UNIT-1

Software Engineering

Software Engineering is the systematic, disciplined, quantifiable approach towards the development, operation, and maintenance of software products and thus supports managing of complexity.

Fundamental Driver of Software Engineering

1. Industrial Strength Software
    a. Refers to the need for software to be robust, reliable, and able to handle real-world industrial demands. It should be capable of performing efficiently and effectively in various conditions and environments.
2. Software is expensive
    a. Highlights the cost associated with software development, maintenance, and support. The expense includes not only initial development but also ongoing costs such as updates, bug fixes, and enhancements.
3. Can influence the life or death of a person
    a. Stresses the critical role that software can play in various applications, such as medical devices or transportation systems, where software failure could have severe consequences, including loss of life.
4. Heterogeneity

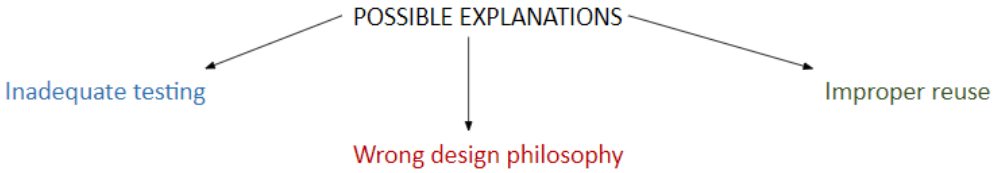
a. Refers to the diverse range of technologies, platforms, and devices that software must be compatible with. Software engineers must consider the heterogeneity of systems when developing solutions.

5. Diversity
    a. Acknowledges the diversity in terms of user demographics, cultural backgrounds, and user needs. Software should be designed to be inclusive and accessible to a wide range of users.
6. Business and Social changes
    a. Recognizes the dynamic nature of both business and societal environments. Software needs to adapt to changing requirements, technologies, and user expectations to remain relevant and effective.
7. Security and trust
    a. Emphasizes the importance of building secure software to protect sensitive data and ensure user trust. Security concerns, such as data breaches and cyber attacks, need to be addressed throughout the software development lifecycle.
8. Scale
    a. Highlights the challenge of developing software that can handle varying levels of scale, from small user bases to large, enterprise-level deployments. Scalability is crucial for accommodating growth and increased usage.
9. Quality and productivity
    a. FLURPS , Portability and Efficient

Case Study – ARIANE Flight 501

- Disintegration after 39 sec
- Caused by large correction for altitude deviation
- Caused by wrong data being sent to On Board Computer
- Caused by software exception in Inertial Reference System after 36 sec

Due to
- Overflow in conversion of variable BH from 64-bit floating point to 16-bit signed integer
- Of 7 risky conversions, 4 were protected; BH was not
- Reasoning: physically limited, or large margin of safety
- In case of exception: report failure on data-bus and shut down



David Hooker's Software Principles

David Hooker's seven principles for software engineering practice encompass fundamental concepts that guide effective software development.

1. **The Reason It All Exists:**
   - This principle emphasizes the importance of understanding and maintaining a clear purpose or goal for the software project. Knowing why the software is being developed helps align efforts and decisions throughout the development process.
2. **KISS (Keep It Simple, Stupid!):**
   - The KISS principle advocates for simplicity in design and implementation. It suggests that solutions should be kept straightforward and uncomplicated, avoiding unnecessary complexity. Simple designs are often easier to understand, maintain, and troubleshoot.
3. **Maintain the Vision:**
   - This principle underscores the need to preserve the original vision or objectives of the software project. As the project evolves, it's crucial to ensure that changes and additions align with the overall vision to prevent scope creep or deviations from the intended goals.
4. **What You Produce, Others Will Consume:**
   - This principle highlights the collaborative nature of software development. Software is often created for use by others, whether it's end-users, other

developers, or stakeholders. Therefore, developers should consider the needs and expectations of these consumers when creating software.

5. **Be Open to the Future:**
    - Encouraging an adaptive mindset, this principle suggests being open to changes and advancements in technology. Software should be designed with flexibility to accommodate future updates, enhancements, and evolving requirements.
6. **Plan Ahead for Reuse:**
    - The principle of planning for reuse emphasizes designing software components with an eye toward future reuse. Identifying and creating reusable modules or components can save time and effort in future projects.
7. **Think!:**
    - This principle underscores the importance of thoughtful and deliberate decision-making throughout the software development process. Critical thinking is essential to solving problems, making design choices, and ensuring the overall success of the project.

1. Software Lifecycle

A software lifecycle, also known as a software development process or model, is a structured framework that guides the systematic development of software from its conception to its retirement. Here's a summary of key aspects:

Entry criteria: What conditions must be satisfied for initiating this phase

Task and its deliverable: What should to be done in this phase

Exit criteria: When can this phase be considered done successful

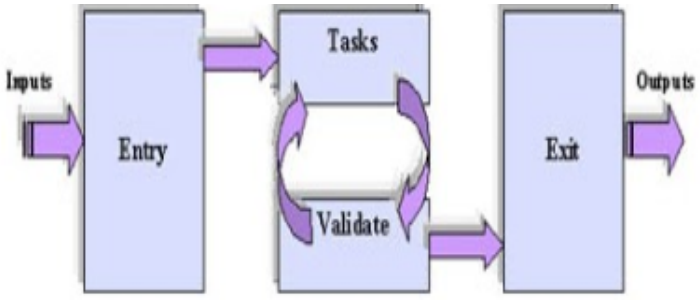
Who: Who is responsible

Dependencies: What are the dependencies for this phase ..etc.

Constraints: Time schedule

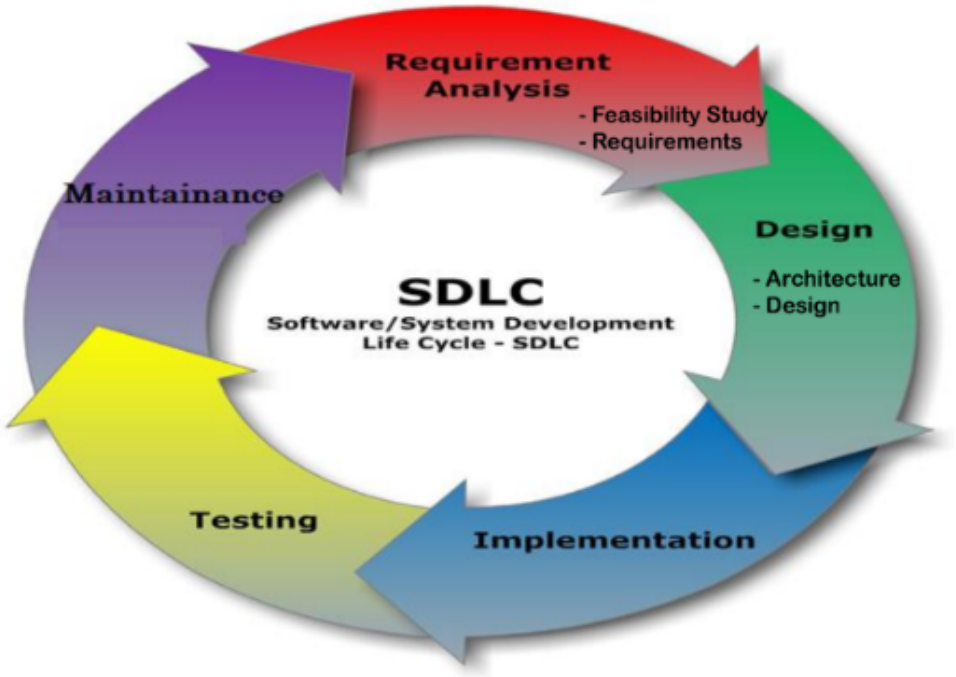
ETVX: Important Gates of each Stage/Phase

> 💡 [Entry, Tasks, Validate, exit]



Products are outcomes of executing a process (or a set of processes) on a project.

2. SDLC

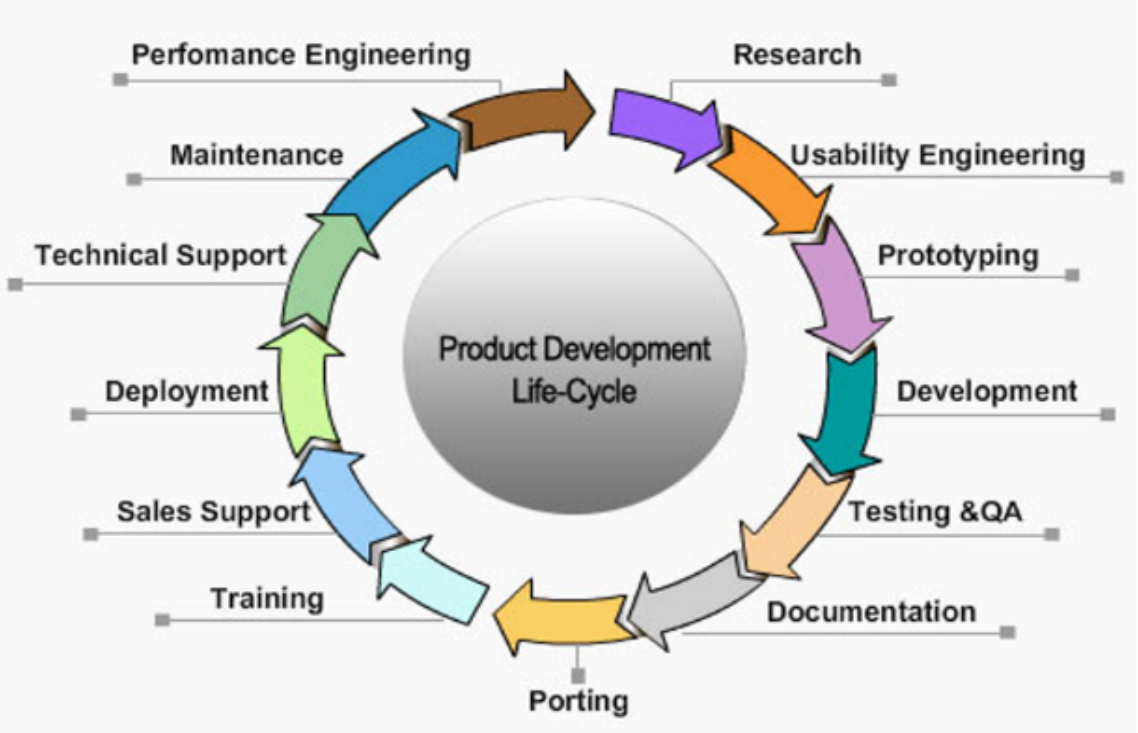


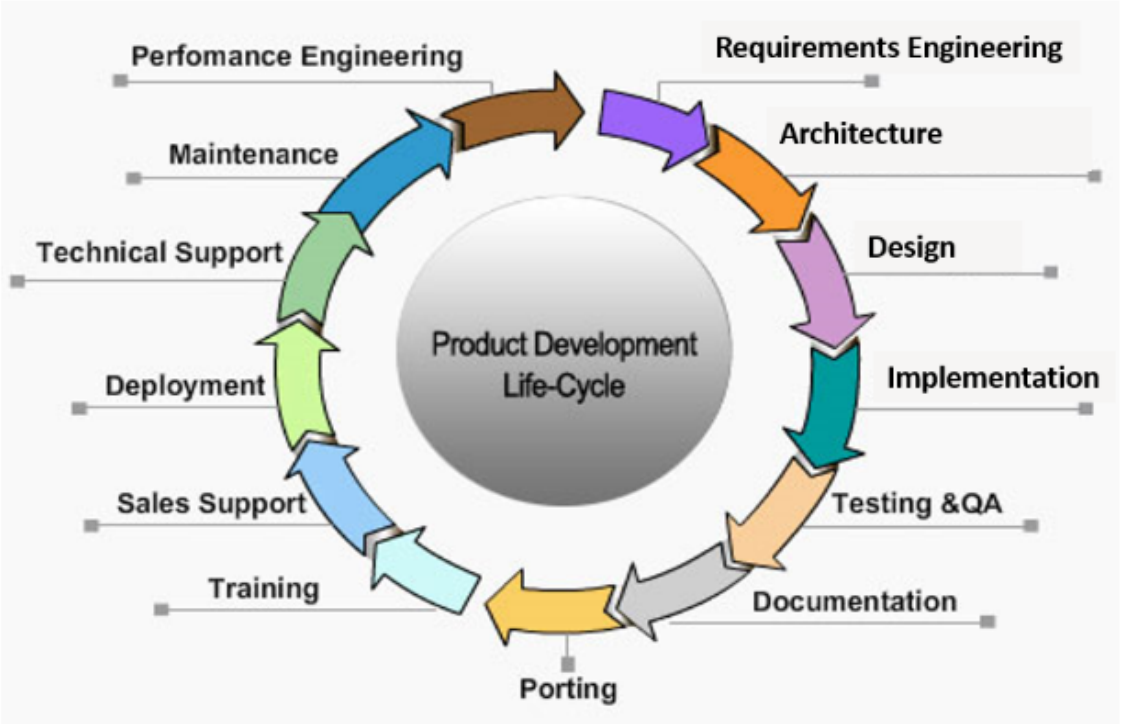
The Software Development Life Cycle (SDLC) is a systematic process for planning, creating, testing, deploying, and maintaining information systems. While different methodologies may have variations, the typical stages in the SDLC include:

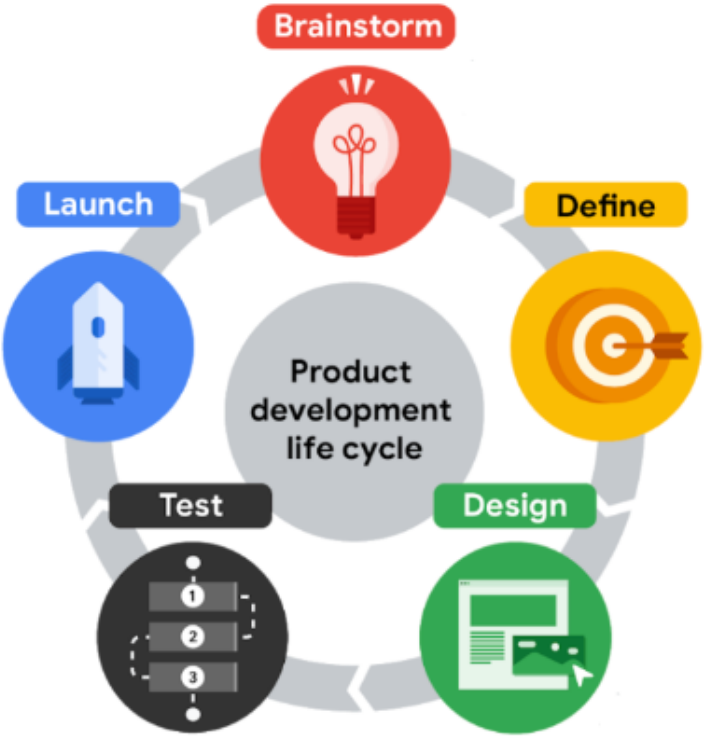
1. **Requirements Gathering and Analysis:**
    - In this initial phase, project stakeholders, including end-users, gather and define the requirements for the software. This involves understanding the needs, constraints, and goals of the system.
2. **System Design:**
    - Based on the gathered requirements, the system design phase involves creating a blueprint for the software system. This includes defining system architecture, modules, interfaces, and data structures.

