Unit-1

Introduction

1. **Software**: A collection of executable computer programs, configuration files, associated libraries, and documentation.
2. **Software Product**: A software solution designed to meet a specific set of requirements.
    - Includes types like generic, custom, system, and application software.
3. **Engineering**: The application of well-defined scientific principles and systematic methods for developing products with:
    - Economic viability
    - Social responsibility
    - Practical considerations
4. **Software Engineering (SE)**:
    - A systematic, disciplined, and quantifiable approach to software development, operation, and maintenance.
    - Software products developed using SE principles are more likely to be reliable.
    - Emphasizes the use of appropriate tools and techniques.
    - Focuses primarily on techniques for software development and maintenance.

Fundamental Drivers of SE

1. **Industrial-Strength Software Requirements**:
    - **Operational**: Ensures functionality, usability, and correctness.
    - **Portability & Interoperability**: Capable of being moved across different platforms.
    - **Maintainability**: Focuses on modularity, scalability, and ease of maintenance.
    - **Comprehensive Documentation**: Detailed and thorough documentation is essential.
    - **Minimal Bugs**: Strives for the absence or minimal presence of bugs.
    - **Business Impact**: Critical for quality, resilience, and overall business operations.
    - **High Cost**: Software development and maintenance are labor-intensive and expensive.

2. **Software Maintenance & Rework**:
    - Involves corrective actions, adaptive updates, or rework.
    - Maintenance costs are significant for both hardware and software.
    - Many projects face delays and reliability issues.
    - Around 35% of projects are considered "runaway" (over budget or off-schedule).
    - In defense systems, 70% of equipment failures are attributed to software issues.
    - Software can directly affect life-or-death situations.

3. **Challenges in Modern Software**:
    - **Heterogeneity**: Systems often operate as distributed networks.
    - **Diversity**: Various software systems require different techniques and methods.
    - **Rapid Business & Social Change**: Constantly adapting existing software and developing new solutions.
    - **Security & Trust**: Ensuring systems are secure and trustworthy.
    - **Scalability**:
        - Maintain quality and productivity.
        - Ensure consistency and repeatability across systems.

Software Lifecycle

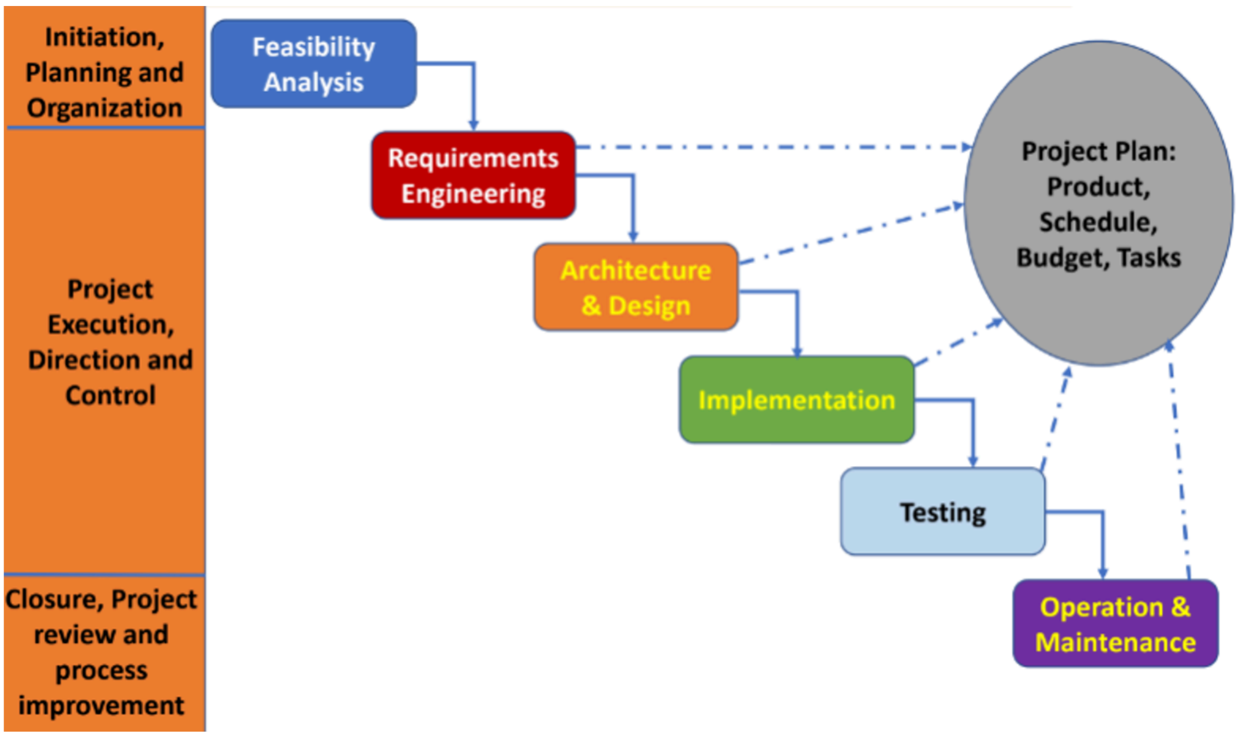
- Software Process/Lifecycle/Process Model/Lifecycle Model:
    - A structured set of activities designed to produce intermediate and final products.
    - Each phase is guided by principles that define its goals and objectives.

Products: Outcomes generated from executing each phase of the process.

Characteristics of Each Phase:

- **Entry Criteria**: Conditions that must be met to begin the phase.
- **Tasks & Deliverables**: Specific activities to be completed and the resulting outputs.
- **Exit Criteria**: Conditions that must be satisfied to move to the next phase.
- **Responsibility**: Clear assignment of who is accountable for each phase.
- **Dependencies**: Relationships and linkages between phases or tasks.
- **Constraints**: Limitations and restrictions impacting the process.

Software Development Life Cycle (SDLC) : Systematic process for building software that ensures quality and correctness that meets customers business requirements.

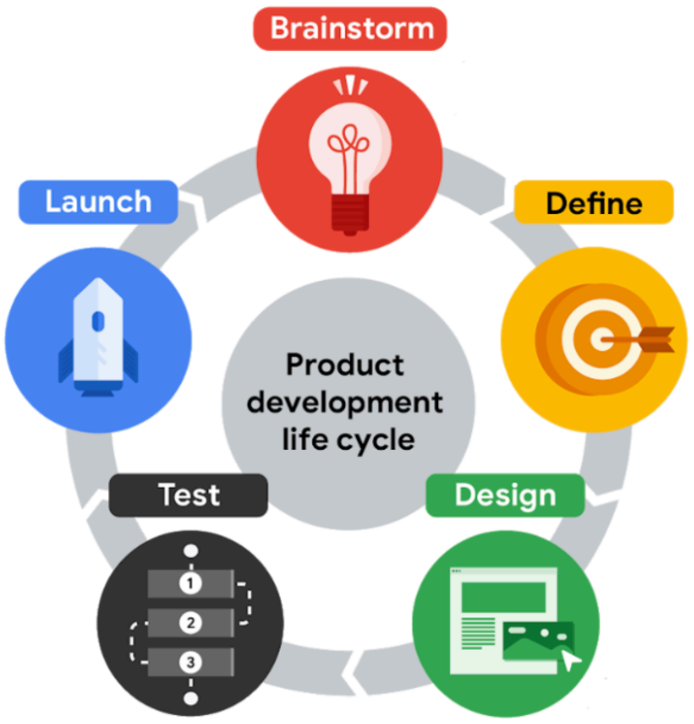

Initiation, Planning and Organization
Project Execution, Direction and Control
Closure, Project review and process improvement
Feasibility Analysis
Requirements Engineering
Architecture & Design
Implementation
Testing
Operation & Maintenance
Project Plan: Product, Schedule, Budget, Tasks

Stages of the Software Lifecycle:

1. **Requirement Analysis:**
   - **Entry Criteria:** Identifying the need for a solution to an existing problem.
   - **Tasks & Deliverables:**
     - Define the project scope and anticipate issues and opportunities.
     - Gather detailed requirements to create a project timeline.
   - **Exit Criteria:** Completion of the Software Requirements Specification (SRS) document.
   - **Responsibility:** Senior team members, stakeholders, and domain experts.
   - **Feasibility Study:** Evaluates economic, legal, operational, technical, and scheduling factors.
2. **Design:**
   - **Entry Criteria:** Availability of the Software Requirements Specification (SRS) document.
   - **Tasks & Deliverables:**
     - Develop system and software design documents.
     - Define overall architecture.
   - **Exit Criteria:** Creation of two design documents:
     - **High-Level Design (HLD):** Brief description and names of modules, functionality, interface relationships, and dependencies.
     - **Low-Level Design (LLD):** Detailed logic for each module, interface specifications, dependencies, error handling, and input/output considerations.
3. **Implementation/Coding:**
   - **Entry Criteria:** Availability of system design documents.
   - **Tasks & Deliverables:** Develop the software modules using the chosen programming language.
   - **Exit Criteria:** Completed software.
   - **Responsibility:** Developers.

4. **Testing:**
   - **Entry Criteria:** Developed software.
   - **Tasks & Deliverables:** Validate system functionality against requirements and resolve any bugs.
   - **Exit Criteria:** Stable, bug-free, and fully functional software.
   - **Responsibility:** Quality Assurance and Testing Team.
5. **Maintenance:**
   - Involves **three** primary activities:
     - **Bug Fixing:** Address issues from scenarios that were not previously tested.
     - **Upgrades:** Update to newer versions.
     - **Enhancements:** Add new features and capabilities.

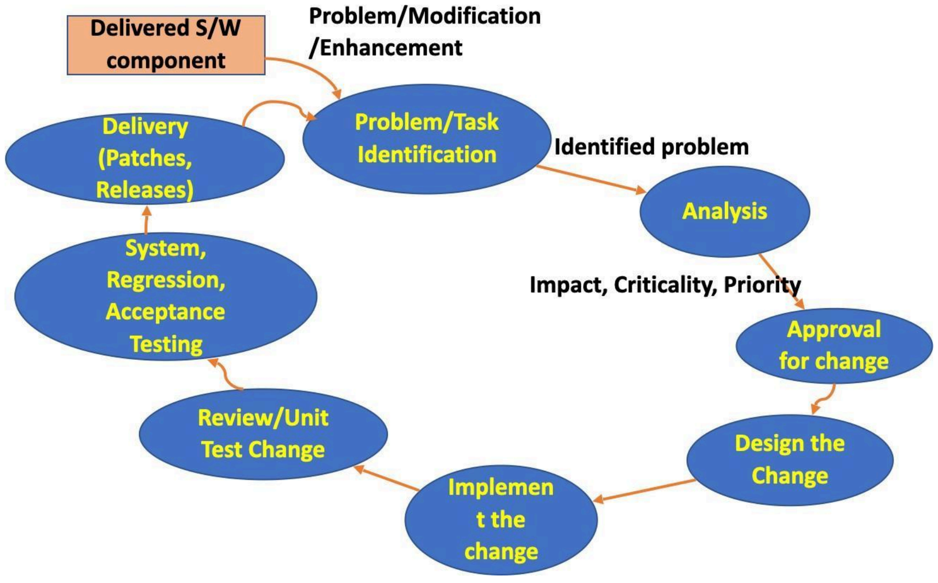
Product Development Lifecycle (PDLC) : Specific set of steps to take a project from first spark to release.



Stages of PDLC:

- **Brainstorm**:
  - Ideate a product that addresses specific user problems.
  - Research competitors and analyze existing products.
  - Ensure the new product either fills a gap or offers improvements over current options.
- **Define**:
  - Map out detailed specifications for the product.
  - Identify the target customers, their needs, and key product features.
  - Narrow down the product's focus to meet specific goals.
- **Design**:
  - Develop design concepts and ideas for the product.
  - Create prototypes and wireframes to define components and structure.
- **Test**:
  - Build functional prototypes and conduct testing in three phases:
    - Internal testing.
    - Stakeholder reviews.
    - External tests with potential users.
  - Gather feedback from each phase and make necessary improvements.
- **Launch**:
  - Finalize the product and make it accessible to the public.

Software Maintenance Life Cycle


Delivered S/W component
Problem/Modification /Enhancement
Problem/Task Identification
Identified problem
Analysis
Impact, Criticality, Priority
Approval for change
Design the Change
Implemen t the change
Review/Unit Test Change
System, Regression, Acceptance Testing
Delivery (Patches, Releases)

The Software Maintenance Life Cycle encompasses the activities involved in managing and updating software after its initial release. The key stages include:

1. **Identification of Maintenance Needs**: Issues or enhancement requests are identified, either through user feedback, system monitoring, or performance reviews.

2. **Analysis**: The impact and feasibility of changes are analyzed to determine if and how they should be implemented.

3. **Design and Planning**: The necessary changes or updates are designed, and a plan is developed to integrate them into the existing system without disrupting functionality.

4. **Implementation**: The changes are coded, tested, and integrated into the software. This may involve bug fixes, performance improvements, or new feature additions.

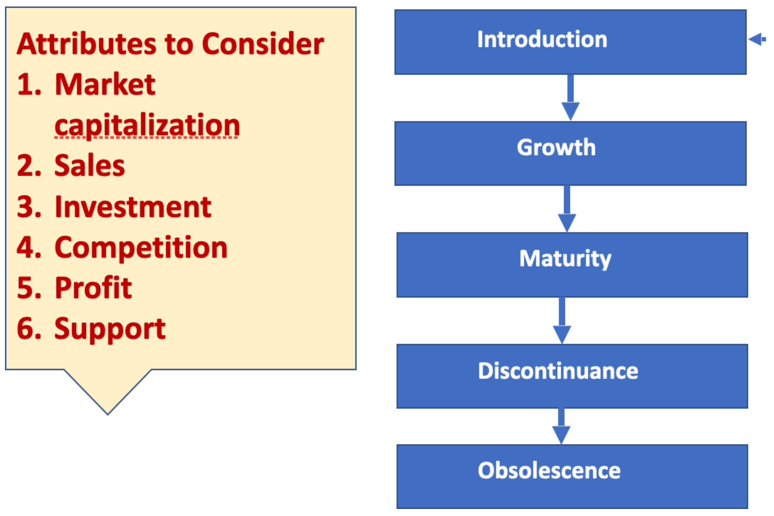
5. **Testing**: Rigorous testing is conducted to ensure that the changes work as intended and that no new issues are introduced.

6. **Deployment**: The updated software is deployed to users, and any necessary documentation is updated.

7. **Review and Monitoring**: The performance of the updated software is monitored, and user feedback is collected to inform future maintenance cycles.

Product Life Cycle

- Dictated by factors such as Market Capitalisation, Sales, Investment and so on.
- The Product Life Cycle refers to the stages a product goes through from its inception to its decline.

