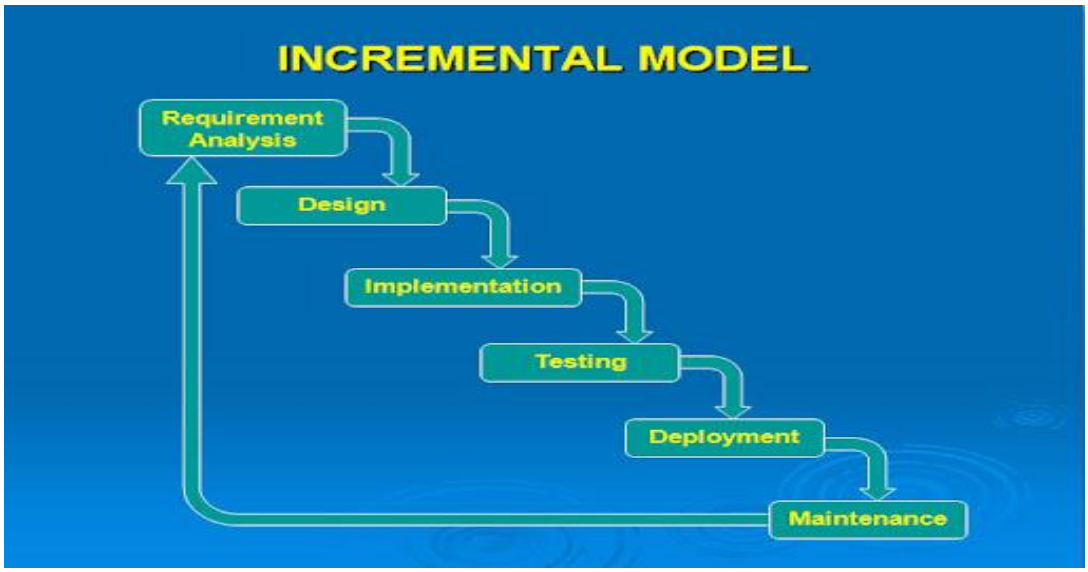
Phase 5: Deployment

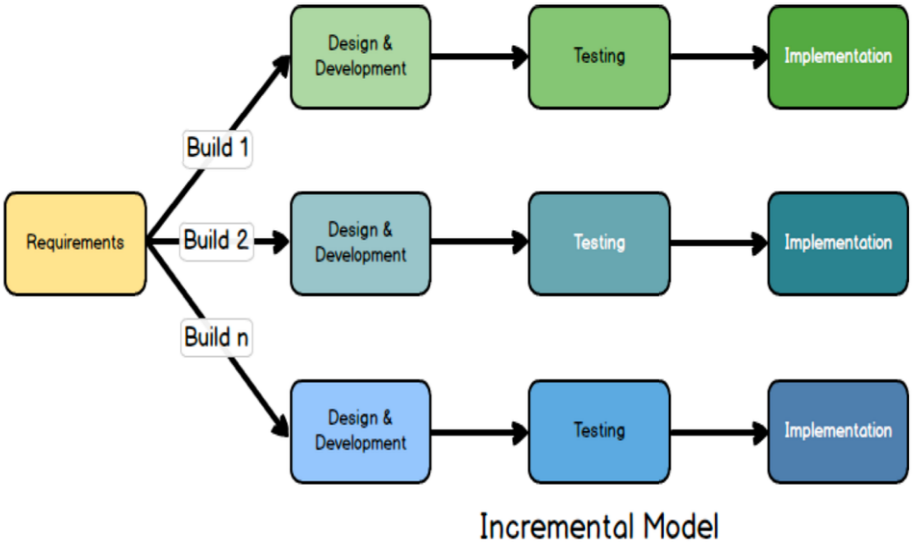
- After testing, each increment is delivered to users.

- Users can start using the system early.
- Feedback is collected after each deployment.
- Feedback helps improve the next increment.

Phase 6: Maintenance

- Support is provided for each released increment.
- Bugs are fixed and performance is improved.
- Small changes are made based on user feedback.
- Maintenance continues until the final system is complete.