3. **Implementation (Coding):**
    - During the implementation phase, developers write the actual code for the software based on the design specifications. This is where the design is translated into a working system through programming.
4. **Testing:**
    - The testing phase involves systematically checking the software for defects, ensuring that it meets the specified requirements, and verifying its functionality. Testing may include unit testing, integration testing, system testing, and user acceptance testing.
5. **Maintenance and Support:**
    - After deployment, the software enters the maintenance phase. This involves addressing any issues that arise, applying updates, and making enhancements as needed. Maintenance ensures the ongoing reliability and functionality of the software.

3. PDLC







Brainstorm

stage is when the team starts thinking of an idea for a product.

Define stage

the goal is to figure out the specifications for the product by answering questions like: Who is the product for? What will the product do? And, what features need to be

included for the product to be successful?

Design stage

you start by drawing wireframes, which are outlines or sketches of the product, then move on to creating prototypes, which are early models of a product that convey its functionality.

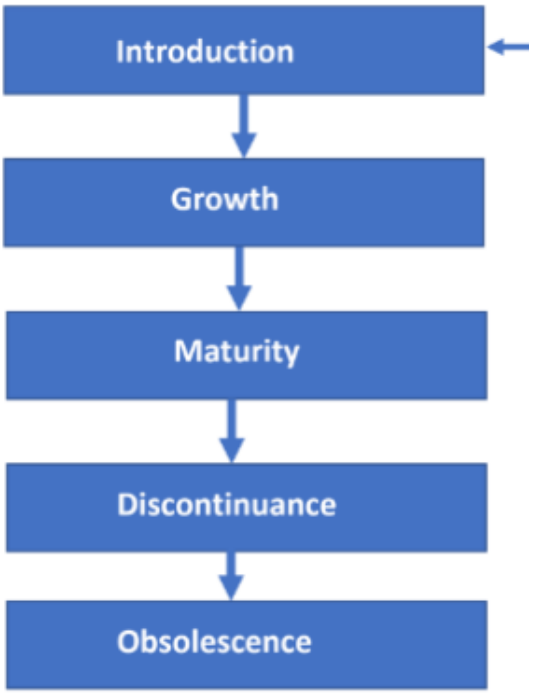
Test stage

means writing the code and finalizing the overall structure of the product.

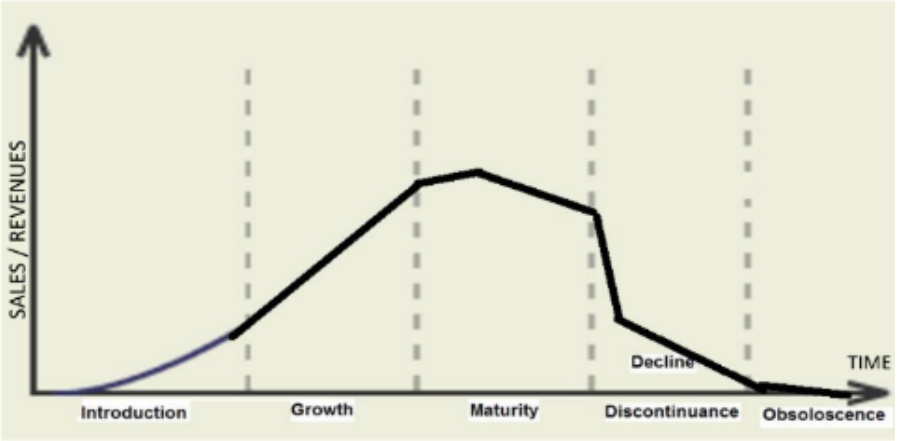
L**aunch** stage

is when the product is released into the world.

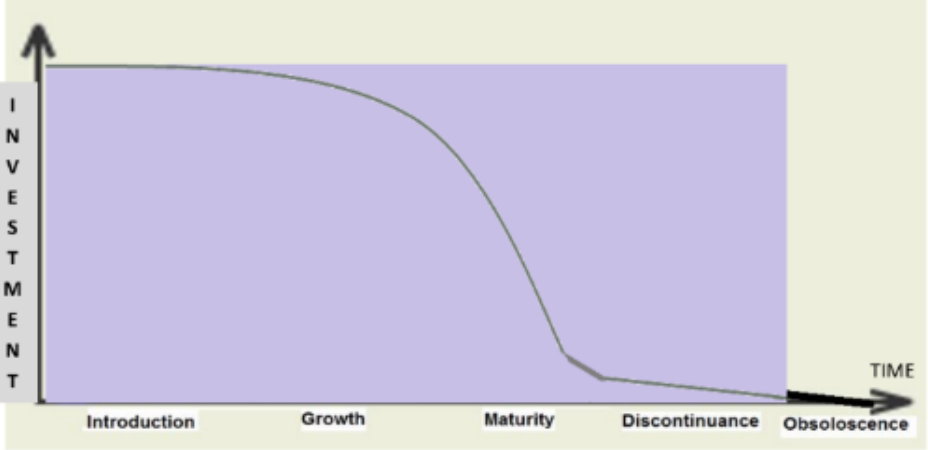
4. Product Lifecycle



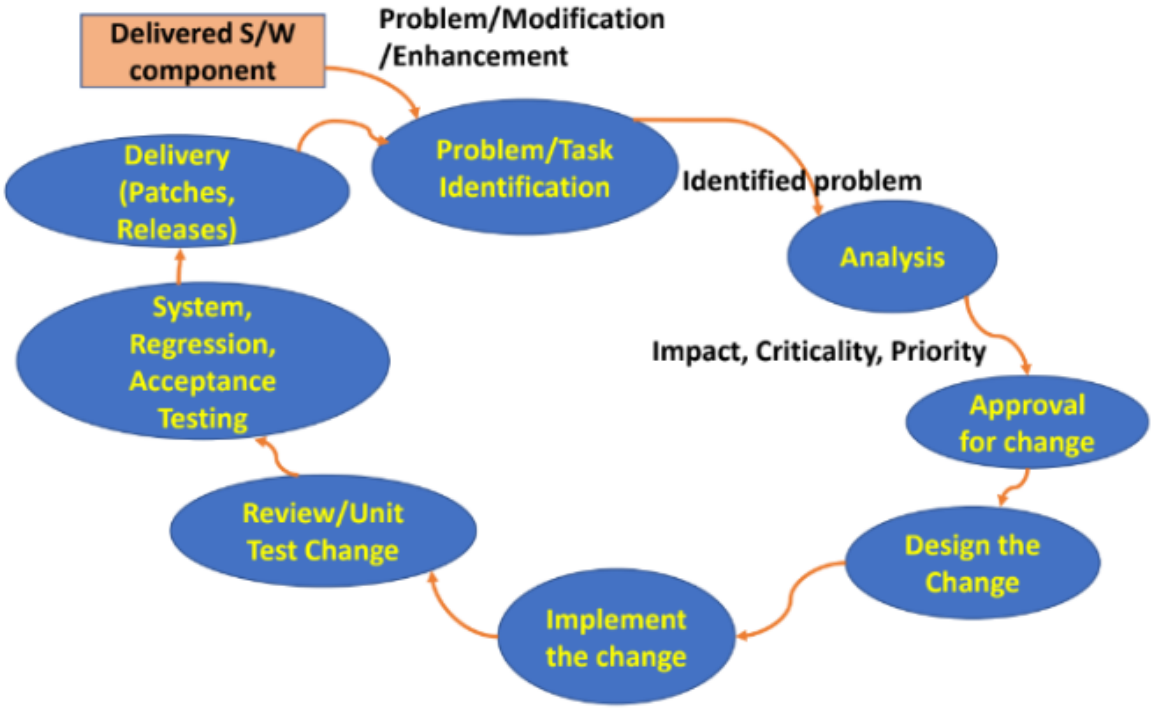
💡 SALES



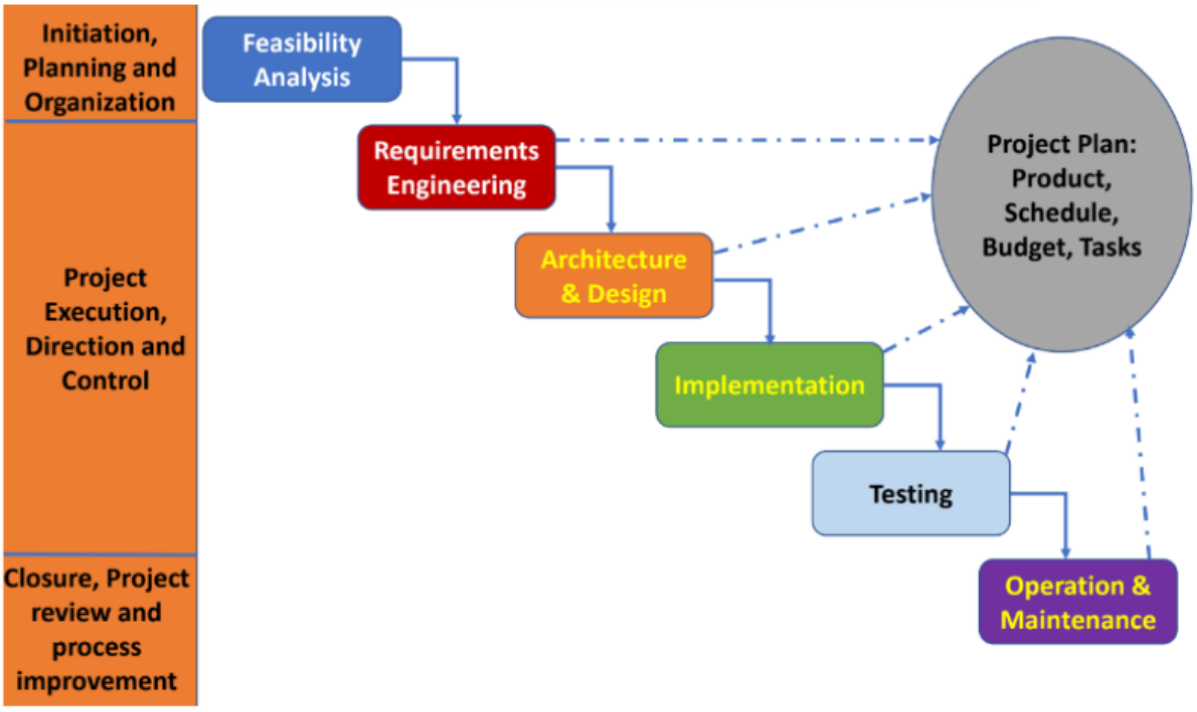
💡 INVESTMENT



5. Software Maintanence Lifecycle

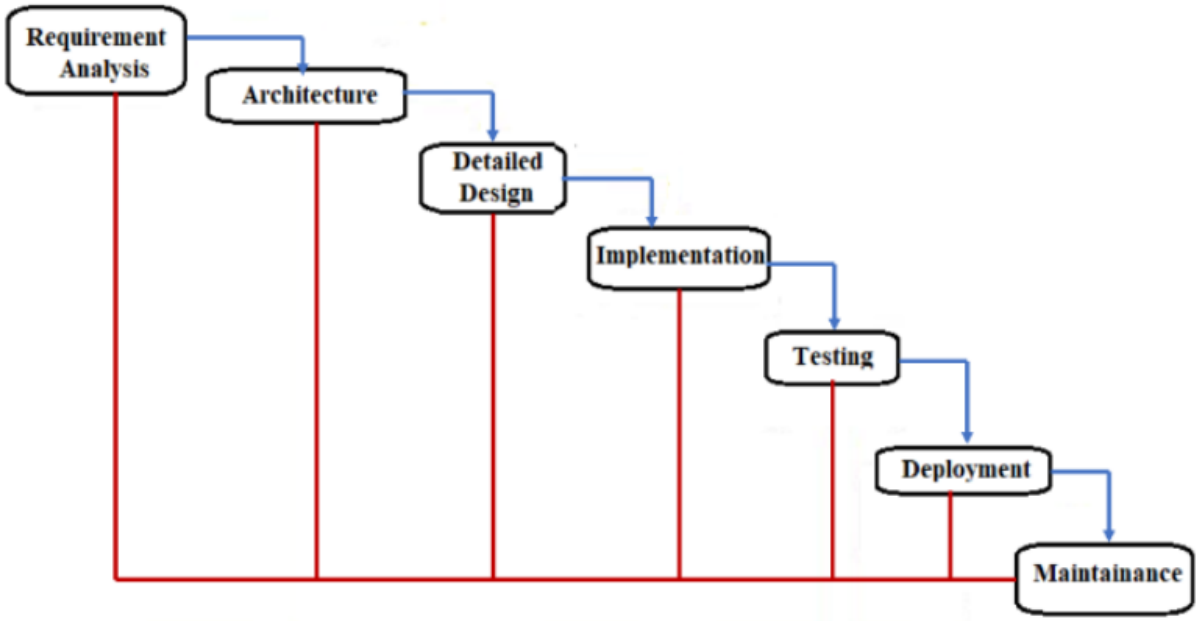


6. Project Management Lifecycle



SDLC Models

7. Waterfall Model



The Waterfall Model is a traditional and sequential approach to software development. In this model, the software development process is divided into distinct phases, and each phase must be completed before the next one begins. It follows a linear and cascading flow, similar to a waterfall, where progress is seen as flowing steadily downward through these phases.

