**User story**

**Identify the Users:**

* Determine who will be using the system or feature. These could be end-users, administrators, or any other relevant roles.

**Determine the User Goals:**

* Understand what the users want to achieve with the system. Conduct interviews, surveys, or observation sessions to gather these goals.

**Write the User Stories:**

* Use the format: "As a [type of user], I want [some goal] so that [some reason]."
* Ensure that each story is concise and specific.

**Prioritize the User Stories:**

* Prioritize the user stories based on their importance to the users and the business goals. This helps in deciding which features to implement first.

**Add Acceptance Criteria:**

* Define the conditions that must be met for the user story to be considered complete. Acceptance criteria help in ensuring that the implementation meets the user’s needs.

**Benefits of User Stories**

* **Focus on User Needs:** Ensures that the development is centered around the user's needs and goals.
* **Encourages Collaboration:** Promotes communication and collaboration between developers, users, and other stakeholders.
* **Flexibility:** User stories can easily be updated or changed as more is learned about the user's needs.
* **Prioritization:** Helps in prioritizing features and functionality based on their value to the user and the business.

**Functional vs. Non-functional Requirements**

* **Functional Requirements:** Specify what the system should do. They describe the functions and features of the system, such as user interactions, data processing, and other specific behaviours.
  + **Example:** The system must allow users to log in with a username and password.
* **Non-functional Requirements:** Specify how the system should perform its functions. They define the quality attributes, performance, and constraints of the system.
  + **Example:** The system must respond to user inputs within 2 seconds.

**Viewpoints of Requirements**

* **Stakeholder Viewpoint:** Focuses on the needs and concerns of stakeholders (e.g., users, customers, management).
* **System Viewpoint:** Focuses on the technical aspects and architecture of the system.
* **Functional Viewpoint:** Focuses on the functionality that the system must provide.
* **Non-functional Viewpoint:** Focuses on the quality attributes and performance constraints.

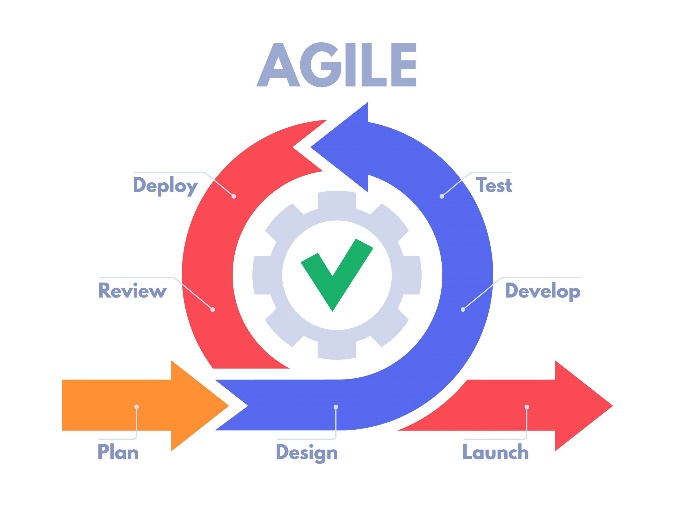
**MoSCoW Method**

* **M:** Must have – Critical requirements that the system must fulfill.
* **S:** Should have – Important requirements that should be included if possible.
* **C:** Could have – Desirable requirements that could be included if time and resources permit.
* **W:** Won't have – Requirements that will not be included in this release but may be considered for future releases.

**Why Do Requirements Need Tests?**

* **Validation:** Ensures that the requirements accurately reflect the needs of the stakeholders.
* **Verification:** Confirms that the system meets the specified requirements.
* **Quality Assurance:** Helps identify and fix issues early in the development process, improving the overall quality of the system.
* **Traceability:** Provides a clear link between requirements and test cases, ensuring all requirements are tested and met.

**Agile vs. Waterfall**

* **Agile:** 
  + Iterative and incremental development.
  + Emphasizes flexibility, collaboration, and customer feedback.
  + Frequent releases with small update, manageable tasks.
  + Continuous integration and testing.
* **Waterfall:**
  + Linear and sequential development.
  + Emphasizes detailed planning and fixed requirements.
  + Each phase must be completed before the next begins.
  + Testing is typically done after the implementation phase.