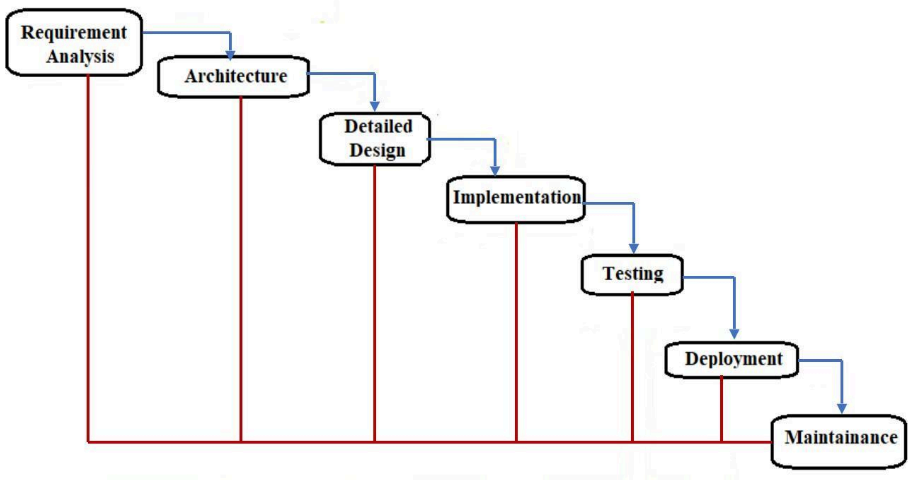


Stages of Product Life Cycle :

- **Introduction:** The product is launched and enters the market, but may have limited popularity initially.
- **Growth:** Customers begin adopting the product widely; additional features may be introduced to stay competitive.
- **Maturity:** Growth slows as product adoption reaches its peak and stabilizes.
- **Discontinuance:** The product is no longer sold but continues to receive maintenance.
- **Obsolescence:** The product is no longer maintained and is phased out.

Legacy SDLCs

1. **Waterfall model** : A linear software development approach where each phase is completed sequentially before moving on to the next. It involves distinct stages like requirement analysis, design, implementation, testing, and maintenance, with no overlap between phases. Progress flows in one direction, like a waterfall, making it best suited for projects with well-defined requirements.



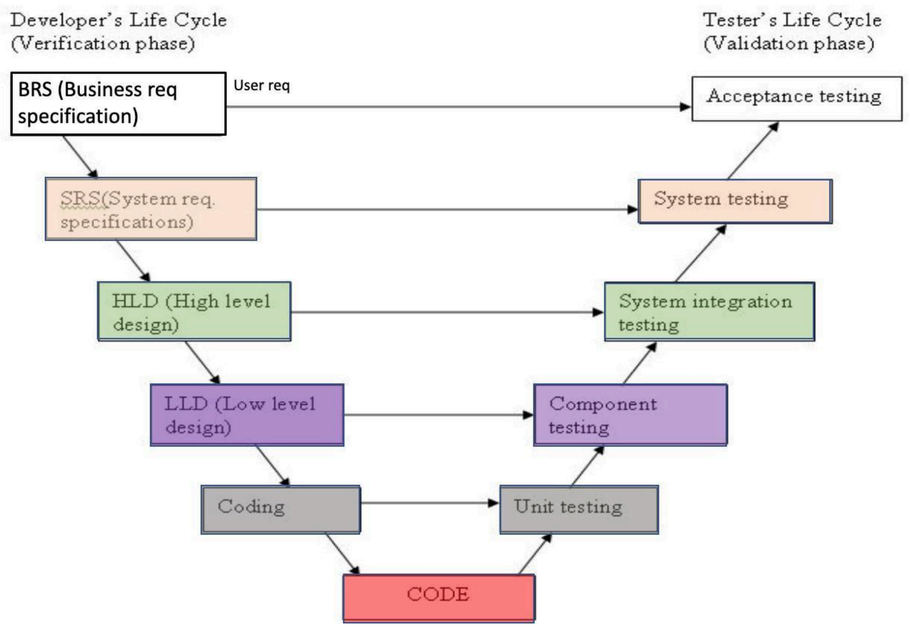
Pros & Cons:

Advantages	Disadvantages
Simple	Assumes frozen requirements
Clear Identified Phases (departmentalised)	Sequential and Big Bang Approach
Easy to Manage	High Risk and Uncertainty
Specific Deliverables and Review at each phase	

Usage:

- Suitable for short projects with clear and well-understood requirements.
- Applicable in large projects as high-level variants.
- Ideal when the product definition is stable and the technology is well-understood.
- Often used in large, globally developed projects in combination with other lifecycle models.

2. **V Model:** Also known as the **Verification and Validation model**, is a software development methodology that emphasizes the importance of testing throughout the development process. It is characterized by its V-shaped diagram, where each development stage corresponds to a testing phase.



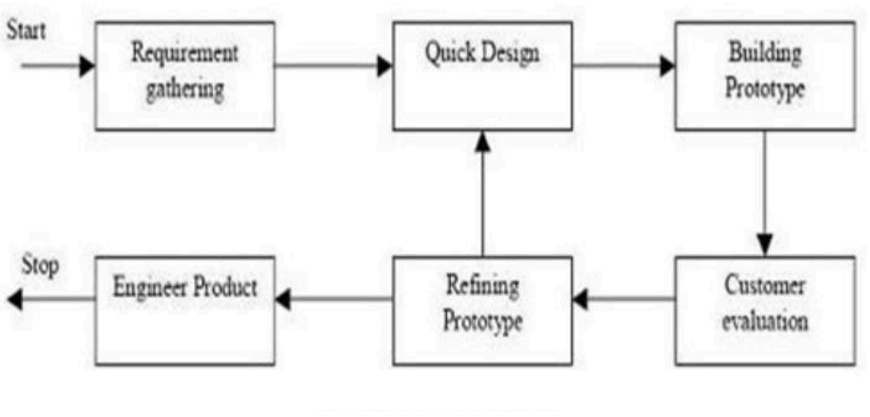
Pros & Cons:

Advantages	Disadvantages
Similar to Waterfall	Similar to Waterfall
Test Activities before Coding	No early prototypes
Higher Probability of success	Change in process = Change in documentation

Usage of the V-Model:

- **Projects with Well-Defined Requirements:** Ideal for projects where requirements are clear and stable from the outset.
- **Regulated Industries:** Commonly used in industries such as aerospace, automotive, and healthcare, where rigorous validation and verification are critical.
- **High Assurance Software:** Suitable for systems where safety, reliability, and compliance are paramount, such as in critical systems or safety-critical applications.
- **Clear Project Phases:** Best for projects where each phase can be distinctly defined and where early and continuous testing is beneficial.

3. **Prototyping Model:** A software development approach that emphasizes the creation of prototypes to explore and refine requirements before finalizing the software product. It involves building preliminary versions of the system to gather user feedback and make iterative improvements. **Relatively a CHEAPER process.**



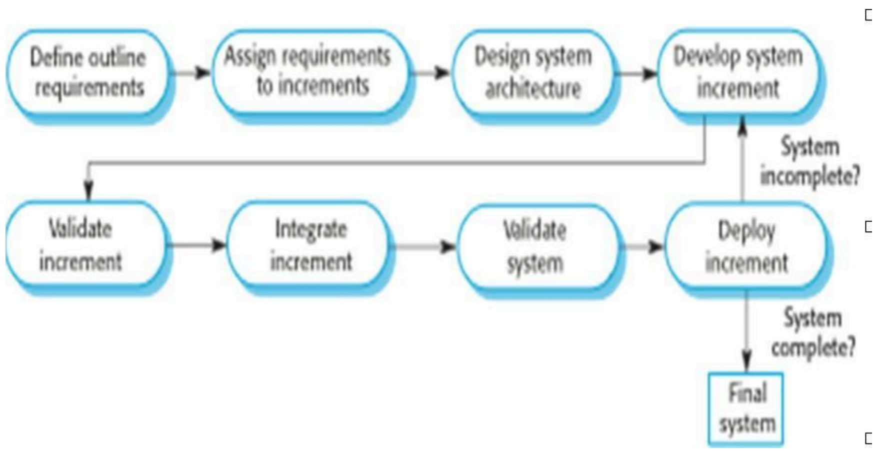
Pros & Cons:

Advantages	Disadvantages
Active user involvement	Increased Complexity
Better Risk mitigation	May not be Optimal
Earlier detection of problems and missing functions	
More Stable	

Usage of the Prototyping Model:

- **Unclear or Evolving Requirements:** Ideal for projects where requirements are not well-defined from the start or are expected to change.
- **User-Centric Applications:** Useful for projects that require close user involvement to ensure the final product aligns with user needs and expectations.
- **Complex Systems:** Effective for complex systems where visualizing and interacting with a prototype helps in understanding and refining the design.
- **Early Feedback Needed:** Suitable when early feedback from users is crucial for shaping the system and ensuring it meets their needs.
- **Innovative Projects:** Beneficial for projects involving new or innovative features where exploration and iterative testing are required to define the final solution.

4. **Incremental Model:** A software development approach where the system is built and delivered in small, manageable segments or increments. Each increment adds additional functionality to the existing system, allowing for gradual development and delivery.



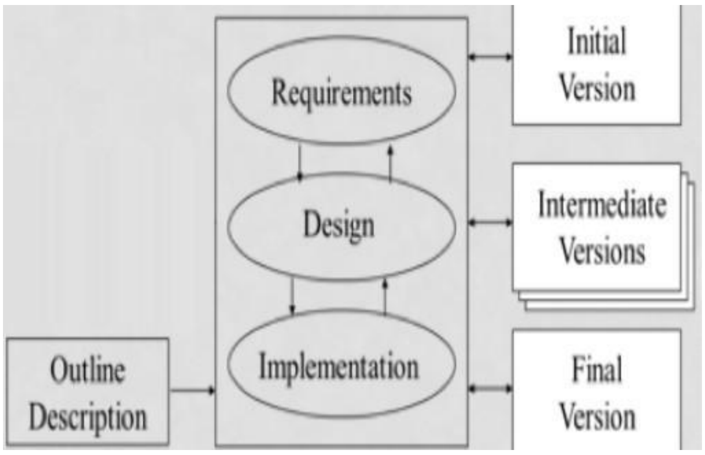
Pros & Cons:

Advantages	Disadvantages
Earlier versions	Needs good planning and design
More flexible	Clear and complete definition
Easier to test	Higher cost than waterfall
Reduces over-functionality	

Usage:

- **Defined Major Requirements:** Suitable for projects where the core requirements are clearly established.
- **Early Market Entry:** Ideal for scenarios where it's important to deliver a product to the market quickly.
- **New Technology:** Effective when implementing new or emerging technologies.
- **Skill Availability:** Useful when the required skills for the project are limited or unavailable, allowing for incremental development and integration.

5. **Iterative Model:** A software development approach where the system is developed through repeated cycles, or iterations, each of which refines and improves upon the previous version.



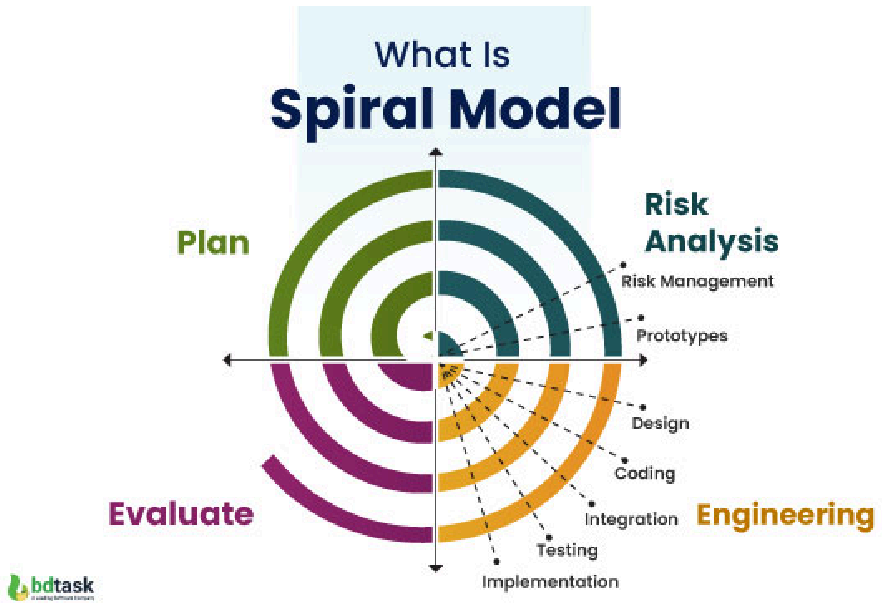
Pros & Cons:

Advantages	Disadvantages
Identify requirements	Rigid with overlaps
Risk mitigation	Costly system
Redesign and Rework	

Usage of the Iterative Model:

- **Evolving Requirements**: Suitable for projects where requirements may change or develop over time, allowing for adjustments and refinements.
- **User Feedback Integration**: Ideal when regular user feedback is needed to guide and improve the development process.
- **Complex Projects**: Effective for complex projects where early versions can be tested and improved in cycles.
- **Risk Management**: Useful for managing risks by identifying and addressing issues early through iterative cycles.
- **Continuous Improvement**: Beneficial when the project demands ongoing enhancement and fine-tuning to meet evolving needs and expectations.

6. **Spiral Model** : A risk-driven software development methodology that combines iterative development with elements of the Waterfall Model. It is designed to address and manage risks through repeated cycles of planning, development, and evaluation.



Pros & Cons:

Advantages	Disadvantages
Risk Management: Early risk handling.	**Complexity**: Difficult to manage.
Flexibility: Adapts to changes.	**Cost**: Potentially high expenses.
User Feedback: Continuous input integration.	**Time-Consuming**: Requires lengthy iterations.

Usage of the Spiral Model:

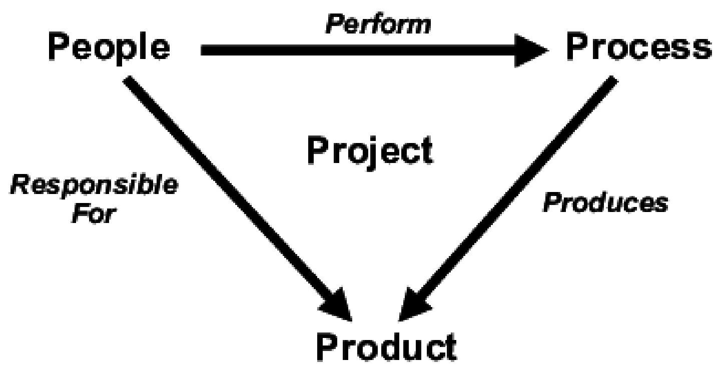
- **Complex Projects**: Suitable for large, complex systems.
- **Unclear Requirements**: Effective when requirements are evolving or not fully defined.
- **High Risk**: Ideal for projects with significant risk factors.
- **Frequent Feedback**: Useful for projects needing regular user feedback and adjustments.

Drawbacks of Legacy SDLCs:

- **Predictive Approach**: Based on rigid, predictive planning rather than adaptive development.
- **Extensive Upfront Planning**: Requires extensive planning before development begins.
- **Limited Customer Interaction**: Lacks mechanisms for regular customer feedback and interaction.
- **Complex Projects Only**: More suited for large projects with intricate dependencies, but less flexible for smaller or evolving projects.

4Ps - Process, Product, People, Project

The 4Ps—Process, Product, People, and Project—are key components in the context of software development and project management. Each element plays a critical role in ensuring the success and effectiveness of a project.



1. Process

Definition: The Process refers to the structured set of activities, methodologies, and procedures used to develop and manage software or projects. It encompasses the workflow and stages involved in achieving project goals.

Key Aspects:

- **Methodology:** The approach used, such as Agile, Waterfall, or Spiral.
- **Phases:** The different stages of development, including planning, design, implementation, testing, and maintenance.
- **Best Practices:** Standard practices and guidelines that ensure efficiency, quality, and consistency throughout the project lifecycle.
- **Continuous Improvement:** The process should be adaptable and allow for refinements based on feedback and evolving needs.

