Software Engineering and Project Management

Introduction to Software Engineering and Process Models

Course Code: BTCS402N

Presented by: Juhi Shrivastava

Institution: Shri Vaishnav Vidyapeeth Vishwavidyalaya, Indore

The Evolving Role of Software

- Software is essential in all domains: healthcare, defense, business, education, entertainment.
- Drives innovation in AI, IoT, robotics, mobile computing, and cloud platforms.
- Used in mission-critical systems: avionics, banking, nuclear control, etc.
- Software is no longer just a support system; it's a strategic asset.
- Increases automation, improves efficiency, enhances decision-making.
- Enables digital transformation and competitive advantage.

Changing Nature of Software

- Highly distributed and networked (Cloud, Microservices).
- Adaptive to change (Agile-friendly).
- Secure, scalable, performance-intensive systems.
- Integration with physical devices (Cyber-physical systems).
- Evolving from monolithic to service-oriented architectures.
- Focus on real-time processing, personalization, and data analytics.

Software Myths** **Management Myths:**

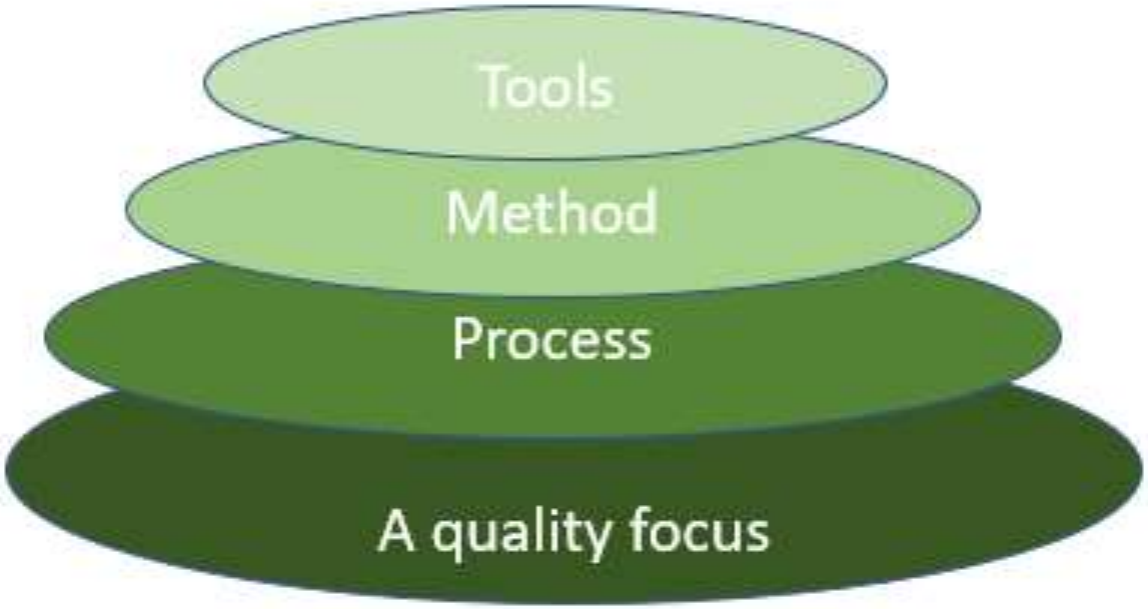
- Adding more people speeds up development.
- Once code is written, the project is finished.
- **Customer Myths:**
- A general statement is enough to start development.
- Software can be easily modified later.
- **Developer Myths:**
- We only need to write code that works.
- Requirements don’t change much.
- **Reality:**
- Late changes are expensive.
- Poor planning leads to project failure.
- Quality assurance must be continuous.
- **Speaker Notes:** Clarify the impact of myths on project failure. Explain how each myth can lead to scope creep, delays, or increased costs.

What is Software Engineering?

- SE is the application of engineering to software development.
- Focuses on structured processes, standards, and best practices.
- Ensures delivery of high-quality software that meets requirements.
- Emphasizes maintainability, scalability, and performance.
- Involves all stages: requirement gathering, design, development, testing, maintenance.
- Incorporates team collaboration, documentation, and lifecycle management.
- **Speaker Notes:** Differentiate SE from ad-hoc coding. Show how it aligns with principles of traditional engineering disciplines.

Layered Technology in SE

- **Quality Focus**: Central objective across all phases. Influences every decision.
- **Process Layer**: Defines framework activities like planning, coding, testing.
- **Methods Layer**: Provides step-by-step technical techniques (e.g., object-oriented design, testing strategies).
- **Tools Layer**: CASE tools, IDEs, automated testing tools that support methods and improve productivity.
- **Speaker Notes:** Show how these layers build upon one another. Give examples such as JIRA (tool), Scrum (process), TDD (method), continuous delivery (quality focus).


Tools
Method
Process
A quality focus

Software Process Framework & Umbrella Activities** **Framework Activities:**

- Communication: Gathering and understanding requirements.
- Planning: Estimating resources, time, cost.
- Modeling: Creating design representations.
- Construction: Actual coding and testing.
- Deployment: Releasing and maintaining the software.
- **Umbrella Activities:**
- Software Configuration Management (version control).
- Software Quality Assurance (standards, audits).
- Risk Management (identifying and mitigating risks).
- Technical Reviews (peer reviews, walkthroughs).
- Documentation (user manuals, SRS, design docs).

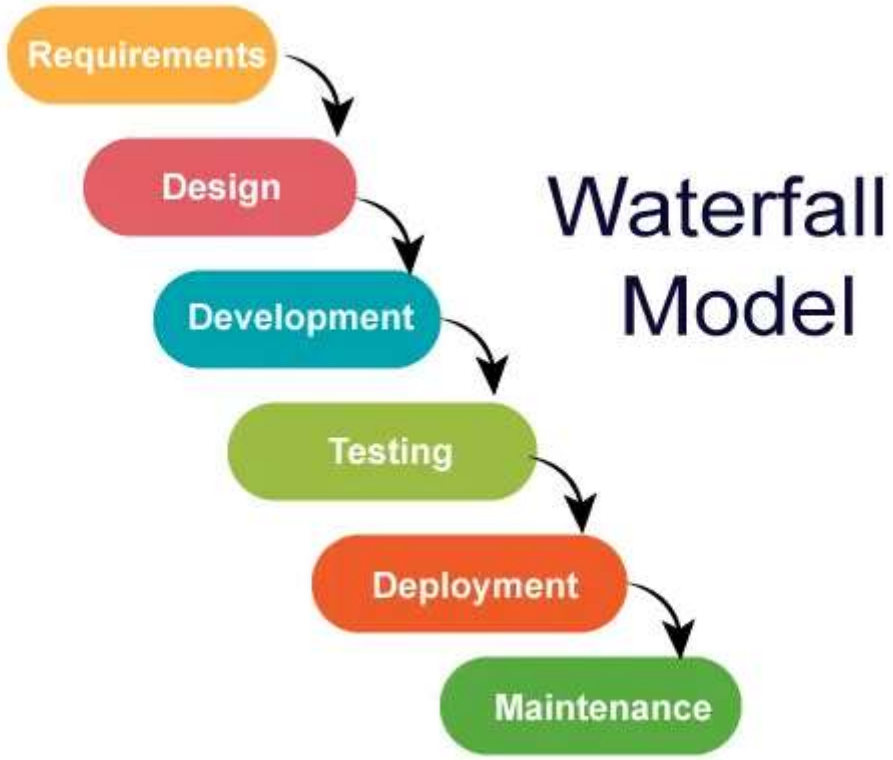
Capability Maturity Model Integration (CMMI)

- Framework to assess and improve software processes.
- Developed by SEI to help organizations improve performance.
- **Maturity Levels:**
  - Level 1: Initial – Ad hoc, chaotic processes.
  - Level 2: Managed – Basic project management.
  - Level 3: Defined – Standardized across organization.
  - Level 4: Quantitatively Managed – Metrics-based management.
  - Level 5: Optimizing – Continuous process improvement.
- Promotes predictability, risk control, and customer satisfaction.

Overview of Software Process Models

- Each model offers a unique approach to managing and executing projects.
- **Waterfall Model**: Sequential, phase-based development.
- **Incremental Model**: Delivers small parts (increments) over time.
- **Spiral Model**: Focuses on risk-driven iteration.
- **Unified Process**: Combines iterative approach with UML modeling.
- **Agile Models**: Focused on collaboration, rapid delivery, and customer feedback.

What is the Waterfall Model?



What is the Waterfall Model?

- A linear and sequential software development model
- Each phase must be completed before the next begins
- Best suited for projects with clearly defined requirements
- Emphasizes documentation and planning

Phases of the Waterfall Model

1. Requirements Analysis
2. System Design
3. Implementation (Coding)
4. Testing
5. Deployment
6. Maintenance

Advantages of the Waterfall Model

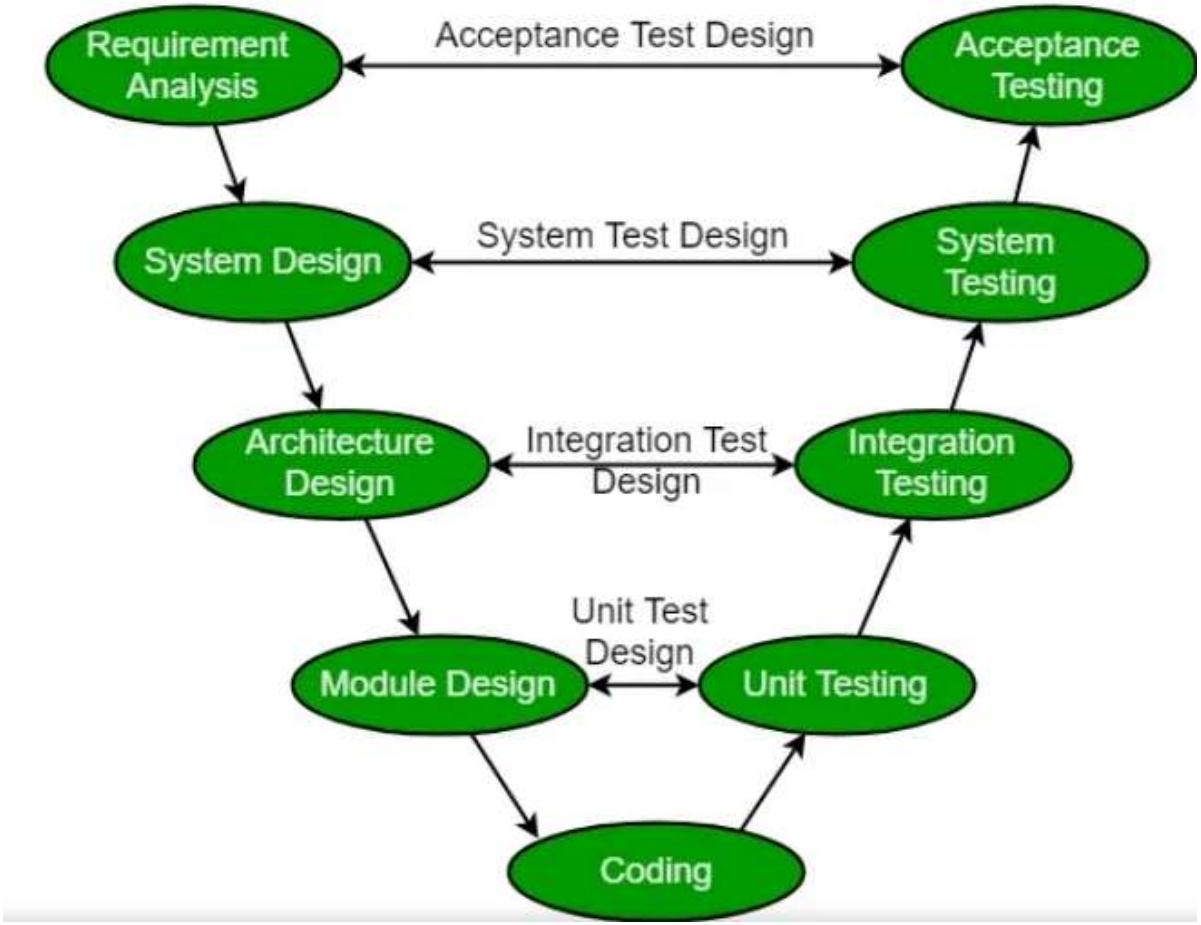
- Simple to understand and use
- Easy to manage due to its rigidity
- Phases are processed and completed one at a time
- Works well for smaller projects with well-defined requirements

Disadvantages of the Waterfall Model

- Inflexible to changing requirements
- Difficult to go back to any phase once completed
- Late discovery of bugs or requirement issues
- Not suitable for complex or long-term projects

When to Use the Waterfall Model

- Projects with fixed and clear requirements
- Projects that require rigorous documentation
- Systems that are not expected to change
- Short-duration and low-complexity projects


Requirement Analysis
Acceptance Test Design
Acceptance Testing
System Design
System Test Design
System Testing
Architecture Design
Integration Test Design
Integration Testing
Unit Test Design
Module Design
Unit Testing
Coding

- The V-Model, which includes the **Verification and Validation** it is a structural approach to the software development.

Introduction to V-Model

- The V-Model is an extension of the Waterfall Model.
- Also known as the Verification and Validation model.
- Development and testing activities are planned in parallel.
- Emphasizes testing in each development phase.

Verification (Left Side of V)

- **Requirements Analysis** – Understand user needs
- **System Design** – Define system architecture
- **High-Level Design** – Outline module interactions
- **Low-Level Design** – Detail module functionalities

Validation (Right Side of V)

- **Unit Testing** – Test individual modules
- **Integration Testing** – Test combined modules
- **System Testing** – Test complete system functionality
- **Acceptance Testing** – Test with user requirements

Advantages of V-Model

- Simple and easy to use
- Clear milestones and deliverables
- Testing activities occur early
- Better quality due to early defect detection

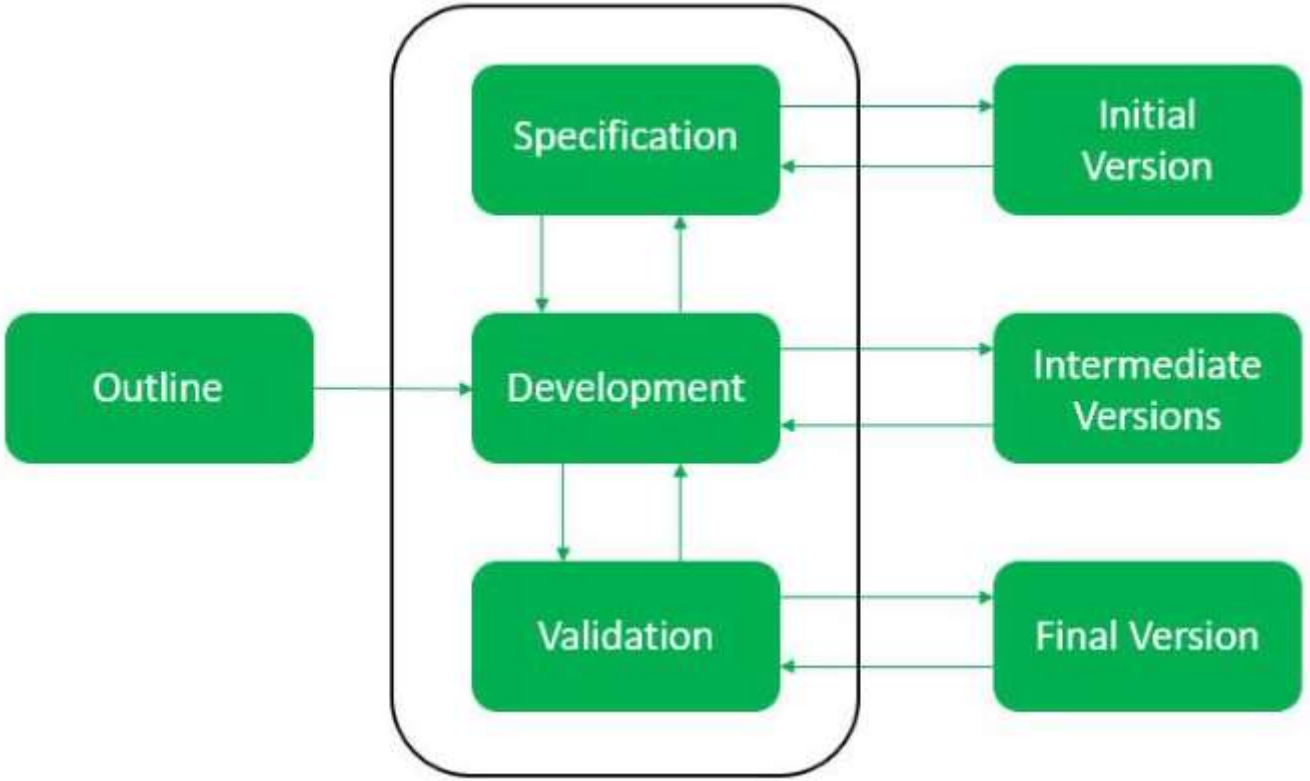
Disadvantages of V-Model

- Rigid and inflexible
- Not suitable for iterative development
- High risk if requirements are unclear
- Limited customer involvement once development starts

When to Use V-Model

- Requirements are well-defined and fixed
- Projects with no expected changes
- Projects with high reliability needs (e.g., medical, defense)

Evolutionary Process Model



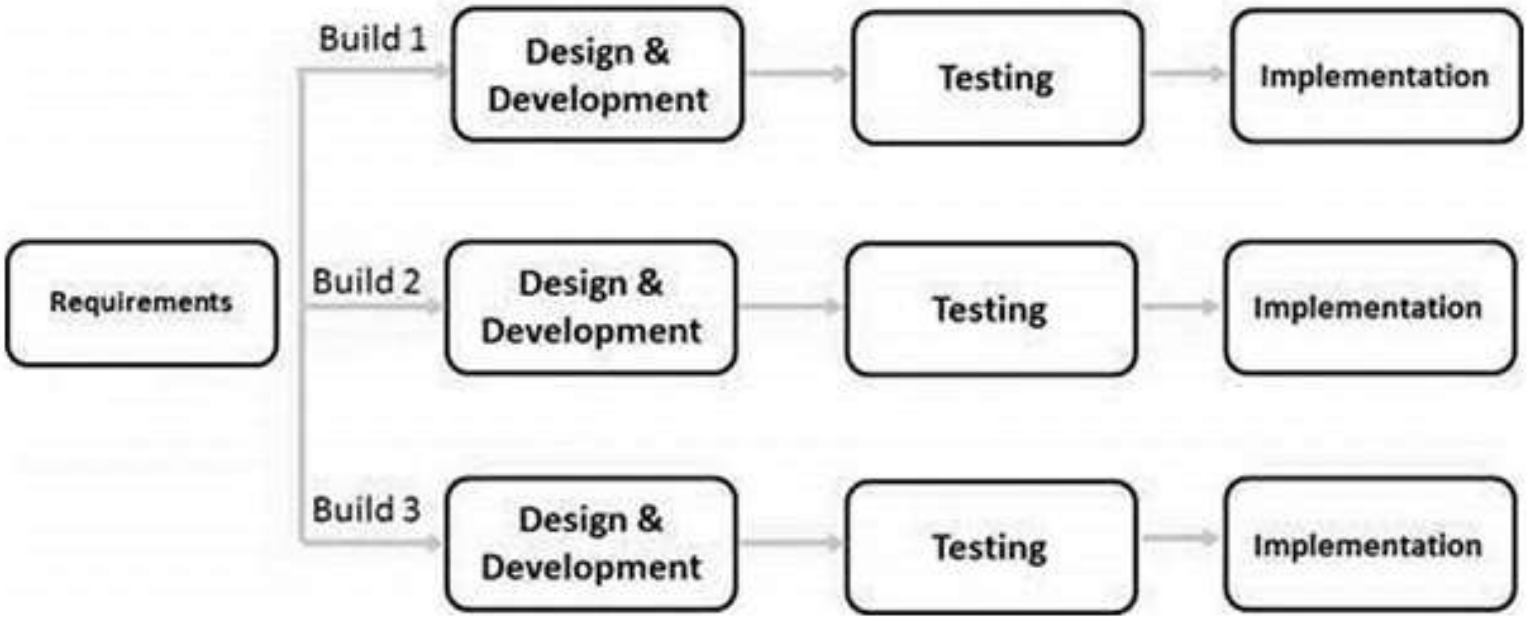
- The evolutionary model is based on the concept of making an initial product and then evolving the software product over time with iterative and incremental approaches with proper feedback.
- In this type of model, the product will go through several iterations and come up when the final product is built through multiple iterations.
- The development is carried out simultaneously with the feedback during the development.

- This model has a number of advantages such as
  - customer involvement,
  - taking feedback from the customer during development,
  - building the exact product that the user wants.
  - Because of the multiple iterations, the chances of errors get reduced and the reliability and efficiency will increase.

Types of Evolutionary Process Models

- Iterative Model
- Incremental Model
- Spiral Model

Iterative Model



- Iterative process starts with a simple implementation of a subset of the software requirements and iteratively enhances the evolving versions until the full system is implemented.
- At each iteration, design modifications are made and new functional capabilities are added.
- The basic idea behind this method is to develop a system through repeated cycles (iterative) and in smaller portions at a time (incremental).