Characteristics of the Waterfall Model:

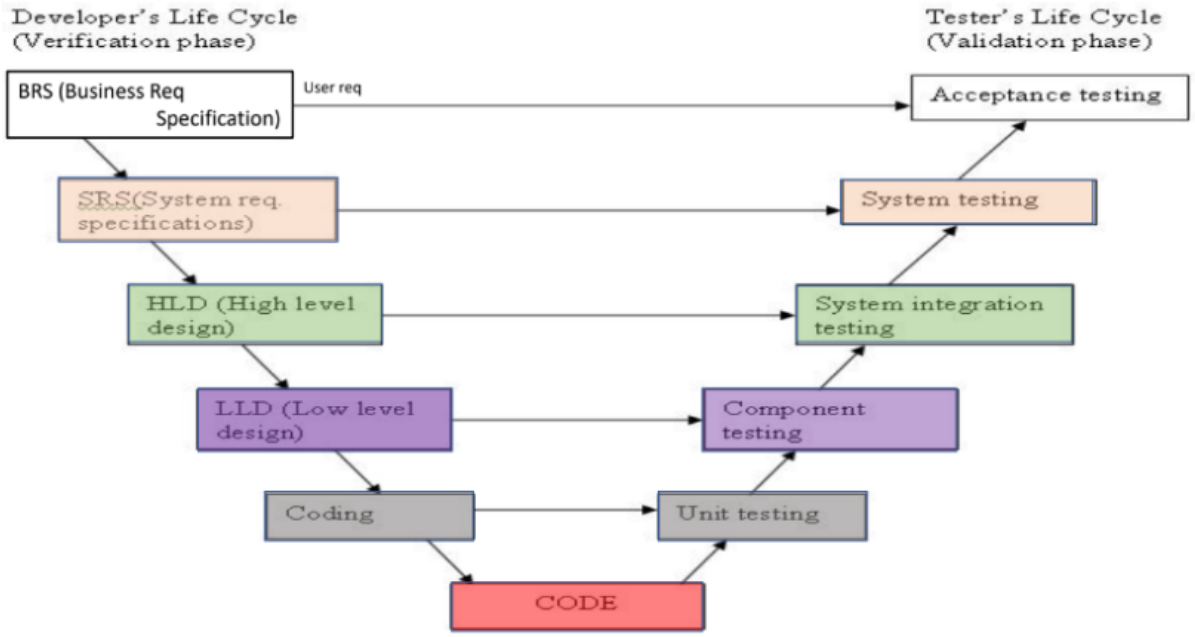
- **Sequential and Linear:** Each phase must be completed before the next one begins, and there is little room for iteration or going back to a previous phase.
- **Document-Driven:** Emphasis is placed on extensive documentation at each phase to provide a clear record of decisions and design choices.
- **Well-Suited for Stable Requirements:** The Waterfall Model works well when requirements are well-defined and unlikely to change significantly during the development process.
- **Rigid and Inflexible:** The model can be less adaptable to changes in requirements or unexpected issues that may arise during development.
- **Risk at the End:** Potential risks and issues may not become apparent until the testing or deployment phase, making it challenging to address them earlier in the process.

Disadvantages of waterfall model:

1. **Rigidity and Inflexibility:**
    - The sequential nature of the Waterfall Model makes it rigid and less adaptable to changes in requirements. Once a phase is completed, it is challenging to go back and make modifications without affecting the entire process.
2. **Late Visibility of the Product:**
    - Stakeholders often do not see a tangible product until the later stages of development. This can lead to a lack of visibility and understanding of the project's progress until significant work has been completed.
3. **High Risk of Project Delays:**
    - If there are changes in requirements or unexpected issues arise late in the development process, it can lead to project delays. Addressing such changes or issues may require going back to earlier phases, which can be time-consuming.
4. **Limited User Involvement:**
    - Users are typically involved mainly in the initial requirement gathering and acceptance testing phases. Limited user involvement throughout the development process may result in a product that does not fully meet user expectations.
5. **Difficulty in Handling Complex Projects:**
    - The Waterfall Model may struggle to accommodate large and complex projects. Breaking down complex systems into well-defined phases can be challenging, and it may be difficult to foresee all potential issues at the beginning of the project.
6. **Costly to Change Requirements:**
    - Making changes to requirements after the project has started can be expensive and time-consuming. The rigid structure of the Waterfall Model may result in increased costs if modifications are needed.
7. **Customer Feedback Limited Until Final Stages:**
    - Customers may have limited opportunities to provide feedback until the later stages of development. If the product does not meet their expectations, making significant changes at that point can be challenging.
8. **Potential for Project Failure:**

- If requirements are not well-understood or if there are significant changes in user needs, the project may not deliver the expected outcomes. This can lead to project failure, especially if issues are discovered late in the development process.

8. V Model



Flow of V Model

Business Requirement Analysis

System Design

Architectural Design

Module Design

Coding Phase

Unit Testing

Integration testing

System Testing

User Acceptance Testing (UAT)

The V-Model, also known as the Verification and Validation Model, is a software development and testing methodology that emphasizes a systematic and parallel approach to both development and testing activities. The model is named after the

shape of the V, which represents the relationship between each phase of development and its corresponding phase of testing. The key characteristic of the V-Model is that testing activities are planned in parallel with development tasks.

Key Characteristics of the V-Model:

- **Parallel Phases:** The development and testing activities proceed in parallel, with each development stage having a corresponding testing stage.
- **Early Detection of Defects:** Defects are detected early in the development process, as testing activities are planned from the beginning.
- **Traceability:** There is a clear and direct relationship between each development phase and its associated testing phase, ensuring traceability between requirements, design, and testing.
- **Structured Approach:** The V-Model provides a structured and systematic approach to software development and testing, making it easier to manage and control the process.

Advantages:

- This is a highly disciplined model and Phases are completed one at a time.
- V-Model is used for small projects where project requirements are clear.
- Simple and easy to understand and use.
- This model focuses on verification and validation activities early in the life cycle thereby enhancing the probability of building an error-free and good quality product.
- It enables project management to track progress accurately.
- Clear and Structured Process: The V-Model provides a clear and structured process for **software development**, making it easier to understand and follow.
- Emphasis on Testing: The V-Model places a strong emphasis on testing, which helps to ensure the quality and reliability of the software.
- Improved Traceability: The V-Model provides a clear link between the requirements and the final product, making it easier to trace and manage changes to the software.
- Better Communication: The clear structure of the V-Model helps to improve communication between the customer and the development team.

Disadvantages:

- High risk and uncertainty.
- It is not a good for complex and object-oriented projects.
- It is not suitable for projects where requirements are not clear and contains high risk of changing.
- This model does not support iteration of phases.
- It does not easily handle concurrent events.
- Inflexibility: The V-Model is a linear and sequential model, which can make it difficult to adapt to changing requirements or unexpected events.
- Time-Consuming: The V-Model can be time-consuming, as it requires a lot of documentation and testing.
- Overreliance on Documentation: The V-Model places a strong emphasis on documentation, which can lead to an overreliance on documentation at the expense of actual development work.
- No Early Prototype

Waterfall VS V-model

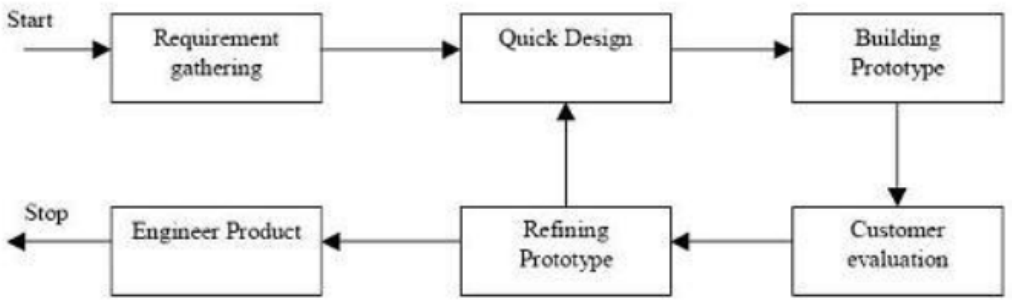
Aspect	Waterfall Model	V-Model
Nature	Linear and Sequential	Linear and Sequential, but with Testing Phases Parallel
Phases	Sequential phases: Requirements, Design, Implementation, Testing, Deployment, Maintenance	Sequential phases: Requirements, Design, Implementation, Testing; Similar Deployment and Maintenance
Feedback	Minimal feedback during development	Limited feedback with formalized testing phases
Flexibility	Less flexible to changes in requirements	Less flexible to changes due to its structured nature
Testing	Testing is a separate phase following development	Testing is planned parallel to each development phase
Adaptability	Limited adaptability to evolving requirements	Limited adaptability to changes once testing begins
User Involvement	User involvement at the beginning and end of the project	Limited user involvement mainly during testing phases
Iterative Nature	Not inherently iterative	May incorporate some iterative elements, especially in testing

Documentation	Extensive documentation at each phase	Documentation is created for each phase, emphasizing traceability
Risk Management	Risks are addressed at the beginning, and changes later in the process can be challenging	Risks are identified and addressed in parallel with development
Suitability	Suitable for well-defined projects with stable requirements	Suitable for projects with reasonably stable requirements and a focus on early testing

Similarities:

- Both models are based on a sequential and structured approach to software development.
- They emphasize the importance of early planning and documentation.
- Both models involve testing activities, with the V-Model incorporating testing in parallel with development phases.
- They assume a defined set of requirements at the beginning of the project.

9. Prototype Model



This model is used when the customers do not know the exact project requirements beforehand.

There are four types of Prototyping Models, which are described below.

- Rapid Throwaway Prototyping
- Evolutionary Prototyping
- Incremental Prototyping
- Extreme Prototyping

Advantages of Prototyping Model

- The customers get to see the partial product early in the life cycle. This ensures a greater level of customer satisfaction and comfort.
- New requirements can be easily accommodated as there is scope for refinement.
- Missing functionalities can be easily figured out.
- Errors can be detected much earlier thereby saving a lot of effort and cost, besides enhancing the quality of the software.
- The developed prototype can be reused by the developer for more complicated projects in the future.
- Flexibility in design.

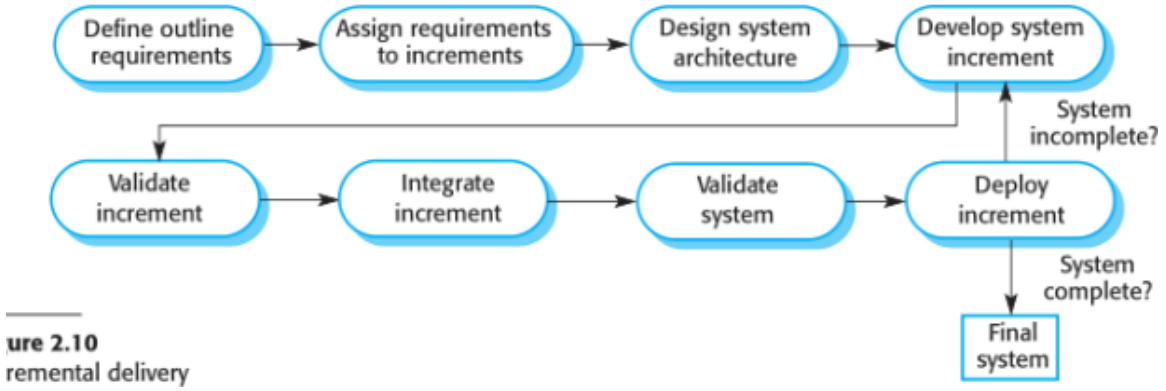
Disadvantages of the Prototyping Model

- Costly with respect to time as well as money.
- There may be too much variation in requirements each time the prototype is evaluated by the customer.
- Poor Documentation due to continuously changing customer requirements.
- It is very difficult for developers to accommodate all the changes demanded by the customer.
- There is uncertainty in determining the number of iterations that would be required before the prototype is finally accepted by the customer.
- After seeing an early prototype, the customers sometimes demand the actual product to be delivered soon.
- Developers in a hurry to build prototypes may end up with sub-optimal solutions.
- The customer might lose interest in the product if he/she is not satisfied with the initial prototype.

