# SOFTWARE ENGINEERING

## UNIT - 3

## TOPIC – 1

# DESIGN PROCESS AND DESIGN QUALITY

## 1. Definition of Design Engineering

Design Engineering is the step where we take **what the software should do** (from the requirements) and figure out **how it will do it**. Think of it like building a house: you know how many rooms you want (requirements), but now you need to plan how to build the house (design). This design becomes the blueprint that guides the entire development.

The main parts of the design include:

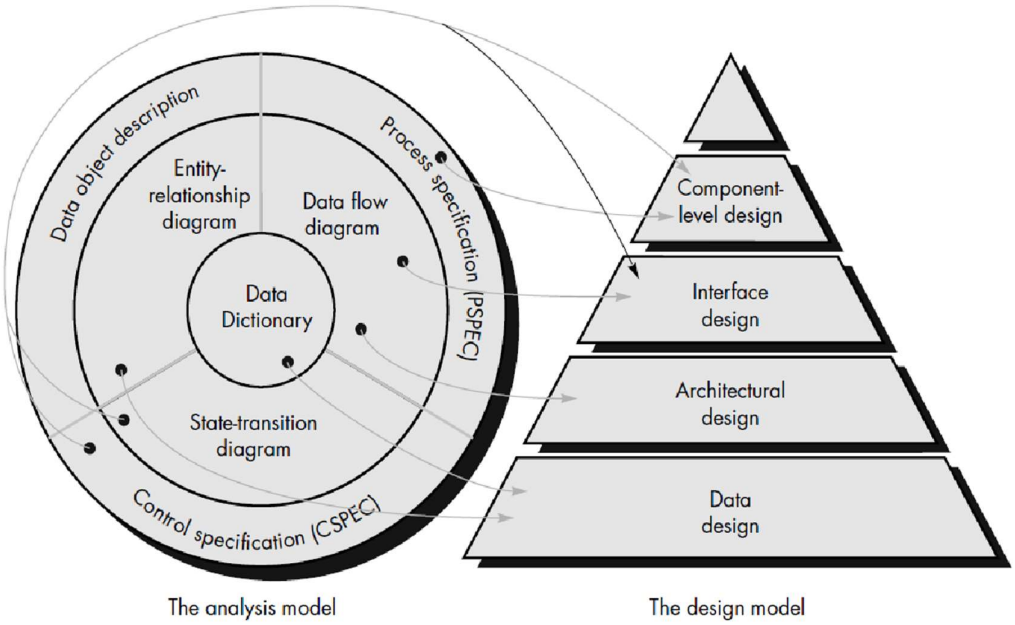
- **Data/Class Design**: Organizing the information the software will use.
- **Architecture**: How the big pieces of the software fit together.
- **Interface Design**: How the software interacts with users or other systems.
- **Component Design**: The smaller parts (or components) of the system that each do a specific job.

## 2. Turning Analysis into Design

We start with the **requirements model** (which describes what the software should do) and turn it into the **design model** (which shows how the software will do it).

This is done in steps:

1. **Data/Class Design**: This step is about defining how data will be structured. Imagine in a school system you need to store data about students, teachers, and courses. This design will tell you how to organize that data into **classes**.




2. **Architecture**: Here, we decide how the **big parts** of the system are connected. For example, how the **student data module** will connect to the **course management module**.
3. **Interface Design**: This step is about planning how the software will communicate with the outside world. It defines how users will interact with the system and how different parts of the software talk to each other.
4. **Component-Level Design**: Finally, we look at the smaller building blocks of the software. Each **component** (like the login feature or registration feature) is designed in detail.

## 3. The Design Process

The design process doesn't happen in one go; it's done in steps and refined over time. With each iteration, the design becomes more detailed and clearer.

Throughout this process, the design is checked for **quality** to ensure:

- The design meets **all the requirements**.
- It is **understandable** so that others can build, test, and maintain it easily.
- The design provides a complete picture of how the software will work.

## 4. Quality Attributes (FURPS)

To ensure the design results in a high-quality product, we follow the **FURPS** model, which stands for:

- **Functionality:** Evaluates if the system has all the required features and works as expected.
    - **Example**: An online banking system should allow users to transfer money, check their balance, and pay bills securely.
- **Usability:** Measures how easy it is for users to interact with the system.
    - **Example**: A weather app that clearly shows the forecast and is easy to navigate.
- **Reliability:** Assesses if the system performs consistently over time without errors.
    - **Example**: A payment gateway should process transactions without failure.
- **Performance:** Refers to how quickly the system operates.
    - **Example**: An e-commerce website should load pages quickly and handle a large number of users during sales.
- **Supportability:** Refers to how easy it is to maintain, upgrade, and test the system.
    - **Example**: A mobile app that can easily be updated to add new features

These attributes guide the design process to ensure the final software product is robust and meets user expectations.