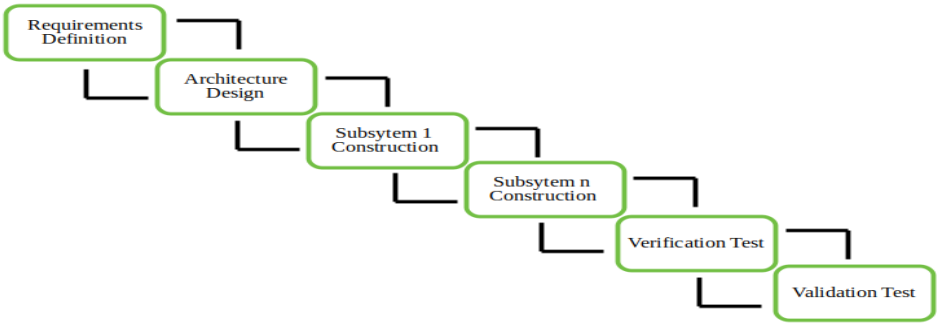


Incremental Model

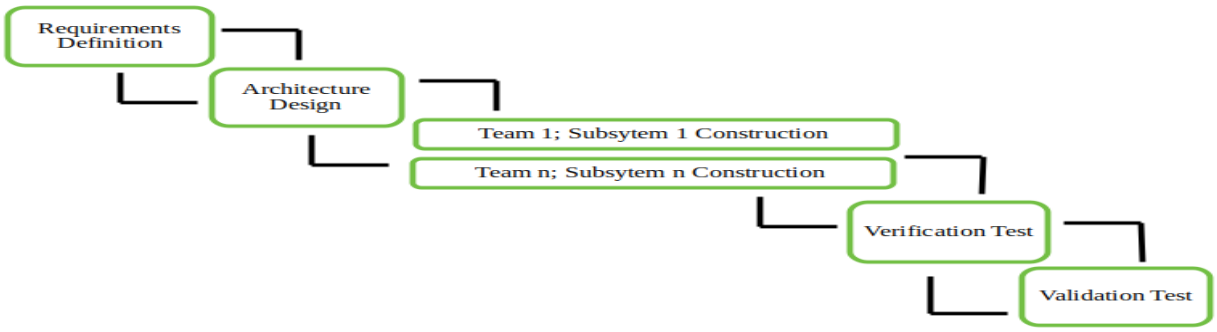
Types of Incremental Model

1. Staged Delivery Incremental Model

- Software is delivered in stages.
- Each stage adds new features.
- Most commonly used type.
- **Staged Delivery Model**
- The Staged Delivery Model develops software in a sequence of planned stages, where each stage delivers a functional part of the system. Each release brings the product closer to completion, allowing it to evolve gradually. Working versions are delivered at regular intervals, making progress visible and manageable throughout the development process. The diagram below shows this model :



-
- **Parallel Development Model**
- The Parallel Development Model divides the system into multiple modules that are developed simultaneously at the same time by different teams. By working on separate components in parallel, the development process becomes faster and more efficient. This approach reduces overall project time and allows teams to focus on specific functionalities concurrently. Given below is the diagram showing the model:



SPIRAL MODEL – SOFTWARE DEVELOPMENT

1. Introduction / Explanation

- The **Spiral Model** is a **risk-driven software development model**.
- It combines **iterative development** (like Incremental) with **systematic, phased approach** (like Waterfall).
- The software is developed in **cycles (spirals)** rather than linear steps.
- Each spiral **consists of planning, risk analysis, engineering, and evaluation**.
- The model is especially suitable for **large, complex, and high-risk projects**.

Key Idea: Build software **gradually while identifying and reducing risks** at each stage.

2. Phases of Spiral Model (4 Main Phases in Each Spiral)

Each cycle (spiral) includes the following phases:

Phase 1: Planning

- Gather requirements from the customer.
- Identify objectives and constraints.
- Decide the goals for the current spiral.
- Plan resources, schedule, and budget.

Phase 2: Risk Analysis

- Identify risks in the project (technical, financial, operational, etc.).
- Analyze each risk and propose solutions BY USING PROTOTYPE.

- High-risk areas are developed first or mitigated.

Phase 3: Engineering / Development

- Design, code, and test the system or increment.
- Each spiral produces a **working prototype or software increment**.
- Development is iterative within the spiral.

Phase 4: Evaluation / Customer Feedback

- Present the developed prototype to the user.
- Collect feedback for improvements.
- Decisions are made to continue, modify, or stop the project.
- After one spiral is completed, the **next spiral begins**, adding new features or refining the software.

---

4. Advantages of Spiral Model

- Reduces project risk through early risk analysis.
- Early detection of defects and errors.
- Allows customer feedback after each spiral.
- Supports **large, complex, and critical systems**.
- Flexible to changes in requirements.

---

5. Limitations of Spiral Model

- Complex to manage and implement.
- Requires **highly skilled team**.
- Can be **costly**, especially for small projects.
- Risk analysis itself can be time-consuming.
- Not suitable for simple or low-budget projects.

---

6. Uses / Applications

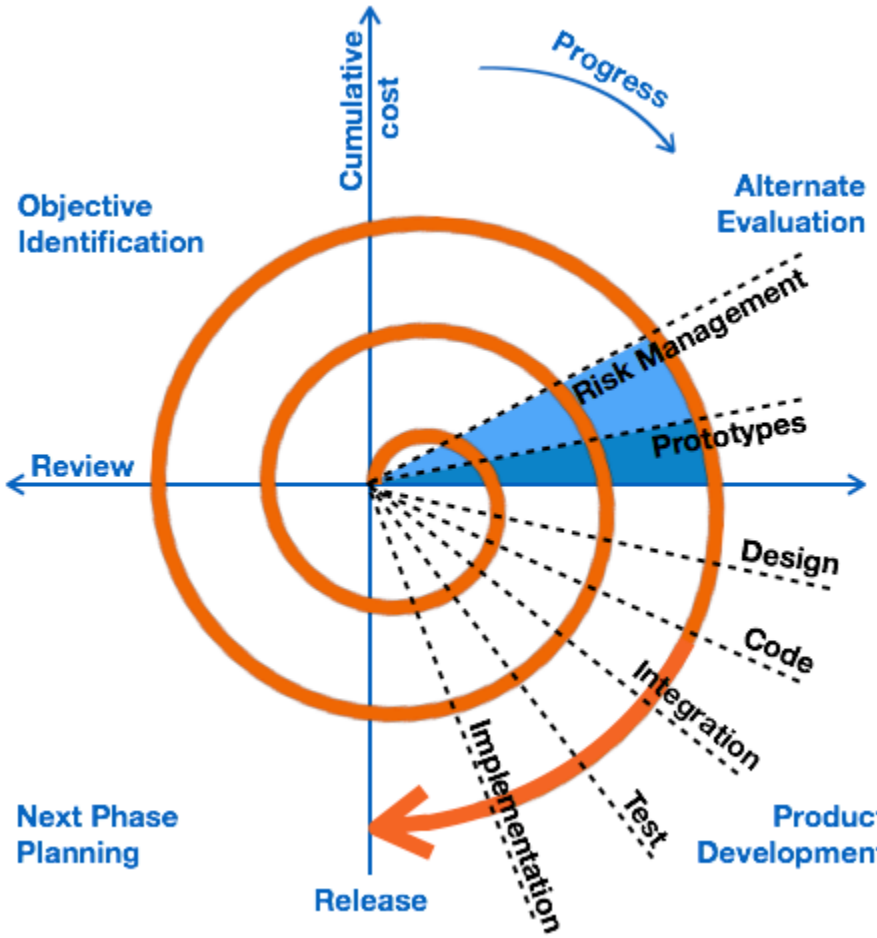
- Large and complex software projects.
- Systems with **high risk** or uncertain requirements.
- Defense, aerospace, banking, and healthcare systems.
- Projects requiring **frequent user feedback**.