Applications of Prototyping Model

- The Prototyping Model should be used when the requirements of the product are not clearly understood or are unstable.
- The prototyping model can also be used if requirements are changing quickly.
- This model can be successfully used for developing user interfaces, high-technology software-intensive systems, and systems with complex algorithms and interfaces.

10. Incremental Model



ure 2.10
remental delivery

Characteristics of an Incremental model –

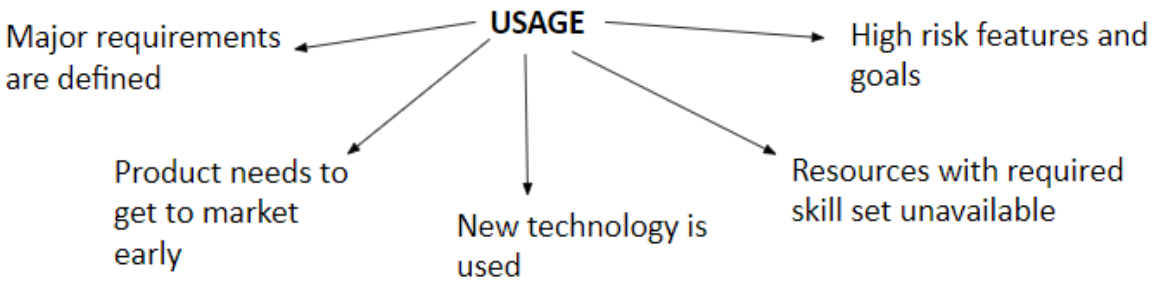
- System development is divided into several smaller projects.
- To create a final complete system, partial systems are constructed one after the other.
- Priority requirements are addressed first.
- The requirements for that increment are frozen once they are created.
- Continuous integration is done until entire system is achieved
- Each subsequent release adds functionality to previous module

Advantages-

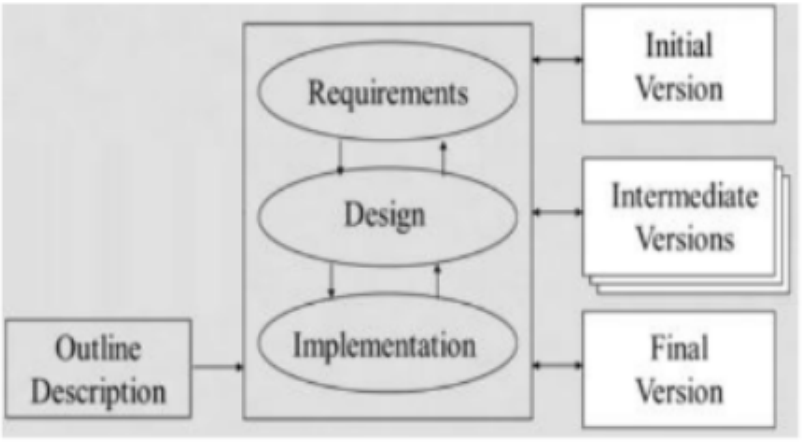
1. Prepares the software fast.
2. Clients have a clear idea of the project.
3. Changes are easy to implement.
4. Provides risk handling support, because of its iterations.
5. Adjusting the criteria and scope is flexible and less costly.
6. Comparing this model to others, it is less expensive.
7. The identification of errors is simple.

Disadvantages-

1. A good team and proper planned execution are required.
2. Because of its continuous iterations the cost increases.
3. Issues may arise from the system design if all needs are not gathered upfront throughout the duration of the program lifecycle.
4. Every iteration step is distinct and does not flow into the next.
5. It takes a lot of time and effort to fix an issue in one unit if it needs to be corrected in all the units.



11. Iterative model



- Initial implementation starts from a skeleton of product
- This is followed by refinement through user feedback & evolution
- Built with dummy modules
- Rapid prototyping
- Successive refinement

ADVANTAGES	DISADVANTAGES
Help identify requirement & solution visualization	Each phase is rigid with overlaps
Support risk mitigation, rework is reduced, incremental investment, increased customer engagement	Costly system architecture may arise

Iterative vs Incremental Model

Aspect	Iterative Model	Incremental Model
Development Approach	Divided into small cycles (iterations).	Divided into functional increments or segments.
Repetition	Repeated cycles with each adding new functionality.	Phased development, each phase adds new features.
Delivery to Customer	Partially functional product after each iteration.	Partial delivery after each completion of an increment.
Feedback and Modification	Feedback and modifications occur after each iteration.	Feedback and modifications occur after each increment.
Risk Management	Suited for projects with evolving or unclear requirements.	Suited for projects where requirements are clear, and delivery in phases is acceptable.
Example Model	Spiral Model, Agile methodologies.	Incremental Model, Waterfall Model with increments.

Similarities:

1. **Iterative Development:**
   - Both iterative and incremental models involve iterative development approaches, where the project is built in stages, and each stage contributes to the final product.
2. **Feedback and Modification:**
   - In both models, there is a provision for gathering feedback from users or stakeholders at certain points in the development process. Modifications and improvements are made based on this feedback.
3. **Phased Delivery:**
   - Both models support phased delivery of the product. In iterative models, each iteration delivers a part of the system, while in incremental models, each increment contributes to the overall system.
4. **Adaptability:**

- Both iterative and incremental models are adaptable to changes in requirements during the development process. They allow for flexibility and adjustment based on evolving needs.

5. **Risk Management:**
   - Both models offer approaches to manage risks effectively. Iterative models manage risks through regular feedback and adaptation, while incremental models manage risks by delivering functional increments progressively.

ITERATIVE MODEL	INCREMENTAL MODEL
Revisit and refine everything	No need to go back and change delivered things
Focus on details of things	Focus on things not implemented yet
Leverage on learnings	Does not leverage on experience or knowledge

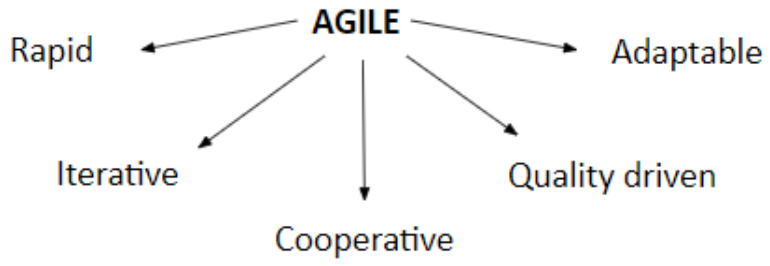
AGILE

The Agile Model was primarily designed to help a project adapt quickly to change requests. So, the main aim of the Agile model is to facilitate quick project completion. To accomplish this task, agility is required. Agility is achieved by fitting the process to the project and removing activities that may not be essential for a specific project. Also, anything that is a waste of time and effort is avoided.

Why was Agile introduced?

Agile was introduced as a response to the limitations of traditional software development methodologies like Waterfall. Traditional approaches struggled with changing requirements, rapid technological advancements, and a lack of adaptability. Agile addressed these challenges by emphasizing flexibility, collaboration, and iterative development. It promotes customer-centricity, empowering development teams to make decisions collectively, and focusing on delivering value to the customer.

Agile Philosophy



Agile Manifesto

1. I**ndividuals and Interactions over Processes and Tools:**
    - Agile places a strong emphasis on the importance of individuals and their interactions within a development team. While processes and tools are necessary, Agile values the human element, effective communication, and collaboration.
2. **Working Solutions over Comprehensive Documentation:**
    - Agile values a focus on delivering working and functional software over extensive documentation. While documentation is essential, the primary measure of progress is the creation of a working product that meets user needs.
3. **Customer Collaboration over Contract Negotiation:**
    - Agile encourages collaboration with customers and stakeholders throughout the development process. Customer feedback is sought regularly, and there is a willingness to adapt to changing requirements, fostering a partnership rather than a contractual relationship.
4. **Responding to Change over Following a Plan:**
    - Agile acknowledges that change is inevitable in software development. It values the ability to respond to changing requirements and priorities over strictly adhering to a predetermined plan. Agile teams embrace change as a means of delivering better solutions.
5. Welcoming changing requirements, even late in development.
6. Delivering working software frequently, with a preference for shorter timescales.
7. Building projects around motivated individuals and giving them the environment and support they need.
8. Trusting motivated individuals to get the job done.
9. Reflecting regularly on processes and adjusting them for continuous improvement.
10. Ensuring sustainable development, maintaining a consistent pace of work to avoid burnout.

Pros	Cons
Is a very realistic approach to software development Promotes teamwork and cross training. Functionality can be developed rapidly and demonstrated. Resource requirements are minimum. Suitable for fixed or changing requirements Delivers early partial working solutions. Good model for environments that change steadily. Minimal rules, documentation easily employed. Enables concurrent development and delivery within an overall planned context. Little or no planning required Easy to manage Gives flexibility to developers	Not suitable for handling complex dependencies. More risk of sustainability, maintainability and extensibility. An overall plan, an agile leader and agile PM practice is a must without which it will not work. Strict delivery management dictates the scope, functionality to be delivered, and adjustments to meet the deadlines. Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction. There is very high individual dependency, since there is minimum documentation generated. Transfer of technology to new team members may be quite challenging due to lack of documentation.

When To Use the Agile Model?

- When frequent modifications need to be made, this method is implemented.
- When a highly qualified and experienced team is available.
- When a customer is ready to have a meeting with the team all the time.
- when the project needs to be delivered quickly.
- Projects with few regulatory requirements or not certain requirements.
- projects utilizing a less-than-strict current methodology

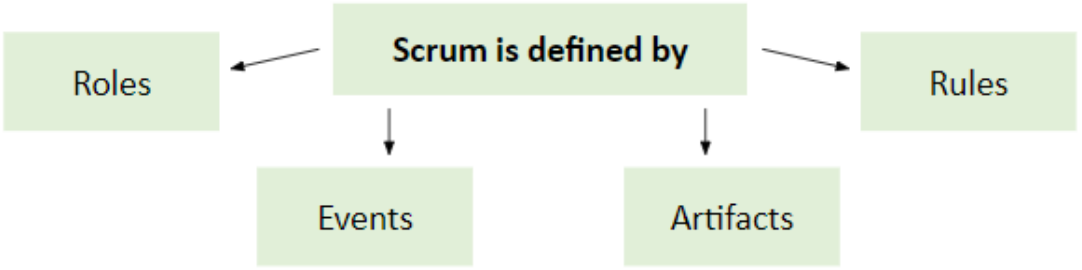
Agile Methodologies

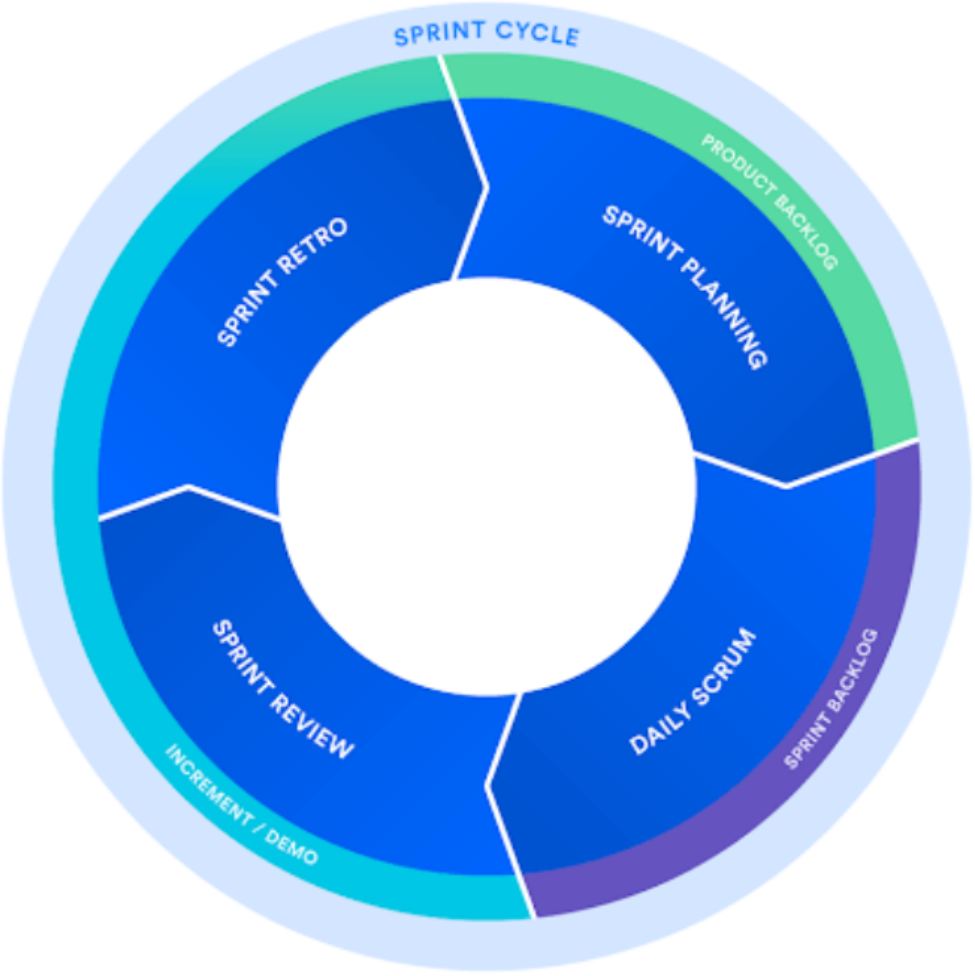
- **Scrum:** Principle-based project management
- **Kanban:** Visual workflows and processes
- **Scrumban:** Hybrid of Scrum and Kanban
- **XP:** Customer-focused product development

Scrum

Scrum is an Agile framework for managing and delivering complex projects, primarily in the realm of software development. It provides a lightweight, flexible, and iterative

approach to project management, emphasizing collaboration, adaptability, and delivering value to the customer.