Importance:

- Ensures structured and systematic development.
- Helps manage project complexity and risks.
- Provides a clear roadmap and accountability.

2. Product

Definition: The Product refers to the final software or system delivered to the user, encompassing all its features, functionalities, and attributes.

Key Aspects:

- **Requirements:** The specific needs and expectations that the product must fulfill.
- **Design and Architecture:** The overall design and technical structure of the product.
- **Quality:** The product's reliability, performance, and adherence to standards and requirements.
- **User Experience:** The usability and satisfaction of the end-users interacting with the product.

Importance:

- Represents the tangible outcome of the development process.
- Directly impacts user satisfaction and business value.
- Requires careful planning and execution to meet user needs and expectations.

3. People

Definition: People refer to the individuals and teams involved in the project, including stakeholders, developers, designers, managers, and end-users.

Key Aspects:

- **Roles and Responsibilities:** The specific functions and tasks assigned to each team member.
- **Skills and Expertise:** The knowledge and capabilities required to complete the project successfully.
- **Collaboration and Communication:** The interactions and coordination between team members and stakeholders.
- **Motivation and Engagement:** Ensuring team members are motivated and committed to the project's success.

Importance:

- Drives the execution and management of the project.
- Influences the quality of the process and product.
- Essential for effective collaboration and achieving project goals.

4. Project

Definition: The Project encompasses the entire endeavor from initiation to completion, including its scope, timeline, budget, and objectives.

Key Aspects:

- **Scope:** The defined boundaries and deliverables of the project.
- **Timeline:** The schedule and milestones for completing project tasks and phases.
- **Budget:** The financial resources allocated and managed throughout the project.
- **Risk Management:** Identifying, assessing, and mitigating risks that may affect project success.

Importance:

- Provides a framework for planning and managing resources.
- Ensures alignment of objectives with organizational goals.
- Helps in tracking progress, managing risks, and delivering the final product on time and within budget.

Agile : Agile is a flexible, iterative approach that emphasizes continuous delivery, collaboration, and adaptation to change.

- **Ongoing Adjustment**: Agile methodologies promote continuously adjusting development goals to meet customer needs and expectations.
- **Reduced Planning Overhead**: Aims to minimize extensive planning by facilitating rapid responses to changes.
- **Flexibility**: Agile is a flexible approach to software development and project management.
- **Iterative Process**: Emphasizes iterative development with frequent cycles or sprints.

- **Adaptability**: Focuses on adapting to evolving requirements and changes.
- **Collaboration**: Encourages strong collaboration and communication among team members.
- **Customer Feedback**: Prioritizes incorporating customer feedback regularly.
- **Incremental Delivery**: Delivers functional software quickly and continuously through smaller, manageable increments.

Agile Manifesto Principles:

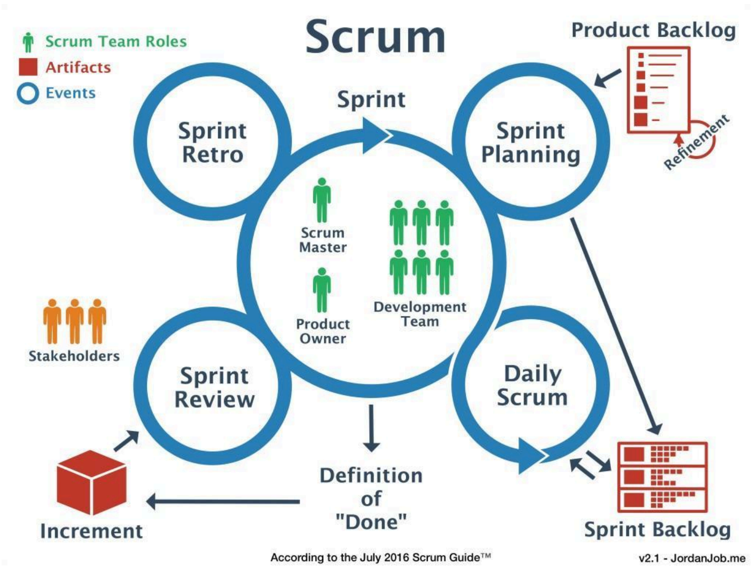
- **Individuals and Interactions Over Processes and Tools**: Emphasizes the importance of people and their collaboration rather than relying solely on processes and tools.
- **Working Software Over Comprehensive Documentation**: Prioritizes delivering functional software over creating extensive documentation.
- **Customer Collaboration Over Contract Negotiation**: Values ongoing collaboration with customers more than adhering strictly to contract terms.
- **Responding to Change Over Following a Plan**: Focuses on adapting to changes rather than rigidly following a set plan.
- **Simplicity in Product and Process**: Advocates for simplicity in both the product and development process.

Pros and Cons:

Advantages	Disadvantages
Realistic approach to development	Not suitable for complex dependencies
Teamwork and Cross training	Risk of sustainability and maintainability
Rapid Development and Demonstration	Depends on customer interaction
Minimum and Flexible requirements	High individual dependency
Minimal rules and easy documentation	
Little to no planning	
Easy to Manage and Flexible	

Scrum Basics:

- **Agile Methodology/Framework:** A framework within Agile methodologies.
- **Iterative Approach:** Uses iterative development to enhance software projects.
- **Application of Agile Practices:** Provides a structured way to implement Agile practices.
- **Lightweight Framework:** A flexible and lightweight framework suited for managing and controlling iterative and incremental projects.



1. **Roles:**
   - **Scrum Team:** Consists of cross-functional, self-organizing members responsible for delivering shippable increments.
   - **Scrum Master:** Acts as a facilitator, not a manager. Removes impediments, facilitates meetings, and ensures adherence to Scrum practices.
   - **Product Owner:** Represents the stakeholder or team, manages the product backlog, and prioritizes features.
2. **Events:**
   - **Sprint:** A fixed-length iteration (2-4 weeks) that produces potentially shippable code. Multiple sprints can be part of a single release.
   - **Sprint Planning Meeting:** Defines which product backlog items will be worked on during the sprint, sets goals, and outlines how to achieve them.

- **Attendees:**
  - **Product Owner:** Prioritizes backlog items and proposes the sprint goal.
  - **Development Team:** Decides which backlog items can be completed and how.
  - **Scrum Master:** Facilitates the meeting, ensuring agreement on goals and scope.
- **Inputs:**
  - Product backlog
  - Team capacity
  - Past performance
- **Activities:**
  - Set sprint goal
  - Select stories
  - Plan capacity

- **Daily Scrum Meeting:** Brief daily meeting to discuss:
  - What was done yesterday?
  - What will be done today?
  - Any obstacles?
  - Update status on Scrum board/burndown chart.
- **Sprint Review:** Demonstrates completed features, evaluates against preset criteria, and gathers feedback from clients and stakeholders to ensure the increment meets business needs and helps reprioritize the product backlog.
- **Sprint Retrospective:** Final team meeting to review the past sprint, discuss improvements, and plan for the future. Led by the Scrum Master.

3. **Artifacts:**
   - **Product Backlog:** A prioritized list of project/product features, including new features, changes, bug fixes, and infrastructure setup. Features are described as user stories, estimated, and ranked.

- **Sprint Backlog:** Features and stories planned for the current sprint, derived from the product backlog.

4. **Alignment with Agile Manifesto:**

   - **Individuals and Interactions Over Processes and Tools:** Emphasizes cross-functional teams, Scrum meetings, and sprint reviews.
   - **Working Software Over Comprehensive Documentation:** Delivers periodic customer-ready increments for review and experience.
   - **Customer Collaboration Over Contract Negotiation:** Involves customers in reviewing sprint outcomes and participating in sprint reviews.
   - **Responding to Change Over Following a Plan:** Accommodates requirement changes through user stories and iterative planning.
   - **Focus on Simplicity:** Short-term planning and straightforward processes keep the approach simple and effective.

Extreme Programming (XP):

1. **Overview**: XP focuses on delivering high-priority user stories as tested software in frequent iterations. It emphasizes working software, continuous feedback, and adaptability to changes.
2. **Key Practices**:

- **Planning Game**: Defines the scope of the next release.
- **Small Releases**: Delivers a simple, functional system in small increments.
- **Communication**: Ensures clear communication of requirements within a small, cohesive team.
- **Simple Design**: Keeps design focused only on the current user story.
- **Customer Involvement**: Involves the customer on-site and continuously throughout development.
- **Feedback**: Gathers feedback through unit testing, customer acceptance, and team discussions.
- **Pair Programming**: Two developers collaborate on a single task.
- **Refactoring**: Updates or discards outdated solutions to improve code.
- **Continuous Integration**: Integrates code multiple times a day.

- **Collective Code Ownership**: Allows any team member to modify the codebase.
- **Coding Standards**: Adheres to standards for better communication and consistency.
- **Metaphor**: Uses a shared vision and common terminology to address issues.
- **Sustainable Pace**: Maintains a standard work schedule to avoid burnout.

3. **Usage**:
   - When requirements are unclear.
   - For smaller systems.
   - When the customer is available on-site.

Lean Agile:

Lean Agile builds on Agile principles but emphasizes faster delivery and sustainable results by focusing on efficiency and continuous improvement.

- **Value Stream Mapping**: Identifies and eliminates activities that do not add value from the customer's perspective.
- **Kaizen**: Embraces continuous inspection and incremental improvement to enhance performance.
- **Active Learning**: Encourages ongoing learning and adaptation.
- **Delayed Decisions**: Advocates making decisions as late as possible to keep options open and adapt to evolving requirements.

Component-Based Software Engineering (CBSE):

CBSE is a reuse-oriented approach focused on defining, implementing, or selecting off-the-shelf components and integrating loosely coupled, independent components into a complete system.

Components are more abstract than object classes and can be considered to be stand-alone service providers. They can exist as stand-alone entities.

- **Motivation**:
  - **Complexity Management**: Addresses the increasing complexity of systems.
  - **Faster Turnaround**: Aims to reduce development time by reusing existing components rather than creating new ones from scratch.
  - **Efficiency**: Promotes efficient system development and maintenance through component reuse.

Pros & Cons:

Advantages	Disadvantages
Black-Box component reduces complexity	Trustworthiness of components
Reduced development time	Component certification
Increased quality and productivity	Emergent property prediction
	Requirement trade off

Essentials for Successful Component-Based Software Engineering (CBSE):

- **Independent Components**: Components must be fully defined by their public interfaces, allowing for clear, standalone functionality.
- **Component Standards**: Adherence to standards that simplify the integration of various components.
- **Middleware**: Utilizes middleware to support and facilitate the integration of components.
- **Development Process**: The development process should be aligned with CBSE practices to ensure effective component utilization.

Software Component:

- **Functionality**: Implements specific functionality regardless of execution environment or programming language.
- **Independence**: Operates as a standalone executable entity, which may consist of one or more executable objects.

- **Interface**: The component interface is publicly available, with all interactions occurring through this defined interface.

Component Interaction:

- **Explicit Dependencies**: Dependencies are specified through "required" interfaces, which detail the component's interaction needs.
- **Component Lifecycle**:
    - **Identification**: Identifying potential software components.
    - **Search and Selection**: Searching for and selecting the appropriate components.
    - **Composition**: If no suitable component exists, composite components are created from existing ones, which may involve sequential composition or the use of adapters or "glue" to reconcile differing interfaces.
    - **Validation**: Ensuring the selected or composed component meets the required specifications.

Component Development Stages:

- **Forms of Component**:
    - **During Development**: Represented using UML diagrams.
    - **Packaging**: Delivered as .zip files or similar packages.
    - **Execution**: Consists of code and data blocks ready for execution.

Elements of a Component Model:

- **Interfaces**: Defines how the component interacts, including operation names, parameters, and exceptions, all detailed in the interface definition.
- **Usage**: Each component has a globally unique name and handle for identification and access.
- **Deployment**: Specifies how the component should be packaged and deployed for use in various environments.

Software Product Lines:

These are engineering techniques for **creating a portfolio of similar software systems** from a **shared set of software assets** where

- **Reuse** is imperative
- Represent a **family of manufactured product**

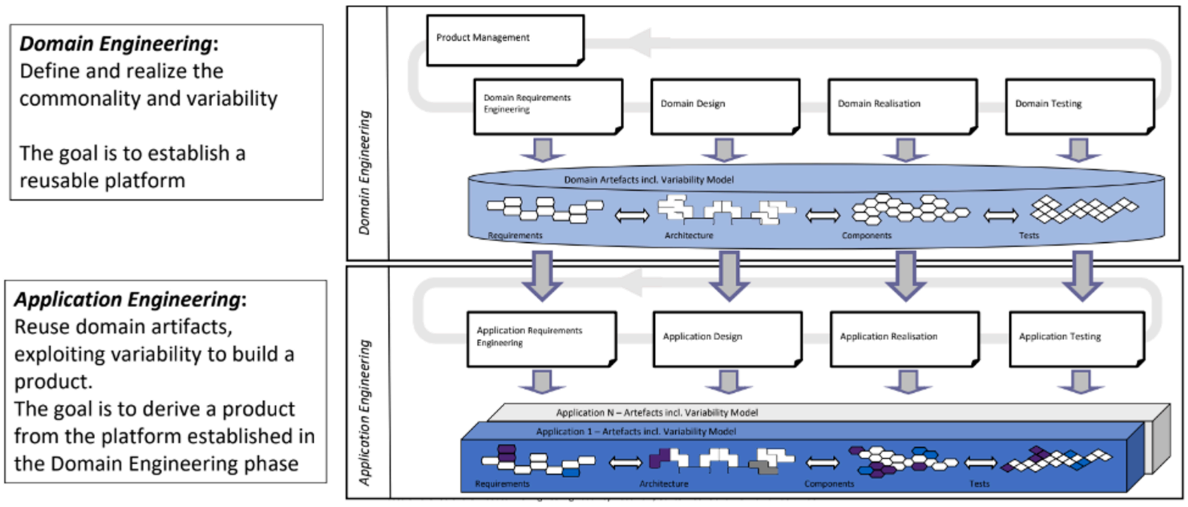
Product Line Architecture: This captures **commonality and variability** of product line components and compositions.

- **Advantages**
    1) Create **software for different products**
    2) **User variability** to customise software to each product

Key Drivers for Reuse:

- **Predictive** software reuse
- Software **artifacts created when reuse is predicted** in one or more products
- Artifacts could be **built as components or design patterns**

Engineering Framework:


Domain Engineering: Define and realize the commonality and variability

The goal is to establish a reusable platform

Application Engineering: Reuse domain artifacts, exploiting variability to build a product.
The goal is to derive a product from the platform established in the Domain Engineering phase

Requirements Engineering:

1) **First step** in any software intensive development lifecycle
2) Difficult, error prone and costly
3) **Errors** introduced in this phase may **propagate to downstream phases** and this makes it **expensive to fix**
4) **Critical** for successful development of all downstream activities

What is a requirement?

A software product/project requirement is defined as a property which **must be exhibited** by software developed/adapted to solve a particular problem.

It should specify **externally visible behaviour** of what the product does and not how the behaviour is achieved. This can be individual or set of requirements.