**Agile Limitations**

1. **Lack of Predictability:**
   * Agile methods rely on iterative development and continuous feedback, which can make it difficult to predict timelines and costs upfront. This can be challenging for projects with fixed budgets and deadlines.
2. **Requires Experienced Team:**
   * Agile practices require a high level of collaboration, communication, and self-management. Teams need to be experienced and skilled in Agile methodologies to effectively implement Agile practices.
3. **Scope Creep:**
   * The flexibility of Agile can lead to scope creep, where new features and changes are continuously added. Without careful management, this can result in a project that never reaches completion or exceeds its budget.
4. **Less Documentation:**
   * Agile focuses on working software over comprehensive documentation. This can be a limitation in environments where detailed documentation is required for regulatory compliance or future maintenance.
5. **Integration with Traditional Business Processes:**
   * Agile can be difficult to integrate with traditional business processes that are not as flexible or iterative. This can create friction between Agile teams and other parts of the organization.
6. May roll back or fail back if review stage found out it does not fit in

**Waterfall Limitations**

1. **Inflexibility:**
   * The Waterfall model is rigid and linear, making it difficult to accommodate changes once a phase has been completed. This can be problematic if requirements change or new insights are gained during development.
2. **Late Testing:**
   * Testing is typically done at the end of the development cycle in the Waterfall model. This can lead to the discovery of significant issues late in the project, making them more costly and time-consuming to fix.
3. **Assumes Stable Requirements:**
   * Waterfall assumes that all requirements can be gathered upfront and remain stable throughout the project. In reality, requirements often evolve, and this model is not well-suited to handle such changes.
4. **Lack of User Feedback:**
   * User feedback is typically not integrated until the end of the development process, which can result in a product that does not fully meet user needs or expectations.
5. **Risk of Project Failure:**
   * If any phase of the Waterfall process is flawed, it can have a cascading effect on subsequent phases, increasing the risk of project failure. Early mistakes can be costly and difficult to correct.

**Verifiable Requirements Include:**

* **Clear and Unambiguous:** Easily understood and interpreted the same way by everyone.
* **Measurable:** Can be objectively tested or measured.
* **Completable:** Contains all necessary information.
* **Consistent:** Does not conflict with other requirements.

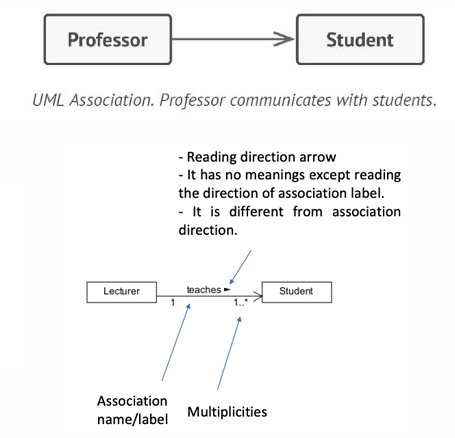
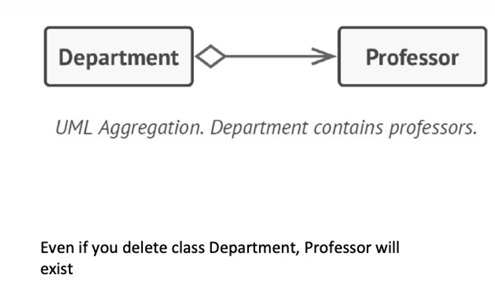
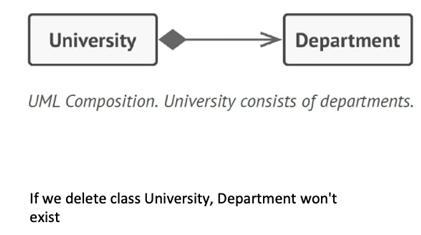
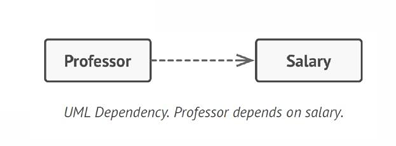
**Non-functional Metrics**

* **Performance:** Speed and responsiveness of the system.
* **Reliability:** Consistency of the system's performance over time.
* **Robustness:** Ability to handle errors and unexpected situations.
* **Portability:** Ease with which the system can be transferred to different environments.
* **Usability:** Ease of use and user-friendliness.