Scrum is a framework for getting work done, whereas agile is a philosophy. The agile philosophy centers around continuous incremental improvement through small and frequent releases

Scrum Team

A scrum team needs three specific roles: product owner, scrum master, and the development team

Roles:

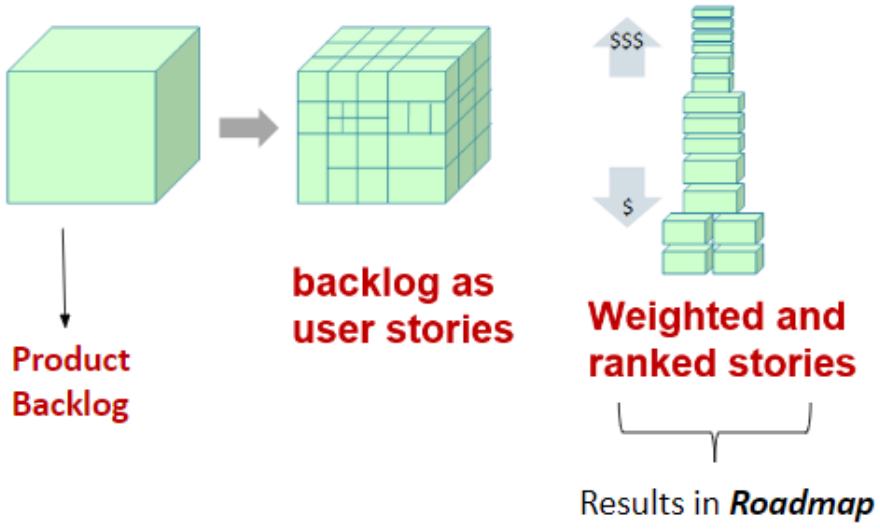
- **Product Owner:** Represents the stakeholders and is responsible for defining and prioritizing the product backlog.
- **Scrum Master:** Facilitates the Scrum process, removes impediments, and ensures that the team follows Scrum practices.
- **Development Team:** Cross-functional, self-organizing group responsible for delivering the product incrementally.

Artifacts

- **Product Backlog:** An ordered list of all features, enhancements, and fixes that constitute the product's roadmap.

  What does Product Backlog include?

  - New features
  - Changes to existing features
  - Bug fixes
  - Infrastructure setups etc.
- **Sprint Backlog:** A subset of the product backlog chosen for a specific sprint, representing the work to be done during that sprint.
- **Increment:** The sum of all completed product backlog items at the end of a sprint. It should be potentially shippable to the customer.



The stories are ranked by importance by estimating the amount of work needed to be done by **Scrum team**

SPRINT

- **Sprint:** A time-boxed period (usually 2 to 4 weeks) during which a potentially shippable product increment is created.
- **Sprint Planning:** A meeting where the team decides which backlog items to include in the upcoming sprint and how to achieve them.
- **Daily Scrum:** A short daily meeting where the team discusses progress, plans for the day, and addresses any obstacles.
- **Sprint Review:** A meeting at the end of a sprint to inspect the increment and adapt the product backlog if needed.
- **Sprint Retrospective:** A meeting at the end of a sprint for the team to reflect on their process and identify improvements.

1. Sprint Pre-planning:

A Pre-Planning meeting enables the business and stakeholders to focus on prioritization

and preparation of requirements far advance before the sprint planning session.

- Every sprint will need to have one prioritized list of requirements (set by business represented by the product owner) so there can be a focus on, and deliver the most valuable and needed requirements in a very short time
- It is quite challenging to prepare the backlog for any upcoming sprint
- Product Owner needs to talk and align requirements of multiple stakeholders which is not easy as every stakeholder has their own priorities that influence the plans of other stakeholders

2. Sprint Planning:

Who attends the Sprint Planning?

Scrum Master	• Facilitate the Sprint planning meeting • Ensures agreement on the Sprint goal and product backlog items

Inputs to Sprint Planning	Activities of Sprint Planning
Product backlog	Identify Sprint goal
Sprint team capacity	Choose user stories
Past performance of Dev team	Plan for capacity

3. Sprint Review

Objectives of the Sprint Review:

Product owner does the following:

- Evaluates against preset criteria
- Gets feedback from clients and stakeholders
- Ensures the delivered increment meets the business need
- Helps support reprioritizing of the product backlog
- Optimize the release plan if needed

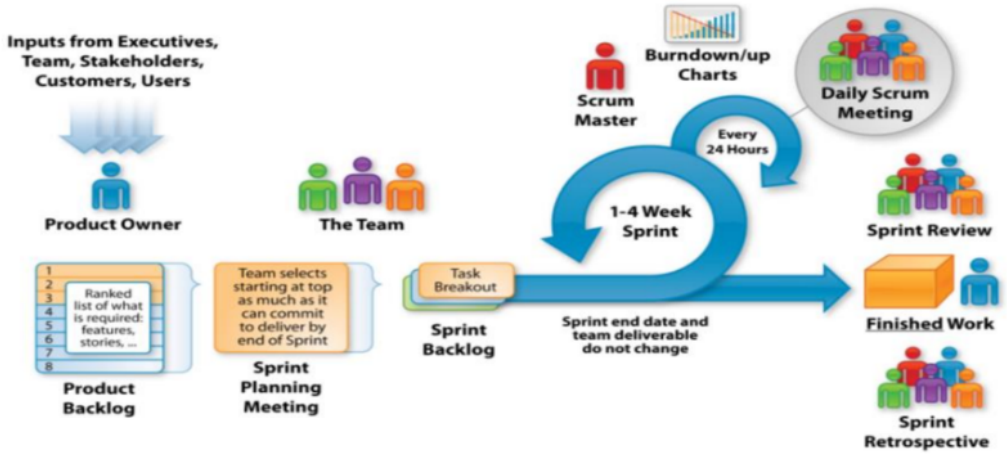
Participants:

- **Scrum Team:** Development Team, Scrum Master, and Product Owner.
- **Stakeholders:** Anyone interested in the product, such as customers, end-users, management, and other relevant parties.

4. Sprint Retrospective

It is the final team meeting in the Sprint to determine what went well, what didn't go well, and how the team can improve in the next Sprint. Attended by the team and the Scrum Master.

It is used for optimizing the process after every iteration.



How Scrum aligns with Agile Manifesto

Individuals and interactions over Processes and tools - Scrum addresses this with Cross functional teams, Scrum Meetings, Sprint reviews

Working software over Comprehensive documentation - Periodic customer experienceable deliverables at the end of every sprint which can be reviewed and experienced

Customer collaboration over Contract negotiation - Having customers to experience the sprint outcomes and participate in sprint reviews to ensure they can visualize and ensure that product meets their needs

Responding to change over Following a plan - User stories which is picked at the beginning of every sprint which can ensure requirement changes can be factored in and prioritized unlike a plan which needs to be followed

Focus on Simplicity in both the product and the process by keeping the process simple, planning is short term focused and hence simple with more interactions and minimal documentation

Extreme Agile Programming

- The development team estimates, plans, and delivers the highest priority user stories in the form of working, tested software on an iteration by iteration basis
- Delivery of working software at very frequent intervals, typically every 1-2 weeks

- Continuous feedback and test-driven development

Core Practices of Extreme Programming:

1. **Test-Driven Development (TDD):**
    - Developers write automated tests before writing the actual code. This ensures that the code meets the specified requirements and allows for easy refactoring.
2. **Pair Programming:**
    - Developers work in pairs, with one writing code (driver) and the other reviewing the work and suggesting improvements (observer). This practice fosters knowledge sharing and improves code quality.
3. **Continuous Integration:**
    - Code changes are integrated frequently into a shared repository, and automated builds and tests are run regularly to identify and address integration issues early.
4. **Refactoring:**
    - Regularly improving the design of the code without changing its external behavior. Refactoring is an ongoing activity to maintain code quality and adapt to evolving requirements.
5. **Collective Code Ownership:**
    - All team members are responsible for the entire codebase. This promotes collaboration, knowledge sharing, and a sense of collective ownership of the software.
6. **Small Releases:**
    - Advocates for delivering small, incremental releases of the software to obtain early feedback and allow for rapid adaptation to changing requirements.
7. **Simple Design:**
    - Encourages the team to keep the design of the system as simple as possible, avoiding unnecessary complexity. The goal is to enhance maintainability and reduce the likelihood of defects.

Extreme Programming is particularly well-suited for small to medium-sized teams working on projects with rapidly changing requirements. Its emphasis on technical excellence, collaboration, and adaptability makes it a valuable approach for delivering high-quality software in dynamic environments.

Aspect	Extreme Programming (XP)	Scrum
Development Focus	Emphasizes technical excellence and coding practices.	Emphasizes flexible project management and adaptive planning.
Roles	Small, cross-functional teams with no specific roles.	Clearly defined roles: Product Owner, Scrum Master, Team.
Iterative Development	Iterative development with frequent releases of small increments.	Iterative development with regular sprint cycles.
Customer Involvement	Constant collaboration with customers and on-site customer presence is encouraged.	Regular collaboration with the Product Owner and stakeholders.
Planning	Adaptive planning and continuous adjustments based on changing requirements.	Fixed-length iterations (sprints) with a predefined plan.
Release Frequency	Frequent, small releases to obtain quick feedback.	Releases occur at the end of each sprint.
Pair Programming	Encourages pair programming (two developers working together at one workstation).	Not a standard practice but may be adopted by some teams.
Testing Approach	Test-Driven Development (TDD) is a core practice.	Testing is integrated into the development process.
Documentation	Minimal upfront documentation. Emphasis on code readability.	Documentation is important, but emphasis on working software.
Continuous Integration	Advocates for continuous integration and automated testing.	Emphasizes continuous integration with regular build and test cycles.
Roles and Responsibilities	Team members share responsibilities, including coding, testing, and customer interaction.	Clear role distinctions, with each role having specific responsibilities.
Adoption Challenges	May be challenging for teams without a strong emphasis on engineering practices.	Easier to adopt due to a simpler framework and clearly defined roles.

Lean Agile

Lean Agile, often referred to simply as Lean or Lean-Agile, is an approach that combines Lean principles with Agile methodologies to enhance efficiency, reduce waste, and deliver value to customers.

- Eliminating waste

- Continuous inspection to adapt and improve. (typically called Kaizen)
- Looks to boost performance
- Focus is to provide a product which addresses the customer needs and expectations in the most efficient fashion

1. **Lean Principles:**
    - Lean principles, originating from manufacturing and the Toyota Production System, focus on eliminating waste, improving efficiency, and delivering value to customers. In Lean Agile, these principles are adapted for software development and project management.
2. **Customer-Centric Focus:**
    - Lean Agile places a strong emphasis on delivering value to customers. Customer feedback and collaboration are key components of Lean Agile practices, ensuring that the product aligns with customer needs and expectations.
3. **Visual Management:**
    - Visual management techniques, such as Kanban boards, are often used to provide transparency into work processes. Visualization helps teams and stakeholders understand the status of work items, identify bottlenecks, and optimize flow.
4. **Reducing Waste:**
    - Lean Agile seeks to minimize or eliminate waste in the development process. This includes reducing unnecessary documentation, avoiding delays in handovers, and optimizing resource utilization.

Lean VS XP VS Scrum

Aspect	Lean	Extreme Programming (XP)	Scrum
Philosophy	Originally from manufacturing, focuses on efficiency, value, and waste reduction.	Focuses on high-quality software development through engineering practices and adaptability.	Agile framework emphasizing collaboration, adaptability, and incremental development.