- Requirements of the complete system are clearly defined and understood.
- Major requirements must be defined; however, some functionalities or requested enhancements may evolve with time.
- There is a time to the market constraint.
- A new technology is being used and is being learnt by the development team while working on the project.
- Resources with needed skill sets are not available and are planned to be used on contract basis for specific iterations.
- There are some high-risk features and goals which may change in the future.

Pros and Cons

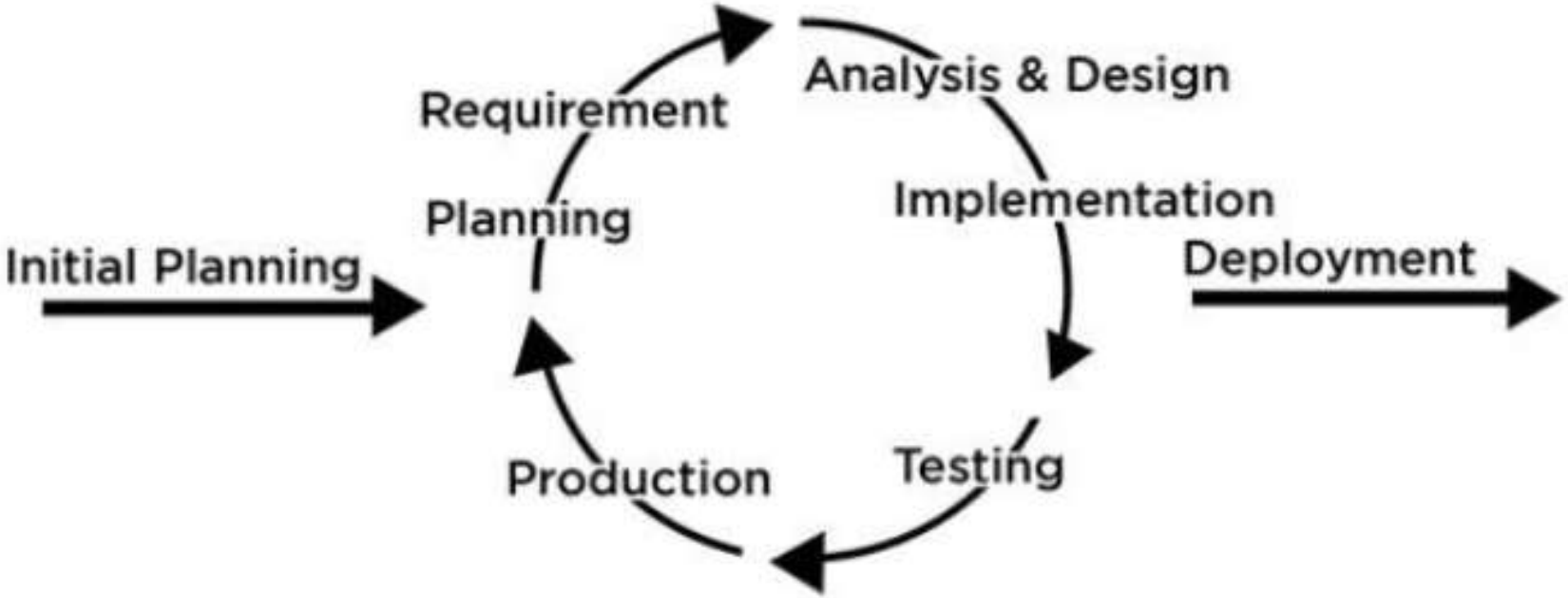
- Some working functionality can be developed quickly and early in the life cycle.
- Results are obtained early and periodically.
- Parallel development can be planned.
- Progress can be measured.
- Less costly to change the scope/requirements.
- Testing and debugging during smaller iteration is easy.
- Risks are identified and resolved during iteration; and each iteration is an easily managed milestone.
- Easier to manage risk - High risk part is done first.
- With every increment, operational product is delivered.
- Issues, challenges and risks identified from each increment can be utilized/applied to the next increment.
- Risk analysis is better.
- It supports changing requirements.
- Initial Operating time is less.
- Better suited for large and mission-critical projects.
- During the life cycle, software is produced early which facilitates customer evaluation and feedback.

Cons

- More resources may be required.
- Although cost of change is lesser, but it is not very suitable for changing requirements.
- More management attention is required.
- System architecture or design issues may arise because not all requirements are gathered in the beginning of the entire life cycle.
- Defining increments may require definition of the complete system.
- Not suitable for smaller projects.
- Management complexity is more.
- End of project may not be known which is a risk.
- Highly skilled resources are required for risk analysis.
- Projects progress is highly dependent upon the risk analysis phase.

Incremental Model

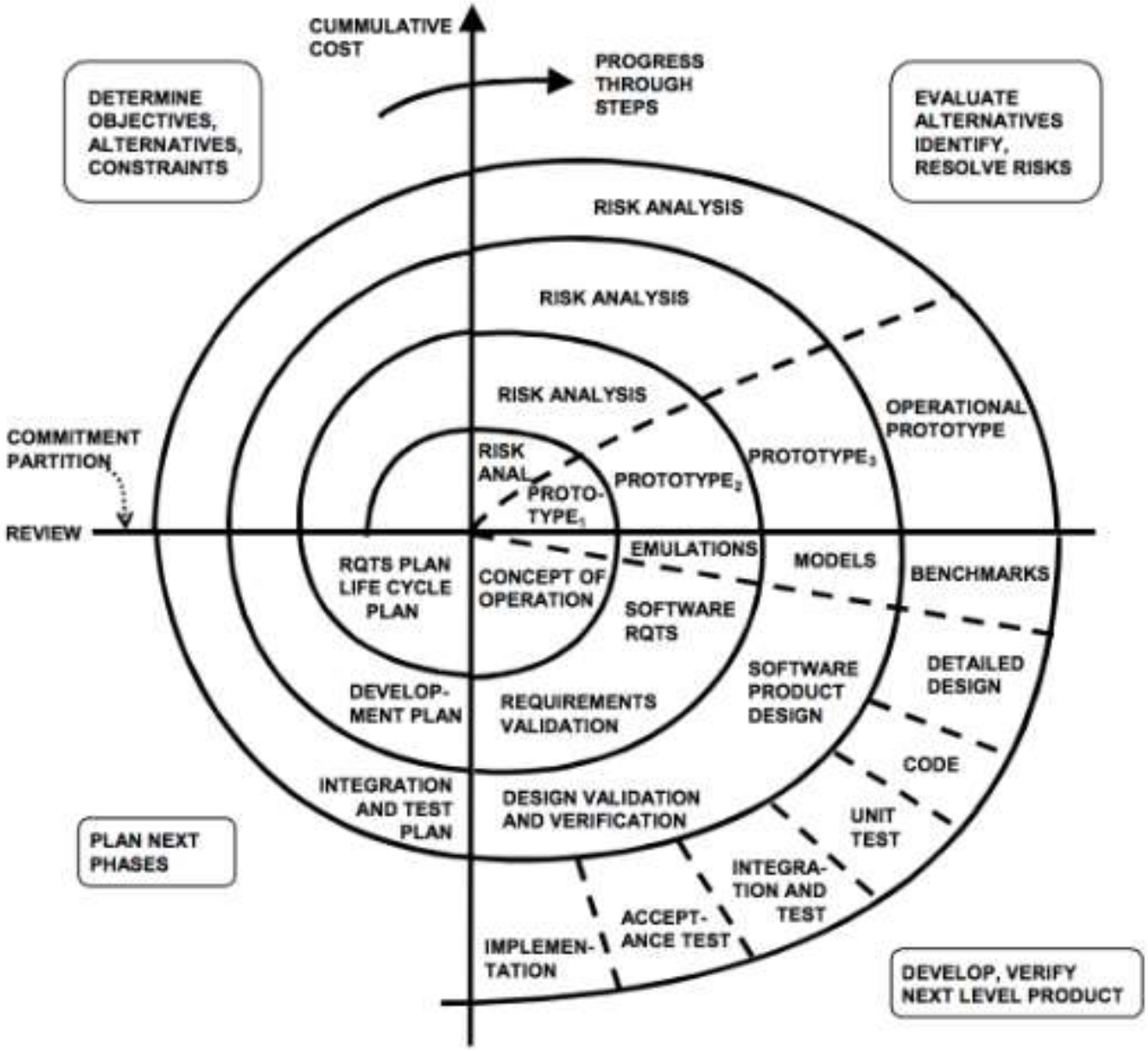
- **Incremental Model Overview**
- Product is developed in small parts (increments)
- Each increment adds functionality
- User feedback after each delivery


Initial Planning
Requirement
Planning
Analysis & Design
Implementation
Testing
Production
Deployment

- **Advantages:**
- Early partial product delivery
- Flexible to changes
- Easier testing and debugging
- **Disadvantages:**
- Needs good planning and architecture
- Integration can be complex
- Final cost may be higher

Spiral model

- The Spiral Model is an SDLC tool that mitigates risk and keeps your team focused on achieving key objectives.


CUMMULATIVE COST
PROGRESS THROUGH STEPS
DETERMINE OBJECTIVES, ALTERNATIVES, CONSTRAINTS
EVALUATE ALTERNATIVES IDENTIFY, RESOLVE RISKS
RISK ANALYSIS
RISK ANALYSIS
RISK ANALYSIS
COMMITMENT PARTITION
REVIEW
RISK ANAL
PROTO-TYPE1
PROTOTYPE2
PROTOTYPE3
OPERATIONAL PROTOTYPE
RQTS PLAN LIFE CYCLE PLAN
CONCEPT OF OPERATION
EMULATIONS
SOFTWARE RQTS
MODELS
BENCHMARKS
DEVELOP-MENT PLAN
REQUIREMENTS VALIDATION
SOFTWARE PRODUCT DESIGN
DETAILED DESIGN
CODE
INTEGRATION AND TEST PLAN
DESIGN VALIDATION AND VERIFICATION
UNIT TEST
INTEGRA-TION AND TEST
ACCEPT-ANCE TEST
IMPLEMEN-TATION
PLAN NEXT PHASES
DEVELOP, VERIFY NEXT LEVEL PRODUCT

- **First Spiral – Planning and Requirements:**
  The team gathers basic requirements (product listings, shopping cart, payment options) and identifies risks (security, scalability). They create a simple prototype to test user interaction and spot design issues.
- **Second Spiral – Risk Analysis and Design Refinement:**
  Based on feedback, they add features like secure payment processing, shopping cart, and user registration. They test security with dummy transactions and assess site performance under more users.
- **Third Spiral – Detailed Implementation:**
  Advanced features such as order tracking, reviews, and search are added. The team addresses scalability risks and tests site performance during high traffic periods.
- **Final Spiral – Full Deployment:**
  The website is fully developed, thoroughly tested, and launched. Remaining risks are monitored and addressed to ensure reliability.

Spiral Model Overview

- Combines iterative nature of prototyping with Waterfall
- Focus on risk analysis at every cycle
- Four phases per spiral: Planning → Risk Analysis → Development → Evaluation

When to use the spiral model

- The spiral model is often best suited for:
- Large, complex, and high-risk projects.
- Projects with undefined or evolving requirements.
- Projects where frequent releases or prototyping are helpful

Advantages of the Spiral Model

Below are some advantages of the Spiral Model.

- **Risk Handling:** The projects with many unknown risks that occur as the development proceeds, in that case, Spiral Model is the best development model to follow due to the risk analysis and risk handling at every phase.
- **Good for large projects:** It is recommended to use the Spiral Model in large and complex projects.
- **Flexibility in Requirements:** Change requests in the Requirements at a later phase can be incorporated accurately by using this model.
- **Customer Satisfaction:** Customers can see the development of the product at the early phase of the software development and thus, they habituated with the system by using it before completion of the total product.
- **Iterative and Incremental Approach:** The Spiral Model provides an iterative and incremental approach to software development, allowing for flexibility and adaptability in response to changing requirements or unexpected events.

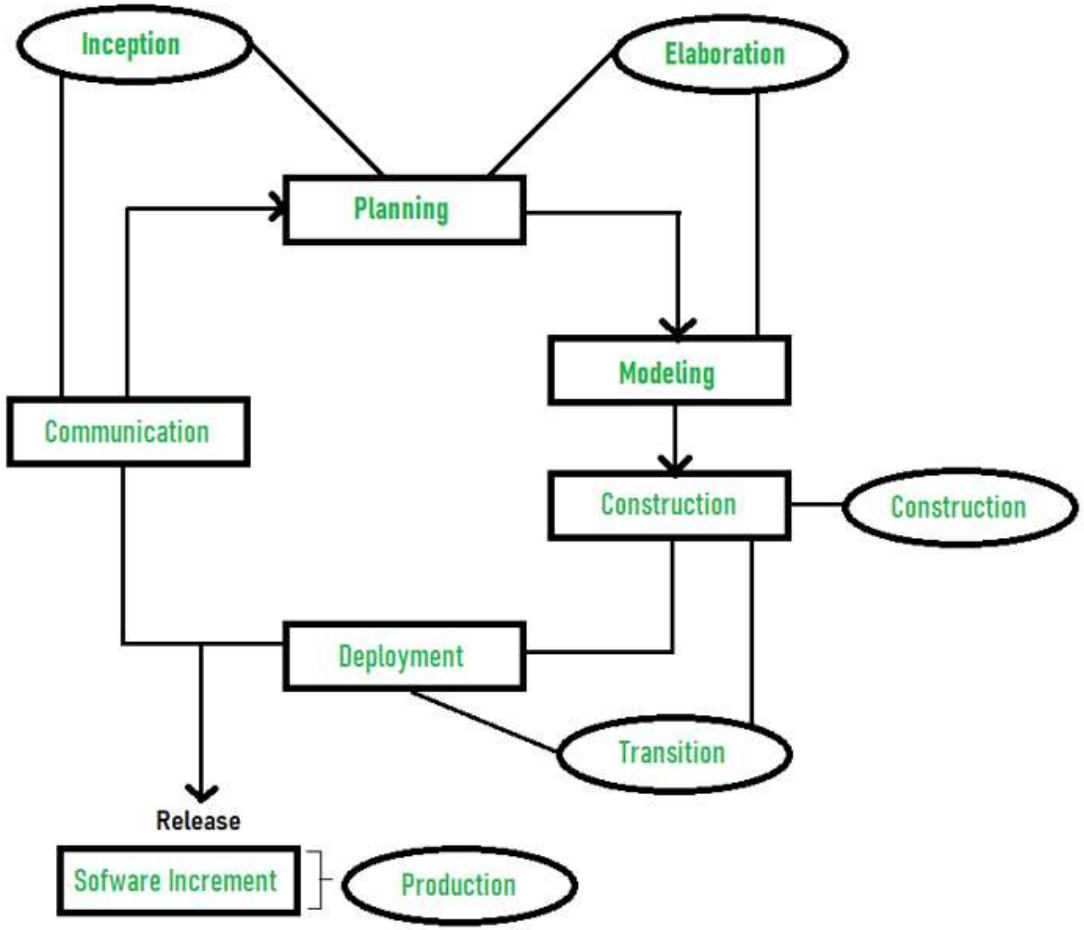
- **Emphasis on Risk Management:** The Spiral Model places a strong emphasis on risk management, which helps to minimize the impact of uncertainty and risk on the software development process.
- **Improved Communication:** The Spiral Model provides for regular evaluations and reviews, which can improve communication between the customer and the development team.
- **Improved Quality:** The Spiral Model allows for multiple iterations of the software development process, which can result in improved software quality and reliability.

Disadvantages of the Spiral Model

Below are some main disadvantages of the spiral model.

- **Complex:** The Spiral Model is much more complex than other SDLC models.
- **Expensive:** Spiral Model is not suitable for small projects as it is expensive.
- **Too much dependability on Risk Analysis:** The successful completion of the project is very much dependent on Risk Analysis. Without very highly experienced experts, it is going to be a failure to develop a project using this model.
- **Difficulty in time management:** As the number of phases is unknown at the start of the project, time estimation is very difficult.
- **Complexity:** The Spiral Model can be complex, as it involves multiple iterations of the software development process.
- **Time-Consuming:** The Spiral Model can be time-consuming, as it requires multiple evaluations and reviews.
- **Resource Intensive:** The Spiral Model can be resource-intensive, as it requires a significant investment in planning, risk analysis, and evaluations.

Rational Unified Process

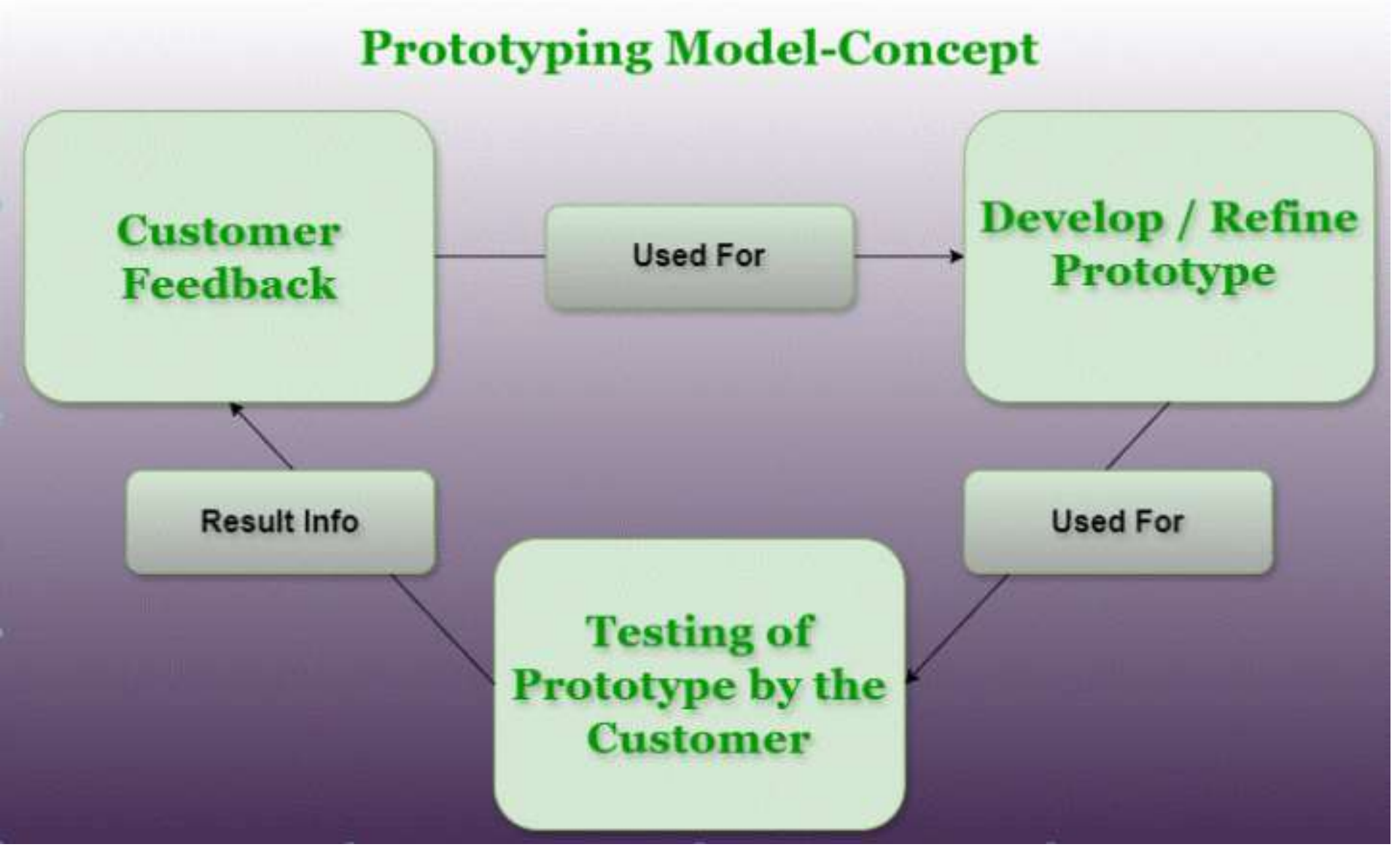


- RUP is an Iterative and incremental approach to improving problem knowledge through consecutive revisions.
- It is an architecture-centric and use-case-driven approach that manages risk and is flexible to change.
- RUP incrementally improves an effective solution through repeated iterations.