Properties of Requirements:

1) **Clear:** precise and simple language; must use active present tense; constant terminology; avoid combining
2) **Concise:** describe a single property with as few words as possible
3) **Consistent**: no requirements should contradict each other
4) **Unambiguous:** should only have one interpretation
5) **Feasible:** realizable in specified time frame
6) **Traceable:** backwards to stakeholder request and forwards to software components
7) **Verifiable:** clear, testable criterion and cost-effective process to check it has been realised
8) **Prioritized**: this helps in achieving ordered workflow
9) **Quantifiable:** aids in testing and verifying

Feasibility Study:

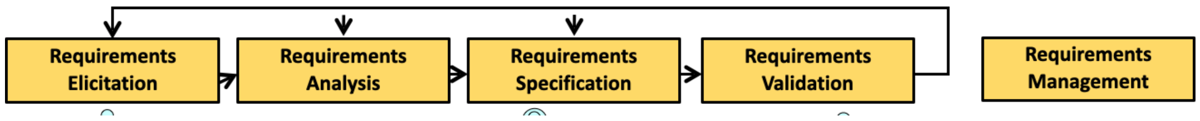
It is defined as a **short, low cost study** conducted before the beginning of the project to assess its **practicality.**

Activities conducted during a feasibility study:

1) **Figure** out **Client/sponsor/user** would have a **stake** in project
2) **Current solution** to the problem?
3) **Target customers** and future market space?
4) Potential **benefits**?
5) **Scope** on a high-level
6) **High block level understanding** of solution
7) Considerations in tech
8) **Marketing** strategy
9) **Financial** projections

10) **Schedule and high level planning**; budget requirements
11) Issues, assumptions, risks and constraints
12) **Alternatives** and their consideration
13) Potential project organisation

Requirements Engineering Process:



1) **Step 1: Requirements Elicitation**

   This phase involves working with all **stakeholders** to **gather their needs**, **articulate** the problem, **identify and negotiate conflicts** and establish a **clear scope and boundary** for the project.

   In this phase,

   1) Stakeholder has been given an opportunity to **explain their problem** and needs
   2) Involves understanding **problem, domain, needs** and **constraints;** identifying **clear objectives and writing business objectives** for the project.

Elicitation Techniques:

Based on the nature of the system being developed and the background experience of stakeholders, techniques are divided into two types:

1) **Active**
- **Ongoing interaction** b/w stakeholders and users
- Interviews; facilitated meeting
- **Role-playing**; prototyping
- **Ethnography** and scenarios

2) **Passive**
- **Infrequent** interaction
- **Use cases;** business process analysis and modelling
- Workflows and questionnaires
- **Checklist and documentation**

2) Step 2: Requirements Analysis

This phase involves understanding **requirements** from both a product **and process perspective. It also involves classifying and organising requirements** into coherent clusters.

Requirements are classified into:

1) **Functional, non-functional and domain:**
   - **Functional:** functionality or services the **system should provide** with different **inputs and expression** on how the system should behave
   - **Non-functional:** constraints on service or functions offered by the system (timing, dev process, etc)
     - Often applied to the system as a whole
     - Specify the **criteria** that is used to judge the operation
   - **Domain:** constraints on system from domain of the operation
2) **System and user:**
   - **User:** statements in natural language + informal context diagrams system/subsystem and their interconnections and operational constraints
   - **System**: structured document; detailed desc of systems functions, services and operational constraints.

This phase also involves modelling the requirements using Unified Modelling Language which is used to visualize, construct, specify and design code.

Primary goals of modelling:

1) provide an understanding of the system
2) analyze and validate requirements
3) communicate the requirements in terms of who, what and interpreting it the same way

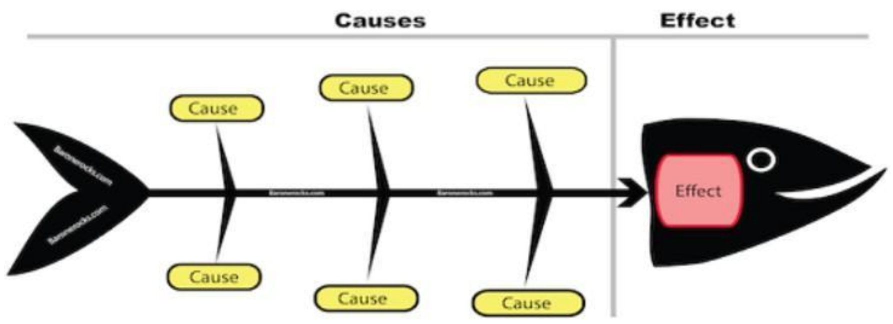
Types of UML models:

1) **Structural**
   - Static aspects of the system
   - Entities in the system
   - How are they related

2) **Behavioural**

   - Dynamic aspects of the system
   - How do entities respond to stimulus

In the analysis phase, use the fishbone diagram to visualize requirements w.r.t causes and effects.



Activities conducted during analysis phase:

- **Recognise and resolve conflicts** (functionality vs time vs cost)
- **Negotiate requirements**
- **Prioritise requirements** (MoSCoW - Must have, Should have, Could have, Won't have)
- **Identify risks** if any
- Decide on **build or buy** and **refine requirements**
  - **Commercial Off The Shelf**

3) **Step 3: Requirements Specification**

This step involves the **documentation** of a set of requirements that is **reviewed and approved** by the **customer** and **provides direction** for software construction activities.

This is usually achieved through a document called the **Software Requirements Specification** (SRS) which serves as:

- basis for customers and contractors/suppliers **agreeing on what they will and will not do**;
- specification for functional and non-functional requirements

Why do we document?

1) **Visibility**
2) **Formalisation** leads to better clarity
3) **User support**
4) **Team communication**
5) **Maintenance and evolution**

Characteristics of Documentation:

1) **Accurate** and **current**
2) **Appropriate** for audience
3) **Maintained online**
4) **Simple** but professional

What should be included in documentation:

1) **Functionality:** what is the **software supposed to do**
2) **External interface:** how does it **interact** with people, hardware and software
3) **Non functionality: quality criteria** for functionality
4) **Performance:** speed, response time, recovery time, etc
5) **Other attributes**: availability, portability, correctness, maintainability
6) Design constraints imposed

- **Required standards** in effect
- Implementation **language**
- Policies for **DataBase integrity**
- **Resource limit**
- **Security**
- **Operating environment**

4) Step 4: Requirements Validation & Verification

When there is an error in requirements engineering, **repairing** it **downstream** can be **expensive.** Hence, we must perform rigorous validation and verification of the requirements gathered.

- **Validation**: Checking whether the software requirements if implemented will **solve the right problem** and satisfy the user's need.
- **Verification:** Checking whether requirements have been **specified correctly.**

Review whether the requirement follows these properties:

- Consistency
- Completeness

- Verifiability
- Comprehensibility
- Traceability
- Adaptability

Prototyping:

- Facilitates **user involvement** and ensures users and engineers have **same interpretation**
- Most **beneficial** in systems with **many user interactions**

Model validation:

- Model represents **all essential functional requirements** (Use case model)
- Demonstrating **each model is consistent** (flow diagrams)
- **Fishbone analysis** to validate if **requirements addresses the reasons for needing a solution** to the problem

Acceptance criteria:

- See if any requirements match the acceptance criteria

NOTE: First 4 steps are iterative.

5) Step 5: Requirements Management

Sometimes, due to some shift in plans, requirements may change. Developers should be prepared to incorporate these changes and have a plan ready. This is known as requirements management.

Potential reasons for requirement change:

- **Better understanding** of the **problem**
- **Customer internalizing**
- **Evolving environment** and **technology**

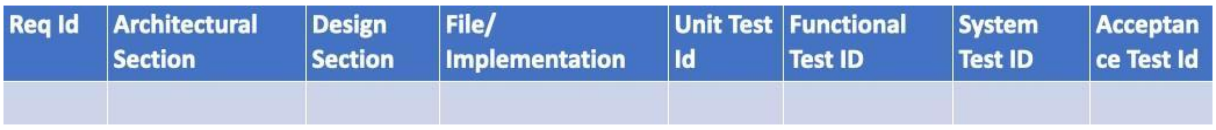
Management involves two facets

- Ensure requirement changes are **addressed in all phases** of the life cycle
- Ensure **changes in requirements are handled appropriately**

Requirements Traceability Matrix (RTM):

It is a document that demonstrates the relationship between requirements and other artifacts. It's used to prove that requirements have been fulfilled. And it typically documents requirements, tests, test results, and issues

- Each phase in the life cycle **progressively fills RTM**
- Requirements **traced along SDLC** using the RTM



Req Id	Architectural Section	Design Section	File/ Implementation	Unit Test Id	Functional Test ID	System Test ID	Acceptance Test Id

Types of Requirements Tracing:

1) **Forward tracing:**
Forward tracing involves tracking the progression of requirements from their origin through the various stages of development, such as design, implementation, and testing. It ensures that each requirement is addressed in the final product by verifying that all necessary design elements, code, and tests are in place. This helps to confirm that the project is on track to meet all specified requirements.

2) **Backward tracing**:
Backward tracing is the process of verifying that each component of the final product, such as code, design, and test cases, can be traced back to specific original requirements. This ensures that every element in the project is justified by a requirement, preventing unnecessary or out-of-scope features from being included. Backward tracing also helps in validating that the product fully satisfies the initial requirements set by stakeholders.

Requirements Change Management:

- **Change** in requirements will **impact plans, work products**, etc
- Uncontrolled changes can have **huge, adverse impac**t (cost, schedule, quality and expectation)
- **Ensure only controlled changes by using a formal change management process as below:**
    1) **Log the request** for change and **assign change request ID**
    2) Info to **log:** who is requesting? Why is the request coming in? What is being requested?
    3) **Perform impact analysis** on work products and items
    4) Estimate impact on scope, effort and schedule
    5) **Review impact** with stakeholders
    6) Solicit formal approval
    7) **Rework** the work products/items
    8) Log the following

  - When was it changed
  - Who all made changes? Reviewed changes? Tested changes?
  - Which release stream is it a part of?

STUDENTS TO NOTE:

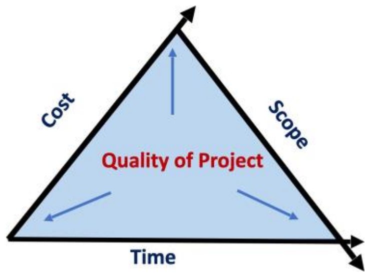
This should not be your only source of preparation. Refer lecture slides, textbooks, reference books, etc. Below are some sources for better information:

1) https://qarea.com/blog/software-development-life-cycle-guide
2) https://xebrio.com/requirements-engineering/
3) https://asana.com/resources/agile-methodology

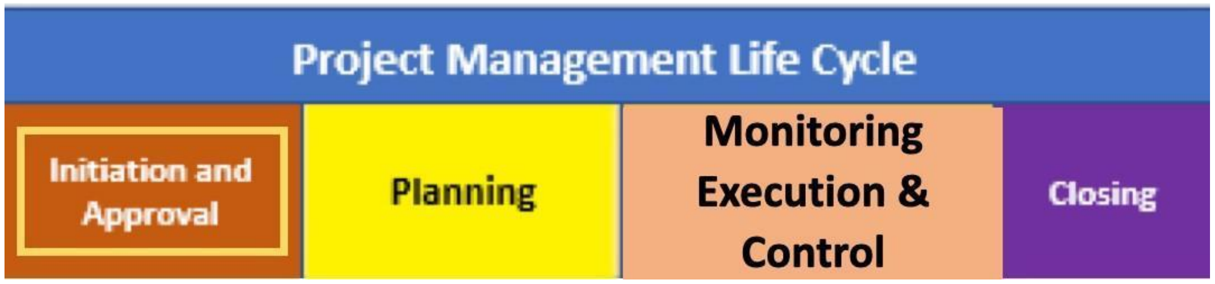
Unit-2

Software project management fundamentals

- **Need for project management**
  - Large projects have **lots of people** working **together** for a **prolonged time**
  - **Coordination** to ensure **integration** and **interoperability**
  - **Business and the Environment change frequently** and rapidly
- **Software project**:
  - individual or collaborative enterprises
  - **carefully planned** to **achieve a certain aim** / **create a unique product** or service
  - **Characteristics of a project**
    - Made up of **unique activities** which do not repeat
    - **Goal specific**
    - **Sequence of activities** to deliver end-product
    - **Time bound**
    - **Inter-related activities**
    - **Need adequate resources** (time, manpower,finance, knowledge-bank)
    - **Intangible;** can claim 90% completion without visible outcomes
- **Software project management:** planning and supervising projects;
  - **planning**, **execution** of plans, **monitoring** and **controlling** projects
- **Project management: planning**, **organising**, **motivating**, **controlling resources** to **achieve certain goals**; project workflow with team collaboration;
  - Planning, scheduling, monitoring, risk management, managing quality and people performance
  - Considers these **3 constraints in equilibrium**



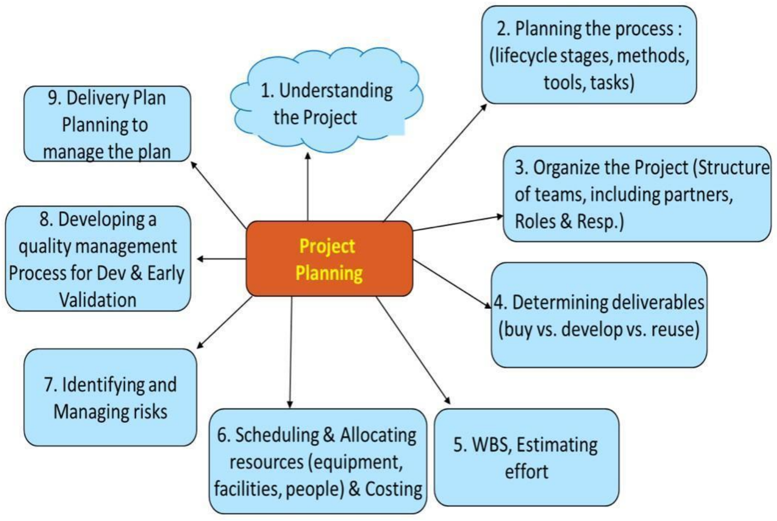
Software project management lifecycle



Project Management Life Cycle

Initiation and Approval	Planning	Monitoring Execution & Control	Closing

- **Initiation and approval**:
  - **Initiation**: happens at **approval or a "go" feasibility study** formally kicking-off
  - Opportunity or reason for project justifies its kick-off
  - A **charter** is created
    - **Purpose** of the project
    - How will it be **structured and executed**
    - **Vision, objectives**
    - **High level scope**
    - **Deliverables**
    - **Responsibilities** of the project teams and stakeholders
  - Initial **project owner/manager**, **budget** along with resources
- **Planning**
  - Focus on **what needs to be achieved and how**
  - **Looking ahead** and **making provisions** for required resources