Development Focus	Efficiency, eliminating waste, and delivering value to customers.	High-quality software development practices, including test-driven development and pair programming.	Flexibility, adaptability, and delivering value in fixed-length iterations (sprints).
Roles	Emphasizes cross-functional teams and minimizing hierarchical structures.	No specific roles prescribed. Team members may take on various responsibilities.	Clearly defined roles: Product Owner, Scrum Master, Development Team.
Customer Involvement	Customer-centric approach with a focus on delivering value to customers.	Constant collaboration with customers, on-site customer presence is encouraged.	Regular collaboration with the Product Owner and stakeholders.
Testing Approach	Testing is integrated into the development process.	Core practice of Test-Driven Development (TDD) and continuous testing.	Testing is integrated into the development process, with emphasis on quality assurance.
Pair Programming	Not explicitly emphasized in Lean.	Encourages pair programming (two developers working together).	Not a standard practice but may be adopted by some teams.
Continuous Integration	Emphasizes continuous integration with regular build and test cycles.	Continuous integration is a fundamental practice.	Continuous integration is a standard practice with regular build and test cycles.
Project Control	Emphasis on self-organizing teams and individual empowerment.	Team members share responsibilities, including coding, testing, and customer interaction.	Employs a structured framework with specific roles and ceremonies.
Adoption Challenges	May require a cultural shift in organizations.	May be challenging for teams without a strong emphasis on engineering practices.	Easier to adopt due to a simpler framework and clearly defined roles.
Visual Management	Utilizes visual management techniques like Kanban boards.	Visual management techniques may be used, such as Kanban boards.	Visual management is often used, such as Scrum boards for tracking work progress.
Continuous Improvement	Core principle, encourages regular reflection and improvement.	Core principle, involves continuous reflection and improvement.	Core principle, regular retrospectives for reflection and

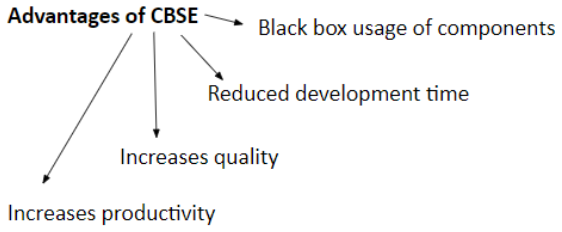
			continuous improvement.
Lean Portfolio Management	Lean principles extend to portfolio level with a focus on value stream mapping.	Lean principles extend to portfolio level.	Lean principles extend to portfolio level, with emphasis on value-driven initiatives and alignment.

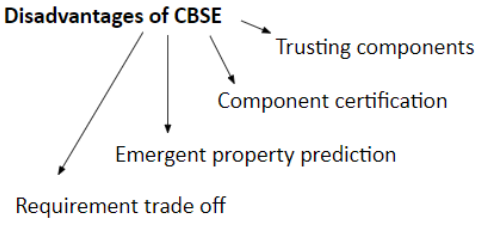
Component Based Software Engineering

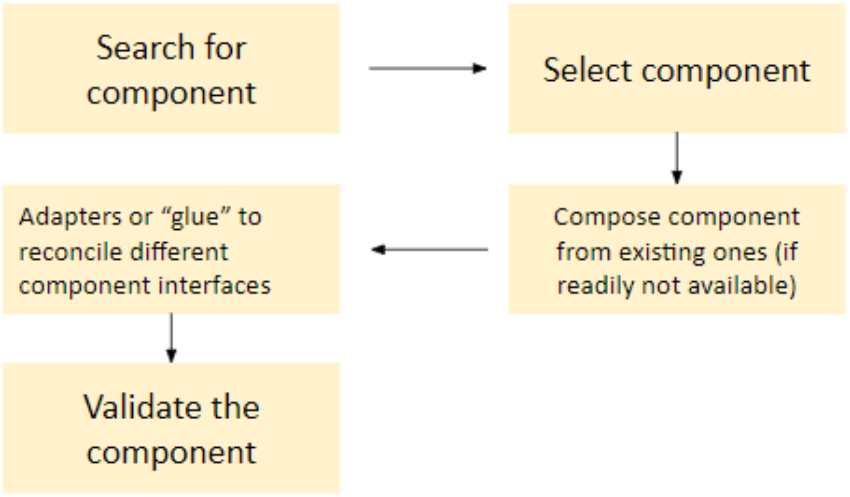
Component-Based Software Engineering (CBSE) is an approach to software development that involves building software systems using pre-defined, loosly coupled, reusable components. In CBSE, a component is a modular and self-contained unit that encapsulates a set of functionalities or services. These components can be independently developed, tested, and maintained, and they can be combined to create complex software systems

Why do we need CBSE?

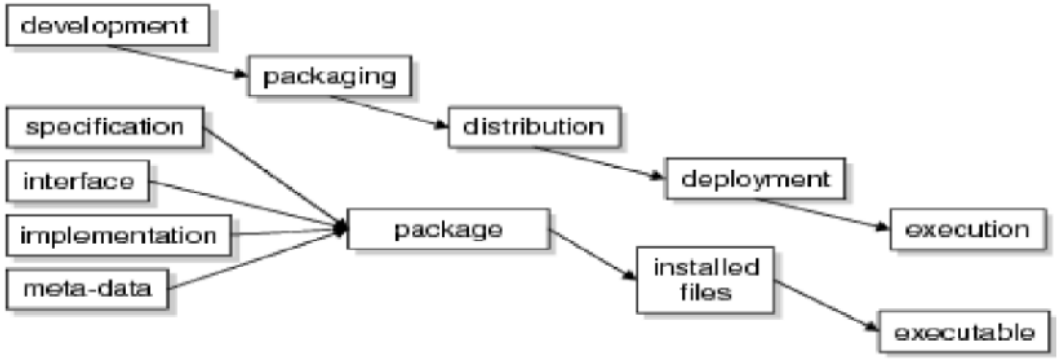
- Increase in complexity of systems
- Reuse rather than re-implement and shorten development time







Component Development Stages



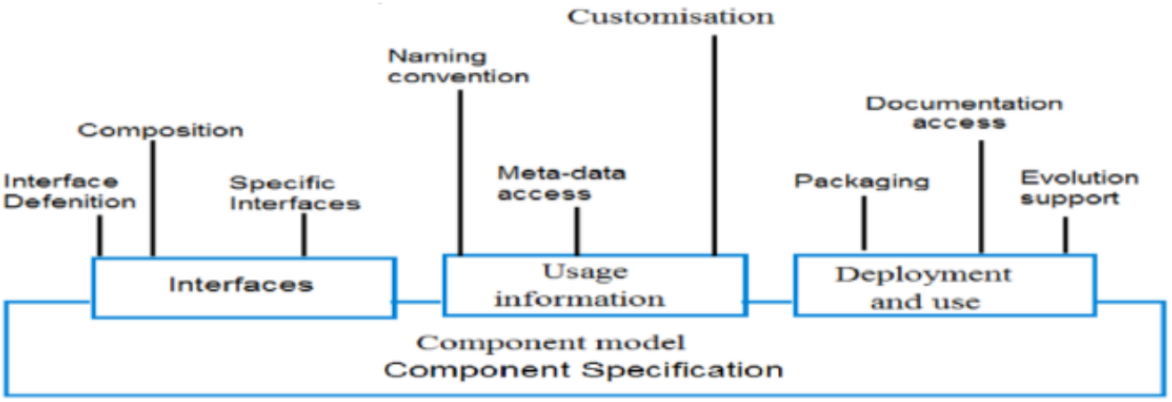
During development – UML

When packaging – .zip

In the execution stage – blocks of code and data

Component Model

Component model – defines the types of building block, which can be composed with other components to create a software system



Product Lines

A product line, in the context of software engineering and product development, refers to a collection or family of related software products that share a common set of features, functionalities, and core architecture.

Software Product Line Engineering makes it possible to

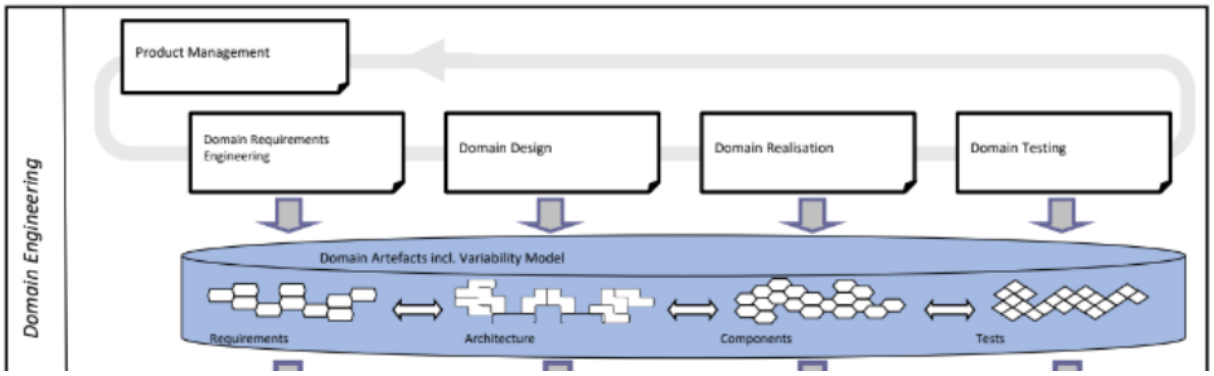
- Create software for different products
- Use variability to customize the software to each different product

Benefits of a software product line approach include:

- **Efficiency and Reusability:** By leveraging common assets and reusable components, development efforts can be more efficient, and the cost of creating new products is reduced.
- **Consistency and Quality:** Common features and components contribute to consistency across products, and improvements or bug fixes made to shared assets benefit all products.
- **Flexibility and Adaptability:** Variability allows for the creation of customized products to address different market needs or customer requirements.
- **Time-to-Market:** The ability to quickly derive new products from a well-established product line can result in faster time-to-market for new offerings.
- **Economies of Scale:** Development efforts are streamlined, and economies of scale are achieved by managing a family of products collectively rather than treating each product as a standalone project.

Domain Engineering

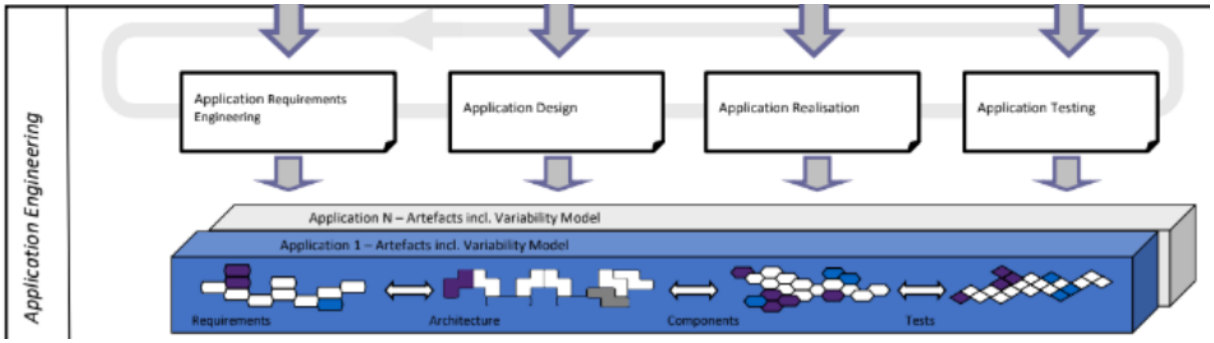
Domain engineering focuses on creating the core assets that form the foundation for the entire product line. This includes the development of reusable components, common architectures, and shared modules. The emphasis is on building a solid foundation that can be leveraged across multiple products.



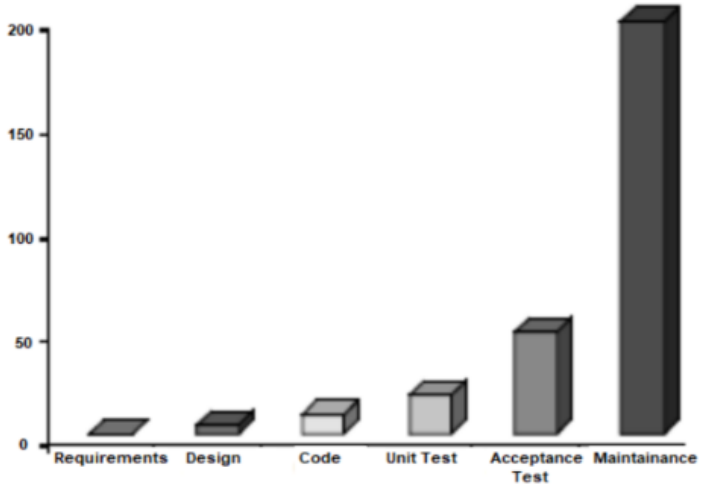
Application Engineering

Derived from the platform set by domain engineering reuses domain engineering artifacts

"Application engineering" typically refers to the process of designing, developing, testing, and maintaining software applications. It encompasses the entire software development lifecycle and involves various stages and activities to create functional and effective applications.



Cost of repair



Life Cycle Stage	Relative cost of Repair
Requirements	0.1 to 0.2
Design	0.5
Coding	1
Unit test	2
Acceptance test	5
Maintenance	20

Requirement Engineering

The primary goal of requirement engineering is to understand, document, and manage the needs and expectations of stakeholders to ensure that the software system meets those requirements effectively.