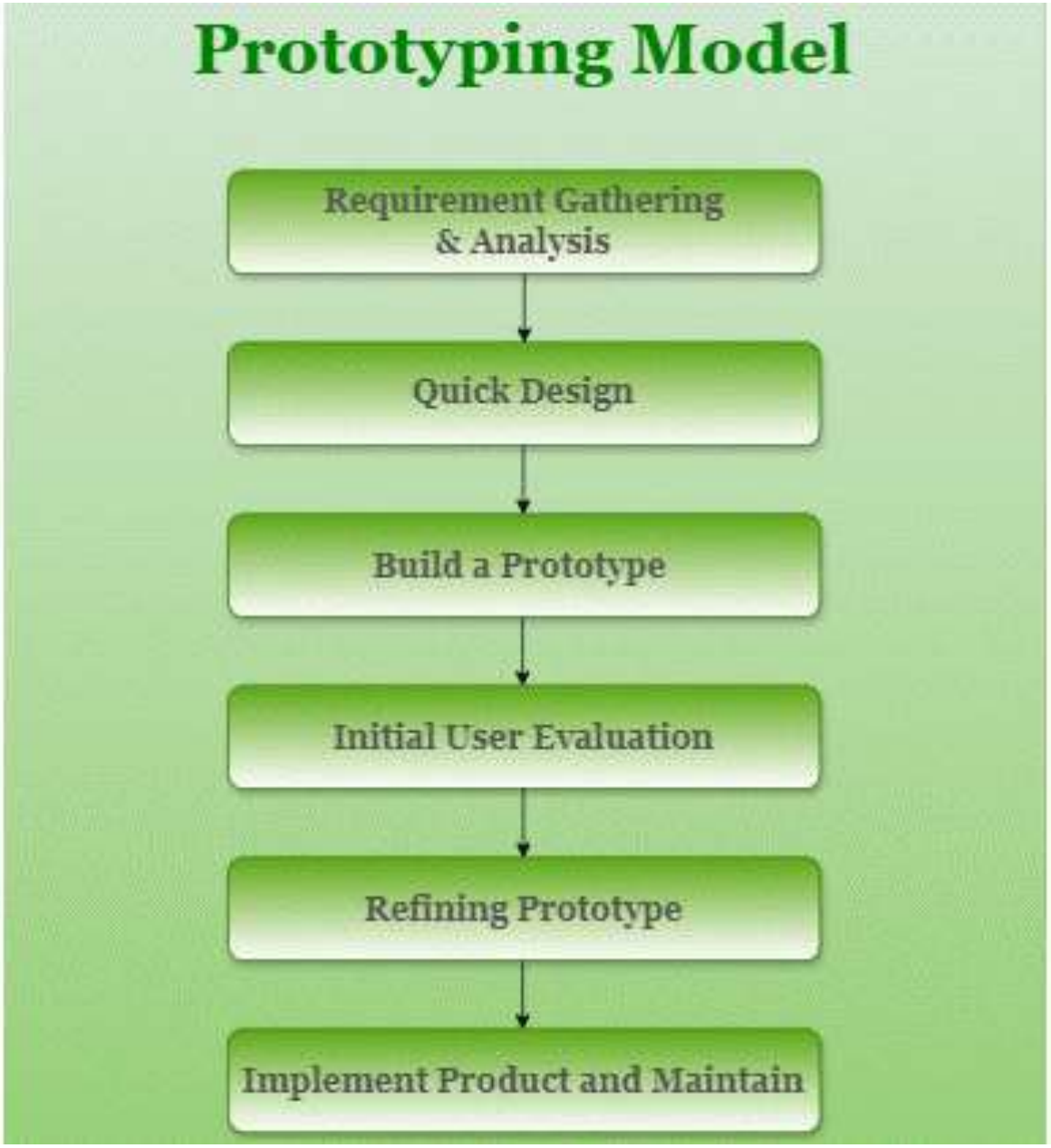
- **Rational Unified Process (RUP)** is a software development process for object-oriented models.
- It is also known as the Unified Process Model.
- It is created by Rational Corporation and is designed and documented using UML (Unified Modeling Language).

- Some characteristics of RUP include being
  - use-case driven,
  - Iterative (repetition of the process),
  - incremental (increase in value) by nature,
  - delivered online using web technology,
  - can be customized or tailored in modular and electronic form, etc.
  - RUP reduces unexpected development costs and prevents the wastage of resources.

Prototyping Model-Concept

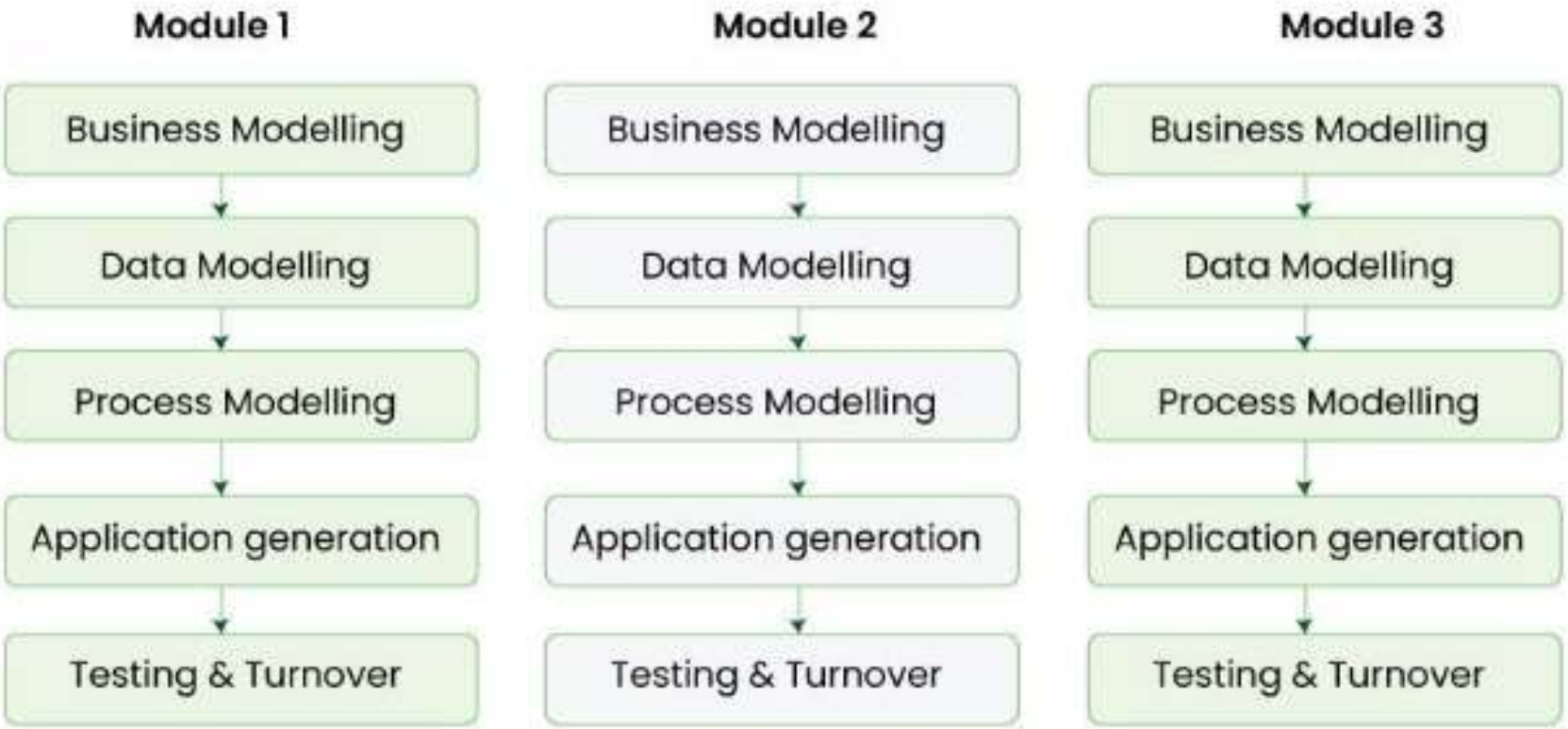


Prototyping Model



RAD Model

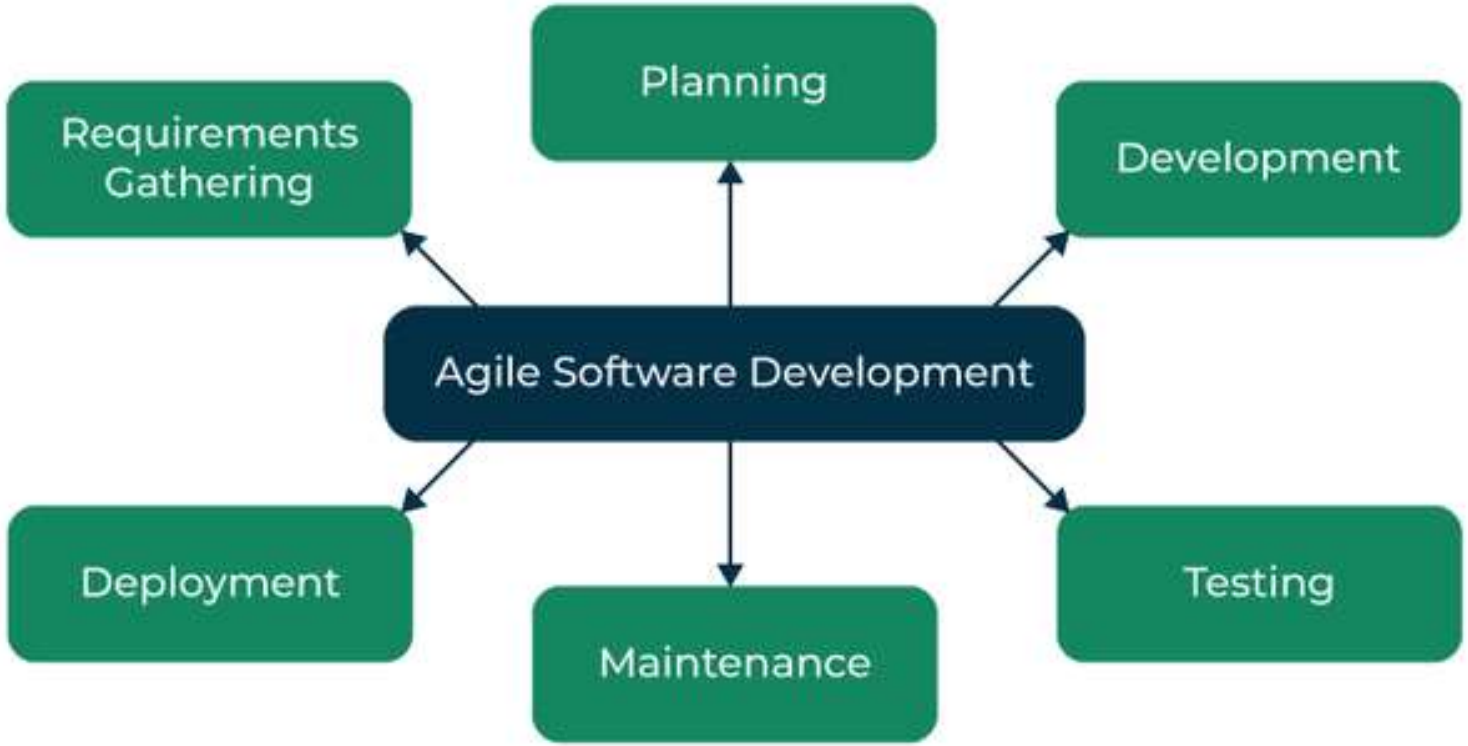
- **RAD Model** stands for rapid application development model.
- The methodology of RAD model is similar to that of incremental or waterfall model.
- It is used for small projects.
- The main objective of RAD model is to reuse code, components, tools, processes in project development.

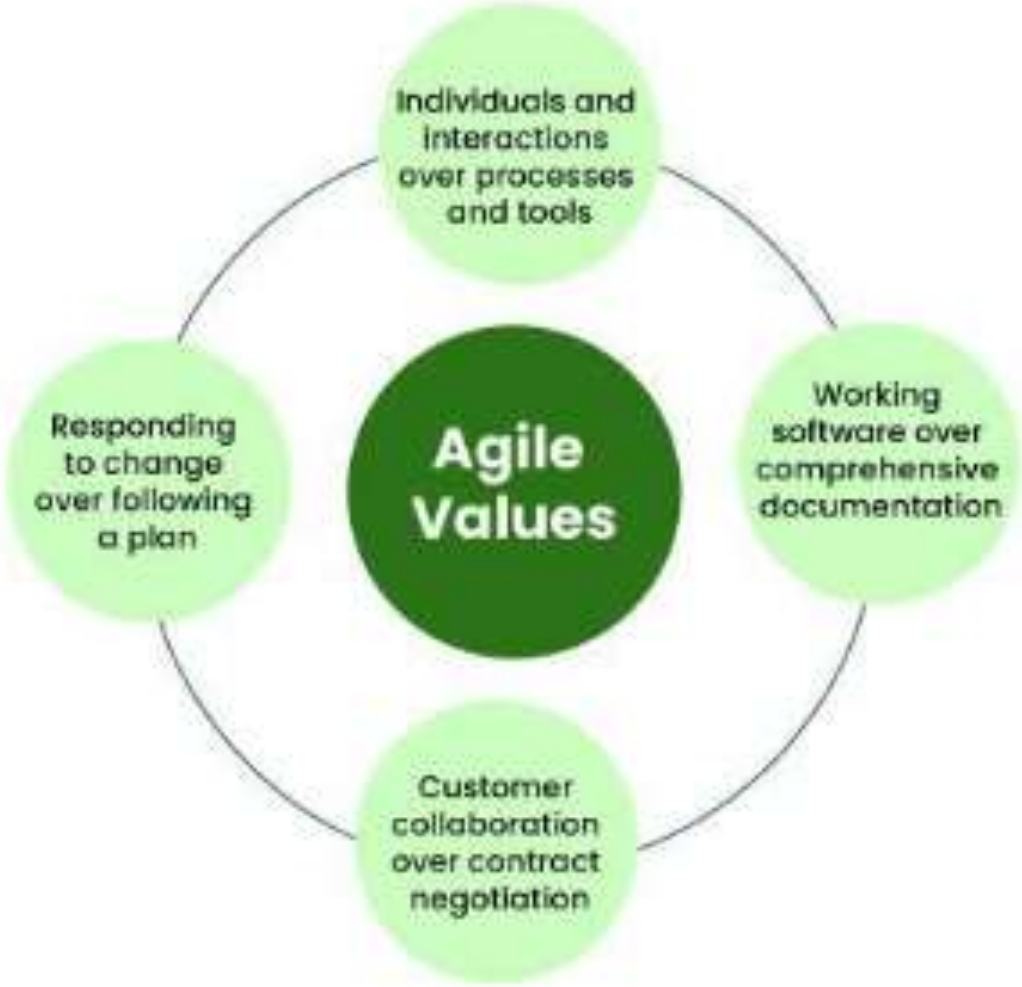

Module 1
Business Modelling
Data Modelling
Process Modelling
Application generation
Testing & Turnover
Module 2
Business Modelling
Data Modelling
Process Modelling
Application generation
Testing & Turnover
Module 3
Business Modelling
Data Modelling
Process Modelling
Application generation
Testing & Turnover

- If the project is large then it is divided into many small projects and these small projects are planned one by one and completed. In this way, by completing small projects, the large project gets ready quickly.
- In RAD model, the project is completed within the given time and all the requirements are collected before starting the project. It is very fast and there are very less errors in it.

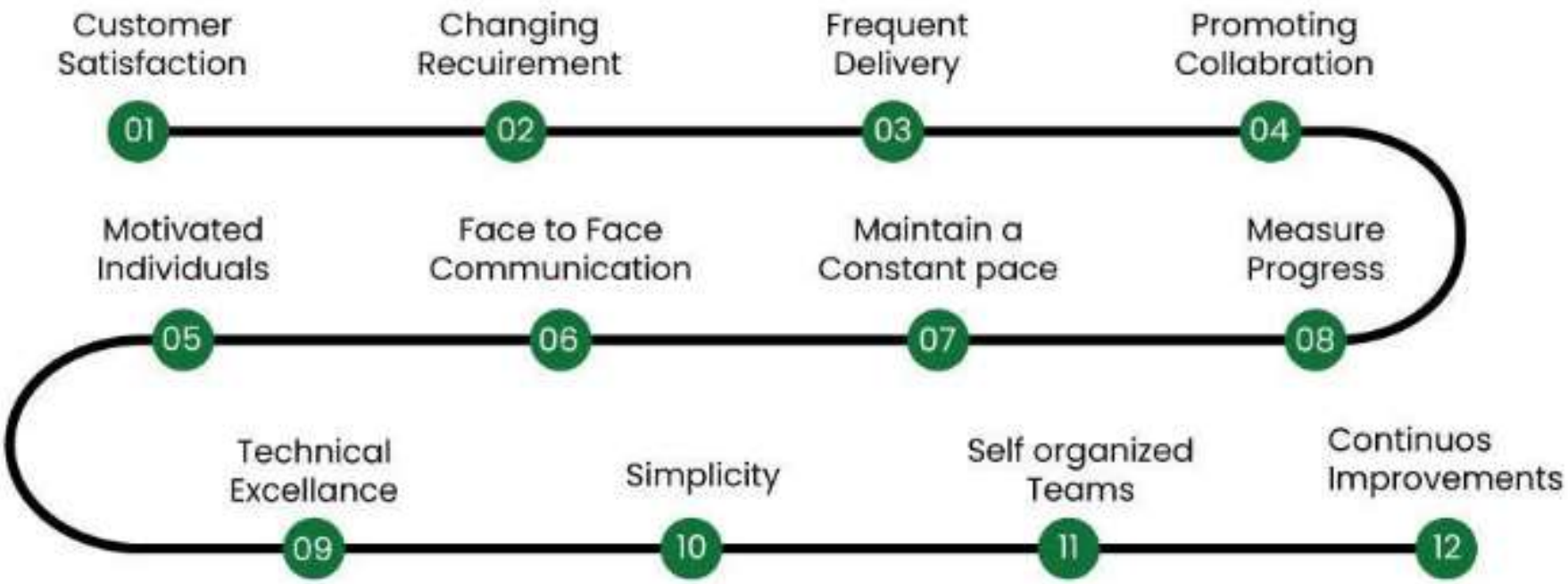
Agile Software Development

- Agile Software Development is a **Software Development Methodology** that values flexibility, collaboration, and customer satisfaction.
- It is based on the Agile Manifesto, a set of principles for software development that prioritize individuals and interactions, working software, customer collaboration, and responding to change.
- Agile Software Development is an iterative and incremental approach that emphasizes the importance of delivering a working product quickly and frequently.
- It involves close collaboration between the development team and the customer to ensure that the product meets their needs and expectations.


Planning
Requirements Gathering
Development
Agile Software Development
Deployment
Maintenance
Testing


Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan
Agile Values

Principles of Agile Software Development



Advantages Agile Software Development

- **Increased collaboration and communication:** Agile Software Development Methodology emphasize collaboration and communication among team members, stakeholders, and customers. This leads to improved understanding, better alignment, and increased buy-in from everyone involved.
- **Flexibility and adaptability:** Agile methodologies are designed to be flexible and adaptable, making it easier to respond to changes in requirements, priorities, or market conditions. This allows teams to quickly adjust their approach and stay focused on delivering value.
- **Improved quality and reliability:** Agile methodologies place a strong emphasis on testing, quality assurance, and continuous improvement. This helps to ensure that software is delivered with high quality and reliability, reducing the risk of defects or issues that can impact the user experience.

- **Enhanced customer satisfaction:** Agile methodologies prioritize customer satisfaction and focus on delivering value to the customer. By involving customers throughout the development process, teams can ensure that the software meets their needs and expectations.
- **Increased team morale and motivation:** Agile methodologies promote a collaborative, supportive, and positive work environment. This can lead to increased team morale, motivation, and engagement, which can in turn lead to better productivity, higher quality work, and improved outcomes.
- Deployment of software is quicker and thus helps in increasing the trust of the customer.
- Can better adapt to rapidly changing requirements and respond faster.
- Helps in getting immediate feedback which can be used to improve the software in the next increment.
- People - Not Process. People and interactions are given a higher priority than processes and tools.
- Continuous attention to technical excellence and good design.

Disadvantages Agile Software Development

- **Lack of predictability:** Agile Development relies heavily on customer feedback and continuous iteration, which can make it difficult to predict project outcomes, timelines, and budgets.
- **Limited scope control:** Agile Development is designed to be flexible and adaptable, which means that scope changes can be easily accommodated. However, this can also lead to scope creep and a lack of control over the project scope.
- **Lack of emphasis on testing:** Agile Development places a greater emphasis on delivering working code quickly, which can lead to a lack of focus on testing and quality assurance. This can result in bugs and other issues that may go undetected until later stages of the project.
- **Risk of team burnout:** Agile Development can be intense and fast-paced, with frequent sprints and deadlines. This can put a lot of pressure on team members and lead to burnout, especially if the team is not given adequate time for rest and recovery.