- Software that needs **multiple prototypes** before final delivery.

7. Example

Air Traffic Control System

- Spiral Model is ideal because:
  - Safety and reliability are critical (risk-driven development).
  - Requirements may evolve with user feedback.
  - Incremental prototypes can be tested and refined repeatedly.
- **First Spiral:** Basic tracking system prototype
- **Second Spiral:** Add flight scheduling
- **Third Spiral:** Add communication & safety modules
- **Fourth Spiral:** Final system with complete functionality



Risk Handling in Spiral Model

Risk Handling in Spiral Model

The **Spiral Model** is a **risk-driven software development model**. Risk management is the main focus at every stage of development.

---

1. Risk Identification

At the beginning of each spiral (iteration), possible risks are identified, such as:

- Technical risks
- Cost risks
- Schedule risks
- Requirement change risks

2. Risk Analysis

Each identified risk is analyzed based on:

- Its severity
- Its impact on the project

This helps in deciding which risks need immediate attention.

3. Risk Resolution Using Prototyping

To reduce or eliminate risks:

- Prototypes are developed
- New technologies are tested
- Alternative designs are evaluated

This helps in detecting problems early.

4. Continuous Risk Handling

- Risks may appear even after development starts

- The Spiral Model allows risks to be identified and handled in every phase
- This makes risk management continuous and flexible

5. Customer Feedback Reduces Risk

- Customer feedback is collected after each spiral
- This reduces misunderstanding of requirements and design errors

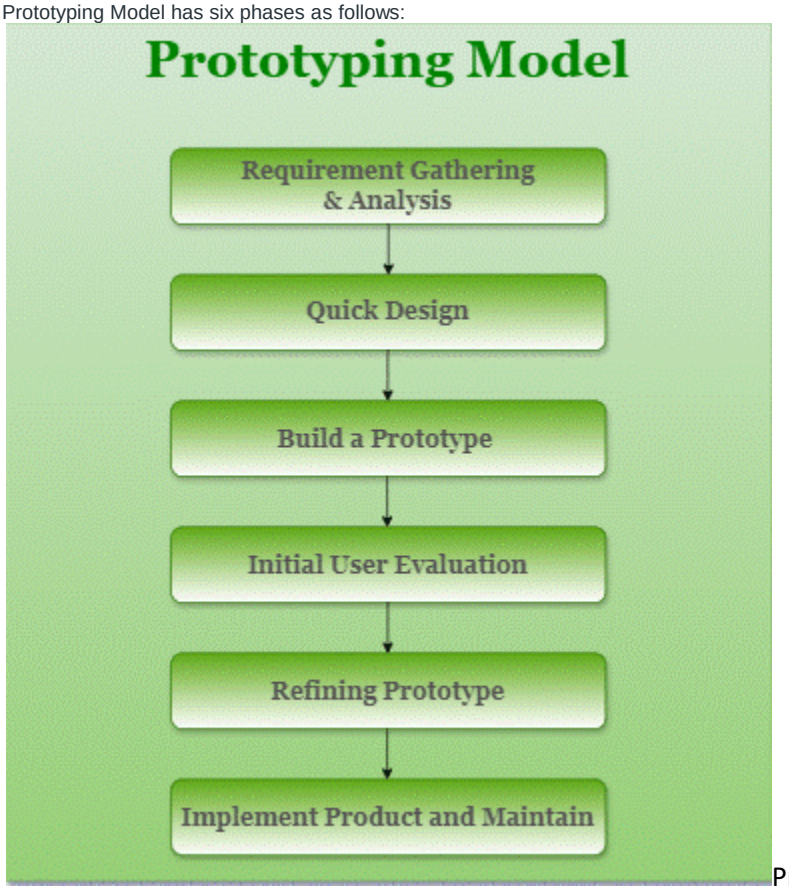
What is Prototyping Model?

The Prototyping Model is one of the most popularly used **Software Development Life Cycle Models (SDLC models)**. This model is used when the customers do not know the exact project requirements beforehand.

In this model, a prototype of the end product is first developed, tested, and refined as per customer feedback repeatedly till a final acceptable prototype is achieved which forms the basis for developing the final product.

Phases of Prototyping Model

Prototyping Model has six phases as follows:



Prototyping Model Phases

1. Requirements gathering and analysis

Requirement analysis is the first step in developing a prototyping model. During this phase, the system's desires are precisely defined. During the method, system users are interviewed to determine what they expect from the system.

2. Quick design

The second phase could consist of a preliminary design or a quick design. During this stage, the system's basic design is formed. However, it is not a complete design. It provides the user with a quick overview of the system. The rapid design aids in the development of the prototype.