**Hard-to-test Requirements**

* **Example:** Backup and recovery processes.
* **How to Test:**
  + **Backup Testing:** Verify that data can be successfully backed up and restored.
  + **Mock Testing:** Use simulated environments to test recovery procedures.

**Domain Modelling**

* **Includes:** Classes, objects, methods.
* **Relationships Include:**
  + **Association:** A relationship between two classes
  + **Aggregation:** one object merely contains a reference to another. 
  + **Composition has-a:** A strong form of aggregation indicating ownership. 
  + **Inheritance is-a:** A relationship where one class inherits attributes and methods from another class.
  + **Dependency**: weaker variant of association, no permanent link between objects. 

**Prototyping**

* **Types:**
  + **Throwaway Prototyping:** Create a prototype to explore ideas, then discard it.
  + **Evolutionary Prototyping:** Build a prototype and evolve it into the final system through iterative refinement.
* **Benefits:**
  + **Feedback:** Helps gather user feedback early in the development process.
  + **Clarification:** Clarifies requirements and reduces misunderstandings.
  + **Flexibility:** Allows for changes and improvements based on user input.

**PEGS**

* **Definition:** PEGS stands for Project Estimation and Goal Setting.
* **Project Requirements vs. Goals Requirements:**
  + **Project Requirements:** Specific needs and functionalities that must be fulfilled.
  + **Goals Requirements:** High-level objectives that the project aims to achieve.
  + **Example:**
    - **Project Requirement:** The system must support 100 concurrent users.
    - **Goal Requirement:** Increase user satisfaction by 20%.

**Analysis vs. Validation**

* **Analysis:** The process of examining requirements to ensure they are complete, consistent, and feasible.
* **Validation:** The process of ensuring that the requirements meet the needs of the stakeholders and that the system fulfills those requirements.

**UML Class Diagram**

* **Exam Importance:** UML class diagrams are critical in understanding the system's structure and relationships between classes.

**Sequence Diagrams**

* **Why:** Illustrates how objects interact in a particular scenario of a use case.
* **What:** Depicts objects and the messages they exchange arranged in a time sequence.

**State Chart Diagram vs. Sequence Diagram**

* **State Chart Diagram:**
  + Used to model the dynamic behaviour of a single object by showing its states and transitions.
  + Suitable for modelling the lifecycle of an object.
* **Sequence Diagram:**
  + Used to model interactions between objects in a specific scenario.
  + Suitable for modelling the sequence of messages exchanged between objects.

**Low Coupling and High Cohesion**

* **Coupling:** The degree of interdependence between modules. Lower coupling is better as it indicates a well-structured, modular system.
* **Cohesion:** The degree to which the elements within a module belong together. Higher cohesion is better as it indicates a well-defined, focused module.