Properties of requirement

- Concise - Requirements should describe a single property
- Unambigious - Requirement should have only one interpretation
- Verifiable - Requirement must have a clear, testable criterion and a cost-effective process to check it has been realized as requested
- Quantifiable – Requirement should be quantifiable
- Consistent – No requirement should contradict another
- Traceable – Backwards to stakeholder request and forward to software components
- Feasible – Realizable with a specified time frame
- Clear – Written precisely
- Prioritized – Requirement should be prioritized

All screens must appear quickly on the monitor
When the user accesses any screen, it must appear on the monitor within 2 seconds (Clear, Concise, Unambiguous, Verifiable, Measurable)

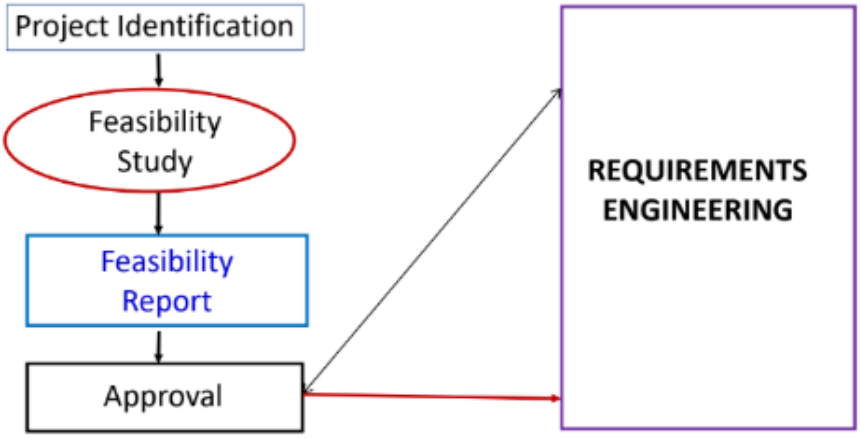
The replacement control system shall be installed with no disruption to production
The replacement control system shall be installed causing no more than 2 days of production disruption (Feasible)

The system must generate a batch end report and a discrepancy report when a batch is aborted
The system must generate a batch end report when a batch is completed or aborted
The system must generate a discrepancy report when a batch is aborted (Traceable)

The system must be user friendly
The user interface shall be menu driven. It shall provide dialog boxes, help screens, radio buttons, dropdown list boxes, and spin buttons for user inputs (Verifiable)

Feasibility Study

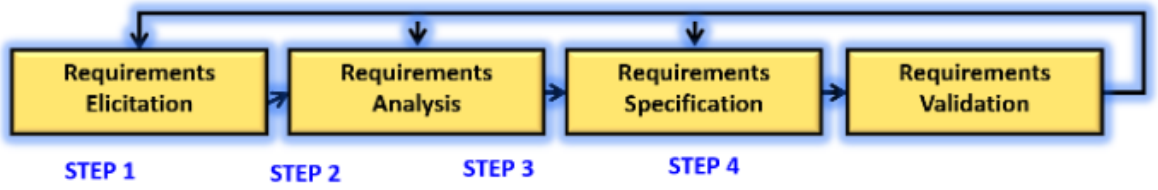
Short, low-cost study to assess the practicality of the project and whether it should be done



- Figure out the client or the sponsor or the user who would have a stake in the project
- Find the current solution to the problem
- Find the targeted customers and the future market place
- Potential benefits
- Scope
- High level block level understanding of the solution
- Considerations to technology

- Marketing strategy
- Financial projection

Requirement Engineering Process



Requirements Elicitation – To gather and discover requirements

Requirements Analysis – Process of reviewing requirements

Requirements Specifications – Process of creating software and system specification documents suitable for further designing and maintenance

Requirements Validation – Helps ensure the right requirements are realized

1. Requirement Elicitation

It is the process of working proactively with all stakeholders gathering their needs, articulating their problem, identify and negotiate potential conflicts thereby establishing a clear scope and boundary for a project.

It involves:

- Understanding the problem
- Understanding the domain
- Identifying clear objectives
- Understanding the needs
- Understanding constraints of the system stake holders
- Writing business objectives for the project

Elicitation Technnique

Active: Ongoing elicitation between the user and the stakeholder

- Interviews

- Facilitated meetings
- Role-playing
- Prototypes
- Ethnography
- Scenarios

Passive: Infrequent elicitation between user and stakeholder

- Use cases
- Business process analysis & modelling
- Workflows
- Questionnaires
- Checklists
- Documentation

2. Requirement Analysis

Requirement Analysis in Software Development

Requirement Analysis is a pivotal phase in the software development lifecycle, focusing on comprehending, capturing, and documenting stakeholder needs effectively.

1. Understand Requirements In-Depth:

- Internalize the problem or requirement, considering both product and process perspectives.
- Ensure a correct interpretation of the requirement to align with stakeholder expectations.

2. Classify Requirements Into Coherent Clusters:

- *Functional Requirements:* Define system functionalities and behaviors in various scenarios.
- *Non-Functional Requirements:* Specify constraints on services, timing, and development processes.
- *User Requirements:* Articulate needs in natural language, often written by customers.

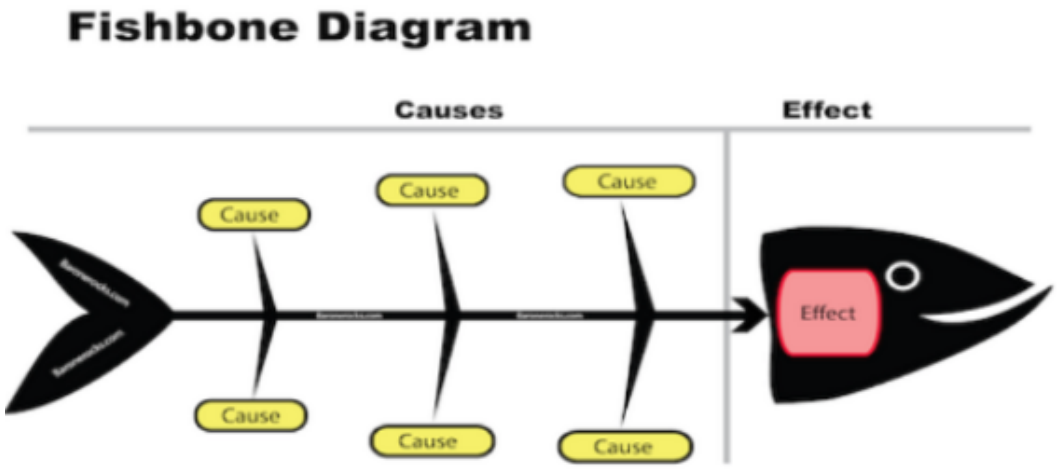
- *System Requirements:* Structured documents detailing functions, services, and constraints.
- *Domain Requirements:* Constraints stemming from the domain of operation.

3. Model the Requirements:

- Employ models to represent the system, enhancing understanding and communication.
- *Structural Models:* Capture static aspects of the system (e.g., Class diagrams).
- *Behavioral Models:* Depict dynamic interactions between entities (e.g., Use Case diagrams).

4. Analyze Requirements Using Fish Bone Diagram:

- Utilize a Fish Bone Diagram to identify causes or reasons behind relevant requirements.
- Understand the root causes contributing to the emergence of each requirement.
- The diagram resembles a fish skeleton, with the "head" representing the problem or effect, and the "bones" representing potential causes or categories of factors that may contribute to the problem



5. Recognize and Resolve Conflicts:

- Address conflicts arising from functionality, cost, and timelines.
- Strive to find optimal solutions that balance competing priorities.

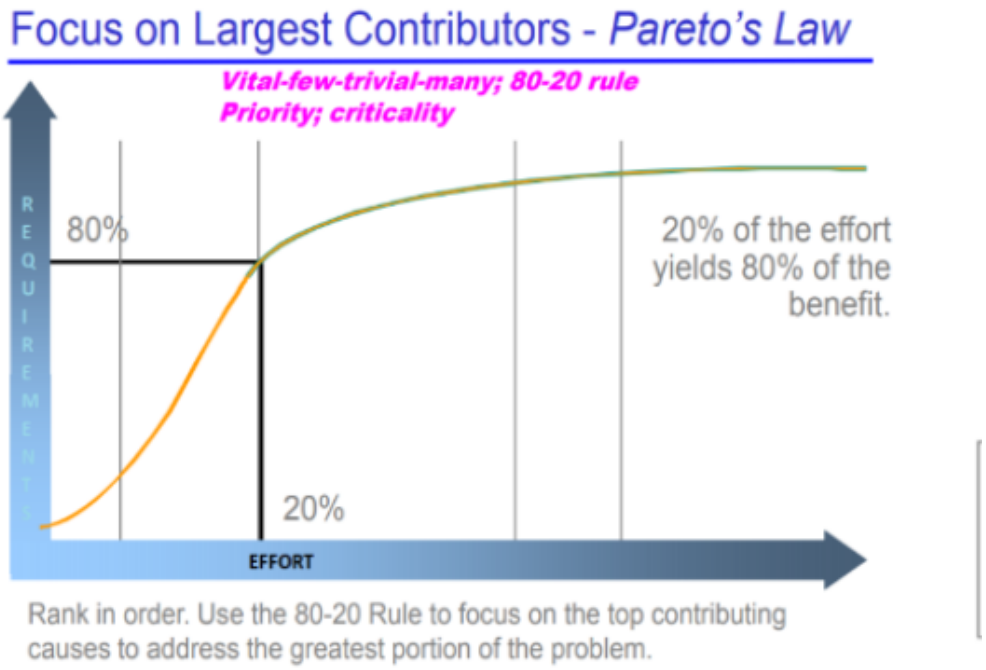
6. Prioritize Requirements - MoSCoW:

- Use MoSCoW prioritization to categorize requirements:

- *Must-haves (M):* Critical and non-negotiable.
- *Should-haves (S):* Important but not critical.
- *Could-haves (C):* Desirable if resources permit.
- *Won't-haves (W):* Excluded from the current scope.

7. Identify Risks:

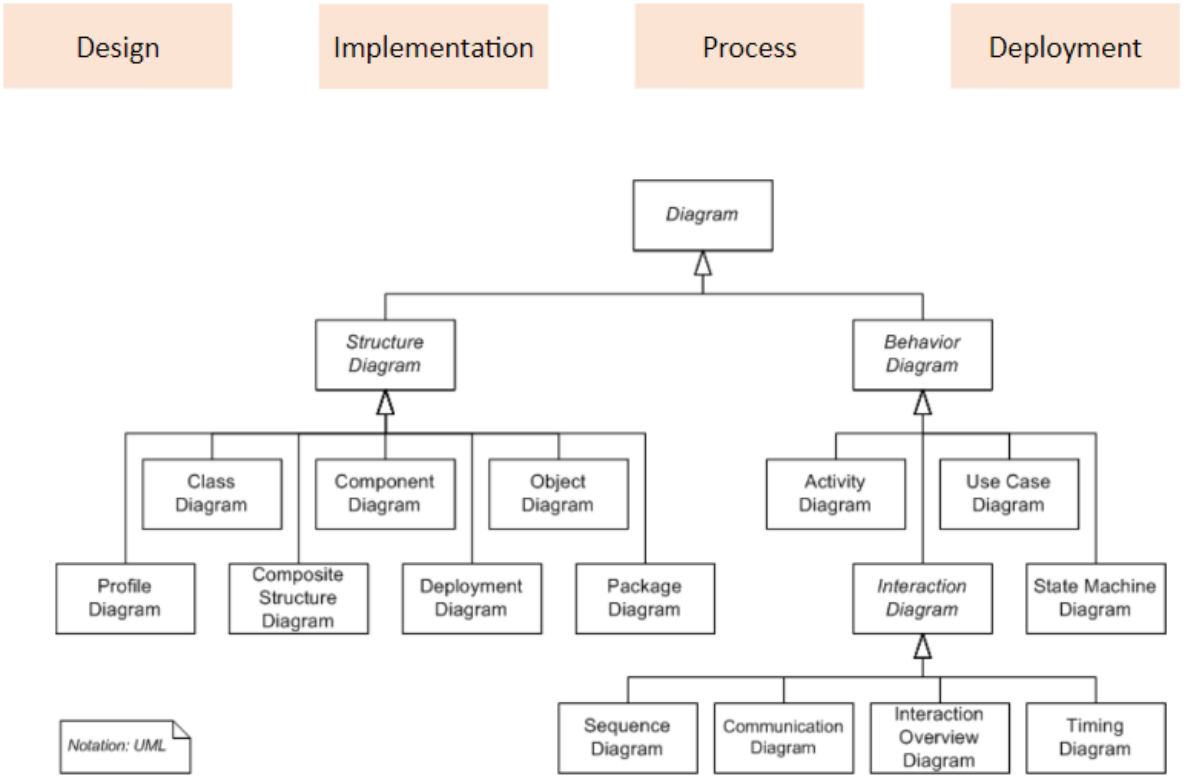
- Conduct a risk analysis to identify potential challenges.
- Consider technical, operational, project management, and business risks.
- Develop strategies for risk mitigation, contingency, or acceptance.



Unified Modeling Language

UML, or Unified Modeling Language, is a standardized modeling language widely used in the field of software engineering for visualizing, specifying, constructing, and documenting the artifacts of a system. It provides a common, visual representation of the various aspects of software systems, making it easier for stakeholders to understand, communicate, and collaborate on complex software projects.

UML Use-case models are predominantly used with Modeling Systems, to discuss the dynamic behavior of the system when it is running/operating. Its often used to used to gather the requirements of a system including internal and external influences.



UML is made up of three conceptual elements.