3. Build a Prototype

During this stage, an actual prototype is intended to support the knowledge gained from quick design. It is a small low-level working model of the desired system.

4. Initial user evaluation

The proposed system is presented to the client for preliminary testing at this stage. It is beneficial to investigate the performance model's strengths and weaknesses. Customer feedback and suggestions are gathered and forwarded to the developer.

5. Refining prototype

If the user is dissatisfied with the current model, you may want to improve the type that responds to user feedback and suggestions. When the user is satisfied with the upgraded model, a final system based on the approved final type is created.

6. Implement Product and Maintain

The final system was fully tested and distributed to production after it was developed to support the original version. To reduce downtime and prevent major failures, the programmer is run on a regular basis.

AGILE MODEL – STEP-WISE EXPLANATION

1. Introduction

- Agile is a **flexible, iterative, and incremental software development model**.
- The software is **developed in small portions (sprints)** instead of building the whole system at once.
- **Customer collaboration** and **continuous feedback** are key to Agile.
- Agile allows **fast delivery, adaptability, and quality improvement**.

2. Build Steps in Agile Model

Step 1: Requirement Gathering

- High-level requirements are collected from the customer.
- Focus is on **user stories** – small, understandable features or tasks.
- Requirements can **evolve during the project**.

Step 2: Plan the Sprint

- The project is divided into **small iterations called sprints** (usually 1–4 weeks).
- Each sprint has **specific features to be developed**.
- Team decides tasks, roles, and timeline for the sprint.

Step 3: Design

- Quick and **lightweight design** is done for the features in the current sprint.
- Agile emphasizes **simplicity** – no heavy documentation.
- The design is enough to guide developers to start coding.

Step 4: Development / Implementation

- Developers **write code for features selected in the sprint**.
- Focus on **delivering a working version quickly**.
- Coding follows best practices but allows **flexibility for changes**.

Step 5: Testing

- Each feature developed in the sprint is **tested immediately**.
- Continuous testing ensures **bugs are fixed early**.
- Agile uses **automated testing** where possible.

Step 6: Customer Review / Feedback

- At the **end of the sprint**, a working software increment is **delivered to the customer**.
- Customer provides **feedback** and suggests improvements.
- Feedback is considered in the **next sprint**.

Step 7: Deployment

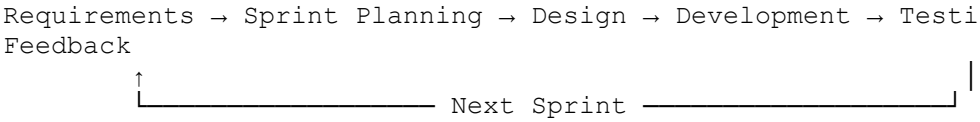
- After each sprint, the working software can be **deployed** in a real environment.

- Users can start using the increment immediately.
- Agile ensures **frequent delivery** of usable software.

Step 8: Repeat Next Sprint

- Steps 2–7 are **repeated for the next set of features**.
- Each sprint adds **new functionality or improves existing features**.
- Iteration continues until **the full system is complete**.

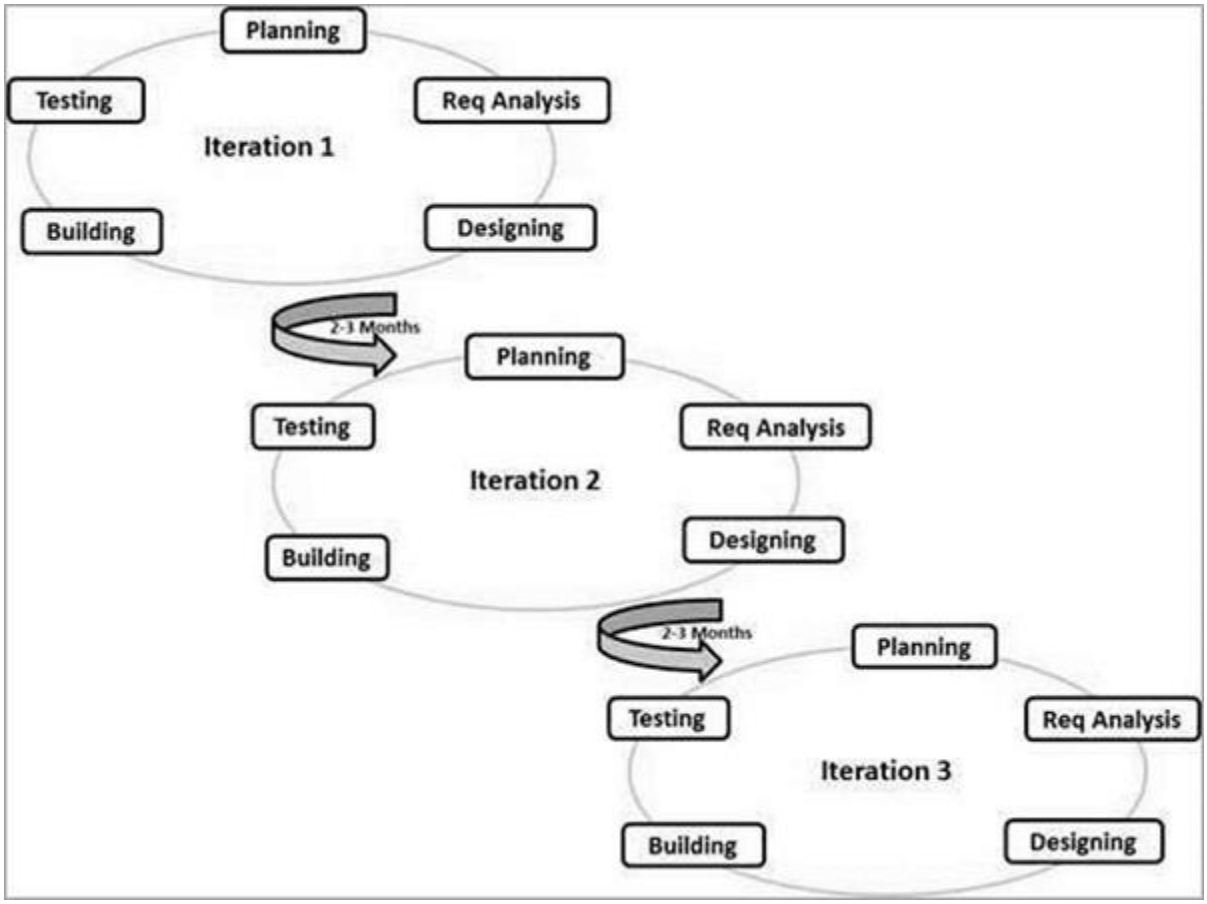
3. Agile Model Diagram (Sprints)