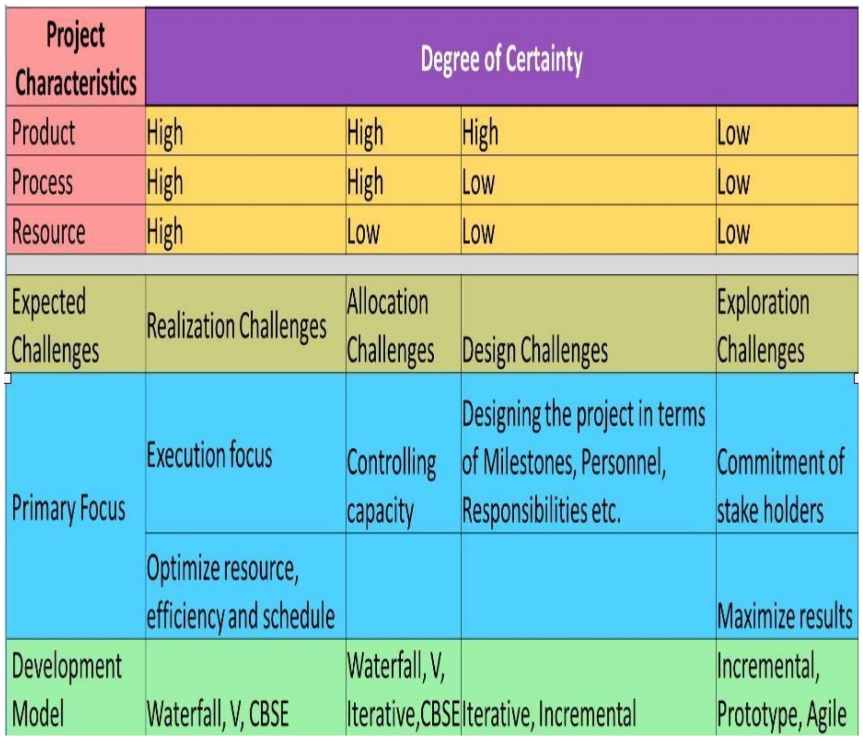

Project Planning
1. Understanding the Project
2. Planning the process : (lifecycle stages, methods, tools, tasks)
3. Organize the Project (Structure of teams, including partners, Roles & Resp.)
4. Determining deliverables (buy vs. develop vs. reuse)
5. WBS, Estimating effort
6. Scheduling & Allocating resources (equipment, facilities, people) & Costing
7. Identifying and Managing risks
8. Developing a quality management Process for Dev & Early Validation
9. Delivery Plan Planning to manage the plan

- **Outcome:** Project plan
  - Level **depends on nature of project** (exploratory, research, development)
  - **Evolves** based on progress, context, risks, etc
  - **Perspectives**

Sponsors	Customer	Execution Stakeholder
Projects role in organization	Team understand the problem?	Software Lifecycle to follow
Address customer requirement	Time to develop	Project Organisation (roles)
Investment vs Revenue	Cost of Solution	Standards, Guidelines, Procedures
Time expenditure and Risks	Delivery plans	Communication Mechanism
Responsibility and Progress Tracking	Metrics to indicate quality	Criteria to Prioritise requirements
Resources and Deliverables	Exit Criteria	Work breakdown and Ownership
Exit Criteria	Project support	
	Interaction/Review plans	

- **Stages of Project Planning**

- **Understanding expected deliverables** of the project
  - **Customer and stakeholder** expectations
  - **Market forces** driving the project (new tech, competition)
  - **High-level decisions** (Make-Buy-Reuse)
  - Supported by **feasibility study** and **requirements elicited**
- **Planning the process**
  - **Choice of lifecycle** is based on: activities, goals, time and so on
    - **Degree of certainty** (high or low) of **product, process, resource**
  - **Models, standards, guidelines and procedures**
    - **Standards**: Technical, interoperability, quality, regulatory
    - Config management, change management, quality plans



Project Characteristics	Degree of Certainty			
Product	High	High	High	Low
Process	High	High	Low	Low
Resource	High	Low	Low	Low
Expected Challenges	Realization Challenges	Allocation Challenges	Design Challenges	Exploration Challenges
Primary Focus	Execution focus	Controlling capacity	Designing the project in terms of Milestones, Personnel, Responsibilities etc.	Commitment of stake holders
	Optimize resource, efficiency and schedule			Maximize results
Development Model	Waterfall, V, CBSE	Waterfall, V, Iterative,CBSE	Iterative, Incremental	Incremental, Prototype, Agile

- **Organize the project**

- **Structure:** organise structure in terms of **people, team and responsibilities**
  - Eg: project manager, programmer, architects and so on
  - **Hierarchical , Flat, Functional, Matrix, Line orgs**
- **Partners**
  - For project build, install, localisation, documentation, product management, product marketing, sales, pre-sales, support
  - **Downstream or Upstream**

- **Determine deliverables**
  - **Buy, develop, reuse**
- **Work break down (WBS)**
  - **Project activities** split into **smaller deliverable components** (iterative)
  - Split till **enough granularity**
  - **Hierarchical tree** structure: WBS (Work breakdown structure)
  - Aggregate tasks into **phases**
  - **Milestone, checkpoints** and so
  - **Entry and exit criteria**: Phases, milestones, etc
  - Identification of **work packages for final product**
  - **Estimation of tasks/activities** (effort and time)
    - Estimation helps in **efficient and effective control**
    - **WBS is estimation** of tasks in project
    - Estimation is done for **size and effort of WBS tasks** (time, cost and so on)

    - **Common estimates:**
      - **Lines of Code** (LOC)
      - **Function points**: business functionality in terms of functions
    - Top down or bottoms up
    - Estimation could be

- **Experience based**: expert judgement, comparative studies

  - Delphi, Modified Delphi

- **Empirical estimation:** Formula derived from past projects (size, process char, experience)

  - **CoCoMo (Constructive Cost Model)**

    - Three categories of projects
      - **Organic:** Small team, problem well understood, experienced people
      - **Embedded:** Large team, complex problem, need people with sufficient experience
      - **Semi-detached**: In between
    - **Estimation approaches**

$$\text{Effort } E = a_b(\text{KLOC})^{b} \text{ Person months}$$

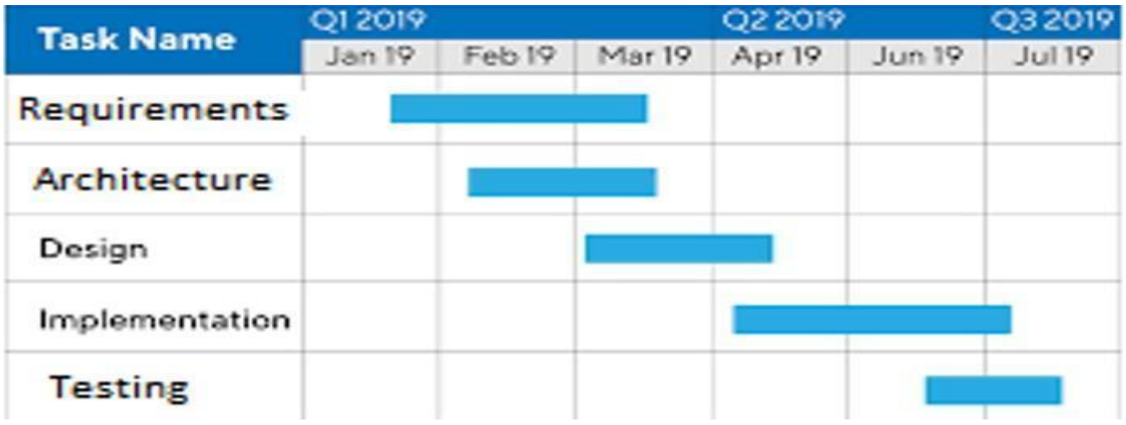
$$\text{Time } T = c_b * (\text{efforts}) d_b \text{ Months}$$

- **Types of CoCoMo**
  - **Basic:** rough calculations
  - **Intermediate:** considers cost drivers
  - **Detailed:** factors dependencies

- **Scheduling and allocating resources**
  - Based on **outcomes of WBS and Estimation**
  - **Calendarization** of work activities
  - **Activities**
    - Brings **concerned individuals** to participate in **building schedule**
    - **Identification and allocation of resources** to WBS tasks
      - Hardware, software, humman

- **Rework estimates;** schedule based on competencies

- **Validate** upstream and downstream dependencies
- **Organize concurrently** to **optimise people** and **sequentially** for **dependencies**
- **Graphic tools**: gantt chart



- **Identify schedule risks** and **mitigation plans**
- **Minimise task dependency** (avoids delays)
- Factor **working conditions** (holidays, work shifts, etc)
- **Multiple iterations:** Due to feature requirements, Schedule, Resources, people or cost
- **Cost** (Budget, capital and expenses)

- **Identify and Manage risks:**
  - **Risk**: **unexpected** event that might **impact** people, tech and resources
  - Can be for **Products, projects, orgs**, etc
- **Steps**
  - **Identify** risk (what may go wrong)
  - **Assessment and analysis** (probability of occurrence and impact)

- **Mitigation**/Fall back plan
- **Identification** and **catching** the **trigger**
- **Execute** and **monitor mitigation** plan
- **Develop Quality management process**
    - Plans for **tracking progress**
    - **Communicatio**n plans
    - **Quality assurance** plans
- **Test completion criteria**
    - **Procedure for release** to customer
- **Early verification**/customer validation (Beta testing)
- **Plans for Tracking Project Plan and Delivery Plan**
    - Plans for **managing project plan**
    - **Procedure for release** to customer

- **Contents of Project Plan**
    - Introduction
    - Deliverables
    - Process model
    - Organisation
    - Standards, Guidelines, Procedures
    - Management activities
    - Risks
    - Staffing
    - Methods and Techniques
    - Quality Criteria/Assurance
    - Work Packages
    - Resources
    - Budget and Schedule
    - Change control Process

- Delivery Means

- **Monitoring and Control**
  - Processes performed to **observe project execution** so that **potential problems** can be **identified promptly** and **corrective action** can be taken
  - Begins once plan is created, runs **in parallel with execution**
  - **Continuously performed**; tasks, measures and metrics to **ensure project is on track** (time, budget, risk)
  - **Quantitative data**
  - **Checkpoints, milestones, toll-gate**s
  - **Project Management and Control dimensions**
    - **Time:** in terms of **effort** (man-months) and **schedule**
      - **Brooks law**: adding people to a late project delays it further
      - **Development models** help in managing time
    - **Information** (availability, propagation): **communication** including documentation
      - **Current state** and **changes agreed** upon
      - **Agile:** focuses on tacit knowledge held by people instead of docs
    - **Organisation** and structure, **roles and responsibilities**
      - Building a **team**
      - **Reorganising** structures
      - Clarify and manage **expectations**
      - **Reorient** roles and responsibilities
  - **Quality is built-in**
    - Designed in, not an afterthought
    - Frequent **interaction with stakeholders**
    - **Quality requirements** may cause **conflict**
  - **Cost, infrastructure and personnel** (capital and expense)

- **Includes**
  - **Monitoring and controlling** project work: **collect measures** and make **corrective/preventive actions**
  - Ensure **change controls** are meticulously followed
  - Scope, deliverables, documents are **updated**
  - **Control quality triangle** (Recollect triangle from page 1)
  - **Manage** team, performance and communication
- **Closure**
  - **Close project** and **report success** to sponsor
- **Steps involved**
  - **Handover deliverable**s to customer and **obtain project/UAT** (User acceptance testing) sign-off from client
    - **Complete** and **pass on documentation**; **cancel supplier** contracts
  - **Release staff**, equipment; inform stakeholders
  - **Post mortem:** determine projects success and lessons learned

---

Software architecture

- **Architectural design: decomposition and organization** of software into **components**
- **Detailed design: specific behavior** of components in Architectural design
- Well written software is **more maintainable**
- To become a **master of software development**
  - Need to **learn the rules**
    - Algorithms, data structures, languages of software
  - **Learn the principles**
    - Structured programming, modular programming
    - Object oriented programming, generic programming

- **Study architecture and designs of other masters**
  - Contain **patterns** that must be understood, memorised and used

- **Software architecture**
  - Top level **decomposition** into **major components** with a characterisation of **how they interact**
  - **Big picture depiction** of the system
    - Used for **communications among stakeholders**
    - **Blueprint** for system and project development
    - **Negotiations and balancing** of **functional and quality goals**

- **Importance of architecture**
  - **Manifests earliest design decisions**
    - **Constraints** on implementation
    - Dictates **organisational structure**
    - **Inhibits or enable quality attributes**, supports WBS
  - **Reuse** at architectural system level
  - **Helps in WBS** (reduce risk, enable cost reduction)
  - **Changes** to architecture **is expensive in later stages** of SDLC
- **Characteristics of software architecture**
  - Address **variety** of stakeholder **perspectives**
  - **Realises all of the use cases and scenarios**
  - Supports **separation of concerns**
  - **Quality driven**
  - **Recurring styles**
  - **Conceptual integrity**
- **Factors that influence Software Architecture**
  - Functional requirements
  - Data profile

- Audience
- Usage characteristics
- Business priority
- Regulatory/Legal obligations
- Architectural standards
- Dependencies and Integration
- Cost constraints
- Initial state
- Architect and staff background
- Technical and organisational environment
- Technical constraints
- Types of user experience

- **Software Architecture Design Factors**
    - Environment
    - User Requirements
    - Business Strategy
    - Software Design
    - Software Quality
- **Architect**
    - Makes **high-level design choices** and **dictates tech standards**
        - Coding standards, tools and platforms
- **Architectural views, styles and patterns**
    - **Views:** ways of **describing software architecture**, enables **different stakeholders** to **view** it from the **perspective** of their interests
        - **Eg**: UI view, Process view
        - Purpose of Views:
            - Communication
            - Documentation
            - Analysis

- **Styles:** how the **subsystems and elements are organised**
  - Way of **organizing code**; (Pipe & Filters, Client-Server, Peer-to-Peer)

Architectural Views	View Points
Representations of different aspects of the system's architecture.	Templates or perspectives for creating architectural views.
To illustrate and analyze specific concerns of the system.	To guide the creation and organization of views.
Specific to individual aspects like performance or design.	Broader, guiding overall view creation strategies.
Used to document and communicate system design.	Used to ensure comprehensive coverage of concerns.
Addresses specific stakeholder needs related to system design.	Provides methodologies to address multiple stakeholder needs.
ex: Logical view, development view, physical view.	ex: 4+1 View Model, C4 Model.

- **Pattern: known or proven approach** of **structuring and functioning**
  - Exists to "solve" a problem
  - **Eg:** MVC solves problem of separating UI from the rest

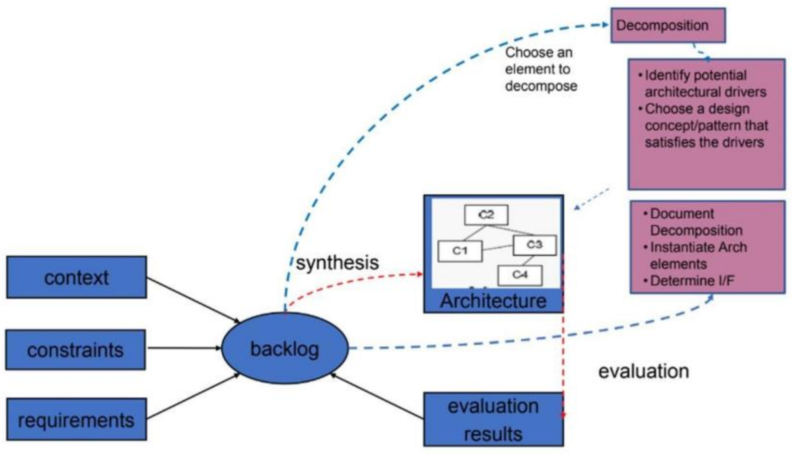
- **Architectural Styles and Patterns**
  - **Service-Oriented Architecture (SOA):** A style where services are provided to the other components by application components, through a network. Commonly used in enterprise systems.
  - **Microservices Architecture:** An evolution of SOA, where applications are structured as a collection of loosely coupled services, each running in its own process and communicating via lightweight mechanisms.

- **Event-Driven Architecture:** A design paradigm that orchestrates behavior around the production, detection, and reaction to events. Often used in real-time or asynchronous processing systems.
- **Layered Architecture:** Organizes code into layers, such as presentation, business logic, and data access layers. Each layer has a specific role and communicates with adjacent layers.