**Software Architecture**

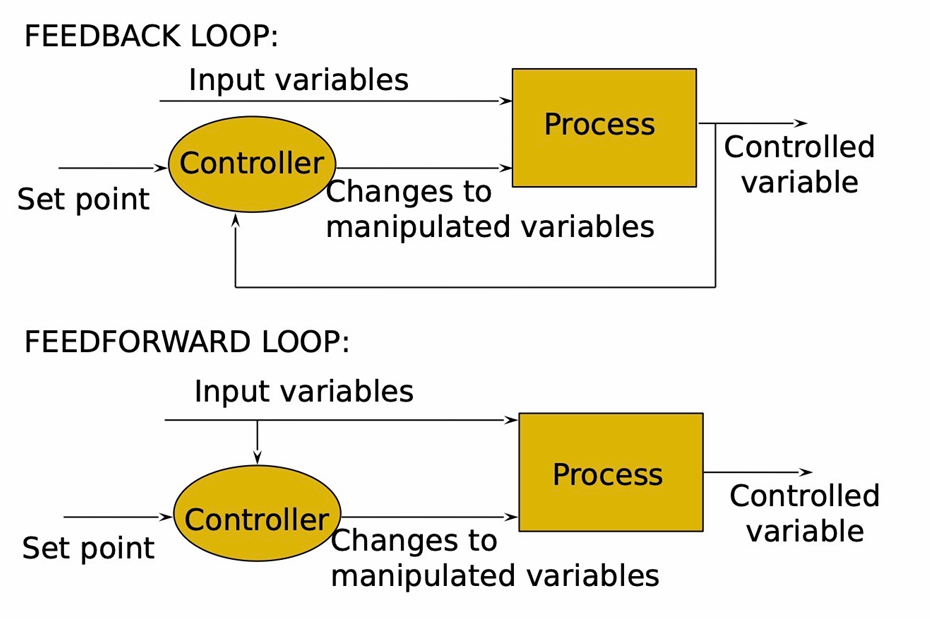
**3 Main Architectures**

1. **Layered Architecture**
   * **Concept:** Separates the system into distinct layers, each with specific responsibilities.
     + **Presentation Layer:** User interface components.
     + **Business Logic Layer:** Core functionality and business rules.
     + **Data Access Layer:** Data storage and retrieval operations.
   * **Advantages:**
     + **Separation of Concerns:** Each layer handles a specific aspect of the application, making the system easier to manage and maintain.
     + **Modularity:** Layers can be developed and updated independently.
     + **Reusability:** Common functionalities (e.g., authentication) can be reused across multiple applications.
   * **Disadvantages:**
     + **Performance Overhead:** Multiple layers can introduce latency.
     + **Rigid Structure:** Strict separation can sometimes lead to inefficiencies if interactions between layers are frequent.
   * **Situations to Use:**
     + When the system complexity requires clear separation of concerns.
     + For enterprise applications with well-defined layers, such as web applications with distinct UI, business logic, and data management.
2. **Client-Server Architecture**
   * **Concept:** Divides the system into two main components: clients (requesters of services) and servers (providers of services).
     + **Client:** User-facing interface.
     + **Server:** Backend processing, data storage, and business logic.
   * **Advantages:**
     + **Centralized Control:** Easier to manage, update, and secure data centrally on the server.
     + **Scalability:** Servers can be scaled independently to handle more clients.
     + **Maintenance:** Backend can be updated without affecting clients.
   * **Disadvantages:**
     + **Single Point of Failure:** If the server fails, clients cannot access services.
     + **Network Dependency:** Performance is dependent on network reliability and bandwidth.
   * **Situations to Use:**
     + For applications requiring centralized data control and processing, like web applications, email services, and database applications.
     + When client devices are diverse and need to interact with a central server.
3. **Service-Oriented Architecture (SOA)**
   * **Concept:** Uses services as the fundamental building blocks, each service being a discrete unit of functionality.
     + **Service:** A self-contained unit that performs a specific business function.
   * **Advantages:**
     + **Loose Coupling:** Services are independent and can be developed, deployed, and updated independently.
     + **Reusability:** Services can be reused across different applications.
     + **Interoperability:** Services can communicate over standard protocols, making it easy to integrate with other systems.
   * **Disadvantages:**
     + **Complexity:** Managing and orchestrating multiple services can be complex.
     + **Performance Overhead:** Network calls between services can introduce latency.
   * **Situations to Use:**
     + For large, complex applications that require high modularity and flexibility.
     + When different parts of the system need to be developed and maintained independently.