- Each sprint produces a **working increment**.
- Continuous feedback improves software **step by step**.

4. Key Features

- Short, iterative sprints
-
- Continuous customer feedback
- Incremental and adaptive development
- Collaboration between team and customer
- Fast delivery of usable software



5. Advantages

- Early delivery of working software
- Flexible to changing requirements
- High customer satisfaction
- Continuous improvement and quality
- Less risk compared to Waterfall

---

6. Limitations

- Requires **active customer involvement**
- Needs a **skilled, collaborative team**
- Less documentation can cause future maintenance issues
- Can be difficult for very large or distributed teams

7. Example

Online Shopping App:

- Sprint 1: User registration & login
- Sprint 2: Product listing & search
- Sprint 3: Cart & checkout
- Sprint 4: Payment gateway & order tracking
- Sprint 5: Reviews, ratings & recommendations

Each sprint delivers a **working feature**, customer reviews it, and the next sprint adds improvements.

Why Agile Fits Real-World Projects

- Requirements change frequently
- Fast delivery is required
- Continuous user feedback is needed

AI Tools for Planning and Estimation

Introduction

Planning and estimation are important activities in project development. Traditional methods depend on human experience and historical data. **AI tools improve accuracy** by analyzing past projects, team performance, and risk factors to provide better estimates for time, cost, and resources.

---

1. Role of AI in Planning

AI helps in:

- Automatic task breakdown (Work Breakdown Structure – WBS)
- Sprint and schedule planning
- Resource allocation
- Risk identification

AI tools use **machine learning and predictive analytics** to optimize project plans.

---

2. Role of AI in Estimation

AI supports:

- Effort estimation (hours, story points)
- Cost estimation
- Schedule prediction
- Workload forecasting

AI learns from **historical project data** to give more realistic estimates.

---

3. AI Tools Used in Project Development

1. Jira with AI Plugins

- Used in Agile projects
- Estimates sprint velocity and predicts completion
- Helps in backlog prioritization

2. ClickUp AI

- Converts requirements into tasks
- Provides time and effort estimates
- Supports sprint planning

3. Asana AI

- Suggests deadlines and workload balance
- Predicts risks and delays

4. Forecast App

- AI-based project and resource planning tool
- Automatically estimates timelines and effort
- Useful for Agile and Scrum teams

5. ChatGPT

- Generates project plans, WBS, and estimation tables
- Estimates time, cost, and story points from requirements
- Useful in early planning stages

6. Microsoft Project with AI

- AI-assisted scheduling
- Predicts delays and resource conflicts
- Optimizes project timelines

UNIT – III: Introduction to AI & Software Engineering

Overview of computer software and software systems.Limitations of traditional software engineering.The software crisis: causes and impact.Need for increased software power.Human–computer responsiveness.Software in dynamic environments.Self-maintaining and adaptive software.Need for AI in modern software systems.

Overview of Computer Software and Software Systems

1. Computer Software

Computer software refers to a collection of programs, procedures, and related documentation that instruct a computer on how to perform specific tasks. Unlike hardware, software is intangible and controls the operation of computer systems.

Key Characteristics

- Intangible (cannot be physically touched)
- Developed using programming languages
- Requires maintenance and updates
- Can be reused and modified

2. Types of Computer Software

a) System Software

System software manages and controls computer hardware and provides a platform for application software.

Examples:

- Operating Systems: Windows, Linux, macOS
- Device Drivers
- Compilers and Interpreters
- Utility Programs (antivirus, disk cleanup)

Functions:

- Resource management (CPU, memory, storage)
- File management
- Hardware–software interaction
- Security and access control

b) Application Software

Application software is designed to perform specific user-oriented tasks.

Examples:

- Word processors (MS Word)
- Spreadsheets (Excel)
- Web browsers (Chrome)
- Educational software
- Banking and shopping applications

Types:

- General-purpose applications
- Customized or tailor-made software

c) Programming Software

Programming software helps developers create, test, and maintain other software.

Examples:

- IDEs (Visual Studio, Eclipse)
- Debuggers
- Code editors
- Build tools

3. Software Systems

A **software system** is an integrated set of software components that work together to achieve a common objective. It may include multiple programs, databases, user interfaces, and communication modules.

Components of a Software System

- User Interface (UI)
- Business Logic
- Database
- Middleware
- APIs and Services

4. Types of Software Systems

a) Standalone Systems

Operate independently on a single machine.

- Example: Calculator, Desktop media player

b) Distributed Systems

Run on multiple computers connected through a network.