- **Architectural conflicts**

  - **Large-grain components improves performance** but **reduces maintainability**
  - **Redundant data**: improves **availability**; makes **security and integrity** difficult
  - **Localising** improves **safety** but **communication** degrades performance

- **Generalised model for architecting**


Decomposition
Choose an element to decompose
• Identify potential architectural drivers
• Choose a design concept/pattern that satisfies the drivers
C2
C1
C3
C4
Architecture
• Document Decomposition
• Instantiate Arch elements
• Determine I/F
synthesis
context
constraints
requirements
backlog
evaluation
evaluation results

- **Common themes of architecture**

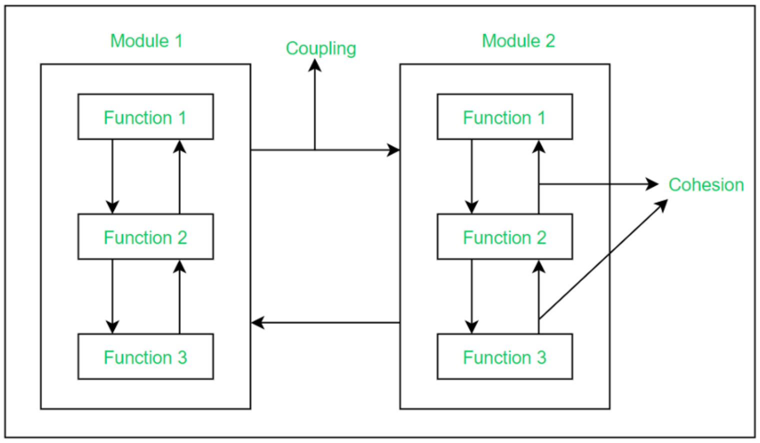
- **Decomposition**: problem -> individual modules/components
  - **Approaches**:
    - **Based on layering**
      - Order **system into layers**; each layer **consumers service from lower** layer and **provides service to higher layers**
      - **Ordering of abstractions**
      - **Eg:** TCP/IP model

- **Based on distribution** (computational resource)
  - **Dedicated task** owns **thread of control;** process need not wait
  - Many clients need access
  - **Greater fault isolation**
- **Based on exposure**
  - How is the **component exposed** and **consumes other components**
  - Service offered, logic and integration
- **Based on functionality**
  - **Grouping with problem domain** and s**eparate based on functions**
    - **Eg:** login module, customer module, so on
  - Mindset of operational process
- **Based on generality**
  - Components which **can be used in other places** as well
- **Based on Volatility**
  - Identify **parts which may change** and **keep them together** (UI)
- **Based on Configuration**
  - Look at **target** for features needing to **support different configurations**
  - **Eg:** security, performance, usability
- **Based on Coupling:** keeping things together; Cohesion: things that work together
  - **Low coupling and high cohesion (good software always obeys this)**
- **Other approaches**
  - **Divide and conquer**
  - **Stepwise refinement**: simple solution and enhance
  - **Top-down approach:** overview of system, detail the subsystems
  - **Bottom-up approach:** specify individual elements and compose
  - **Information hiding**

- **Architectural views**

- **4 ways to view**

  - **Structure of modules** (Module view point)



- **Module:** unit of code with a functional responsibility
- **Structure system as set of code units**
- Architect enumerates **what units of software will have to do** and assigns **each item to a module**
- **Larger module** may be divided into **smaller modules**
- **Less emphasis** on how **software manifests at runtime**

  - **Component-and connector structure** (View point)

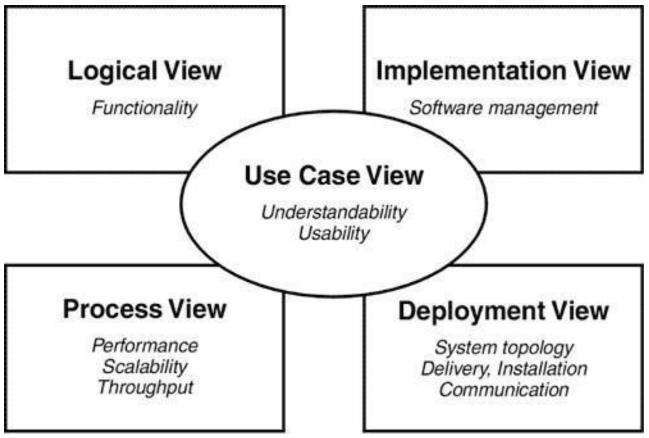
- **Dynamic view** of system in **execution/runtime**
  - **Eg:** process view includes set of processes connected by sync links
- **Components/processing element**: software structure that **converts input to Output**; series of processes
  - **Computational**: does computation; Eg: function, filter
  - **Manager:** contains state+operations; State retained between invocations
  - **Controller:** governs time sequence of events
- **Connecting elements: glue** to component and data element
  - **Communication and sync link**
  - **Eg:** Procedure calls, RPCs, comms protocols, etc

- **Data element: information** needed **for processing/to be processed**
  - Memory in data; persistent

- **As allocation structure**
  - **Deployment structure: software** assigned to **hardware and communication paths**
    - **Relations:**
      - **"allocated-to"**: shows on which **physical units software elements reside**
      - **"Migrates-to"**: if allocation is **dynamic**
    - Allows to reason about **performance, data integrity, availability, security**
    - Important in **distributed or parallel systems**
  - **Implementation structure**
    - How **software is mapped onto file structures** in systems development, integration or config control envs
  - **Work assignment structure**
    - **Who is doing what** and the knowledge needed

- **Krutchens (4+1 view)**
  - **Use case view:** exposing requirements or scenarios
  - **Design view:** exposes vocabulary of problem and solution space
  - **Process view:** dynamic aspects of runtime behaviour
    - Threads and processes; addresses performance, concurrency
  - **Implementation view**: realisation of the system; UML diagrams
  - **Deployment view**: focus on system engineering

- **Architectural styles**



- Way of **organisation of components**, characterised by **features** that make it **notable**
- Used to **construct software modules**
- **Structure and behaviou**r of the system
- **Provides four things:**
  - **Vocabulary:** set of **design elements** (Pipes, filters, client, severs,)
  - **Design rules:** constraints that **dictates** how **processing elements** would be **connected**
  - **Semantic interpretation**: meaning of connected design elements
  - **Analysis**: performed on the system (Deadlock detection, scheduling)
- **Recognised architectural styles**
  - **Main-program with subroutines**
    - **Generic:** traditional language-influenced style
    - **Problem:** system is hierarchy of functions; natural outcome of functional decomposition

      - **Top-level module is the main program** that invokes the rest

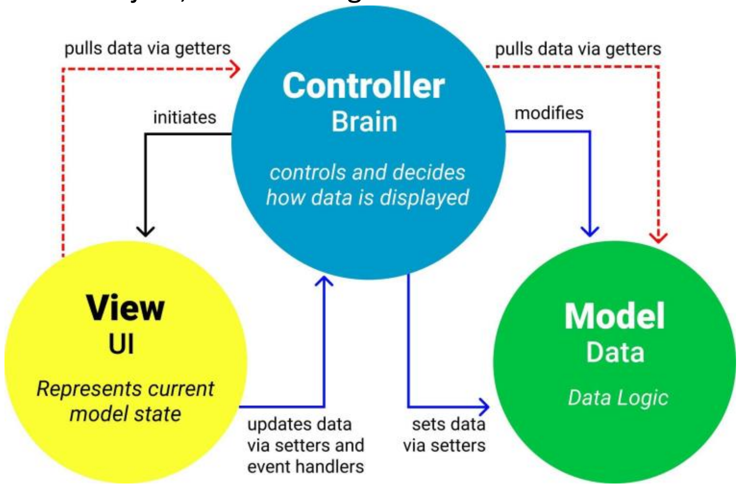
      - Single thread of control
    - **Context:** language with nested procedures
    - **Solution**
      - **System model:** procedures and modules are defined in a **hierarchy**

        - **Higher level modules call the lower level**
        - Hierarchy: strict or weak
      - **Components:** procedures resigning in main program; local and global data
      - **Connectors**: procedure call and shared access to global data
      - **Control structure:** single centralised thread of control
    - Implicit invocation
    - Pipes and filters
    - Repository
    - Layers of abstraction
    - Client server
    - Component based system
    - Service oriented architecture
    - Object oriented arch
- **Architectural pattern**
  - **Proven solution** to a **recurring architectural problem** (structuring, function and solutioning of subsystems)
  - Named collection of architectural designs that
    - Has **resulted in a successful solution** in a given development context
    - **Constrain** such **design decisions** that are specific to a particular system
    - Elicit **beneficial qualities** in each resulting system,
    - **Good starting point** for solution
    - **Number of layers** between user and data
  - **Tiered architecture**
    - **Single tiered or monolithic**
      - Single app layer that supports UI, business rules and manipulation of data
      - **Eg:** microsoft word
      - Used in **client server apps**
    - **Two tiered arch**

- **Client app:** business rules + UI
- **Server app:** data retrieval + manipulation; physically separate system
- Eg: SQL server
- Uses in **traditional client-server App**

- **Three tiered arch/Model - View - Controller**

- **Model**
  - Central component; **application data**, **business rules**, logic and functions; **abstraction layer for features**
  - True domain object; no knowledge of UI



  - **Establish and collect information** to be displayed
  - Notify associated views and controllers of about change in state
- **View**
  - **Output representation** of information (chart or diagram)
  - Graphics design and layout; get **information from model** as prompted by the **controller**
- **Controller**
  - **Accepts input** and **converts it to commands** for model/view
- **Use cases**
  - **UI logic** tends to **change more frequently** than **business logic**
  - App needs to **display same data in different ways**

- Developing UI and Business logic require **different skill sets**; separate development teams easily
- **UI code** is more **device-dependent** than business logic

- **Architectural Risk Management**

- **Risk-Driven Design:** Prioritizing architectural concerns based on risks, focusing on areas that have the most significant potential impact on the project.
- **Technical Debt Management:** Architectural decisions need to consider the trade-offs between immediate functionality and long-term maintainability, managing technical debt strategically.

- **Example of Architectural Document:**

Example : E-commerce application

1. **Introduction**
    1. **Purpose**
       To describe the architecture of the Nico e-commerce system.
    2. **Scope**
       Covers system design, deployment, and performance considerations.
    3. **Definitions, Acronyms and Abbreviations**
       API - Application Programming Interface, DB - Database.
    4. **References**
       [1] Nico System Design Document, [2] E-Commerce Standards Guide.

2. **Architectural Representation**
   The system is represented using the 4+1 View Model.
3. **Architectural Goals and Constraints**
   Ensure scalability, security, and high availability within a budget constraint.
4. **Use-Case View**
    1. **Architecturally-Significant Use Cases**
       User registration, product purchase, and order management.

5. **Logical View**

- **Architecture Overview – Package and Subsystem Layering**

Divided into User Interface, Business Logic, and Data Access layers.

- **Process View**
  - **Processes**
    User management, inventory control, and transaction processing.
  - **Process to Design Elements**
    User management maps to the User Interface and Authentication modules.
  - **Process Model to Design**
    Data flow diagrams showing interactions between modules.

6. Model Dependencies

User Interface depends on the Business Logic layer.

7. Processes to the Implementation

Transaction processing implemented using a microservices architecture.

8. Deployment View

- **External Desktop PC**
  Runs the client application.
- **Desktop PC**
  Hosts the local web server for testing.
- **Registration Server**
  Manages user registrations and authentication.
  - **Course Catalog**
    Provides the list of available products.
- **Billing System**
  Handles payment processing and invoicing.

9. Performance

System designed to handle 10,000 concurrent users with a response time under 2 seconds.

10. Quality

Adheres to industry best practices for security, maintainability, and usability.

Software design

- **Design principles**

- **Further decomposition** post architecture If necessary

- **Description of behaviour** of components/sub-systems as identified in Architecture design
- How **interfaces will be realised** (data structures+algorithms)
- System will **facilitate interactio**n with user
- Use of apt **structural and behavioural design patterns**
- Maintenance and reuse

- **Techniques that enable design**
  - **Abstraction:** focus on essential properties
    - Expose only relevant functions
    - Procedural/data abstraction
  - **Modularity, coupling and cohesion**
    - **Modularity:** degree/extent to which large module is decomposed
      - Best to be self-contained
    - **Cohesion:** extent to which components are dependent on each other
      - Strong is good
      - **Could be**
        - Adhoc
        - Logical (input routines)
        - Temporal (initialisation sequence)
        - Sequential
        - Procedural (read and print)
        - Functional (contribute to same function)
    - **Coupling:** how strongly modules are connected
      - Loose is good
      - **Types**
        - **Content:** one component directly impacts another
        - **Common**: two components share overall constraints
        - **External:** components communicate through external medium

- **Control:** one component controls the other (passes info)
- **Stamp:** complete data structures are passed
- **Data:** only one type of interaction

- **Information hiding**
  - **Need to know**; each module has a secret
  - Done via
    - **Encapsulation**: hides data and only allows access via specific functions
    - **Separation of interface and implementation:** define a public interface but separate details of how it is realised

      - Enables independence
- **Limiting complexity**
  - **Effort required to build** the solution (lines of code, depth of nesting)
  - Criterion to asses a design

  - Higher value => higher complexity => higher effort = worse design
  - Kinds of module complexity
    - **Intra-modular:** complexity of single module; based on size (LOC)
    - **Inter-modular:** between module
      - Based on size and structure
      - Eg: local flow, global flow and so on
- **Hierarchical structure**

- **Issues to be handled**
  - **Concurrency:** parallel execution of more than one program; deadlocks/race conditions
  - **Event handling**: messages sent between objects
  - **Distribution of components**
    - Distributed apps are supported by middleware
    - Consider communication breakdown
  - **Non functional requirement:** may have system-wide impact

- **Error, exception handling, fault tolerance**
  - **Mistake diverts program execution** or **create incorrect result/action**
  - **Exception** could lead to **termination**
  - **Ensure that Faults** do not need **lead to errors** which **lead to system failures**
  - **Approaches:** fault avoidance, fault detection and removal
- **Interaction and presentation**
  - React to user input **effectively and efficiently**
- **Data persistance**
  - Storage of information **between executions**

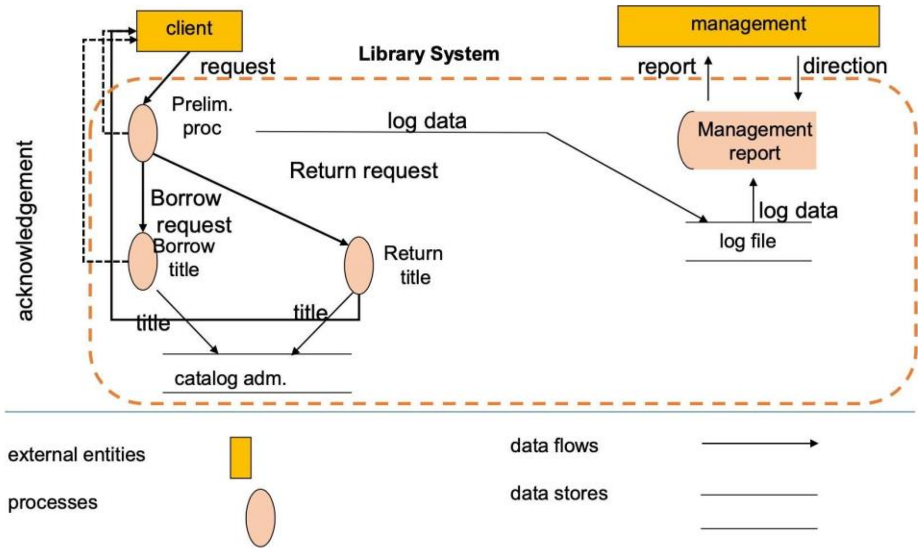
- **Differences between architecture and design**