**5 Other** **Architectures**

1. **Pipe and Filter Architecture**
   * **Concept:** Uses a series of processing elements (filters) connected by pipes, where data flows from one filter to the next.
   * **Advantages:**
     + **Modularity:** Filters can be developed and tested independently.
     + **Reusability:** Filters can be reused in different contexts.
     + **Flexibility:** Filters can be rearranged or replaced easily.
   * **Disadvantages:**
     + **Performance Overhead:** Data transfer between filters can introduce latency.
     + **Limited Interaction:** Filters typically process data in isolation, which can limit complex interactions.
   * **Situations to Use:**
     + For data processing applications, such as compilers, data transformation pipelines, and image processing systems.
2. **Event-Driven Architecture**
   * **Concept:** Uses events to trigger interactions between loosely coupled components.
     + **Publisher:** Generates events.
     + **Subscriber:** Listens for and processes events.
   * **Advantages:**
     + **Scalability:** Can easily scale by adding more event handlers.
     + **Decoupling:** Components are loosely coupled, facilitating independent development and deployment.
   * **Disadvantages:**
     + **Complexity:** Managing events and ensuring proper sequencing can be complex.
     + **Debugging:** Tracing the flow of events can be challenging.
   * **Situations to Use:**
     + For applications with asynchronous processing needs, such as real-time systems, user interfaces, and event-driven microservices.
3. **Microservices Architecture**
   * **Concept:** Decomposes the system into small, independent services that communicate over a network.
     + **Microservice:** A small, self-contained unit that performs a specific business function.
   * **Advantages:**
     + **Scalability:** Services can be scaled independently.
     + **Flexibility:** Enables continuous deployment and independent updates.
     + **Resilience:** Failures in one service do not affect others.
   * **Disadvantages:**
     + **Complexity:** Managing a large number of services can be complex.
     + **Latency:** Network communication between services can introduce latency.
   * **Situations to Use:**
     + For large, complex applications that require high modularity, flexibility, and scalability.
     + When frequent updates and independent deployment are essential.
4. **MVC (Model-View-Controller)**
   * **Concept:** Separates the application into three interconnected components.
     + **Model:** Manages the data and business logic.
     + **View:** Displays the data and handles user interactions.
     + **Controller:** Acts as an intermediary between Model and View, handling user input and updating Model and View.
   * **Advantages:**
     + **Separation of Concerns:** Each component has a distinct responsibility, making the system easier to manage and maintain.
     + **Reusability:** Models and controllers can be reused across different views.
     + **Testability:** Components can be tested independently.
   * **Disadvantages:**
     + **Complexity:** Managing interactions between components can be complex.
     + **Performance Overhead:** The separation of concerns can introduce performance overhead.
   * **Situations to Use:**
     + For web applications and desktop applications where a clear separation between the user interface and business logic is beneficial.
     + When you need a well-organized codebase that supports independent development and testing of components.
5. **Component-Based Architecture**
   * **Concept:** Builds the system from reusable components, each encapsulating a specific piece of functionality.
     + **Component:** A self-contained unit with a well-defined interface.
   * **Advantages:**
     + **Reusability:** Components can be reused across different systems.
     + **Modularity:** Components can be developed, tested, and deployed independently.
     + **Maintainability:** Easier to update and maintain components.
   * **Disadvantages:**
     + **Complexity:** Managing dependencies and interactions between components can be complex.
     + **Integration:** Integrating components from different sources can be challenging.
   * **Situations to Use:**
     + For large systems that require high modularity and reusability.
     + When different parts of the system need to be developed and maintained independently.

**Summary of Situations to Use Each Architecture**

1. **Layered Architecture:**
   * Use when you need clear separation of concerns and have a complex application with distinct layers like web applications.
2. **Client-Server Architecture:**
   * Use when you need centralized control and processing, such as in web applications, email services, and database applications.
3. **Service-Oriented Architecture (SOA):**
   * Use for large, complex applications requiring high modularity, flexibility, and independent development of services.
4. **Pipe and Filter:**
   * Use for data processing applications that benefit from modular, reusable components, such as compilers and data transformation pipelines.
5. **Event-Driven Architecture:**
   * Use for applications with asynchronous processing needs, such as real-time systems, user interfaces, and event-driven microservices.
6. **Microservices Architecture:**
   * Use for large, complex applications requiring high modularity, flexibility, scalability, and independent deployment.
7. **MVC (Model-View-Controller):**
   * Use for web and desktop applications where separation between user interface and business logic is beneficial, and a well-organized codebase is needed.
8. **Component-Based Architecture:**
   * Use for large systems requiring high modularity, reusability, and independent development and maintenance of components.
9. Process Control Architecture 

**MVC**

* **Definition:** An architectural pattern that separates an application into three main logical components: Model, View, and Controller.
* **How it Works:**
  + **Model:** Manages the data and business logic.
  + **View:** Displays the data and handles user interactions.
  + **Controller:** Acts as an intermediary between Model and View, handling user input and updating the Model and View.
* In a web application, the **Model** might be the database, the **View** is the web page, and the **Controller** handles user inputs like clicking a button or submitting a form.