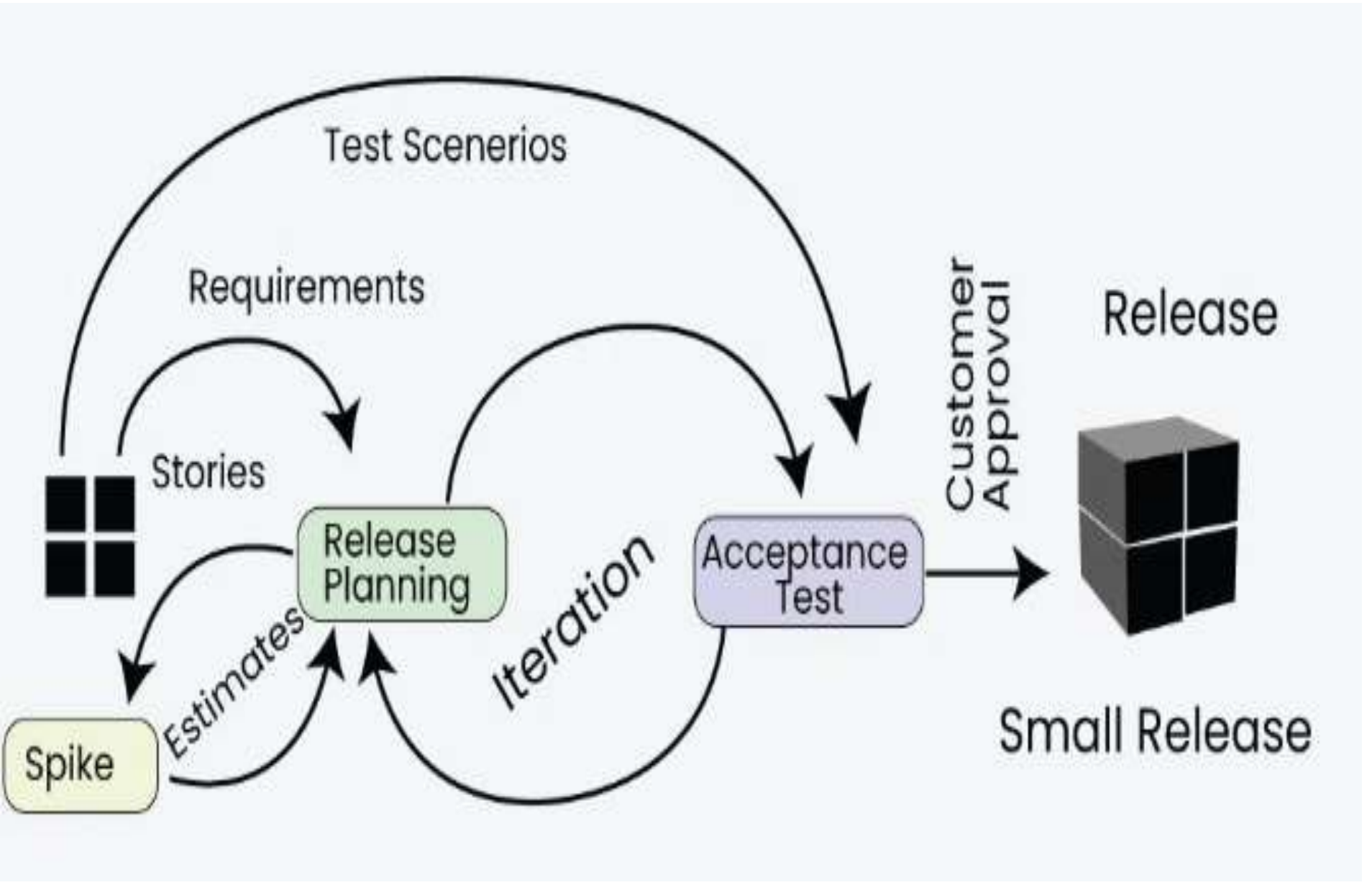
- **Lack of structure and governance:** Agile Development is often less formal and structured than other development methodologies, which can lead to a lack of governance and oversight. This can result in inconsistent processes and practices, which can impact project quality and outcomes.
- In the case of large software projects, it is difficult to assess the effort required at the initial stages of the software development life cycle.
- Agile Development is more code-focused and produces less documentation.
- Agile development is heavily dependent on the inputs of the customer. If the customer has ambiguity in his vision of the outcome, it is highly likely that the project to get off track.
- Face-to-face communication is harder in large-scale organizations.
- Only senior programmers are capable of making the kind of decisions required during the development process. Hence, it's a difficult situation for new programmers to adapt to the environment.

Practices of Agile Software Development

- **Scrum:** Scrum is a framework for agile software development that involves iterative cycles called sprints, daily stand-up meetings, and a product backlog that is prioritized by the customer.
- **Kanban:** Kanban is a visual system that helps teams manage their work and improve their processes. It involves using a board with columns to represent different stages of the development process, and cards or sticky notes to represent work items.
- **Continuous Integration:** Continuous Integration is the practice of frequently merging code changes into a shared repository, which helps to identify and resolve conflicts early in the development process.
- **Test-Driven Development:** Test-Driven Development (TDD) is a development practice that involves writing automated tests before writing the code. This helps to ensure that the code meets the requirements and reduces the likelihood of defects.
- P**air Programming:** Pair programming involves two developers working together on the same code. This helps to improve code quality, share knowledge, and reduce the likelihood of defects.

Extreme Programming (XP)

- Extreme Programming (XP) is an Agile software development methodology
- It focuses on delivering high-quality software
- Use of frequent and continuous feedback, collaboration, and adaptation.
- XP emphasizes a close working relationship between the development team, the customer, and stakeholders
- It emphasize on rapid, iterative development and deployment.


Test Scenerios
Requirements
Stories
Release Planning
Estimates
Spike
Iteration
Acceptance Test
Customer Approval
Release
Small Release


Good Practices in Extreme Programming
Code Review
Testing
Incremental development
Simplicity
Design
Integration testing

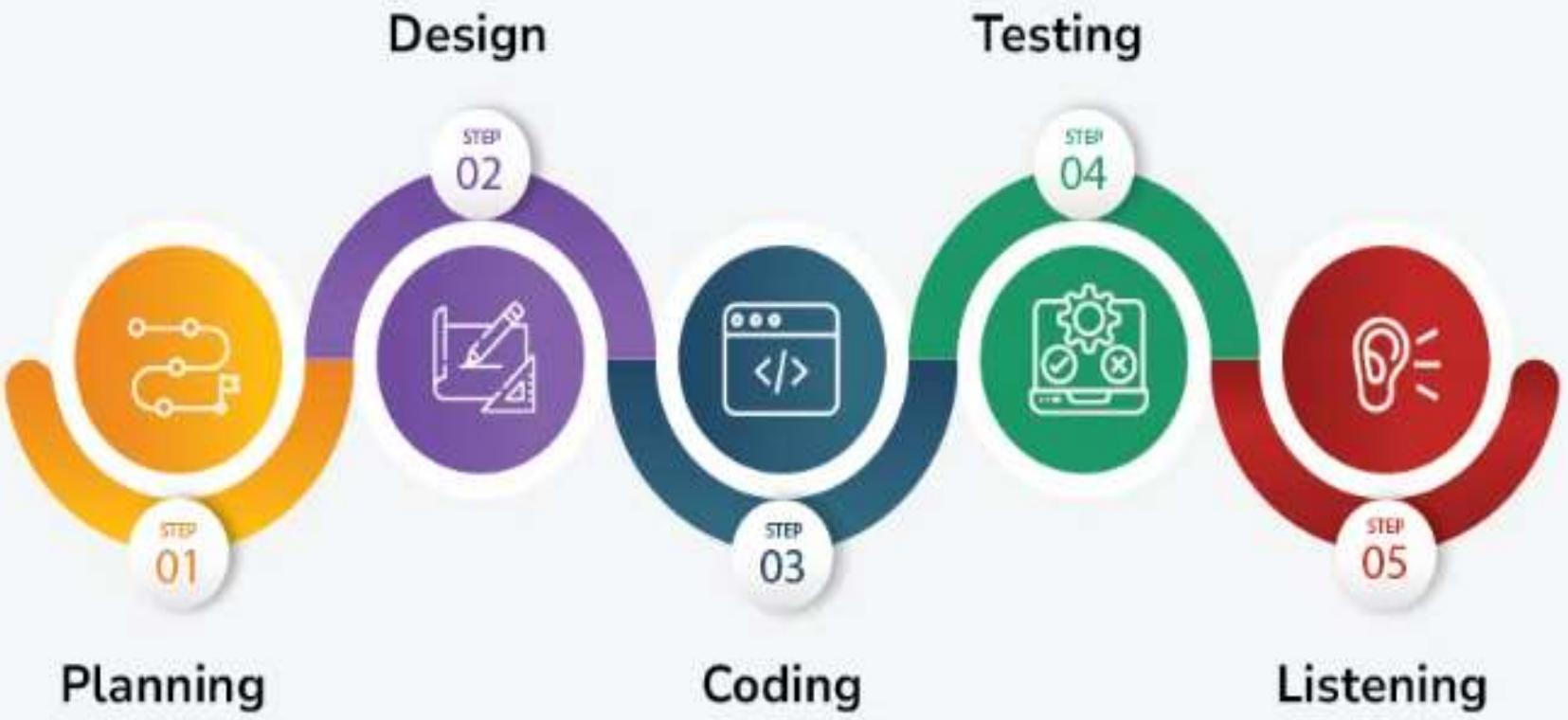
Good Practices in Extreme Programming

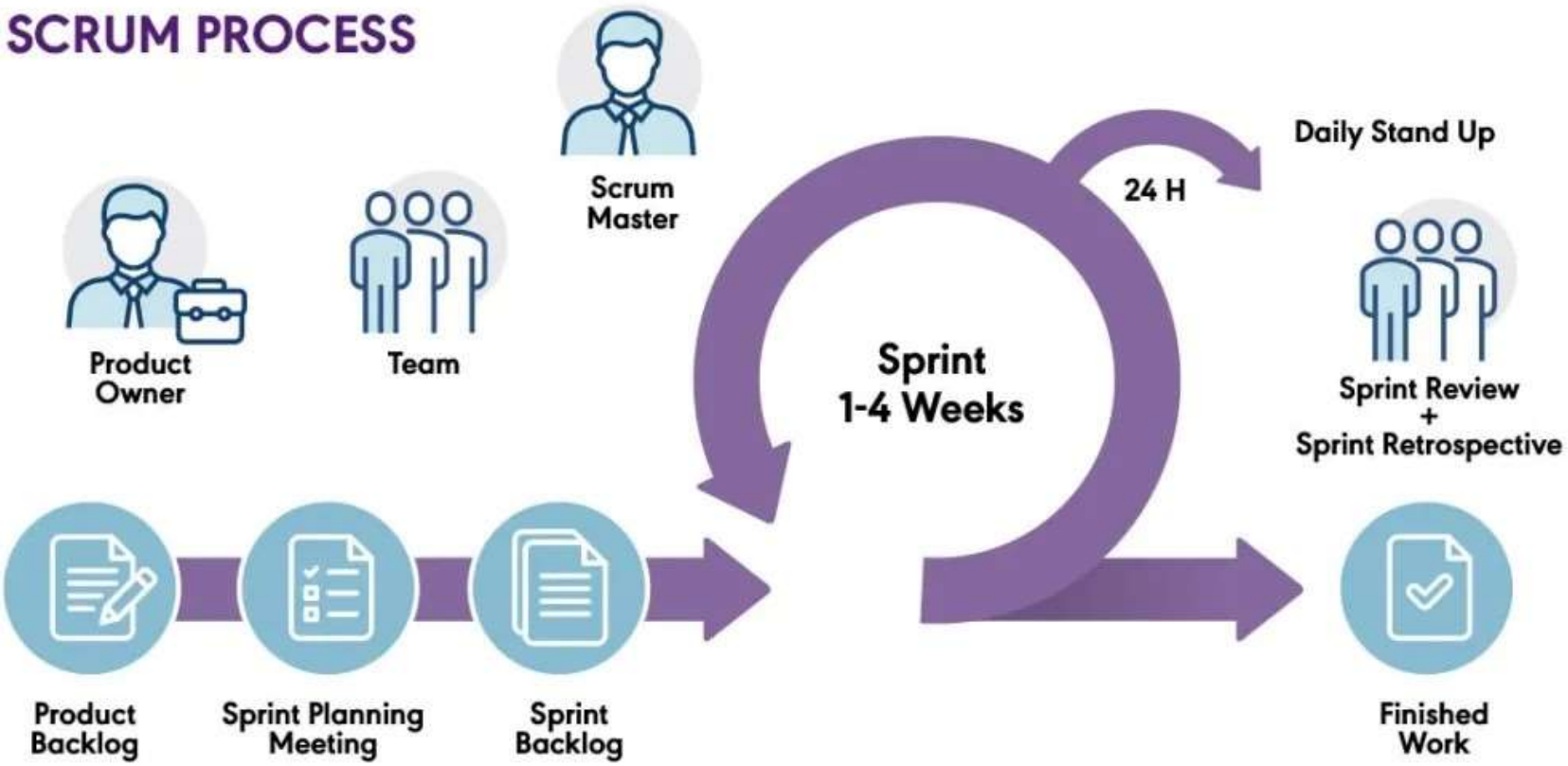
- **Code Review:** Code review detects and corrects errors efficiently. It suggests pair programming as coding and reviewing of written code carried out by a pair of programmers who switch their work between them every hour.
- **Testing:** Testing code helps to remove errors and improves its reliability. XP suggests test-driven development (TDD) to continually write and execute test cases. In the TDD approach, test cases are written even before any code is written.
- **Incremental development:** Incremental development is very good because customer feedback is gained and based on this development team comes up with new increments every few days after each iteration.
- **Simplicity:** Simplicity makes it easier to develop good-quality code as well as to test and debug it.
- **Design:** Good quality design is important to develop good quality software. So, everybody should design daily.
- **Integration testing:** Integration Testing helps to identify bugs at the interfaces of different functionalities. Extreme programming suggests that the developers should achieve continuous integration by building and performing integration testing several times a day.

Basic Principles of Extreme programming

- XP is based on the frequent iteration through which the developers implement User Stories.
- User stories are simple and informal statements of the customer about the functionalities needed.
- A User Story is a conventional description by the user of a feature of the required system.
- It does not mention finer details such as the different scenarios that can occur.
- Based on User stories, the project team proposes Metaphors.
- Metaphors are a common vision of how the system would work.
- The development team may decide to build a Spike for some features.
- A Spike is a very simple program that is constructed to explore the suitability of a solution being proposed. It can be considered similar to a prototype.

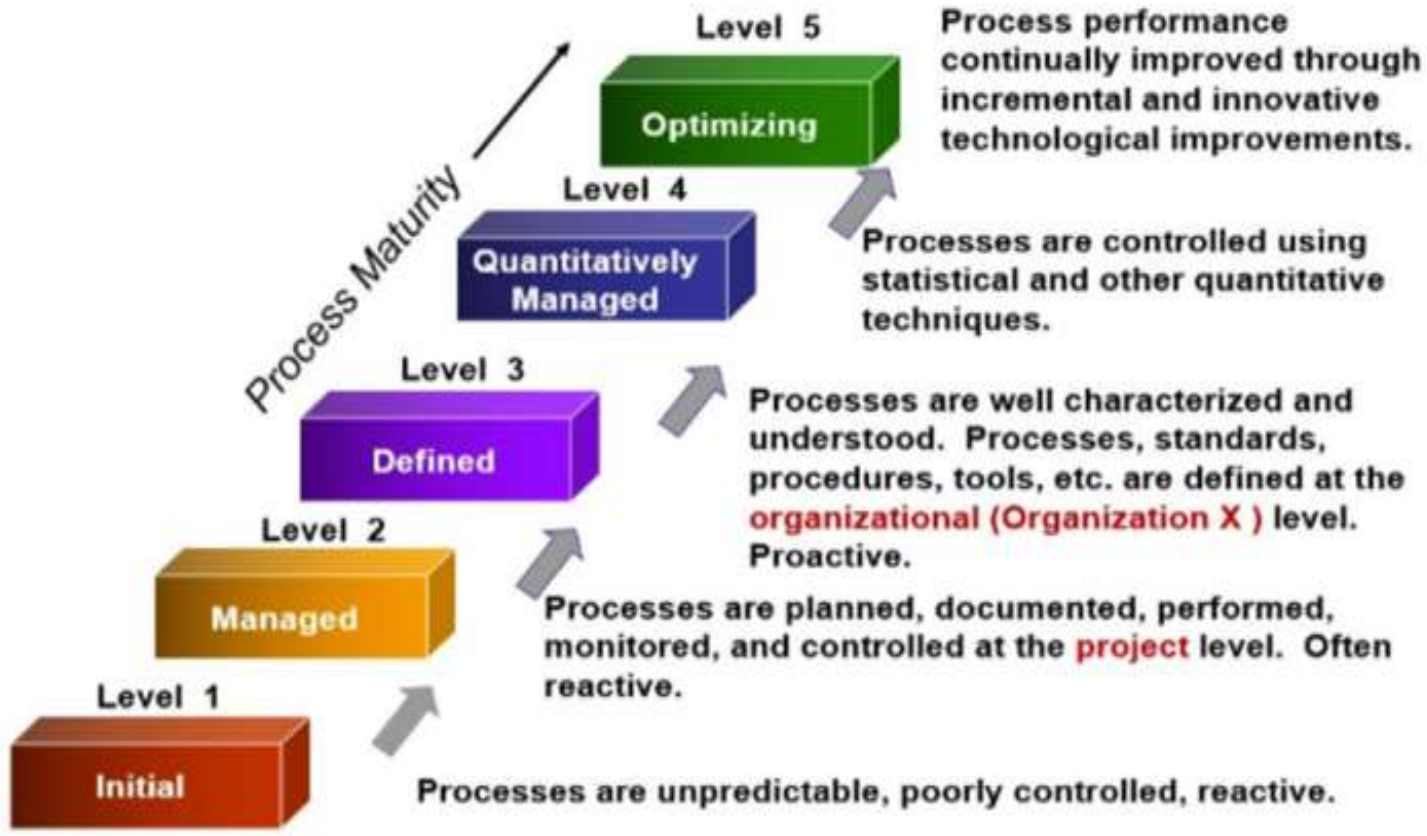
Life Cycle of Extreme Programming (XP)




SCRUM PROCESS
Scrum Master
Product Owner
Team
Daily Stand Up
24 H
Sprint 1-4 Weeks
Sprint Review + Sprint Retrospective
Product Backlog
Sprint Planning Meeting
Sprint Backlog
Finished Work

Capability Maturity Model (CMM)

- The Capability Maturity Model (CMM) is a tool used to improve and refine software development processes.
- It provides a structured way for organizations to assess their current practices and identify areas for improvement.
- CMM consists of five maturity levels: initial, repeatable, defined, managed, and optimizing.
- Organizations can systematically improve their software development processes, leading to higher-quality products and more efficient project management.


Level 5
Optimizing
Process performance continually improved through incremental and innovative technological improvements.
Process Maturity
Level 4
Quantitatively Managed
Processes are controlled using statistical and other quantitative techniques.
Level 3
Defined
Processes are well characterized and understood. Processes, standards, procedures, tools, etc. are defined at the organizational (Organization X) level. Proactive.
Level 2
Managed
Processes are planned, documented, performed, monitored, and controlled at the project level. Often reactive.
Level 1
Initial
Processes are unpredictable, poorly controlled, reactive.

Unit II

Software Requirements:

- The process to gather the software requirements from client, analyze and document them is known as requirement engineering.
- The goal of requirement engineering is to develop and maintain sophisticated and descriptive ‘System Requirements Specification’ document.

Types of Requirements:

- **User Requirements**: It is a collection of statement in natural language and description of the services the system provides and its operational limitation. It is written for customer.
- **System Requirement**: It is a structured document that gives the detailed description of the system services. It is written as a contract between client and contractor.

User Requirements

- Describe **what the user needs** from the system.
- Written in **natural language**, accessible to non-technical stakeholders.
- Should cover:
- Functional needs
- Non-functional expectations (performance, usability)
- Example:
- *“The user shall be able to search for products by name, category, or price.”*

System Requirements

- Detailed description of **system functionalities** and constraints.
- Types:
  - **Functional Requirements**: Describe *what* the system should do.
  - **Non-functional Requirements**: Define *how* the system performs tasks.
- Example:
  - *The system shall handle 1000 transactions per second.*

Software Requirement Specification: -

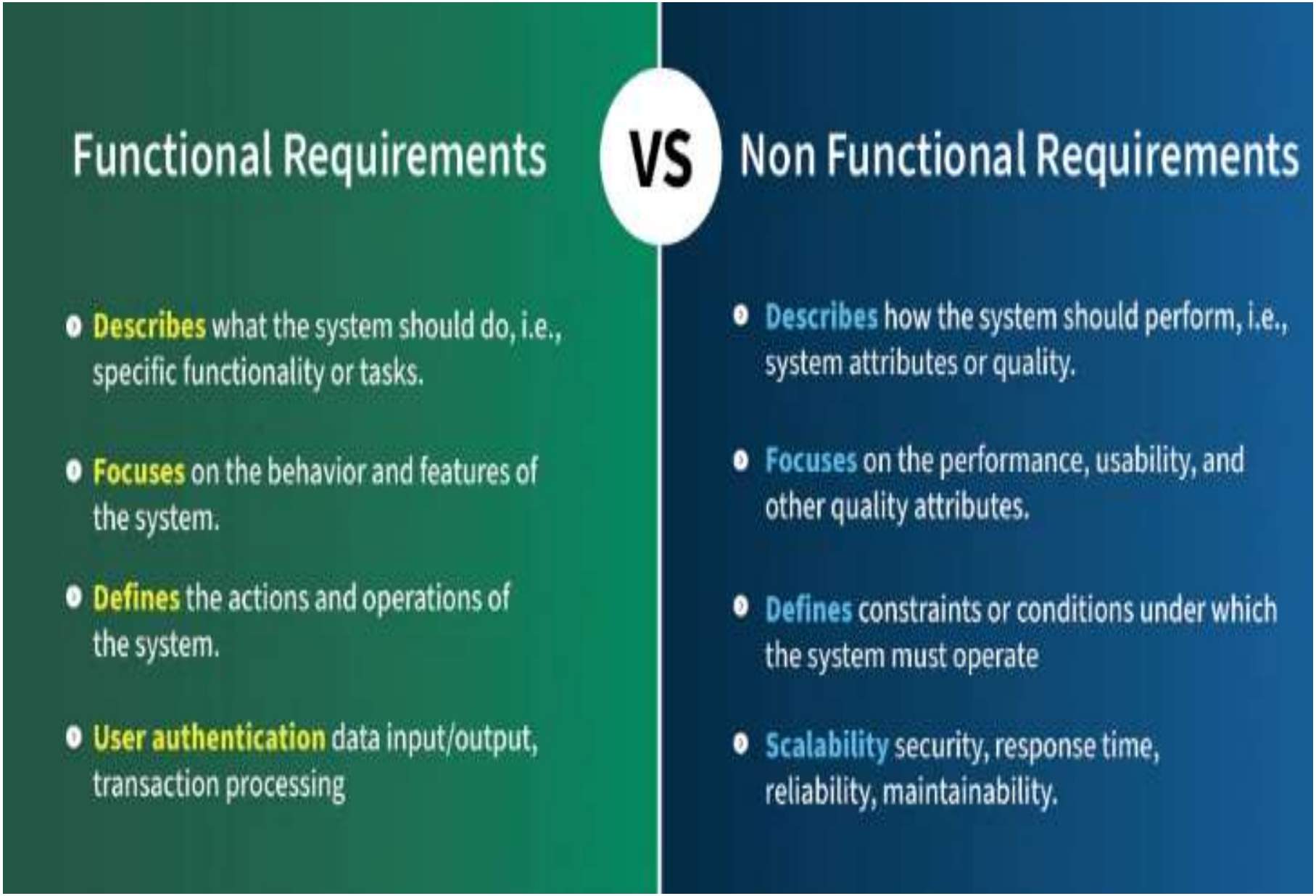
- SRS is a document created by system analyst after the requirements are collected from various stakeholders.
- SRS defines how the intended software will interact with
  - hardware,
  - external interfaces,
  - speed of operation,
  - response time of system,
  - portability of software across various platforms,
  - maintainability,
  - speed of recovery after crashing,
  - Security,
  - Quality,
  - Limitations etc.