- Example: Online banking systems

c) Real-Time Systems

Respond to inputs within strict time constraints.

- Example: Air traffic control systems

d) Embedded Systems

Integrated into hardware devices.

- Example: Washing machines, automobiles

Limitations of Traditional Software Engineering

Traditional software engineering approaches (such as **Waterfall model**) follow a linear and sequential process. While they were effective for small and stable projects, they have several limitations in modern, dynamic environments.

1. Rigid and Inflexible Process

- Once a phase is completed, it is difficult to go back and make changes.
- Not suitable for projects with frequently changing requirements.

2. Late Detection of Errors

- Testing is usually done after implementation.
- Errors discovered late increase cost, time, and effort.

3. Poor Handling of Changing Requirements

- Requirements must be clearly defined at the beginning.
- Any change later in the project causes rework and delays.

4. Limited Customer Involvement

- Customer interaction is minimal after the requirements phase.
- Final product may not fully meet user expectations.

5. Long Development Time

- Sequential phases increase total development time.
- No working software is delivered until the end.

6. High Risk and Uncertainty

- Risks are identified late in the process.
- Difficult to manage technical and business risks early.

7. Documentation Overload

- Emphasis on heavy documentation.
- Increases effort and reduces development speed.

8. Not Suitable for Complex and Large Systems

- Poor adaptability for large-scale, evolving systems.
- Integration issues arise late in development.

9. Low User Satisfaction

- Users see the product only at the final stage.
- Mismatch between user needs and final output is common.

10. Poor Support for Innovation

- Does not encourage experimentation or rapid prototyping.
- Limits creativity and iterative improvement.

The Software Crisis: Causes and Impact

1. Introduction

The **software crisis** refers to the difficulty of developing **large, complex, and reliable software systems** within **time, budget, and quality constraints**. The term became popular during the **1960s and 1970s** when software projects frequently failed or were delivered late and unreliable.

2. Causes of the Software Crisis

1. Increasing Software Complexity

- Growth in size and functionality of software systems
- Complex logic and interdependencies make development difficult

2. Poor Requirement Analysis

- Incomplete, unclear, or changing requirements
- Misunderstanding user needs leads to rework

3. Lack of Standardized Methods

- Early development lacked proper models, tools, and frameworks
- No structured approach to design and testing

4. Inadequate Project Management

- Poor planning and estimation
- Unrealistic schedules and budgets

5. Shortage of Skilled Software Engineers

- Limited trained professionals
- Developers lacked experience in large-scale systems

6. Poor Communication

- Ineffective coordination among developers, users, and managers
- Results in incorrect implementation

7. Late Testing and Debugging

- Testing performed only after coding
- Defects discovered late increase cost

3. Impact of the Software Crisis

1. Project Failures

- Many projects were cancelled or never completed
- Systems failed to meet requirements

2. Cost Overruns

- Projects exceeded allocated budgets
- High maintenance costs

3. Schedule Delays

- Late delivery of software products
- Missed business opportunities

4. Poor Quality Software

- Software contained bugs and performance issues
- Low reliability and maintainability

5. User Dissatisfaction

- End users rejected systems
- Reduced trust in software solutions

6. Maintenance Problems

- Difficult and expensive to modify or enhance software
- Poor documentation increased maintenance effort

7. Safety and Security Risks

- Failures in critical systems (medical, aviation, defense)
- Loss of data and system crashes

4. Measures Taken to Overcome the Software Crisis

- Introduction of **Software Engineering discipline**
- Development of **software process models** (Waterfall, Spiral, Agile)
- Use of **structured programming**
- Adoption of **CASE tools**
- Emphasis on **quality assurance and testing**
- Improved **project management practices**

Causes (Why it happens)	Impact (What happens because of it)	Measures (How to solve it)
Poor requirement understanding	Wrong software developed	Clear requirement gathering
Lack of proper planning	Project delay	Proper project planning
Poor project management	Cost increases	Use project management tools
Frequent requirement changes	Confusion & rework	Change management process
Lack of skilled developers	Low quality software	Training & hiring skilled staff
Poor communication	Misunderstanding in team	Regular meetings & communication
No proper testing	Software errors & bugs	Proper testing methods

Causes (Why it happens)	Impact (What happens because of it)	Measures (How to solve it)
Increasing software complexity	Hard to maintain	Use modular design
No documentation	Difficult maintenance	Maintain proper documentation

Need for Increased Software Power

Introduction

Software power refers to the ability of software to handle **complex tasks, large data volumes, high user loads, and intelligent decision-making** efficiently. With rapid technological advancements, traditional software capabilities are no longer sufficient, creating a strong need for increased software power.

Reasons for Increased Software Power

1. Growing Software Complexity

- Modern systems are large and highly interconnected.
- Applications integrate databases, networks, cloud services, and AI components.

2. Explosion of Data (Big Data)

- Massive amounts of structured and unstructured data are generated daily.
- Software must process, analyze, and extract insights in real time.

3. Real-Time Processing Requirements

- Many applications require immediate responses.
- Examples: stock trading, medical monitoring, autonomous vehicles.

4. Increased Number of Users

- Software systems now serve millions of users simultaneously.
- Requires high performance, scalability, and reliability.

5. Advanced User Expectations

- Users expect fast, responsive, and personalized software.
- Poor performance leads to user dissatisfaction.

6. Automation and Intelligence

- Need to reduce human effort and errors.
- Software must support intelligent decision-making and automation.

7. Dynamic and Distributed Environments

- Software runs across cloud, mobile, IoT, and edge devices.
- Must adapt to changing environments and workloads.