**User Interface**

* **Definition:** The space where interactions between humans and machines occur.
* **Three Important Components:**
  + **Usability:** Ease of use and learning.
  + **Accessibility:** Usable by people with a wide range of abilities.
  + **Aesthetics:** Visual appeal and layout.

**Hardware Interfaces**

* **Definition:** Interfaces that allow hardware components to communicate with each other or with software.
* **Example:** USB ports, network interfaces.

**Software Interfaces**

* **Definition:** Interfaces that allow software components to communicate with each other.
* **How it Works:** Defines the methods and protocols for software interaction.
* **Example:** APIs (Application Programming Interfaces).

**Top-down vs. Bottom-up**

* **Top-down:**
  + **How it Works:** Starts from the highest level of the system and breaks it down into smaller components.
  + **Common Use:** Used in large systems where high-level design is essential.
  + **Why:** Ensures that the overall system architecture is well-defined.
* **Bottom-up:**
  + **How it Works:** Starts from the lowest level components and integrates them to form higher-level systems.
  + **Common Use:** Used when individual components are well-understood and need to be integrated.
  + **Why:** Allows for the development of independent, reusable components.

**Design Patterns**

* **Why:** Provide reusable solutions to common design problems, improving code readability, maintainability, and flexibility.
* **Kinds of Patterns:**
  + **Creational Patterns:** Deal with object creation mechanisms.
  + **Structural Patterns:** Deal with object composition and structure.
    - **Examples:** Adapter, Facade, Proxy.
  + **Behavioral Patterns:** Deal with object interaction and responsibilities.
    - **Examples:** Observer, Strategy, Command.

**Facade Pattern**

* **Principle:** Provides a unified interface to a set of interfaces in a subsystem, making the subsystem easier to use.
* **Why:** Simplifies interactions with a complex system.
* **Example:** A video conversion system where the Facade pattern simplifies the interaction with multiple components.

**Adapter Pattern**

* **Principle:** Converts the interface of a class into another interface that a client expects.
* **Why:** Allows incompatible interfaces to work together.
* **Example:** Using a third-party library that works with JSON while your system uses XML.

**Proxy Pattern**

* **Principle:** Provides a placeholder for another object to control access to it.
* **Why:** Controls access and reduces the cost of accessing objects.
* **Example:** Loading images where a Proxy Image displays a placeholder until the real image is loaded.

**Observer Pattern**

* **Principle:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
* **Why:** Facilitates the implementation of distributed event handling systems.
* **Example:** A customer wants to be notified when a specific product becomes available in a store.

**Strategy Pattern:**

* **Justification:** Allows the system to define a family of algorithms (e.g., different methods for route calculation or map display) and make them interchangeable. The Strategy pattern promotes flexibility and extensibility by enabling the dynamic selection of algorithms at runtime.
* **Example:** The system could use different strategies for route calculation based on user preferences (e.g., shortest route, fastest route, scenic route). By implementing the Strategy pattern, the Route class can select and execute the appropriate route calculation algorithm without changing its interface. This makes it easier to add new route calculation methods in the future.

**Refactoring**

* **What:** The process of modifying software to improve its internal structure without changing its external behaviour.
* **Why:** Improves design, makes software easier to understand, helps find bugs, and speeds up development.
* **When:** During initial development (Agile) or after the software has been written (to address design decay).
* **How:** Test-refactor-test process.
* **Advantages:**
  + **Improves Design:** Prevents design decay and maintains the integrity of the original design.
  + **Eases Addition of New Features:** A cleaner codebase makes it easier to add new functionality.
  + **Helps Find Bugs:** Cleaning up code makes bugs easier to spot.
  + **Increases Development Speed:** Better design and readability lead to fewer bugs and faster development.