Architecture	Design
Bigger picture; frameworks, tools, languages, scope, goals, etc	Smaller picture; local constraints, design patterns, programming idioms, code org
Strategy, structure and purpose	Implementation and practice
Software components, visible properties and relationships	Problem solving and planning for internal software
Harder to change	Simpler and have less impact
Influence non-functional requirements	Influence functional requirements

- **Design methods:** support in **decomposing components** and representing **system requirements as components**
  - **Module hierarchy**
  - **Data flow design**
    - **Two step** process
      - **Structured analysis: logical** design; **data flow** diagrams
        - Logical consisting of set of DFD's augmented by minispecs and data dictionary
      - **Structured design**: transform **logical design into program structure**; **structure charts**

  - Heuristics based on **coupling and cohesion**
  - Transform centered

- **Data flow diagram**
  - **External entity**
    - **Source and destination** of transaction
    - Located **outside** the domain
    - **Squares** in the diagram
  - **Processes**
    - **Transform** the data; depicted as **circle**
  - **Data stores**
    - Lie **between processes** and places where **data structures** reside
    - **Two parallel lines**
  - **Data flows**
    - **Data travels** between processes, entities, data stores
    - Depicted as an **arrow**



- **Minispecs**: process in DFDs become sufficiently straight forward and **doesn't warrant further decompositions**
- **Data dictionary: contents** of DFDs after we are at **logical decomposed state**
  - Precise description of the structure of data

- **Design patterns**
  - **Procedural patterns**
    - **Analyse** problem **prior** to and **during construction**
  - **Object oriented pattern:** Gang of Four solutions

Structural decomposition pattern	Breaks down a large system into subsystems and complex components into co-operating parts, such as ***a product breakdown structure***
Organization of work pattern	defines how components work together to solve a problem, such as ***master-slave and peer-to-peer***
Access control pattern	describes how access to services and components is controlled, such as through a ***proxy***
Management pattern	defines how to handle homogeneous collections in their entirety, such as a ***command processor and view handler***
Communication pattern	defines how to organize communication among components, such as a ***forwarder-receiver, dispatcher-server, and publisher-subscriber***

- **Creational patterns** that focus on creation of objects (Singleton, builder)
- **Structural patterns** that deal with composition (Adapter, Bridge)
- **Behavioural pattern** that describe interaction (Command, interpreter)
- **Distribution patterns** that deal with interface for distributed systems
- **Singleton pattern**
  - Intent: only **one instance of class is created;** global access point to object
  - Motivation: **one object to coordinate** actions across the system
    - One class responsible to instantiate itself
    - **Global point of access** to the instance
  - Eg: centralised management of global resources

- **Anti Patterns**
  - Describes **situations** a developer should **avoid**
  - **In agile** approaches, **refactoring** is **applied when anti pattern** is **introduced**
  - Patterns describe desirable behavior and anti patterns describe situations
  - one had better avoid
  - In agile approaches (XP), refactoring is applied whenever an anti pattern has

- been introduced

Example,

- **God class:** class that holds most responsibilities
- ***Lava flow****:* dead code which gets carried forward indefinitely

Six Types of Antipatterns:

1. Spaghetti Code
Code with a tangled, unstructured control flow that is hard to follow and maintain.
Example: Using numerous goto statements leading to a confusing, non-linear flow of execution.

2. Golden Hammer
Applying a single tool or technology to all problems, regardless of its suitability.
Example: Using a relational database for every application, even when a NoSQL solution would be more appropriate.

3. Boat Anchor
Retaining obsolete or unnecessary components that are no longer needed, but are kept due to their historical significance.
Example: Continuing to use a legacy library for new development when it’s been superseded by more efficient alternatives.

4. Dead Code
Code that is never executed or used, which adds unnecessary complexity and can lead to confusion.
Example: A function in a codebase that is never called by any other part of the program.

5. God Object and God Class
An object or class that performs too many functions and handles too many responsibilities, leading to poor maintainability.
Example: A single class that handles database access, user interface management, and business logic all in one.

6. Copy and Paste Programming
Reusing code by copying and pasting rather than abstracting common functionality into reusable components.
Example: Duplicating the same block of code in multiple places to handle similar logic instead of creating a reusable function.

- **Contrasting structural approach vs object oriented approach**

- **Comparison**

Comparison Factor	Structural Approach	Object Oriented approach
Abstraction	Basic abstractions are real world functions, processes and procedures	Basic abstraction are not real world functions but data representing real world entities
Lifecycles	Uses SDLC methodology	Incremental or Iterative methods
Function	Grouped together, hierarchically	Functions grouped based on data
State information	In centralised shared memory	State information distributed among objects
Approach	Top Down	Bottom up approach
Begin basis	Considering use case diagram and scenarios	Begins by identifying objects and classes
Decompose	Function level decomposition	Class level decomposition
Design approaches	Use Data flow diagram, structured English, ER diagram, Data dictionary, Decision tree/table	Class, component and deployment for static design; Interaction and State for dynamic

Comparison Factor	Structural Approach	Object Oriented approach
Design techniques	Design enabling techniques need to be implemented	Communicates with objects via message passing and has design enabling techniques
Design Implementation	Functions are described and called; data isn’t encapsulated	Components have attributes and functions; class acts as blueprint
Ease of development	Easier; depends on size	Depends on experience of team and complexity of program
Use	Computation sensitive apps	Evolving systems that mimic a business or business case

---

Service oriented architecture

- Make **software components reusable** via **service interfaces**

- Utilise **common communication standards** and can be **rapidly incorporated** into new applications without deep integration

- Each **service** embodies **code and data integrations** to execute a **complete, discrete business function**; provide **loose coupling**
- Exposed using **standard network protocols** (SOAP - Simple object access protocol/HTTP or JSON/HTTP) to send **requests** to read or change data
- **Structured collections of discrete software modules** that collectively provide functionality
- **Benefits**
  - **Greater business agility, faster time to market**
  - Ability to **leverage legacy functionality** in new markets
  - **Improved collaboration** between business and IT
  - **Service reusability**
  - **Service Compensation**

- **Service**
  - **Logical representation of a repeatable business activity** that has **specified outcome**
  - Discrete pieces of software written in **any language**
  - **"Callable entities"** accessed via **exchange of messages**
  - **Application functionality with wrappers**
- **Service characteristics**
  - Services **adhere to service contract**
    - Adhere to **communications agreement**
  - Services are **loosely coupled**: minimise dependencies
    - Maintain awareness of each other
  - Services are **stateless**
    - **Minimise resource consumption**
  - Services are **autonomous**

- Service **abstraction**
- Services are **reusable**
- Services use **open standards**
- Services **facilitate interoperability**
- **Service Monitoring**
- **Service Orchestration**
- Services can be **discovered:** metadata
    - **Composed to form larger services**

- **SOA roles**
    - **Service provider: creates web services** and provides them to a **registry**
      - Responsible for terms of service
    - **Service broker/registry**: responsible for **providing information** about the service
    - **Service requester/consumer: finds a service** in the broker and **connects** to the provider
- **SoA vs Microservices**

SoA	**Microservice**
Enterprise-wide approach to architecture	Implementation strategy with app dev teams
Communicates with components using Enterprise service bus (ESB)	Communicate statelessly using APIs
Less tolerant and flexible	More tolerant and flexible

Unit-3

Software Implementation

- **Purpose:** Converts program structures into actual code in a programming language.

Entry and Exit Criteria:

- **Entry:**
  - Software Requirements Specifications (SRS) document.
  - Design specifications document.
- **Exit:**
  - Stable and executable build.
  - Unit test cases completed.

Level of Details:

- Dependent on whether the language is compiled or interpreted.
- Involves selecting:
  - Programming language.
  - Development environment.
  - Configuration management plan.
  - Building code.

Software Creation:

- Involves detailed coding, reviews, and unit testing.

Goals of Implementation:

- Minimize complexity.
- Anticipate change.
- Ensure verification readiness.
- Facilitate code reuse.
- Make code understandable for others.

Characteristics of Software Construction:

- Produces a high volume of configuration items, such as source files and test cases.
- **Tool-Intensive:**
  - Relies on compilers, debuggers, GUI builders, etc.
- **Related to Software Quality:**
  - Code is the ultimate deliverable of a software project.
- Involves extensive use of computer science knowledge, such as algorithms and detailed coding practices.

Software Construction and Languages

- **Language Choice Based on Abstraction:**
  - **Assembly Languages:** Map directly to CPU architecture.
  - **Procedural Languages:** Provide modest abstraction from underlying architecture.
  - **Aspect-Oriented Programming:** Allows separation of aspects within a program.
  - **Object-Oriented Languages:** Provide further abstraction, using objects to structure programs.

Development Environment:

- Choosing developers and the implementation environment is restricted by:
  - Language.
  - Methodology.
  - Application type.
  - Development and target hardware platforms.
- **Factors to Consider:**
  - **Commercial vs. Open Source:** Developers may prefer open source, but organizations may consider it untrustworthy.
  - **Support for Development Process:** Availability of debuggers, test builders, and analysis tools.
  - **Security:** Considerations around securing the environment.
  - **Future Capabilities:** Account for product size and the distribution of the development team.
  - **Degree of Integration:** How well tools integrate with other systems and processes.

Coding Practices

- Coding begins once the development environment is selected.
- **Where to Catch Bugs:**
  - During coding.
  - Through code reviews.
  - In unit testing.
  - Via integration testing.

Testing and Production:

- Involves system testing, field trials, and eventual production deployment.

Programming Considerations:

- **Code is for the programmer, not the computer.**
    - Written once but read and modified multiple times.
    - **Goal:** Minimize complexity.

Characteristics of Good Code:

- **Simplicity and Clarity:**
    - Well thought-out, designed, and structured.
    - Avoid too many lines per function, nesting, or arguments per function.
- **Naming Conventions:**
    - Use short, descriptive names.
    - Avoid overly long names or starting with underscores.
    - Follow consistent casing (Pascal, Camel, or underscore prefixes).

Structuring Code:

- Ensure logical dependencies between components.
- Use proper file structure (e.g., grouping `#includes`, `#defines`).
- Apply abstract data types and encapsulation.

Readability:

- Focus on readable identifier names and good visual layout.
- Use spaces, parentheses, and new lines effectively.
- Comment code well to explain complex logic.

Good Programming Practices:

- Follow agreed-upon coding standards.
- Use GOTO sparingly and in a disciplined manner.
- Hide data structures behind user-defined data types.

Handling Errors and Security:

- **Error Handling:**
  - Use exception handling in object-oriented languages.
  - Check error flags in procedural languages.
  - Use assertions to prevent processing errors.
- **Security:**
  - Guard against vulnerabilities like buffer overflows and SQL injection.
  - Use principles like **default deny** and **least privilege**.
  - Sanitize all inputs to prevent attacks.

Code Documentation:

- **Types of Documentation:**
  - **About Code:** Descriptions of functionality.
  - **In Code:** Embedded comments and explanations.
  - **Literate Programming:** Combines human-readable documentation with machine-readable code.

Code Optimization and Standards:

- **Optimization:**
  - Use performance monitoring tools to identify and improve slow-running sections of the code.
- **Coding Standards:**
  - Ensure a uniform code appearance to improve readability and maintainability.
  - Follow guidelines for defensive, secure, and testable programming.

Code Reuse and Defensive Programming:

- **Code Reuse:** Reuse sound practices to increase efficiency.

- **Defensive Programming:**
    - Anticipate errors (Murphy's law) by adding redundant checks.
- **Secure Programming:**
    - Protect against logic flaws and defects by:
        - Validating input.
        - Heeding compiler warnings.
        - Denying access by default.
        - Sanitizing all data.

Testable Programming Techniques:

- **Assertions:** Catch out-of-range or inappropriate values.
- **Test Points:** Methods to retrieve current module status and variable contents.
- **Scaffolding:** Code that emulates missing or future system features.
- **Test Harness:** Simulates parts of a system to drive code testing.
- **Test Stubs:** Return fixed values to test functions not under development.
- **Instrumentation:** Logs execution details to track system activity.
- **Test Data Sets:** Build valid and invalid data for testing scenarios.

Refactoring Code:

- **Goals of Refactoring:**
    - Improve the design, structure, and implementation of software without changing its external behavior.
    - Improve attributes like code length and duplication to simplify maintenance.
    - Enhance understanding of the codebase.
- **Refactoring does not include:**
    - Rewriting code.
    - Fixing bugs.
    - Changing observable system behavior, such as the user interface.

Managing Construction

- **Key Issues**
  - Minimize complexity
  - Anticipate change

Constructing Software Solutions for Verification

- **Construct Software Using Standards**
  - **Construction Quality**
    - Proceeding as planned
  - **Development Progress and Productivity Measures**
    - Measures: number of active days; assignment scope completed; productivity; efficiency; code churn
    - Metrics: Lines of Code (LoC)/effort days; LoC generated/purged
  - **Technical Quality**
    - Ease of debugging, troubleshooting, maintenance, integration, and extension
    - Measures: Static code analysis (lines of code, code complexity, instruction path length)
    - Metrics: Number of review errors/KLoC
- **For Agile Scrum Projects**
  - **Sprint breakdown**: completion of work through sprint story points
  - **Team velocity metric**: amount of software/stories completed during a sprint
  - **Throughput**: total value-added work output
  - **Cycle time**: time from the start of an item to completion
- **Activities for Construction Quality**
  - **Peer Review**: developers show code to other developers for advice and comments
  - **Unit Testing**: code calls functions/methods and tests them
  - **Test First**: design and build test harness before writing code to balance tendency to write tests that match code
  - **Code Stepping**: execute one step at a time to see intermediate variable values
  - **Pair Programming**: two developers work on the same code—one

focuses on logic, the other on accuracy

- **Debugging**: analyze code to find defects
- **Code Inspections**: developers present code for review to code inspectors
- **Static Analysis**: examine code without execution

- **Code Review**
  - Consciously and systematically check each other's code
  - Review Checklists: correctness, error handling, readability, standards
- **Code Inspection**
  - Examine source code to discover anomalies and defects
  - Conformance with specification, but not with requirements
- **Inspection Procedure**
  - Planning: select team location; present system overview
  - Code and associated documents distributed in advance
  - Inspection takes place; errors noted
  - Owners identify and make modifications
  - Re-inspection may or may not occur
  - Document and archive all findings
- **Inspection Checklist**
  - Common errors that drive the inspection
  - Checklists are programming language-dependent and reflect characteristic errors

Unit Testing Tools

- **Unit Testing Frameworks**
  - Allow testers to enter method names, parameters, and expected results
  - Supply method call, return value comparison, and error reporting
- **Code Coverage Analyzers**
  - Identify code that is not covered by test cases

- **Wizards**
    - Tools that generate tests from input parameters
- **Record Playback Tools**
    - Start session and record every keystroke (e.g., Selenium for web application testing)

Software Configuration Management (SCM)