SRS:

- The requirements received from client are written in natural language.
- It is the responsibility of system analyst to document the requirements in technical language so that they can be comprehended and useful by the software development team.
- SRS should come up with following features:
  - User Requirements are expressed in natural language.
  - Technical requirements are expressed in structured language, which is used inside the organization.
  - Design description should be written in Pseudo code.
  - Format of Forms and GUI screen prints.
  - Conditional and mathematical notations for DFDs etc.

Functional vs. Non Functional Requirements

- functional requirements define the specific behavior or functions of a system
- non-functional requirements specify how the system performs its tasks, focusing on attributes like performance, security, scalability, and usability.


Functional Requirements
VS
Non Functional Requirements
Describes what the system should do, i.e., specific functionality or tasks.
Focuses on the behavior and features of the system.
Defines the actions and operations of the system.
User authentication data input/output, transaction processing
Describes how the system should perform, i.e., system attributes or quality.
Focuses on the performance, usability, and other quality attributes.
Defines constraints or conditions under which the system must operate
Scalability security, response time, reliability, maintainability.

Functional Requirements?

- These are the requirements that the end user specifically demands as basic facilities that the system should offer.
- All these functionalities need to be necessarily incorporated into the system as a part of the contract.
- These are represented or stated in the form of input to be given to the system, the operation performed and the output expected.
- They are the requirements stated by the user which one can see directly in the final product, unlike the non-functional requirements.

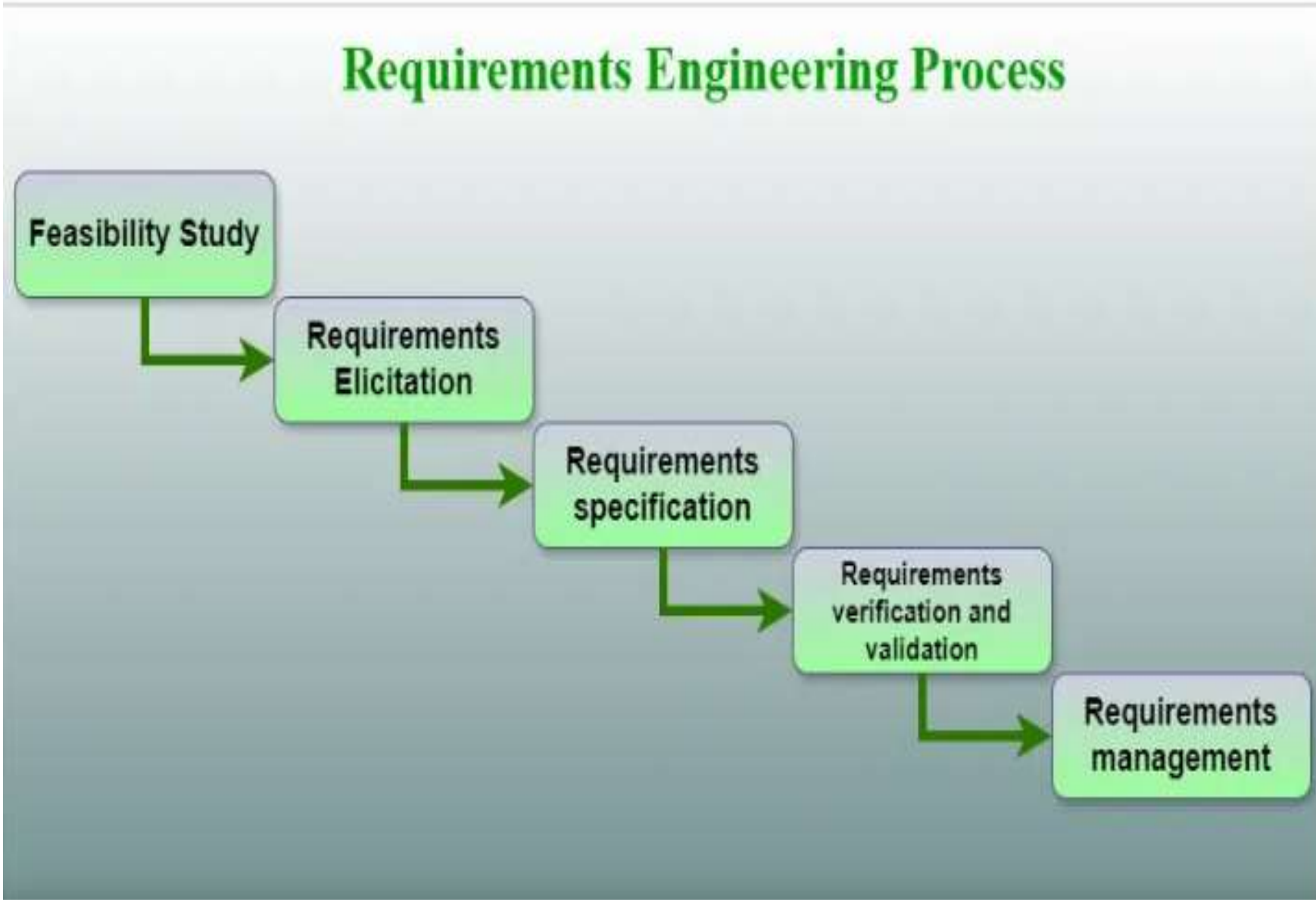
What are Non-Functional Requirements?

- These are the quality constraints that the system must satisfy according to the project contract.
- The priority or extent to which these factors are implemented varies from one project to another.
- They are also called non-behavioral requirements.
- They deal with issues like:
  - Portability
  - Security
  - Maintainability
  - Reliability
  - Scalability
  - Performance
  - Reusability
  - Flexibility

Interface Specification

- Interface Specification defines **how different components of the system interact** with each other.
- It provides a **formal description** of the inputs, outputs, and communication rules between:
- Software ↔ Users
- Software ↔ Hardware
- Software ↔ Software

Requirements Engineering Process



Requirement Engineering

- A systematic and strict approach to the definition, creation, and verification of requirements for a software system is known as requirements engineering.
- To guarantee the effective creation of a software product, the requirements engineering process entails several tasks that help in understanding, recording, and managing the demands of stakeholders.

1. Feasibility Study

- **Feasibility Study** in Software Engineering is a study to evaluate feasibility of proposed project or system.
- Feasibility study is one of stage among important four stages of Software Project Management Process.
- it is a measure of the software product in terms of how much beneficial product development will be for the organization in a practical point of view.
- Feasibility study is carried out based on many purposes to analyze whether software product will be right in terms of development, implementation, contribution of project to the organization etc.

Types of Feasibility Study

- **Technical Feasibility:** current resources both hardware software along with required technology are analyzed/assessed to develop project.
- **Operational Feasibility:** degree of providing service to requirements is analyzed along with how much easy product will be to operate and maintenance after deployment, determining usability of product, Determining suggested solution by software development team is acceptable or not etc.
- **Economic Feasibility:** cost and benefit of the project is analyzed. After that it is analyzed whether project will be beneficial in terms of finance for organization or not.
- **Legal Feasibility:** project is analyzed in legality point of view, analyzing barriers of legal implementation of project, data protection acts or social media laws

- **Schedule Feasibility:** timelines/deadlines is analyzed for proposed project which includes how much time teams will take to complete final project
- **Cultural and Political Feasibility:** assesses how the software project will affect the political environment and organizational culture. It is essential that cultural and political factors be taken into account in order to execute projects successfully.
- **Market Feasibility:** evaluating the market's willingness and ability to accept the suggested software system. Analyzing the target market, understanding consumer wants and assessing possible rivals are all part of this study.
- **Resource Feasibility:** if the resources needed to complete the software project successfully are adequate and readily available. Financial, technological and human resources are all taken into account in this study.

2. Requirements Elicitation

- It is related to the various ways used to gain knowledge about the project domain and requirements.
- The various sources of domain knowledge include customers, business manuals, the existing software of the same type, standards, and other stakeholders of the project.
- The techniques used for requirements elicitation include interviews, brainstorming, task analysis, Delphi technique, prototyping, etc.
- Elicitation does not produce formal models of the requirements understood. Instead, it widens the domain knowledge of the analyst and thus helps in providing input to the next stage.

Requirements elicitation

- Requirements elicitation is the process of gathering information about the needs and expectations of stakeholders for a software system.
- This is the first step in the requirements engineering process and it is critical to the success of the software development project.
- The goal of this step is to understand the problem that the software system is intended to solve and the needs and expectations of the stakeholders who will use the system.

Several techniques can be used to elicit requirements, including:

- **Interviews**: These are one-on-one conversations with stakeholders to gather information about their needs and expectations.
- **Surveys**: These are questionnaires that are distributed to stakeholders to gather information about their needs and expectations.
- **Focus Groups**: These are small groups of stakeholders who are brought together to discuss their needs and expectations for the software system.
- **Observation**: This technique involves observing the stakeholders in their work environment to gather information about their needs and expectations.
- **Prototyping**: This technique involves creating a working model of the software system, which can be used to gather feedback from stakeholders and to validate requirements.

3. Requirements Specification

- This activity is used to produce formal software requirement models.
- All the requirements including the functional as well as the non-functional requirements and the constraints are specified by these models in totality.
- During specification, more knowledge about the problem may be required which can again trigger the elicitation process.
- The models used at this stage include ER diagrams, data flow diagrams(DFDs), function decomposition diagrams(FDDs), data dictionaries, etc.

Requirements specification:

- Requirements specification is the process of documenting the requirements identified in the analysis step in a clear, consistent, and unambiguous manner.
- This step also involves prioritizing and grouping the requirements into manageable chunks.
- The goal of this step is to create a clear and comprehensive document that describes the requirements for the software system. This document should be understandable by both the development team and the stakeholders.

types of requirements are commonly specified

- **Functional Requirements:** These describe what the software system should do. They specify the functionality that the system must provide, such as input validation, data storage, and user interface.
- **Non-Functional Requirements**: These describe how well the software system should do it. They specify the quality attributes of the system, such as performance, reliability, usability, and security.
- **Constraints:** These describe any limitations or restrictions that must be considered when developing the software system.
- **Acceptance Criteria**: These describe the conditions that must be met for the software system to be considered complete and ready for release.

4. Requirements Verification and Validation

- **Verification:** It refers to the set of tasks that ensures that the software correctly implements a specific function.
- **Validation:** It refers to a different set of tasks that ensures that the software that has been built is traceable to customer requirements. If requirements are not validated, errors in the requirement definitions would propagate to the successive stages resulting in a lot of modification and rework.

The main steps for this process include:

- The requirements should be consistent with all the other requirements i.e. no two requirements should conflict with each other.
- The requirements should be complete in every sense.
- The requirements should be practically achievable.
- Reviews, buddy checks, making test cases, etc. are some of the methods used for this.
- Requirements verification and validation (V&V) is the process of checking that the requirements for a software system are complete, consistent, and accurate and that they meet the needs and expectations of the stakeholders.

- Verification is checking that the requirements are complete, consistent, and accurate.
- It involves reviewing the requirements to ensure that they are clear, testable, and free of errors and inconsistencies.
- This can include reviewing the requirements document, models, and diagrams, and holding meetings and walkthroughs with stakeholders.
- Validation is the process of checking that the requirements meet the needs and expectations of the stakeholders.
- It involves testing the requirements to ensure that they are valid and that the software system being developed will meet the needs of the stakeholders.
- This can include testing the software system through simulation, testing with prototypes, and testing with the final version of the software.

Verification and Validation is an iterative process that occurs throughout the software development life cycle.

It is important to involve stakeholders and the development team in the V&V process to ensure that the requirements are thoroughly reviewed and tested.

- It's important to note that V&V is not a one-time process, but it should be integrated and continue throughout the software development process and even in the maintenance stage.

5. Requirements Management

- Requirement management is the process of analyzing, documenting, tracking, prioritizing, and agreeing on the requirement and controlling the communication with relevant stakeholders.
- This stage takes care of the changing nature of requirements.
- It should be ensured that the SRS is as modifiable as possible to incorporate changes in requirements specified by the end users at later stages too.
- Modifying the software as per requirements in a systematic and controlled manner is an extremely important part of the requirements engineering

- Requirement management is the process of analyzing, documenting, tracking, prioritizing, and agreeing on the requirement and controlling the communication with relevant stakeholders.
- This stage takes care of the changing nature of requirements.
- It should be ensured that the SRS is as modifiable as possible to incorporate changes in requirements specified by the end users at later stages too.
- Modifying the software as per requirements in a systematic and controlled manner is an extremely important part of the requirements engineering

Several key activities are involved in requirements management

- **Tracking and controlling changes:** This involves monitoring and controlling changes to the requirements throughout the development process, including identifying the source of the change, assessing the impact of the change, and approving or rejecting the change.
- **Version control**: This involves keeping track of different versions of the requirements document and other related artifacts.
- **Traceability**: This involves linking the requirements to other elements of the development process, such as design, testing, and validation.
- **Communication:** This involves ensuring that the requirements are communicated effectively to all stakeholders and that any changes or issues are addressed promptly.
- **Monitoring and reporting**: This involves monitoring the progress of the development process and reporting on the status of the requirements.

Requirements management is a critical step in the software development life cycle as :

- it helps to ensure that the software system being developed meets the needs and expectations of stakeholders
- Ensures that it is developed on time, within budget, and to the required quality.
- It also helps to prevent scope creep and to ensure that the requirements are aligned with the project goals.

Context Diagrams

- Context diagrams serve as a foundational tool, helping designers and stakeholders grasp the scope and boundaries of a system under consideration.
- These diagrams provide a high-level view, illustrating how the system interacts with external entities and the environment.

Importance of Context Diagrams in Systems Analysis

- **Scope Definition:** Context diagrams define the system's boundaries by highlighting its interactions with external entities, ensuring that the analysis focuses on pertinent components and processes.
- **Requirement Gathering:** These diagrams visualize how the system interacts with its environment, aiding in identifying both functional and non-functional requirements. They offer clarity on the system's objectives and its external interactions.
- **Communication:** Acting as a bridge between stakeholders, such as business users, developers, and project managers, context diagrams foster shared understanding of the system's scope and context. They streamline discussions and decision-making throughout the development process.
- **Risk Identification:** Context diagrams assist in spotting potential risks stemming from the system's interactions with external entities. They help stakeholders assess the implications of external factors on the system's performance, security, and reliability.

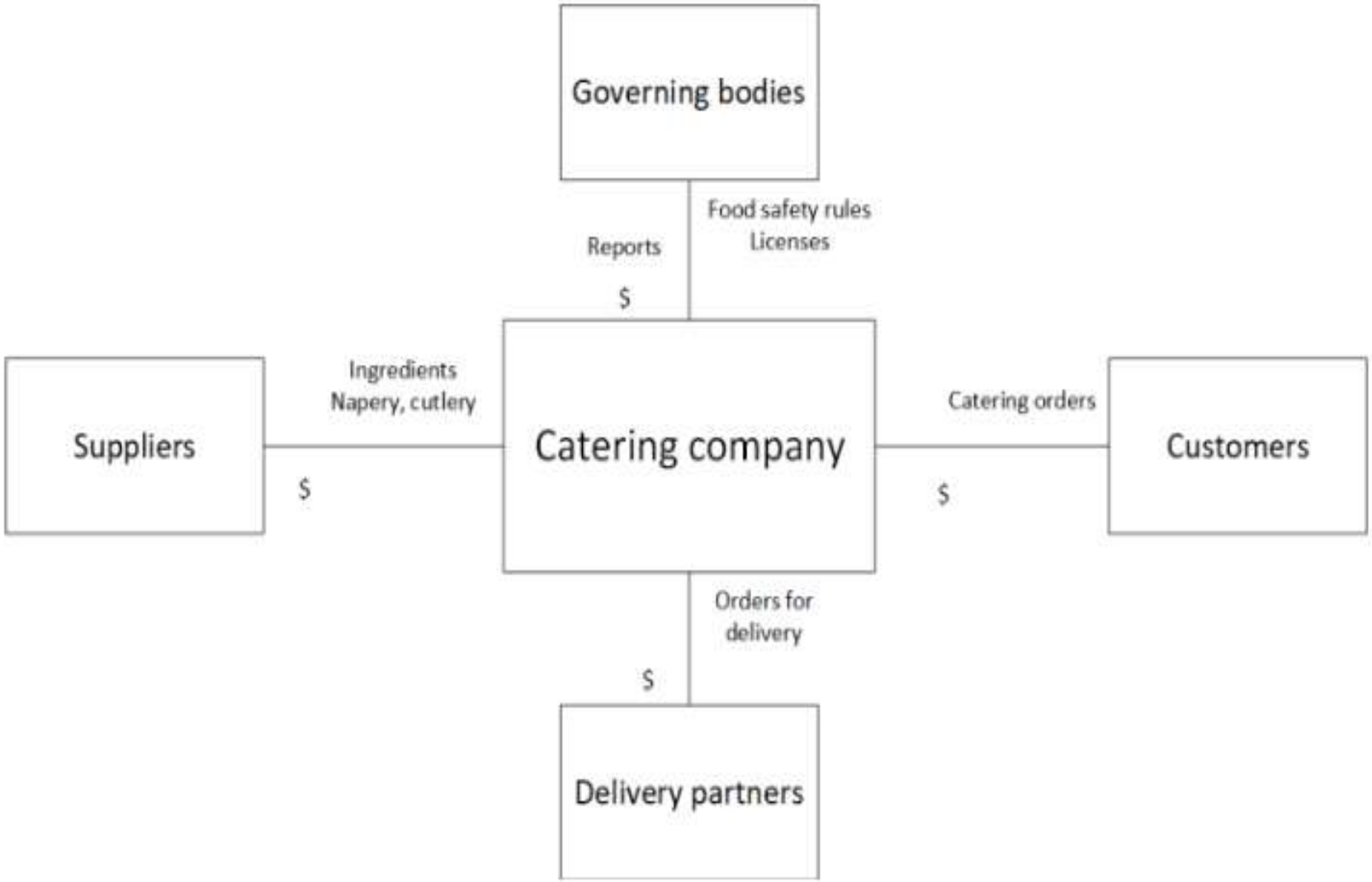
Components of Context Diagrams

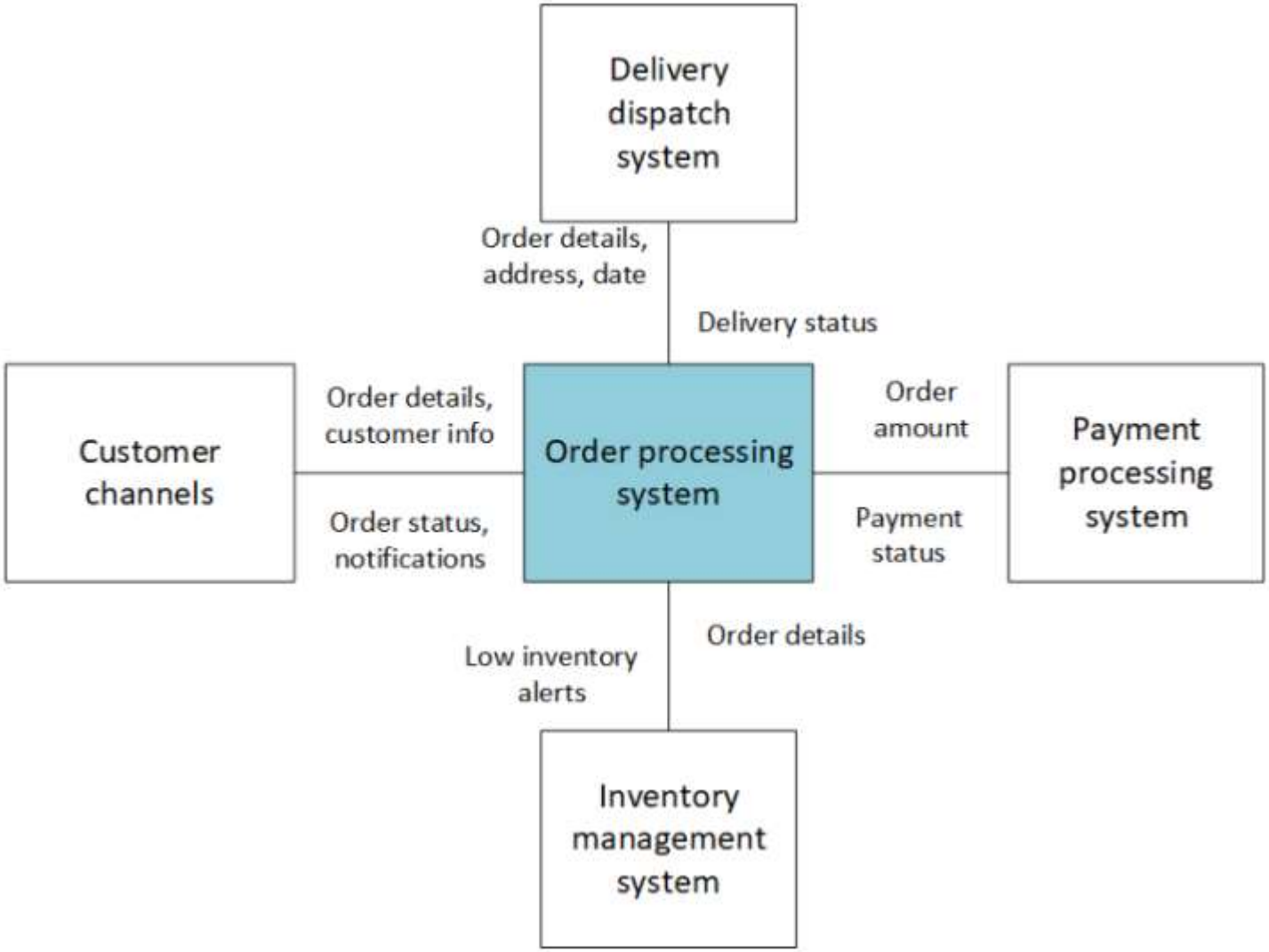
- **1. System/Product:** This is the primary focus of the diagram, representing the system being analyzed or designed.