- Building blocks which make up the UML
- Rules that dictate how these building blocks can be put together
- Common mechanisms that apply through out UML

Use Case Diagrams:

A *Use Case Diagram* is a graphical representation that depicts actors, use cases, and their relationships within a system. This visual tool is crucial for visualizing, specifying, constructing, and documenting the intended behavior of a system during the requirements capture and analysis phase. Let's delve into the key components and concepts:

1. Elements of Use Case Diagrams:

a. Actors:

- Actors represent entities outside the system that interact with it. They can be human users or external systems. Actors are named by nouns and are crucial for outlining the roles that interact with the system's functions.

b. Use Cases:

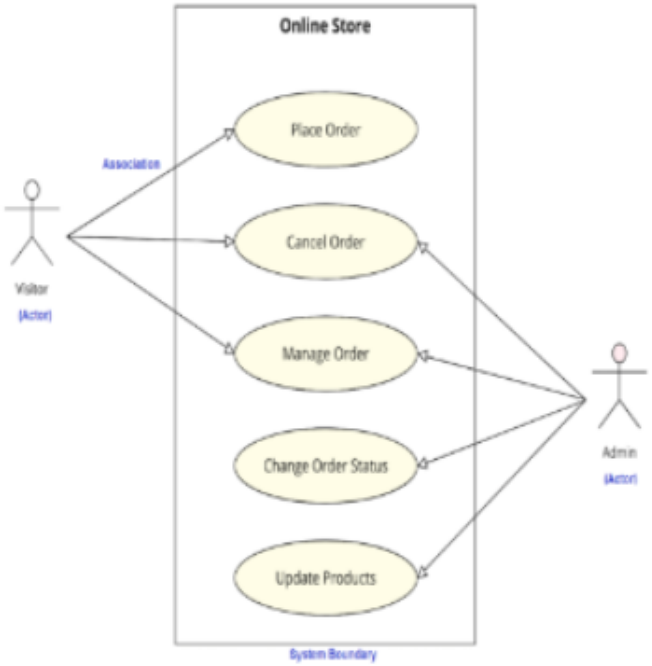
- Use cases represent specific functionalities or tasks that the system performs to fulfill a user's goal. They describe how the system responds to a request or task from an actor. Use cases are typically named with a verb-noun phrase, outlining actions the system can take.

c. Relationships:

- Relationships in use case diagrams connect actors and use cases, illustrating the interactions between them. These relationships denote how actors utilize specific functionalities within the system.

d. System Boundary:

- The system boundary, often depicted as a box, encapsulates all the use cases of the system. It defines the scope of the system and helps distinguish internal elements from external actors.



20

2. Types of Relationships:

a. Association:

- Describes a general relationship between an actor and a use case. It signifies that the actor is somehow associated with the functionality but doesn't specify the nature of the association.

b. Include:

- Denotes that one use case includes the functionality of another. It is used to show a modular or hierarchical structure within the system's functionalities.

c. Extend:

- Illustrates optional or conditional behavior in a use case. An extended use case is invoked under certain conditions, providing additional functionalities.

d. Generalization:

- Represents a specialization-generalization relationship between use cases or actors. It signifies that one use case or actor is a specialized version of another.

3. Use Case from a User's Point of View:

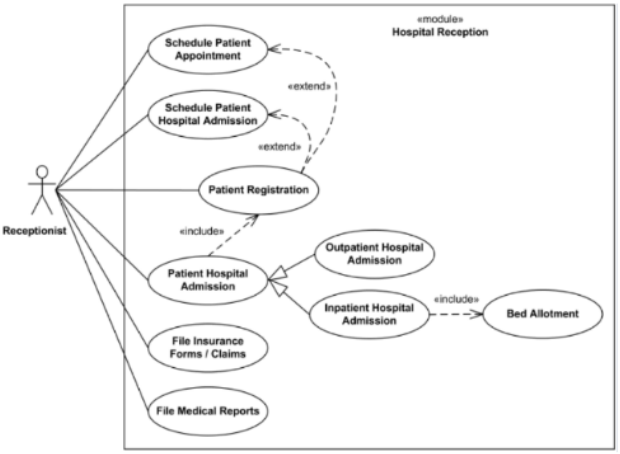
- Use cases, when viewed from a user's perspective, outline how the proposed system will perform tasks in response to a request from a role, actor, or user. It provides a user-centric lens to understand system behavior.

4. Importance and Users of Use Case Diagrams:

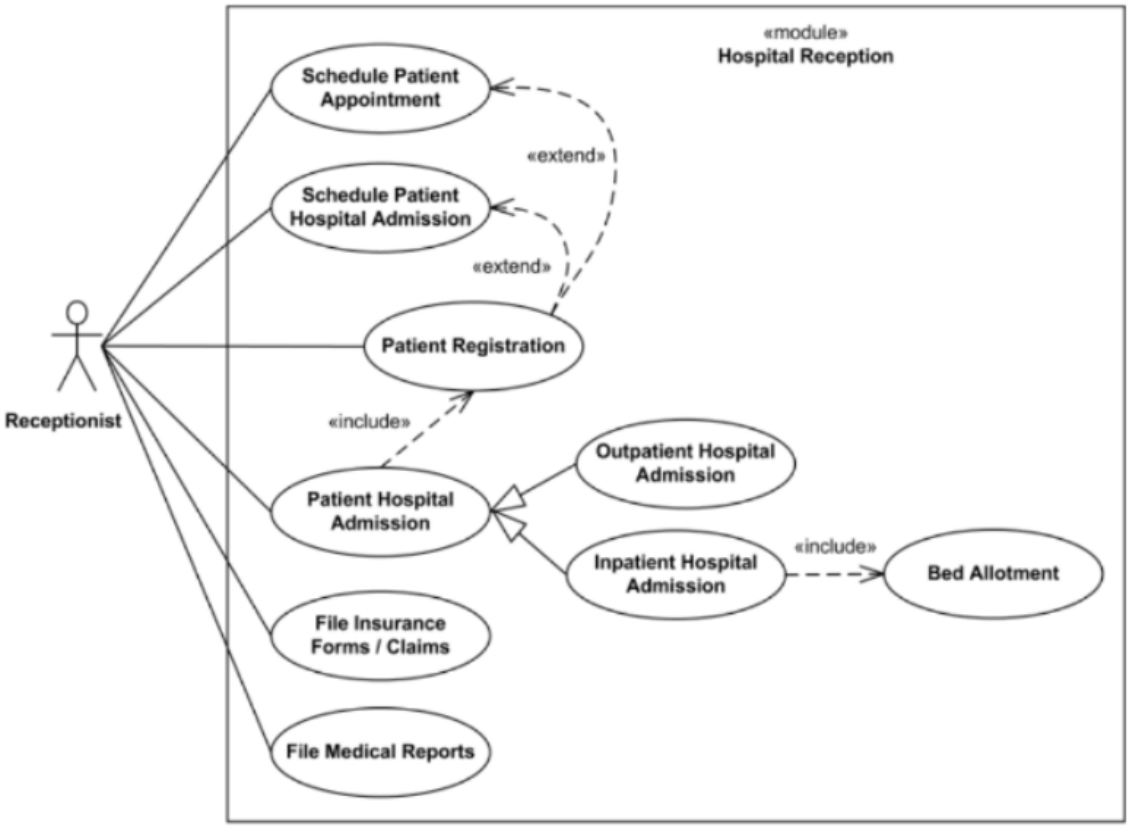
- Use case diagrams are utilized by developers, domain experts, and end-users for:
    - Visualizing system behavior.
    - Specifying and understanding requirements.
    - Constructing a blueprint for system functionalities.
    - Documenting how actors interact with the system.

Use Case Diagram – In-Class Exercise

Using Use Case diagram, to depict the job of a hospital receptionist. Include scheduling appointments, admissions, bed allotment, filing insurance and filing medical reports as some of the use cases.



21



- **Use cases:** Horizontally shaped ovals that represent the different uses that a user might have.
- **Actors:** Stick figures that represent the people actually employing the use cases.
- **Associations:** A line between actors and use cases. In complex diagrams, it is important to know which actors are associated with which use cases.
- **System boundary boxes:** A box that sets a system scope to use cases. All use cases outside the box would be considered outside the scope of that system. For example, Psycho Killer is outside the scope of occupations in the chainsaw example found below.
- **Packages:** A UML shape that allows you to put different elements into groups. Just as with component diagrams, these groupings are represented as file folders.

3. Software Requirement Specification

The *software requirements specification (SRS)* document is the basis for customers and contractors/suppliers agreeing on what the product will and will not do. It describes both the functional and nonfunctional requirements

Functionality: What is the software supposed to do?

External interfaces: How does the software interact with people, the system's hardware, other hardware, and other software?

Non Functionality: This includes all of the Quality criteria which drive the functionality. Performance, Availability, Portability etc.

Design constraints imposed on an implementation:

- Required standards in effect
- Implementation language
- Policies for database integrity
- Resource limits
- Security
- Operating environment(s) etc.

4. Requirement Validation

Validation determines whether the software requirements if implemented, will solve the right problem and satisfy the intended user needs.

Verification determines whether the requirements have been specified correctly

Prototyping

Prototype facilitates user involvement during requirements engineering phase and ensures engineers and users have the same interpretation of the requirements.

Prototyping is most beneficial in systems – With many user interactions

Model Validation

- Ensuring that the models represent all essential functional requirements
- Demonstrating that each model is consistent in itself
- Usage of the Fish Bone Analysis technique for validation

Requirement Management

Requirement Management is a systematic and structured process that involves the planning, monitoring, and controlling of requirements throughout the software development lifecycle. It encompasses activities related to the identification, documentation, organization, and tracking of requirements from their inception to their implementation and beyond. The primary goals of requirement management include ensuring clarity, traceability, and consistency in understanding the project's scope and objectives.

- Ensuring that the requirements are all addressed in each phases of the lifecycle
- Ensuring that the changes in the requirements are handled appropriately

Requirement Change Management

Requirement Change Management specifically focuses on the process of handling changes to requirements throughout the software development lifecycle. It ensures that changes are properly evaluated, documented, and implemented while minimizing negative impacts on the project.

1. Log the Request for Change:

- Capture the request for change and assign a unique identifier. Document the following details:
  - **Requester:** Identify who is requesting the change.
  - **Reason:** Understand why the request is being made.

- **Change Details:** Clearly specify what is being requested to change.

2. Perform Impact Analysis & Estimate Impact:

- Analyze the potential impact of the proposed change on the project. Estimate the effects on scope, schedule, cost, and other relevant factors. This step is crucial for informed decision-making.

3. Review Impact with Stakeholders:

- Engage relevant stakeholders to discuss and review the impact analysis. Ensure that all perspectives are considered, and the implications of the change are thoroughly understood.

4. Log Post-Change Information:

- After implementing the change, log relevant information to maintain a clear record:
  - **Timestamp:** Record when the change was made.
  - **Contributors:** Document who made the changes.
  - **Reviewers:** Specify individuals who reviewed the changes.
  - **Testers:** Identify those responsible for testing the changes.
  - **Release Stream:** Determine which release stream the change will be part of.

5. Rework the Work Products/Items:

- Make necessary adjustments to work products, documentation, or any items affected by the change. Ensure that all artifacts align with the modified requirements.

6. Solicit Formal Approval:

- As part of the approval process, formally seek approval for the change. This may involve obtaining signatures or documented approval from relevant stakeholders.

Additional Considerations:

- **Documentation Management:**
  - Maintain a well-organized and accessible repository for change documentation, ensuring transparency and traceability throughout the process.
- **Change Control Board (CCB):**

- If applicable, involve a Change Control Board or a similar governing body to review and make decisions regarding proposed changes.

- **Communication:**
  - Keep stakeholders informed at each stage of the process. Effective communication is crucial for managing expectations and ensuring alignment with project goals.
- **Lessons Learned:**
  - Periodically review the change management process and capture lessons learned. Use these insights to continuously improve the effectiveness of handling future changes.

Read grey part from slides

Aspect	Requirement Elicitation	Requirement Analysis	Requirement Specification	Requirement Validation
Definition	The process of gathering, identifying, and discovering requirements from stakeholders.	The process of understanding, organizing, and prioritizing gathered requirements.	The process of documenting requirements in a clear and structured manner.	The process of ensuring that the documented requirements meet stakeholder needs and expectations.
Focus	Identifying what stakeholders need or expect from a system.	Understanding the nature of requirements, their relationships, and potential conflicts.	Documenting requirements in a standardized and structured format.	Confirming that the documented requirements accurately represent stakeholder needs.
Activities	- Interviews - Surveys - Workshops - Observations	- Requirement organization - Prioritization - Identifying conflicts - Creating models (e.g.,	- Creating requirement specifications - Use of standard notations (e.g., UML)	- Reviews - Inspections - Prototyping - Testing

		use case diagrams)		
Output	- List of requirements - Use cases - User stories	- Organized requirements - Prioritized requirements - Requirement models	- Requirement documents - Use case diagrams - System models	- Verified requirements - Validated system models - Test cases
Purpose	To uncover, identify, and understand stakeholder needs and expectations.	To analyze and make sense of the gathered requirements, ensuring clarity and consistency.	To document requirements in a standardized format for easy understanding and communication.	To confirm that the documented requirements accurately represent stakeholder expectations and are feasible to implement.
Participants	Stakeholders, end-users, domain experts, project managers, etc.	Business analysts, project managers, system architects, domain experts, etc.	Business analysts, system architects, developers, testers, stakeholders, etc.	Quality assurance professionals, testing teams, stakeholders, etc.
Tools/Techniques	Interviews, surveys, brainstorming, use of prototypes, etc.	Use of models (e.g., use case diagrams, class diagrams), prioritization techniques, etc.	Use of UML, requirement documentation tools, etc.	Inspections, walkthroughs, reviews, testing tools, etc.
Challenges	Understanding unstated needs, conflicting requirements, and evolving expectations.	Managing changing requirements, resolving conflicts, and ensuring completeness.	Maintaining consistency, addressing ambiguity, and ensuring that requirements are testable.	Identifying and resolving discrepancies between requirements and system behavior.