- **Overview**
    - Code must be converted into reliable and executable builds for testing
    - Systematic organization, management, and control of changes in documents, code, and other entities
- **SCM Goals**
    - Increase productivity and coordination to eliminate confusion and mistakes
- **SCM Involves**
    - Identifying elements and configurations
    - Tracking changes
    - Version selection
    - Control and baselining
- **Need for SCM**
    - Large projects with multiple teams across countries
    - Multiple people working concurrently on the same software
    - Managing more than one version of software
    - Coordinating configuration changes
    - Effective management of simultaneous source files

Build Management and Defect Tracking

- **SCM in Agile**
    - Responsibility of the whole team; automate as much as possible
    - Definitive versions held in a shared project repository
    - Developers copy into their workspace, modify, and update once satisfied

- **Benefits of SCM**
    - Allows orderly development of software configuration items
    - Ensures orderly release of new and revised products
    - Only approved changes are implemented and deployed
    - Updates documents accordingly
    - Evaluates and communicates the impact of changes
    - Prevents unauthorized changes

SCM Roles

- **Configuration Manager**
    - Identifies configuration items
    - Defines procedures for creating, promoting, and releasing
- **Change Control Board Members**
    - Approve or reject change requests
- **Developer**
    - Creates versions triggered by change requests
    - Checks changes and resolves conflicts
- **Auditor**
    - Validates processes for selection and ensures consistency and completeness

SCM Planning

- **Configuration Manager Responsibilities**
    - Outcome: SCM Plan
        - Define configuration items to be managed
        - Assign responsibilities for SCM procedures
        - Create baselines
        - Establish policies for change control and version management
        - Define tools for SCM processes
        - Create an SCM Database for configuration information

Configuration Management Activities

- **Configuration Item Identification**
  - Configuration item: independent/aggregation of hardware, software, or both, treated as a single entity
  - Examples include code files, test drivers, requirements, analysis, design, test documents, and user manuals
- **Challenges**
  - Identifying which items need configuration control
  - Maintaining items for the software's lifetime

SCM Directories

- **Programmer's Directory (Dynamic Library)**
  - Newly created or modified entities
  - Controlled by the developer
- **Master Directory (Controlled Library)**
  - Manages the current baseline; controls changes made
  - Entry is controlled and verified
- **Software Repository (Static Library)**
  - Archive for baselines in general use
  - Copies of baselines made available
- **Baselines**
  - Specification/product that has been reviewed and agreed upon
  - Serves as the basis for future development
  - Changes only through formal procedures

Branch Management

- **Codeline**
  - Progression of source files and artifacts that make up software components
- **Branch**
  - Copy/clone of all or part of the source code

- **Need for Code Branching**
    - Supports concurrent development
    - Captures solution configurations
    - Supports multiple versions
    - Enables isolated experimentation
    - Keeps overall product stable
- **Merging**
    - Integrating branches so everyone can use it
    - Frequent merging decreases likelihood and complexity of merge conflicts

Branching Strategies

- May be applied in combination:
    - Single branch
    - Branch by customer/organization
    - Branch by developer/workspace
    - Branch by module/component
- **Version Management**
    - Keeping track of different versions of software components
    - Tools like Git
- **Features of Version Control**
    - All changes are traceable
    - Change history is recorded and can be reverted
    - Better conflict resolution
    - Easier code maintenance and monitoring
    - Reduces software regression
    - Improves organization and communication

Build Management

- **Creating Application Programs for Software Release :** Source code and libraries are compiled and linked to produce build artifacts (bins and execs)

- **Tools**
    - Make, Apache Ant, MS Build, Maven
    - Compilation of files in the appropriate order
    - If source code is unchanged, recompilation may not be necessary

Build Automation

- **Build Process**
    - Fetch the code from the repository
    - Compile code and check dependencies
    - Link libraries and code
    - Run automated/manual unit tests
    - Generate build artifacts and store them
    - Archive build logs
    - Send notification emails
    - Change version numbers
- **Promotion Management**
    - Changes made by programmers need to be promoted to the central master directory based on specific policies

Change Management

- **Change Request Process**
    - Unique ID assigned to each change request
    - Change assessed for impact
    - Decision made on the change (accept/reject)
- **Tool-Driven Process**
    - If accepted, the change is implemented and validated
    - Plans executed for documents, versioning, merging, and development
    - Audited for compliance

Complexity of Change Management

- **Small Projects:** Change requests are informal and fast

- **Complex Projects**
  - Detailed change requests and approvals required from the Change Control Board (CCB)
- **Change Documentation**
  - Description of proposed changes
  - Reasons for making changes
  - List of items affected

Tools, Resources, and Training for Baseline Change

- **File Comparison Tools**
  - Identify changes
  - Control changes at two points
- **Change Policies**
  - Informal (research-type environments)
  - Formal (externally developed configuration items)

Release Policy and Management

- **Movement from Master Directory to Software Repository**
  - Gating quality criteria
  - Managing, planning, scheduling, and controlling software builds

Testing and Deployment

- **Making Software Components Available for System Release**
  - Version vs Revision vs Release
    - Release: formal and approved distribution
    - Version: initial release or re-release of configuration item
    - Revision: changes that correct errors without altering functionality

Bug/Defect Management

- **Definitions**
  - Bug: consequence of a coding fault

    - Defect: variation from expected business requirements
- **Process**
    - Discover, report/log, validate, analyze and categorize (critical, high, medium, low)
    - Request and approval for fixes
    - Resolution and verification
    - Merging, updating version numbers, planning release of fixes, closure, and reporting

SCM Tools

- **Categories of Tools**
    - **Source Code Administration: Source Code Control**
        - **RCS (Revision Control System):**
            - GNU; very old; only VCS.
        - **CVS (Concurrent Version Control):**
            - Enables concurrent working without locking.
            - **CVSWeb:** Interface for CVS.
        - **ClearCase:**
            - Supports Linux, Solaris, Windows.
            - Features multiple servers, process modeling, and policy checks.
        - **GitHub:**
            - Development platform for hosting code and version control.
            - **Key Features:**
                - Create repositories and manage contributions.
                - Access via SSH and HTTPS.
                - Changes pushed via commits with commit numbers and messages.
                - Different branches for code additions; master branch auto-cloned.
            - **Common Commands:**

      - `git init`
      - `git status`
      - `git add`
      - `git commit -m`
      - `git remote add origin`
      - `git remote -v`
      - `git push`
      - `git push origin`

- **Software Build:**
  - **Building Source Code:**
    - Compilation of source code into executable code.
    - **Make:**
      - Builds executable programs and libraries from source code.
      - **Makefiles** specify how to derive target programs.
- **Continuous Software Builds:**
  - **CruiseControl:** Open-source tool for continuous software builds.
  - **FinalBuilder:** Automated build and release management tool; user-friendly.
  - **Maven:**
    - Based on Project Object Model (POM).
    - Manages build, reporting, and documentation.
    - Simplifies and standardizes the build process.
    - Handles compilation, distribution, documentation, collaboration, etc.
    - Supports multiple development environments and increases reusability.
- **Software Installation:**
  - Packaging and installing products.
  - Cross-platform tools for all OSs.
  - Custom installers or specific installers.

- **Bootstrapper:**
  - Small installer that runs first; handles prerequisites and updates larger bundles.
  - Includes rollback and clean-up features.
- **Tools:**
  - **DeployMaster:** Distributes Windows files via internet/CD/DVD.
  - **InstallAware:** Windows installation.
  - **InstallShield:** De facto standard for MSI installations.
  - **Wise Installer:** Tool for creating installations.

Software Bug Tracking

- **Purpose:**
  - Tracks bugs and changes through an issue tracking system.
  - Provides a clear, centralized overview of development requests.
  - Supports the life cycle for a bug.

Software Quality

- **Complexity of Software Systems:**
  - Software systems are complex and evolve, necessitating quality assessment and evaluation.
  - Issues can lead to customer dissatisfaction.
- **Perspectives on Quality:**
  - **Transcendent:** Exceeds normal expectations.
  - **User-based:** Fitness for use.
  - **Product-based:** Based on attributes of the software.
  - **Manufacturing-based:** Conformance to specifications.
  - **Values-based:** Balances time, cost, and profits.
- **Product Quality Attributes:**
  - **Product Operation Perspective:**
    - **Correctness:** Does it function as intended?
    - **Reliability:** Does it perform accurately all the time?

    - **Efficiency:** Does it run well on my hardware?
    - **Integrity:** Is it secure?
    - **Usability:** Can I use it easily?
    - **Functionality:** Does it include all necessary features?
    - **Availability:** Is the product operational and available consistently?
  - **Product Revision Attributes:**
    - **Maintainability:** Can I fix it?
    - **Testability:** Can it be tested easily?
    - **Flexibility:** Can it be changed easily?
  - **Product Transition Attributes:**
    - **Portability:** Can it be used on another machine?
    - **Reusability:** Can parts of the software be reused?
    - **Interoperability:** Can it interface with other systems?
- **Overall Environment:**
  - Factors influencing quality include responsiveness, predictability, productivity, and people involved.

Approaches to Quality Assessment

- **Quality of Product vs. Process Used**
- **Verification, Validation, and Audits**
- **Improvements:** By product or process.

Quality Attributes: FLURPS

- **Functionality**
- **Localisation**
- **Usability**
- **Reliability**
- **Performance**
- **Supportability**
- **Extensibility**

Metrics and Measures

- **Purpose of Measurement:**
  - **Feedback:** Quantifies aspects of quality for improvement.
  - **Diagnostics:** Identifies issues and supports evaluations.
  - **Forecasting:** Assists in estimating, budgeting, and costing.
- **Measures:**
  - **Measure:** Quantitative indication of extent, amount, dimension, etc. (e.g., number of errors).
  - **Metric:** Quantitative measure of the degree to which a system possesses an attribute (e.g., number of errors per person-hour).

Characteristics of Metrics

- Quantitative
- Understandable
- Applicable
- Repeatable
- Economical
- Language-independent

Example Measures for Quality Attributes

- **Correctness:** Degree of operation according to specifications (e.g., defects/KLOC, failures/hours of operation).
- **Maintainability:** Openness to change (e.g., mean time to change, cost to correct).
- **Integrity:** Resistance to external attacks (e.g., fault tolerance, security).
- **Usability:** Ease of use (e.g., training time required, skill level necessary).

Categorization of Metrics

- **Direct Measures (Internal Attributes):**
  - Depends only on the value of the attribute.
  - Valid and measured in relation to other attributes (e.g., cost, effort, LOC, duration of testing).

- **Indirect Measures (External Attributes):**
  - Derived from direct measures (e.g., defect density, programmer productivity).

Types of Metrics

- **Size-oriented Metrics:** Size of software produced (e.g., LOC and KLOC).
- **Complexity-oriented Metrics:** Measures like LOC, fan-in, fan-out, and Halstead's software science.
- **Categories of Metrics:**
  - **Product Metrics:** Assess project state, track risks, uncover problem areas, adjust workflow, and evaluate team quality control.
  - **Process Metrics:** Provide insights into software engineering tasks, work products, and milestones for long-term processes.
  - **Project Metrics:** Include number of developers, staffing patterns, costs, schedules, and productivity.

Using Metrics

1. Understand the environment and product.
2. Formulate and select metrics.
3. Educate team members.
4. Collect data.
5. Analyze results.
6. Interpret findings.
7. Implement course corrections.
8. Provide feedback.

Cost of Software Quality

- **Quantifying Business Value:**
  - Costs incurred through meeting vs. not meeting quality.
- **Cost of Good Quality:**
  - **Prevention Costs:** Error-proofing, capability measurements.
  - **Appraisal Costs:** Quality assurance, inspections, and reviews.
  - **Management Control Costs.**

- **Cost of Bad Quality:**
  - **Internal Failure Costs:** Rework, retesting, etc.
  - **External Failure Costs:** Support calls, patches, etc.
  - **Technical Debt:** Structural problems and increased complexity.

Software Quality Assurance

- **Purpose:** Monitor software engineering processes and ensure quality.
- **Involves:**
  - Planning, goal setting, commitments, activities, measurements, and verifications.
  - Encompasses the entire software development process and activities.
  - Includes planning oversight, record keeping, analysis, and reporting.

Involvement in Software Quality Assurance

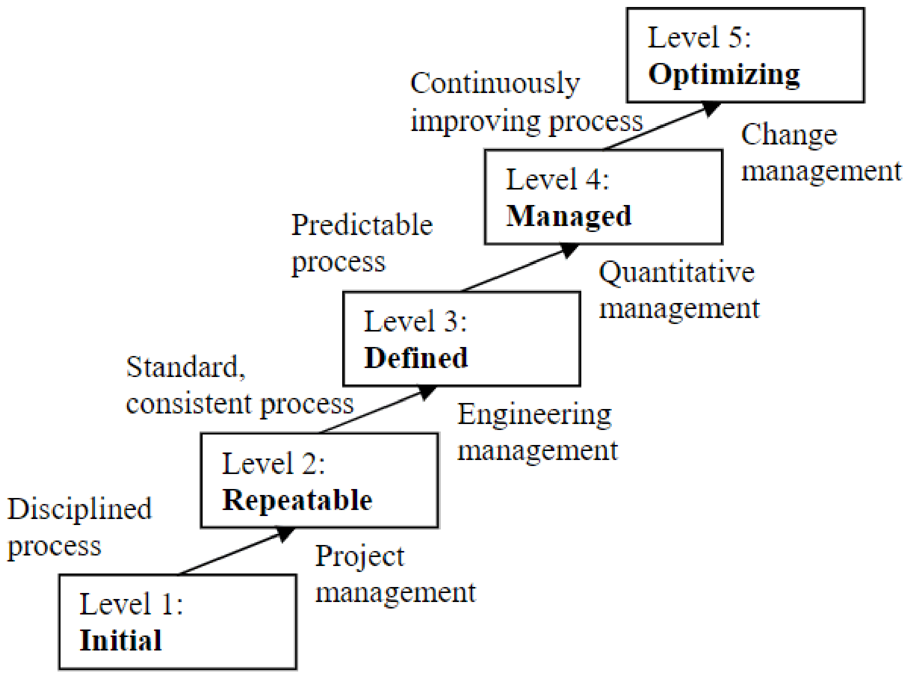
- **Project Managers:** Establish processes and methods for the product and provide oversight.
- **Stakeholders:** Perform actions relevant to quality.
- **Software Engineers:** Apply technical methods, conduct reviews, and perform testing.
- **SQA Group:** Oversees quality assurance planning, record keeping, and reporting.

Contents of the Software Quality Assurance Plan

- Responsibility management
- Document management and control
- Requirements scope
- Design control
- Development control and rigor
- Testing and QA
- Risks and mitigation strategies
- Quality audits
- Defect management
- Training requirements

SEI - CMM (Software Engineering Institute - Capability Maturity Model)

- **Maturity Model:** Structured levels describing organizational behavior, practices, and processes for reliable software production.
- **Evolutionary Improvement Path:** From ad-hoc, immature processes to reliable, sustainable practices.
- **Benefits:**
    - Establishes common language and vision.
    - Builds on processes developed with community input.
    - Provides frameworks for prioritization and consistent appraisals.
    - Allows for industry-wide comparisons.
- **Risks and Limitations:**
    - Models are simplifications of reality.
    - Not comprehensive; require careful interpretation and tailoring.
    - Only effective if implemented early in the software development life cycle.


Level 5: Optimizing
Continuously improving process
Change management
Level 4: Managed
Predictable process
Quantitative management
Level 3: Defined
Standard, consistent process
Engineering management
Level 2: Repeatable
Disciplined process
Project management
Level 1: Initial