8. Security and Reliability Demands

- Increased cyber threats require powerful security mechanisms.
- Software must be fault-tolerant and resilient.

Human–Computer Responsiveness

Definition

Human–computer responsiveness refers to the ability of a computer system or software to **respond quickly, accurately, and appropriately to user actions or inputs**. It plays a crucial role in determining the **usability and effectiveness** of a software system.

Importance of Human–Computer Responsiveness

1. **Improves User Experience**
    - Fast responses make systems easy and pleasant to use.
    - Reduces user frustration.
2. **Increases Productivity**
    - Users can complete tasks faster with minimal waiting time.
    - Essential in professional and real-time systems.
3. **Enhances Interaction Quality**

    - Supports natural interaction such as touch, voice, and gestures.
    - Makes software more intuitive.
4. **Builds User Trust**
    - Reliable and consistent responses increase confidence in the system.

Key Aspects of Responsiveness

1. **Response Time**
    - Time taken by the system to react to user input.
    - Shorter response times are preferred.
2. **Feedback**
    - Immediate visual, audio, or textual feedback after an action.
    - Example: loading indicators, progress bars.
3. **Accuracy**
    - Correct and meaningful output for user input.
4. **Consistency**
    - Similar actions produce similar responses.

Examples

- **ATM systems** responding immediately to user selections
- **Search engines** displaying results within milliseconds
- **Voice assistants** responding instantly to spoken commands
- **Online gaming systems** requiring real-time interaction

Challenges

- High system load
- Network latency
- Complex computations
- Poor system design

Software in Dynamic Environments

Definition

Software in dynamic environments refers to software systems that operate in conditions where **requirements, inputs, resources, users, and operating contexts change continuously**. Such software must adapt to these changes while maintaining correct and efficient operation.

Software in dynamic environments refers to software systems that operate in situations where conditions change continuously and unpredictably.
These changes may occur in users, data, hardware, network conditions, or the external environment.

In simple words:
 It is software that must **adapt quickly to changes**.

Why Dynamic Environments Matter

Modern systems are not static. They work in:

- Changing user requirements
- Real-time data streams
- Network fluctuations
- Device mobility
- Security threats

Example areas:

- Cloud computing
- Mobile applications
- IoT systems
- Online banking
- E-commerce websites

Characteristics of Software in Dynamic Environments

1. **Adaptability**
    - Adjusts to new conditions automatically.
    - Example: Auto-scaling in cloud systems.
2. **Real-Time Processing**
    - Responds immediately to events.
    - Example: Traffic management systems.
3. **Scalability**
    - Handles increasing users or workload.
4. **Fault Tolerance**
    - Continues working even if some parts fail.
5. **Flexibility**
    - Easy to modify or update.

Examples

1. **Online Shopping System**
    - Must handle sudden traffic during sales.
    - Example: Amazon website during festival sales.
2. **Navigation Systems**
    - Update routes based on live traffic.
    - Example: Google Maps adjusting path automatically.
3. **Stock Market Applications**
    - React to real-time price changes.
4. **Cloud Platforms**
    - Example: Microsoft Azure automatically increasing server resources.

Challenges

1. Requirement changes frequently
2. Maintaining system stability
3. Handling security risks
4. Managing performance
5. Data consistency issues

Techniques Used

- Agile development
- DevOps practices
- Microservices architecture
- Machine learning for prediction
- Continuous integration & deployment

Self-Maintaining and Adaptive Software & Need for AI in Modern Software Systems

---

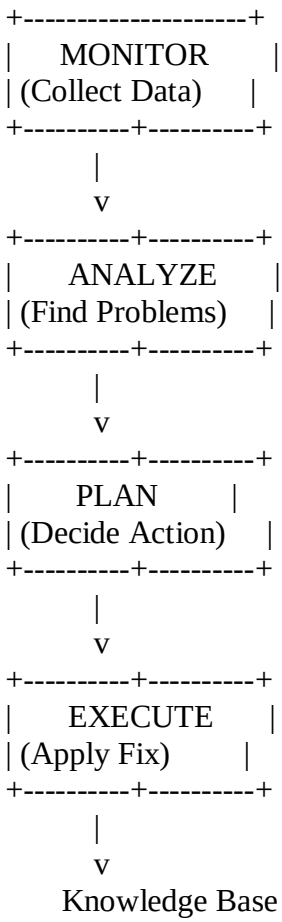
1. Self-Maintaining and Adaptive Software

Definition

Self-Maintaining Software is a system that can monitor its own performance, detect faults, and automatically take corrective actions without human involvement.

It follows the principle of:

"Monitor → Detect → Diagnose → Repair → Continue"



Explanation:

- **Monitor** → Checks system health
- **Analyze** → Detects abnormal behavior
- **Plan** → Chooses corrective action
- **Execute** → Performs automatic repair

Characteristics of Self-Maintaining Software

1. Self-Monitoring

Continuously checks system performance such as CPU usage, memory, errors, and response time to detect problems early.

2. Self-Diagnosis

Analyzes detected issues and identifies the root cause (hardware, software, or overload problem).

3. Self-Healing

Automatically fixes problems by restarting services or switching to backup servers.

4. Automatic Updates

Installs patches and security fixes without manual intervention to keep the system secure and updated.

5. Performance Optimization

Adjusts system resources automatically to maintain smooth and efficient performance.

Real-World Examples

- Microsoft Azure auto-recovers failed virtual machines.
- Windows Update installs patches automatically.
- Amazon Web Services replaces unhealthy instances automatically.