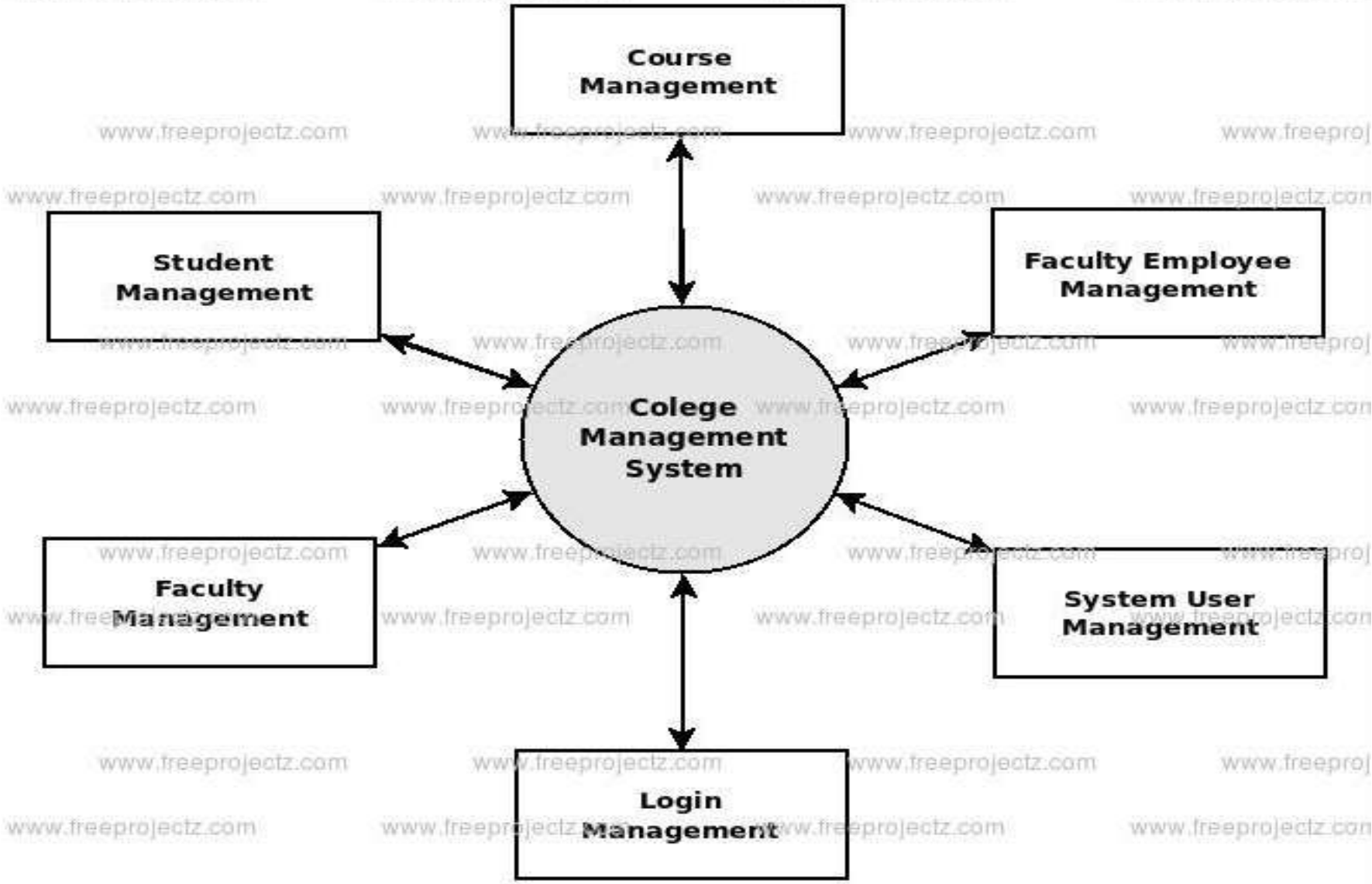
2. External Entities

- These are entities outside the system boundary that interact with the system. They could be users, other systems, or processes that provide input to or receive output from the system.

External Entity


Governing bodies
Food safety rules
Licenses
Reports
$
Suppliers
Ingredients
Napery, cutlery
$
Catering company
Catering orders
$
Customers
Orders for
delivery
$
Delivery partners


Delivery dispatch system
Order details, address, date
Delivery status
Customer channels
Order details, customer info
Order processing system
Order amount
Payment processing system
Order status, notifications
Payment status
Low inventory alerts
Order details
Inventory management system


Course Management
Student Management
Faculty Employee Management
Colege Management System
Faculty Management
System User Management
Login Management

Behavioral Diagrams in Software Engineering

- Behavioral diagrams are a category of diagrams in software engineering, particularly within Unified Modeling Language (UML), used to capture the dynamic aspects of a system. These diagrams model the interactions, activities, and changes that occur within the software over time.

Purpose of Behavioral Diagrams

- Represent how system components interact and behave.
- Show the flow of control and data in various scenarios.
- Help in understanding system functionality, requirements, and possible use cases.
- Useful for both analysis and design phases of software development.

Types of Behavioral Diagrams

Use Case Diagram

- Describes the functional requirements of a system.
- Shows actors (users/external systems) and their interactions with use cases (system functions).
- Usage: Early stages of development to capture user requirements.

Sequence Diagram

- Visualizes object interactions in a specific scenario of a use case.
- Shows how messages are exchanged in a chronological sequence.
- Usage: Detailing system logic, identifying responsibilities of objects.

Activity Diagram

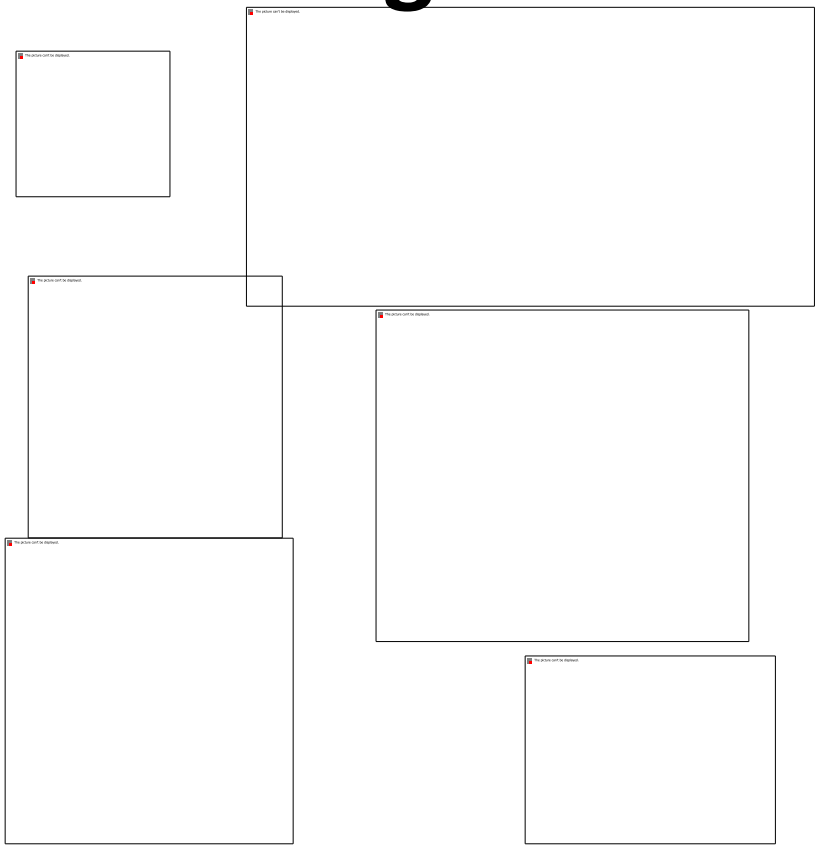
- Represents workflows of stepwise activities and actions.
- Illustrates flow of control from one activity to another.
- Usage: Modeling business processes, describing algorithm logic.

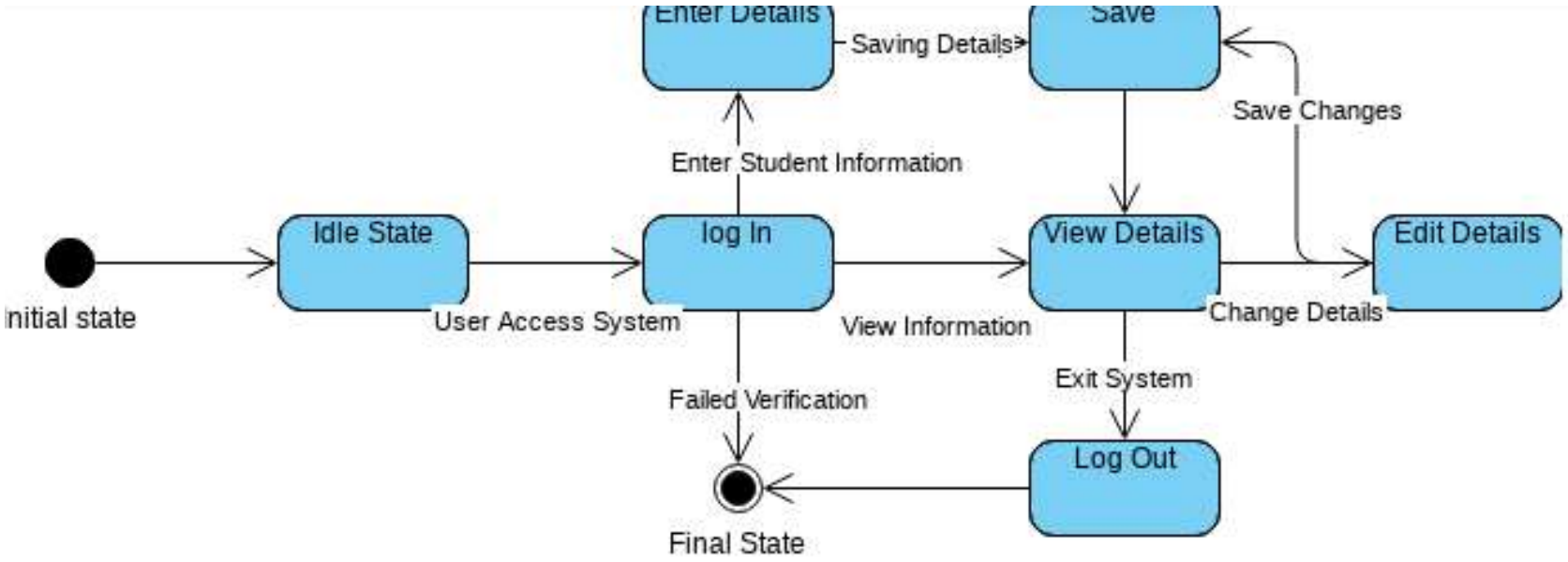
State Machine Diagram (Statechart)

- Depicts the states of an object and transitions triggered by events.
- Shows lifecycle of an object, from creation to destruction.
- Usage: Modeling complex object behavior and event-driven systems.
- State Machine diagrams are also known as **State Diagrams** and **State-Chart Diagrams**.

Basic Components and Notations of a State Machine Diagram

1. Initial state
2. Transition
3. State
4. Fork
5. Join
6. Final State

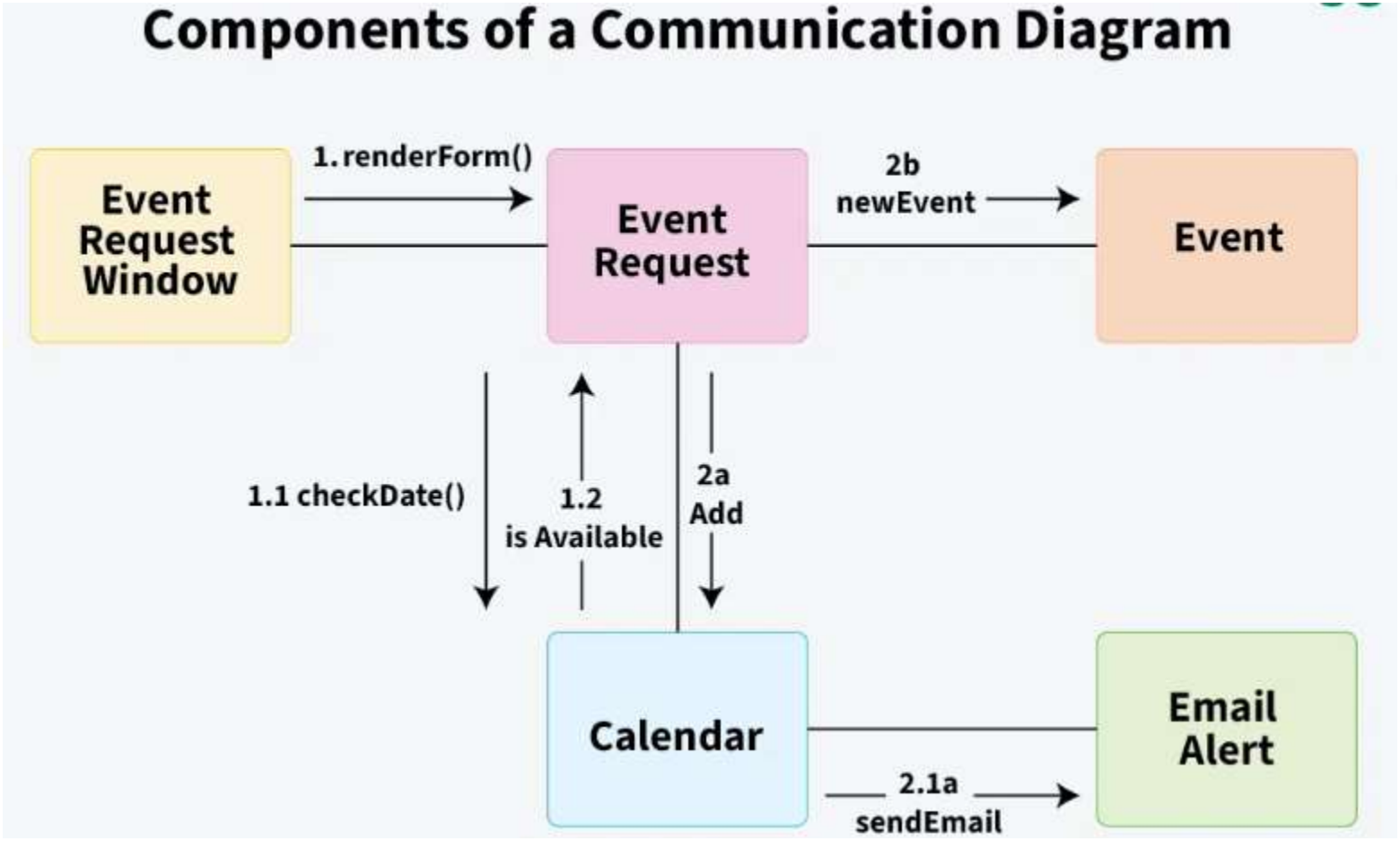



Enter Details
Saving Details
Save
Save Changes
Enter Student Information
initial state
Idle State
log In
View Details
Edit Details
User Access System
View Information
Change Details
Failed Verification
Exit System
Log Out
Final State

Communication Diagram

- Describes interactions between objects focusing on message flow and links between participants.
- Alternative to sequence diagrams for showing relationships.
- visually represents the interactions between objects or components in a system. It focuses on how messages are exchanged between these elements, highlighting the flow of information in a sequence.

Components of a Communication Diagram



Key Elements in Behavioral Diagrams

- Actors: Entities interacting with the system.
- Objects: Components participating in interactions.
- States: Conditions or situations during the life of an object.
- Events: Triggers that cause transitions or actions.
- Transitions: Movement from one state to another.
- Messages: Information or invocation sent between objects.

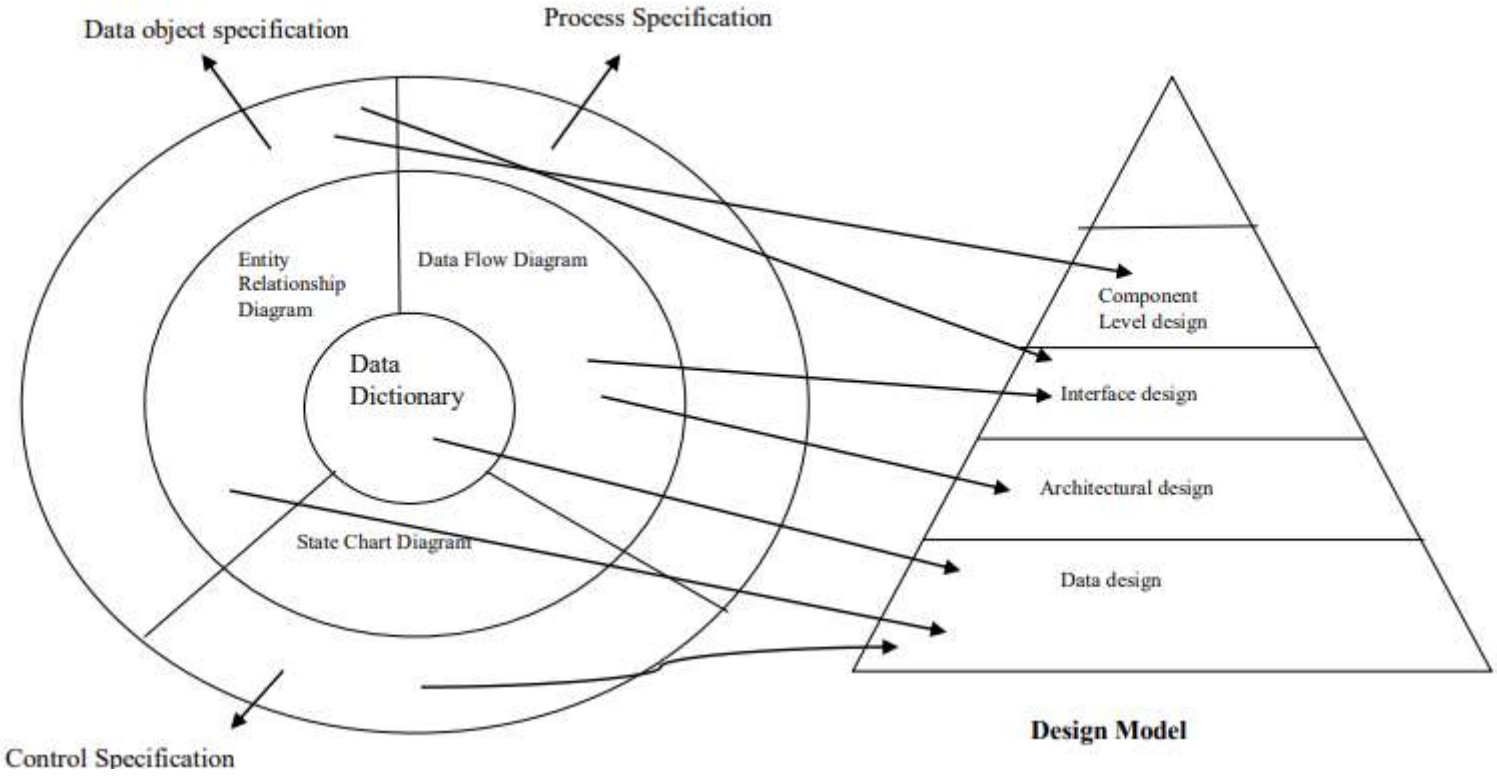
Applications of Behavioral Diagrams

- Clarifying complex system behavior.
- Validating system requirements and expected responses.
- Facilitating communication among stakeholders.
- Assisting in system design and documentation.
-

Unit 3:
Software Design, UML, Architecture, UI Design, Metrics

SOFTWARE DESIGN PROCESS:

- Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.
- For assessing user requirements, an SRS document is created
- for coding and implementation, there is a need of more specific and detailed requirements in software terms.
- The output of this process can directly be used into implementation in programming languages


Data object specification
Process Specification
Entity
Relationship
Diagram
Data Flow Diagram
Data
Dictionary
State Chart Diagram
Control Specification
Component
Level design
Interface design
Architectural design
Data design
Design Model

The Design Process:

- Software design is an iterative process through which requirements are translated into blue print for constructing the software.