Adaptive Software

Definition

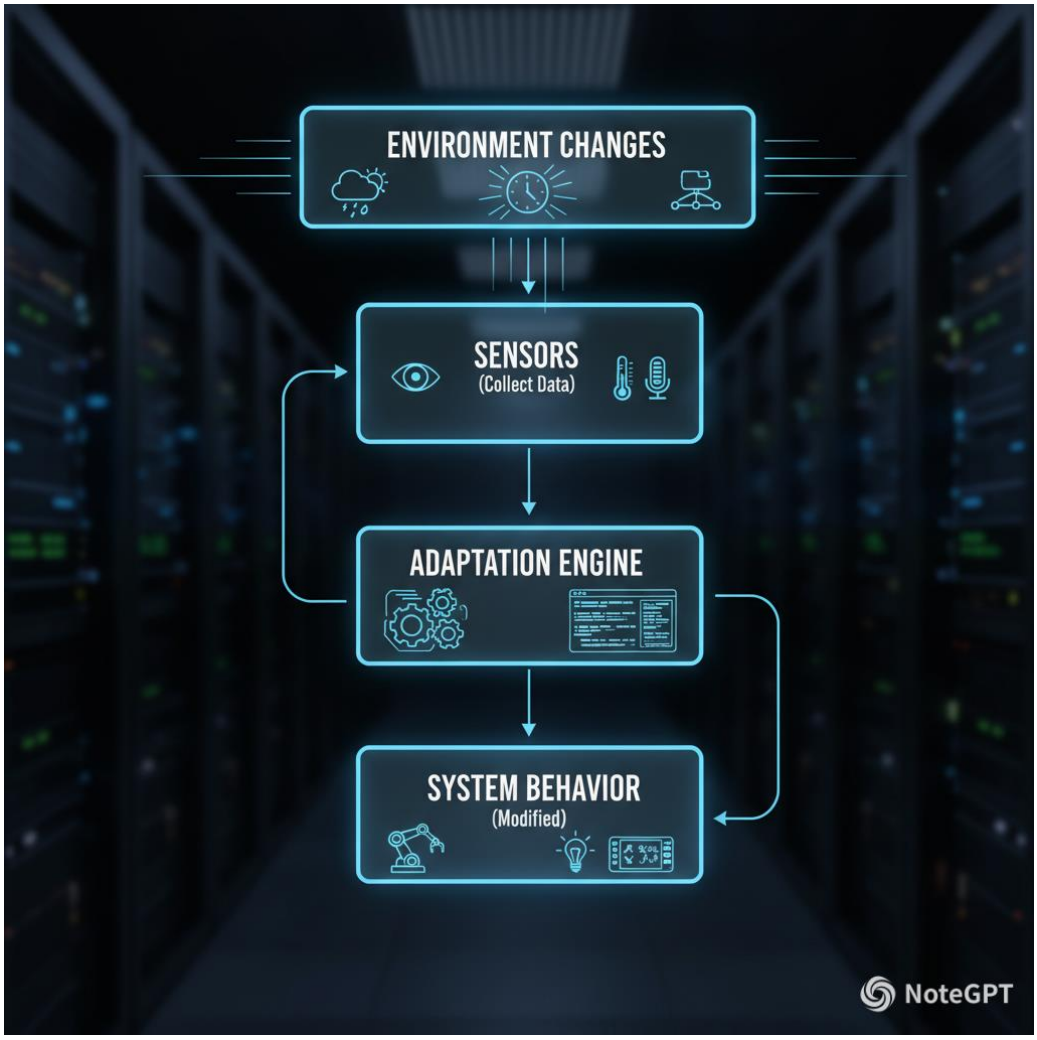
Adaptive Software is a system that automatically adjusts its behavior according to environmental changes, user behavior, or system conditions.

It focuses on:

"Sense → Learn → Adapt → Improve"

Core Characteristics

1. **Context Awareness**
    - Understands environment (location, device, time)
2. **Dynamic Configuration**
    - Changes system parameters automatically
3. **Learning Capability**
    - Uses AI/ML to improve performance
4. **Personalization**
    - Adjusts according to user preferences
5. **Scalability**
    - Handles workload growth dynamically



Real-World Examples

- Google Maps changes routes based on traffic.
- Amazon recommends products based on user behavior.
- Netflix adjusts video quality based on internet speed.

Comparison: Self-Maintaining vs Adaptive Software

Feature	Self-Maintaining	Adaptive
Main Goal	Fix internal problems	Adjust to external changes
Focus	System health	Performance & user experience
Trigger	Failures	Environment changes
Example	Auto restart server	Change route based on traffic

2. Need for AI in Modern Software Systems

Introduction

Modern software systems are **complex, large-scale, and dynamic**. Traditional rule-based programming is insufficient to manage such complexity. **Artificial Intelligence (AI)** enables software to **learn, reason, and make decisions autonomously**.

Reasons for Using AI

1. **Handling Complexity**
   - AI manages systems too complex for manual control.
2. **Automation**
   - Reduces human intervention in monitoring and maintenance.
3. **Learning and Adaptation**
   - Systems improve performance using historical data.
4. **Real-Time Decision Making**
   - AI enables quick and accurate responses.
5. **Personalization**
   - Tailors software behavior to individual users.

Applications of AI in Software Systems

- Chatbots and virtual assistants
- Fraud and anomaly detection
- Predictive maintenance systems
- Intelligent recommendation engines
- Autonomous vehicles and robotics

Relationship Between AI and Adaptive Software

AI provides the **intelligence** required for software to:

- Monitor itself
- Learn from data
- Adapt to changes

- Maintain optimal sperformance