- Initially, the blueprint depicts a holistic view of software.

- The design is represented at a high level of abstraction at level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements.
- As design iterations occur, subsequent refinement leads to design representations at much lower level of abstraction.

DESIGN CONCEPTS AND PRINCIPLES:

- Software design is both a process and a model.
- The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built.
- The design process is combination of Creative skill, past experience, and a sense of what makes “good” software

- The design model is equivalent of an architect’s plan for a house.
- It begines by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house)
- And slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout).

- Similarly, the design model that is created for software provides a variety of different views of the computer software. Basic design principles enable the software engineer to navigate the design process.

Principles for software design:

- The design process should avoid tunnel vision.
- The design must be traceable to the analysis model.
- Do not reinvent the wheel – use existing solutions where possible.
- Minimize the intellectual gap between software and the real-world problem.
- Software design structure should, whenever possible, mimic the problem domain structure.
- The design should have uniformity and integration.
- It should be structured to handle changes easily.
- The design should degrade gracefully (fail without catastrophic breakdown).
- Assess design quality while creating it, not afterward.
- Review the design to minimize conceptual or semantic errors.

Design concepts

Abstraction:
Abstraction allows a person to focus on a problem at a certain level without worrying about lower-level details. At the highest level of abstraction, the solution is described in broad terms using the language of the problem environment. At a lower level, the solution takes a more procedural form. Finally, at the lowest level of abstraction, the solution is described in a way that can be directly implemented.

Types of Abstraction:

Procedural Abstraction

Data Abstraction

- **Refinement:**
  Refinement is the process of elaboration. A function defined at the abstract level is gradually decomposed into more detailed steps until it can be expressed as programming language statements.
- **Modularity:**
  Modularity means dividing software into separately named and addressable components called modules. It follows the "divide and conquer" concept, where a complex problem is broken down into several manageable parts.

SOFTWARE MODELING AND UML:

- Software modeling: Software models are ways of expressing a software design. Usually some sort of abstract language or pictures are used to express the software design. For object-oriented software, an object modeling language such as UML is used to develop and express the software design.

Unified Modeling Language (UML):

- UML allows a software engineer to express an analysis model using a modeling notation that is governed by a set of syntactic, semantic, and pđagŵatiĐ đules. IŶ UML, a sLJsteŵ is đepđeseŶted usiŶg fi|e diffeđeŶt┬ţie‖s that describe the system from distinctly different perspectives. Each view is defined by a set of diagrams

The following views are present in UML:

- **User Model View:**
  This view represents the system (or product) from the user's perspective, where users are called *actors* in UML. The *use-case* diagram is the preferred modeling technique for this view. It describes how the system will be used, focusing on the end-user's interaction and experience.
- **Structural Model View:**
  This view focuses on the internal structure of the system — including data, classes, objects, and their relationships. It represents the static aspects of the system.

- **Behavioral Model View:**
  This view represents the dynamic behavior of the system. It shows how different structural elements interact or collaborate, as described in the user and structural model views.
- **Implementation Model View:**
  This view shows how the structural and behavioral aspects of the system are actually implemented or built in practice.
- **Environment Model View:**
  This view represents the structural and behavioral aspects of the environment in which the system operates or is implemented.

Refactoring or **Code Refactoring**

- **It is** systematic process of improving existing computer code, without adding new functionality or changing external behaviour of the code.
- It is intended to change the implementation, definition, structure of code without changing functionality of software.
- It improves extensibility, maintainability, and readability of software without changing what it actually does.

UML Diagram Types:

- **There are several types of UML diagrams:**
- **User Model View** is represented through:
  Use-Case Diagram: It shows the *actors*, *use-cases*, and the *relationships* between them.
- **Structural Model View** is represented through:
  Class Diagram: It shows the relationships between classes and important details about each class.
  Object Diagram: It shows a specific configuration of objects at a particular moment in time.

Behavioral model view represents through **Interaction Diagrams:**

- Show an interaction between groups of collaborating objects.
- Two types:
  - Collaboration diagram and sequence diagram
- **Package Diagrams**: Shows system structure at the library/package level.
- **State Diagram**: Describes behavior of instances of a class in terms of states, stimuli, and transitions.
- **Activity Diagram**: Very similar to a flowchart— shows actions and decision points, but with the ability to accommodate concurrency.

Environment model view represent through **Deployment Diagram**

- Shows configuration of hardware and software in a distributed system.
- **Implementation model** view represent through **Component Diagram**: It shows code modules of a system. This code module includes application program, ActiveX control, Java beans and back end databases.
- It representing interfaces and dependencies among software architect.

Design Classes

- **Types:**
- There are 5 different types of design classes that represent a different layer of design architecture that can be developed:
- **User interface classes** define abstraction that mandatory for human-computer interaction [HCI]. In cases, HCI occurs within the context of metaphor, and design classes for the interface may be visible representations of elements of metaphor.
- **Business domain classes** are often refinements of analysis classes defined earlier. The class identifies the attributes that are required to implement some elements of the business domain.

- **Process classes implement** lower-level business Preoccupation need to manage business domain classes.
- **Persistent classes** represent the data stores that will persist beyond the execution of software.
- **System classes implement** software management and control function that permits the system to operate and convey within its computing environment and with the outside world

Characteristics:

- **Complete and sufficient:** A design class should be complete encapsulation of all attributes and method that can be reasonably be expected to exist for class. For example, the class **scene** defined for video editing software is complete only if it contains all attributes and methods that can agreeably be associated with the creation of a video scene. Sufficiently ensure that design class contains only those methods that are sufficient to achieve the intent of class, no more and no less.
- **Primitiveness:** Method associate with design class should be focused on accomplishing one service for class. Once service implemented with the method, the class should not provide another way to accomplish the same thing. For example, the class **Video Clip** for video editing software might have attributes *start point* and *end point* to specify start and endpoint of clip.

- **High Cohesion:/strong> A** cohesion design class has a small, concentrated set of authority and single-mindedly applies attributes and methods to implement those responsibilities. For example, the class video clip might contain set of method for editing the video clip. As long as each method focus solely on attributes associated with video clip, cohesion is maintained.
- **Low Coupling:** Within the design model, it is necessary for design classes to get together with one another. However, get together should be kept to an acceptable minimum. If the design model is highly coupled, the system is difficult to implement to test and to maintain over time, In general, design classes within subsystem should have only limited knowledge of other classes. This restriction called the Law of Demeter, suggest that method should only send message to methods in neighboring classes.
-

Package Diagram

- A package diagram is a type of structural diagram in UML (Unified Modeling Language) that organizes and groups related classes and components into packages. It visually represents the dependencies and relationships between these packages, helping to illustrate how different parts of a system interact with each other.

Basic Elements of Package Diagrams

- **Package**: This is the fundamental unit of a package diagram, serving as a container for various elements like classes and interfaces. It's depicted as a folder-like icon with a name label, making it easy to identify.
- **NameSpace**: This denotes the name of the package and usually appears at the top of the package symbol. It helps uniquely identify the package within the diagram.
- **Package Merge**: This relationship illustrates how one package can be merged with another. It's represented by a direct arrow between the two packages, indicating that their contents can combine.
- **Package Import**: This relationship shows that one package can access the contents of another package, depicted with a dashed arrow.
- **Dependency**: Dependencies indicate that changes in one package may affect another. This relationship signifies that one element or package relies on another, highlighting how interconnected they are.
- **Element**: An element is a single unit within a package, which can be a class, interface, or subsystem. Elements reside inside packages and are connected to the main package. For instance, a class may contain various functions and variables, all of which are considered elements tied to that class.
- **Constraint**: This represents a condition or requirement associated with a package, typically shown in curly braces. Constraints help define the rules or limitations for how the package operates.

1. Package

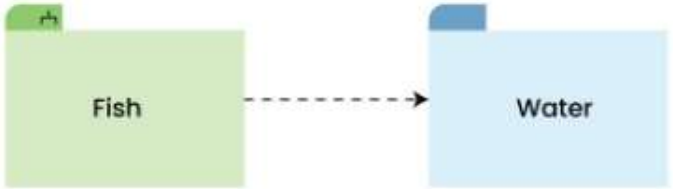
Package

2. Subsystem

Subsystem

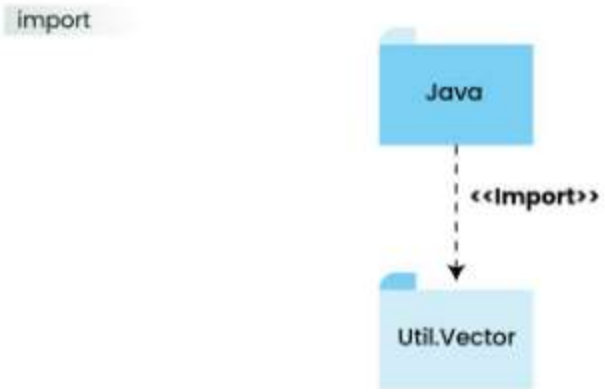
Dependency

Dependency



Fish is Dependant on Water

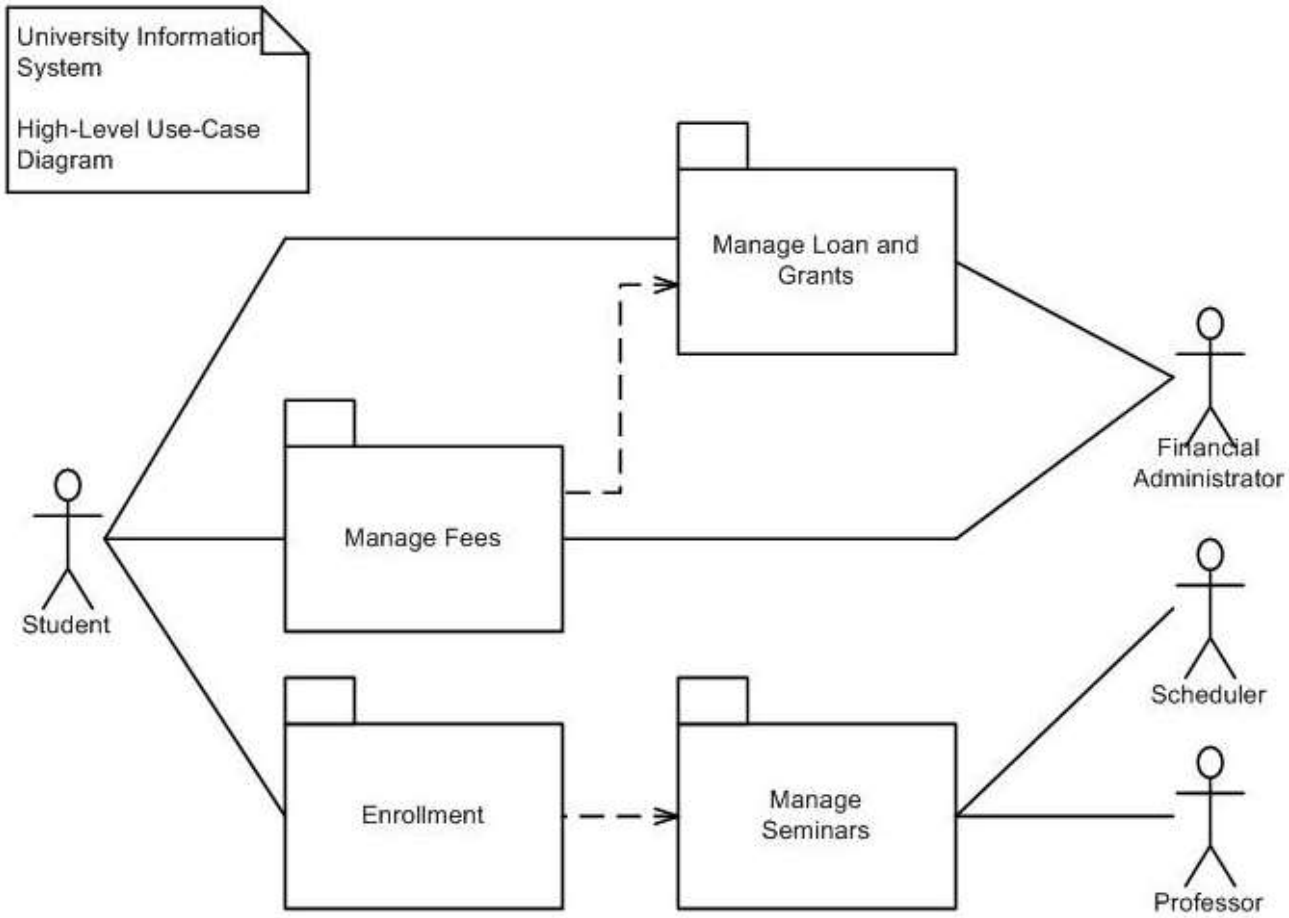
Import



Merge

Package Relationships

- **Package Merge Relationship**
- This relationship is used to represent that the contents of a package can be merged with the contents of another package. This implies that the source and the target package has some elements common in them, so that they can be merged together.
- **Package Dependency Relationship**
- A package can be dependant on other different packages, signifying that the source package is somehow dependent on the target package.
- **Package Import Relationship**
- This relationship is used to represent that a package is importing another package to use. It signifies that the importing package can access the public contents of the imported package.
- **Package Access Relationship**
- This type of relationship signifies that there is a access relationship between two or more packages, meaning that one package can access the contents of another package without importing it.

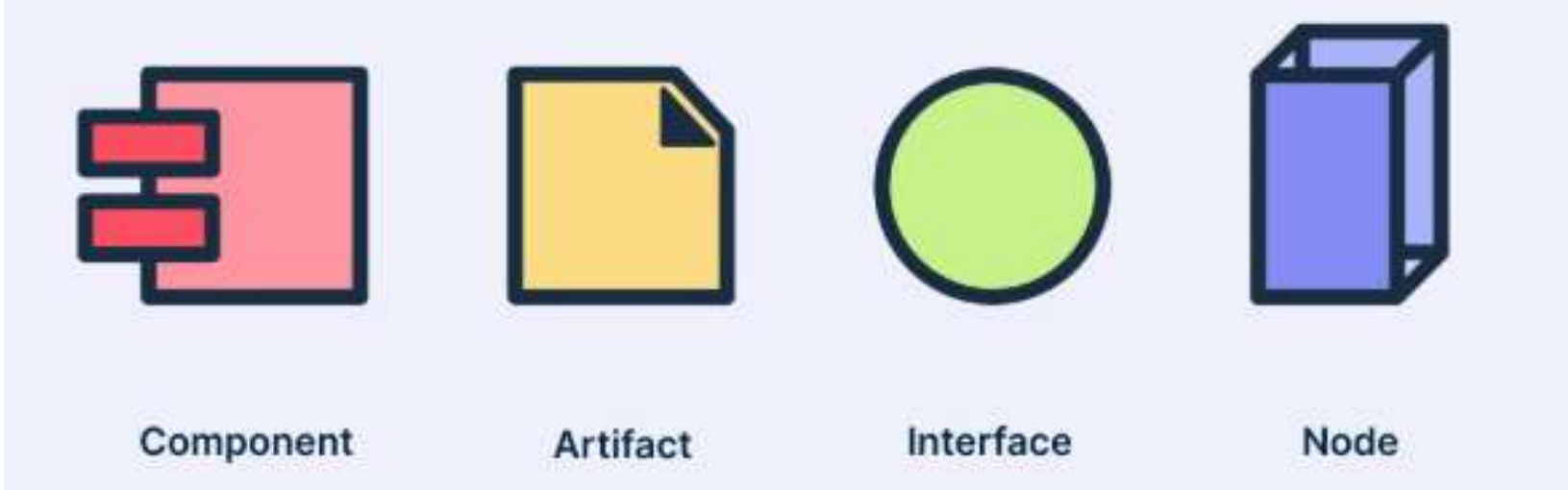

University Information System
High-Level Use-Case Diagram
Manage Loan and Grants
Financial Administrator
Manage Fees
Student
Scheduler
Enrollment
Manage Seminars
Professor

Deployment Diagram

- A Deployment Diagram shows how the software design turns into the actual physical system where the software will run. They show where software components are placed on hardware devices and shows how they connect with each other. This diagram helps visualize how the software will operate across different devices

Key elements of a Deployment Diagram

- **Nodes**: These represent the physical hardware entities where software components are deployed, such as servers, workstations, routers, etc.
- **Components**: Represent software modules or artifacts that are deployed onto nodes, including executable files, libraries, databases, and configuration files.
- **Artifacts**: Physical files that are placed on nodes represent the actual implementation of software components. These can include executable files, scripts, databases, and more.
- **Dependencies**: These show the relationships or connections between nodes and components, highlighting communication paths, deployment constraints, and other dependencies.
- **Associations**: Show relationships between nodes and components, signifying that a component is deployed on a particular node, thus mapping software components to physical nodes.
- **Deployment Specification**: This outlines the setup and characteristics of nodes and components, including hardware specifications, software settings, and communication protocols.
- **Communication Paths**: Represent channels or connections facilitating communication between nodes and components and includes network connections, communication protocols, etc.


Component
Artifact
Interface
Node

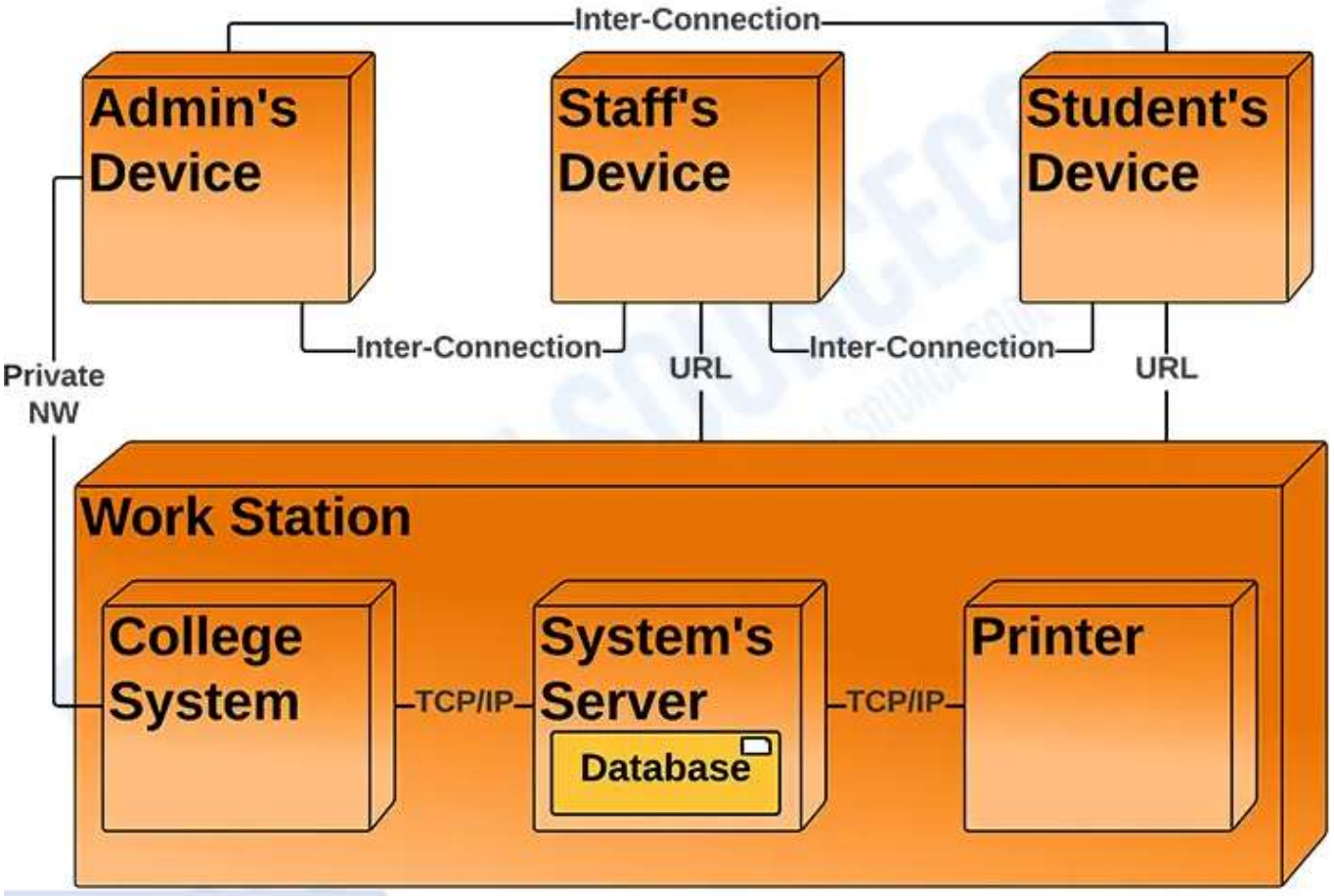
Use Cases of Deployment Diagrams

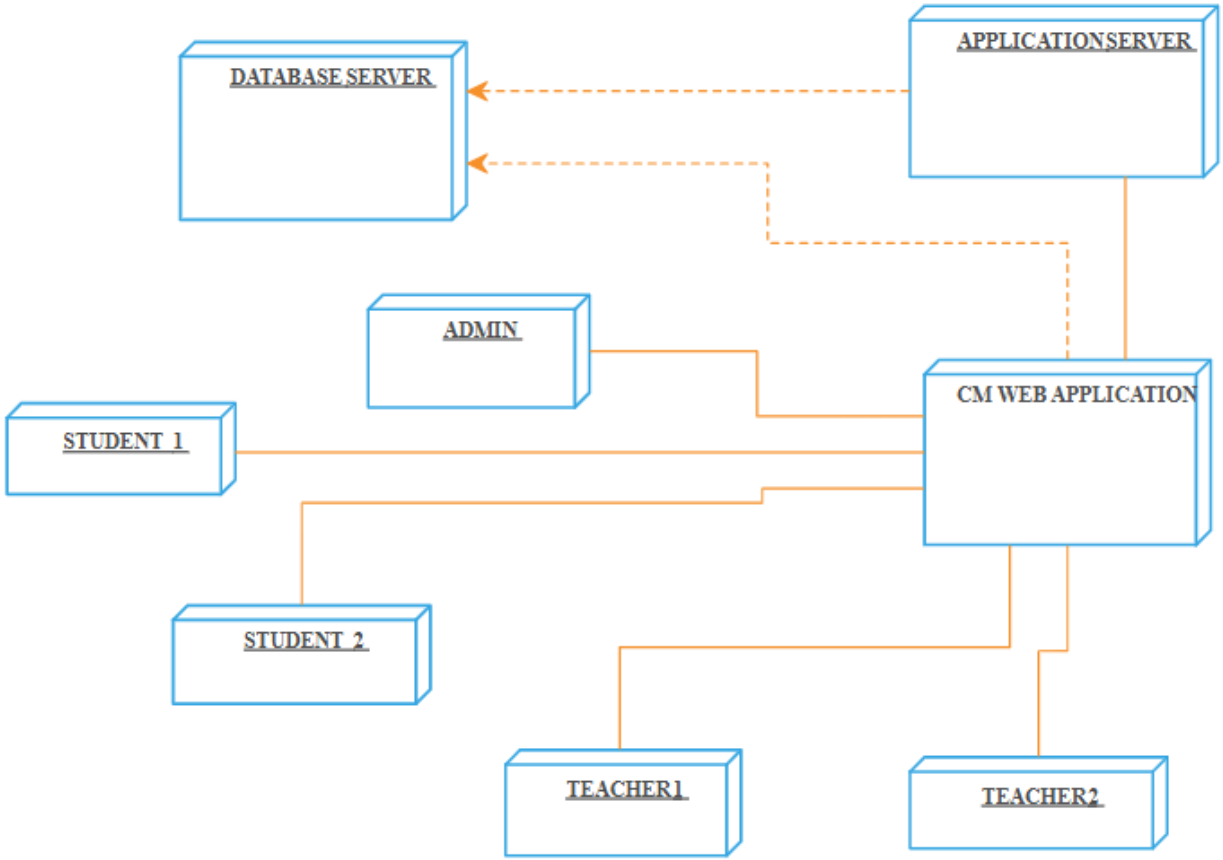
- Deployment diagrams help plan how software systems will be set up on different devices.
- They help design the hardware needed to support the software. By showing which software parts go where, they help decide what devices and networks are needed.
- Deployment diagrams make sure each part of the software has enough resources, like memory or processing power, to run well.
- They show how different parts of the software depend on each other and on the hardware.
- By seeing how everything is set up, teams can find ways to make the software run faster and smoother.

Steps for creating a Deployment Diagram

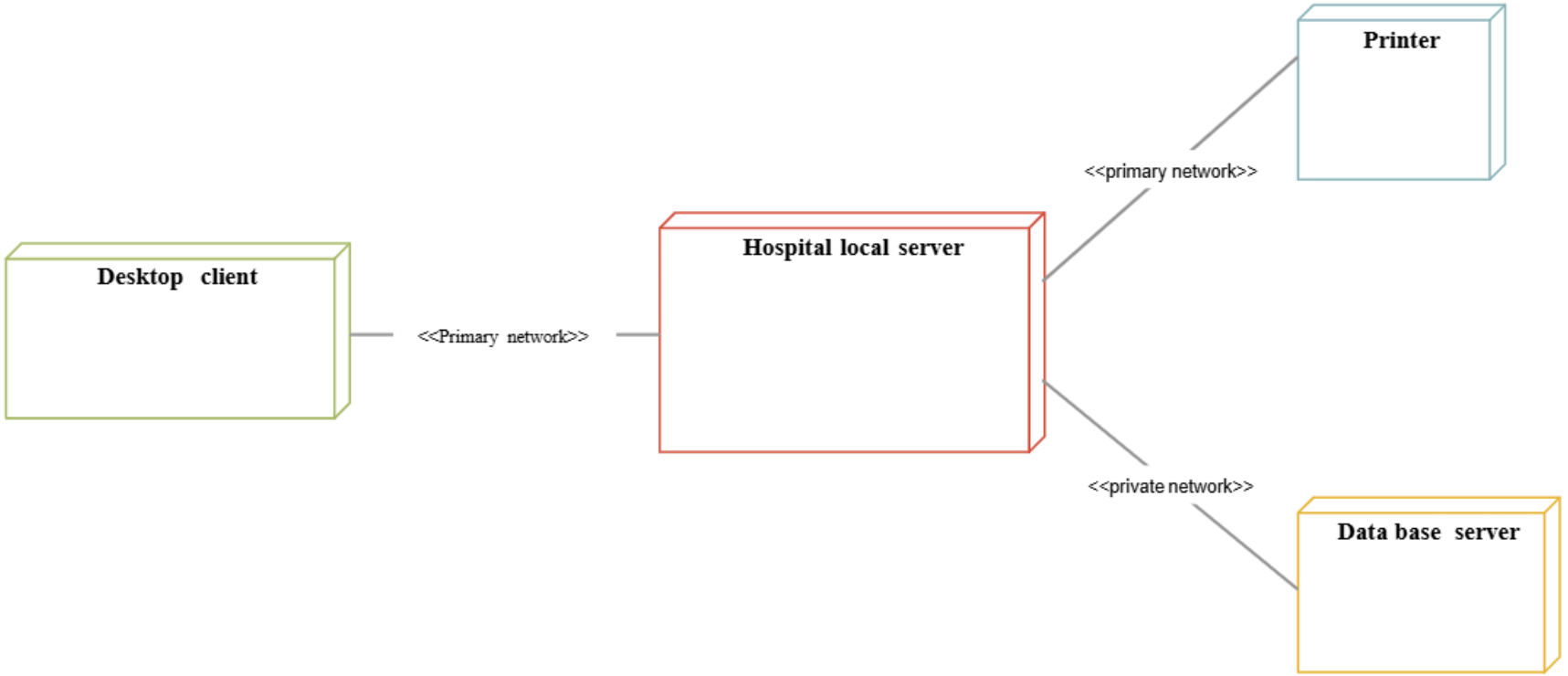
- **Step1: Identify Components:** List all software parts and hardware devices that will be in the deployment diagram.
- **Step 2: Understand Relationships**: Figure out how these parts connect and work together.
- **Step 3: Gather Requirements:** Collect details about hardware, network setups, and any special rules for deployment.
- **Step 4: Draw Nodes and Components:** Start by drawing the hardware devices (nodes) and software parts (components) using standard symbols roughly at first improvise it and draw the final one.
- **Step 5: Connect Nodes and Components:** Use lines or arrows to show how nodes and components are linked.
- **Step 6: Add Details:** Label everything clearly and include any extra info, like hardware specs or communication protocols.

College management




APPLICATIONSERVER
DATABASE SERVER
ADMIN
CM WEB APPLICATION
STUDENT_1
STUDENT_2
TEACHER1
TEACHER2

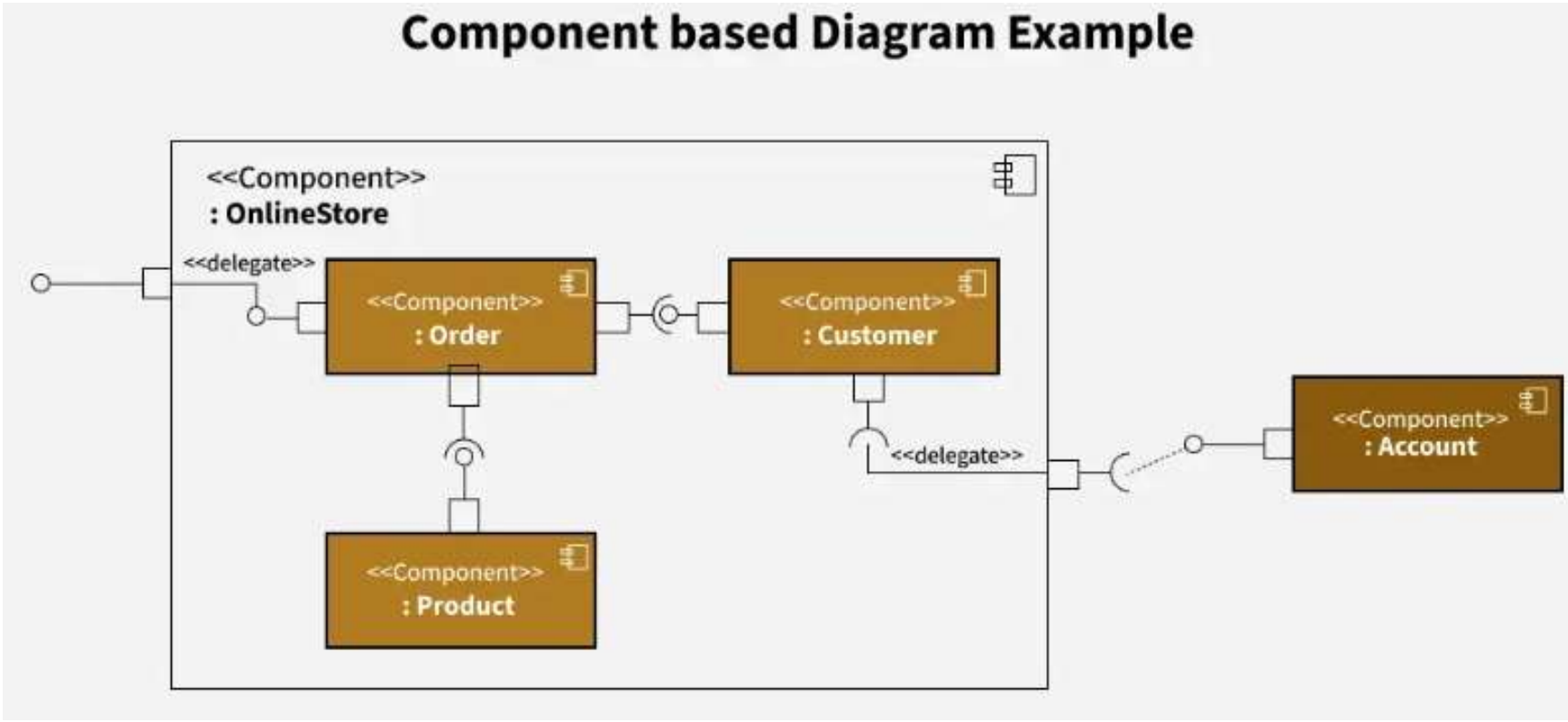
HOSPITALMANAGEMENTSYSTEM

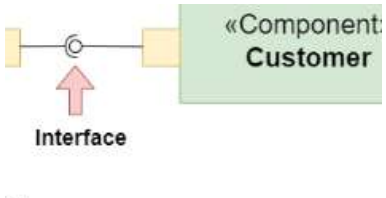


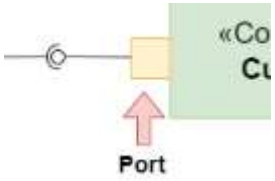
Component Based Diagram

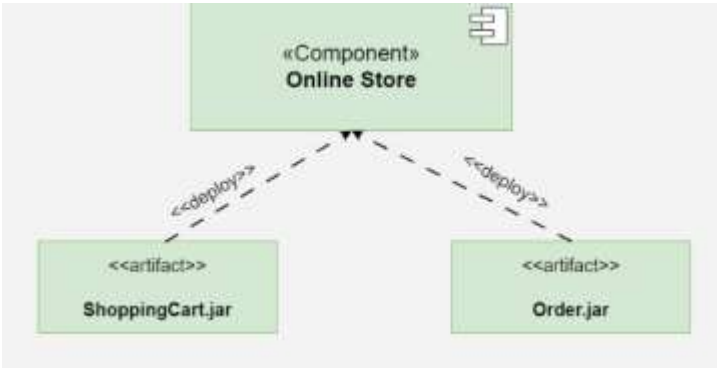
- Component-based diagrams are essential tools in software engineering, providing a visual representation of a system's structure by showcasing its various components and their interactions. These diagrams simplify complex systems, making it easier for developers to design, understand, and communicate the architecture.
- One kind of structural diagram

Component based Diagram Example







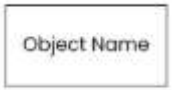


Collaboration Diagrams

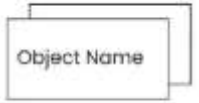
- In UML (Unified Modeling Language), a Collaboration Diagram is a type of Interaction Diagram that visualizes the interactions and relationships between objects in a system. It shows how objects collaborate to achieve a specific task or behavior. Collaboration diagrams are used to model the dynamic behavior of a system and illustrate the flow of messages between objects during a particular scenario or use case.
- A collaboration diagram is a behavioral UML diagram which is also referred to as a communication diagram. It illustrates how objects or components interact with each other to achieve specific tasks or scenarios within a system.

Components and their Notations in Collaboration Diagrams

- **Objects/Participants**



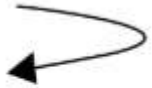
Object



Multiple Objects

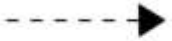
Actor

Message



Self Message

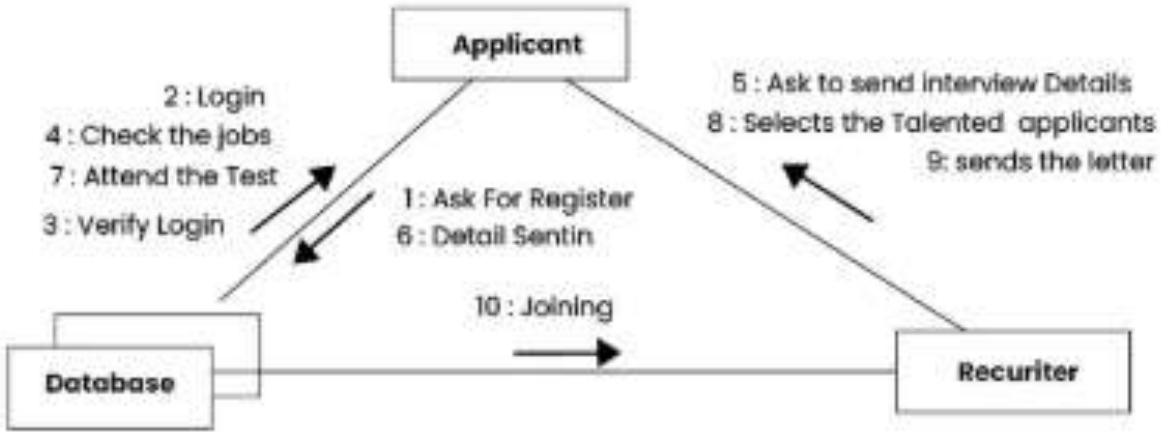
Link



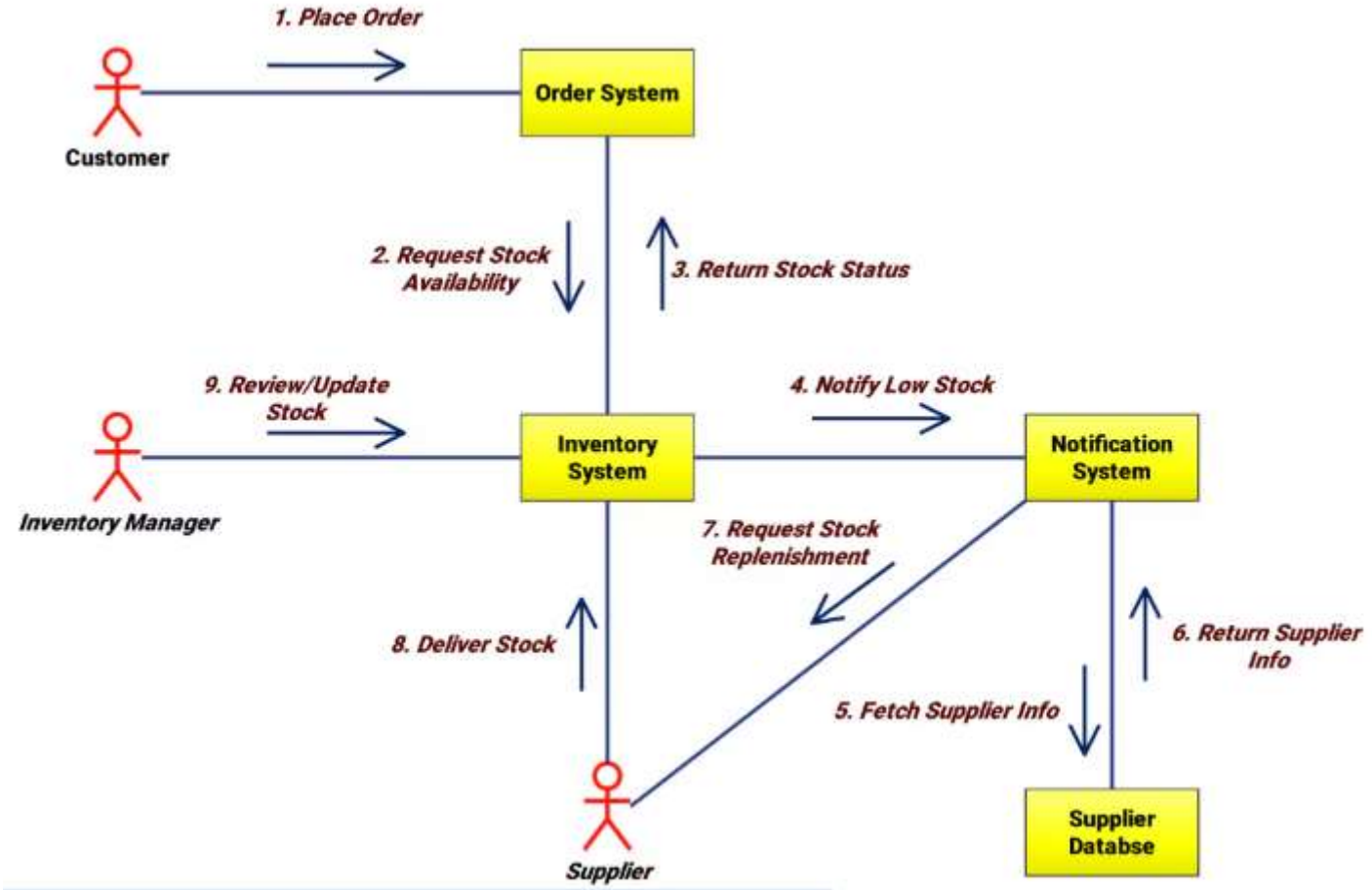
Return Message

How to draw Collaboration Diagrams?

- **Step 1: Identify Objects/Participants:** Start by figuring out the objects or participants in the system. These can be classes, modules, actors, or any other important entities.
- **Step 2: Define Interactions:** Determine how these objects work together to complete tasks or scenarios in the system. Identify the messages they exchange during these interactions.
- **Step 3: Add Messages:** Draw arrows between lifelines to show the messages exchanged between objects. Label each arrow with the message name and any relevant parameters or data being sent.
- **Step 4: Consider Relationships:** If there are connections or dependencies between objects, show these using the right notations, like dashed lines or arrows.
- **Step 5: Documentation:** Once you're done, document the collaboration diagram with any necessary explanations or notes. Make sure the diagram clearly communicates the system's interactions to stakeholders



Job Recruitment system


1. Place Order
Customer
Order System
2. Request Stock Availability
3. Return Stock Status
9. Review/Update Stock
Inventory Manager
Inventory System
4. Notify Low Stock
Notification System
7. Request Stock Replenishment
8. Deliver Stock
6. Return Supplier Info
5. Fetch Supplier Info
Supplier
Supplier Databse