## 5. Design Guidelines for Good Software

To make sure the design is good, we follow these rules:

1. **Familiar Structure**: Use well-known design patterns or styles and create parts that are designed well and can grow over time.

2. **Modular Design**: Split the system into smaller, manageable parts (modules) that can work independently.
3. **Clear Representation**: The design should clearly show how data, architecture, interfaces, and components are organized.
4. **Good Class Structure**: Make sure the design leads to proper class structures that will be used in the software.
5. **Independent Parts**: Each part should be able to function on its own without needing other parts too much.
6. **Simple Connections**: Keep the connections between parts of the system simple, especially how the system interacts with the outside world.
7. **Based on Requirements**: The design should come from the information gathered during the requirements phase.
8. **Clear Communication**: Use simple diagrams or explanations to show how the design works.

# SOFTWARE ENGINEERING

## UNIT - 3

## TOPIC – 2

# DESIGN CONCEPTS AND DESIGN MODEL

## 1. Key Design Concepts

Several important ideas are used in the design process. These concepts help us plan and build software better:

### a. Abstraction

Abstraction means **simplifying** complex things by breaking them into smaller parts. For example, instead of focusing on how a car's engine works, you can just think of the car as something that gets you from place A to B.

There are three types:

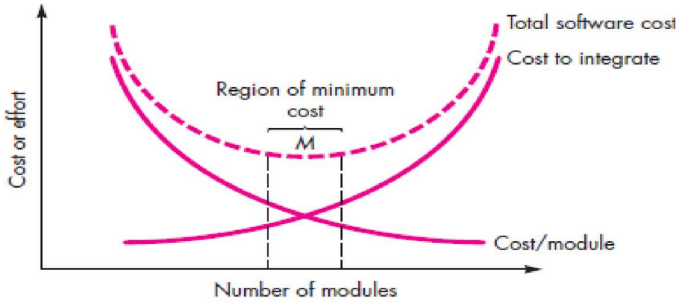
- **Procedural Abstraction**: Describes a process or task without getting into details.
- **Data Abstraction**: Groups related data together, like having all the details of a "Student" in one place.
- **Control Abstraction**: Simplifies the control flow (like making it easier to manage how different parts of a program interact).

### b. Refinement

**Refinement** is about starting with a big idea and **breaking it into smaller, more detailed steps**. For example, instead of just saying "manage student records," we can break that down into "add a student," "update student details," and "delete a student."

### c. Modularity

**Modularity** means splitting the software into small, independent parts called **modules**. Each module handles one task. For example, in a school system, there could be a module for managing students and another module for managing courses.




### d. Architecture

The **architecture** is like the **blueprint** of the software. It shows how the **big parts** of the software (like different modules) will work together and how they will communicate. A good architecture makes the software easier to build and maintain.

### e. Information Hiding

This concept is about **keeping details hidden** inside each module, only showing what's necessary. For example, a module that handles student grades might keep its internal calculations hidden and only give out the final grades. This reduces the risk of errors spreading through the system when changes are made.

### f. Control Hierarchy

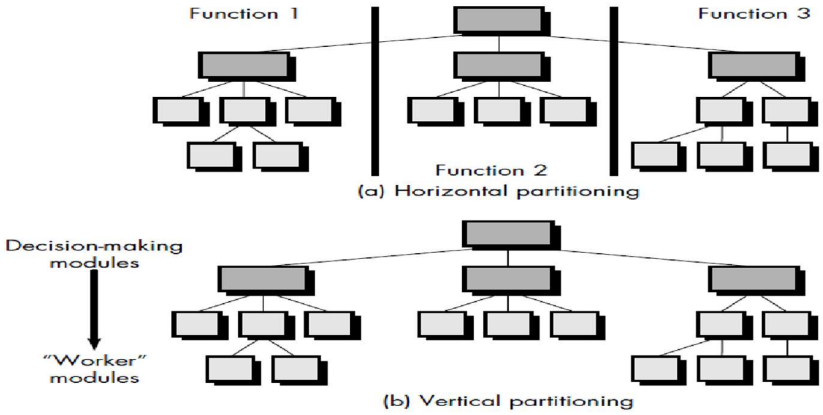
This defines how **control flows** through the system. Some parts of the system control others. For example, in a school system, the **admin module** might control the **student registration module**. There's a hierarchy, just like how a manager controls employees in a company.

### g. Structural Partitioning

We divide the system into smaller parts, based on the way the system works:

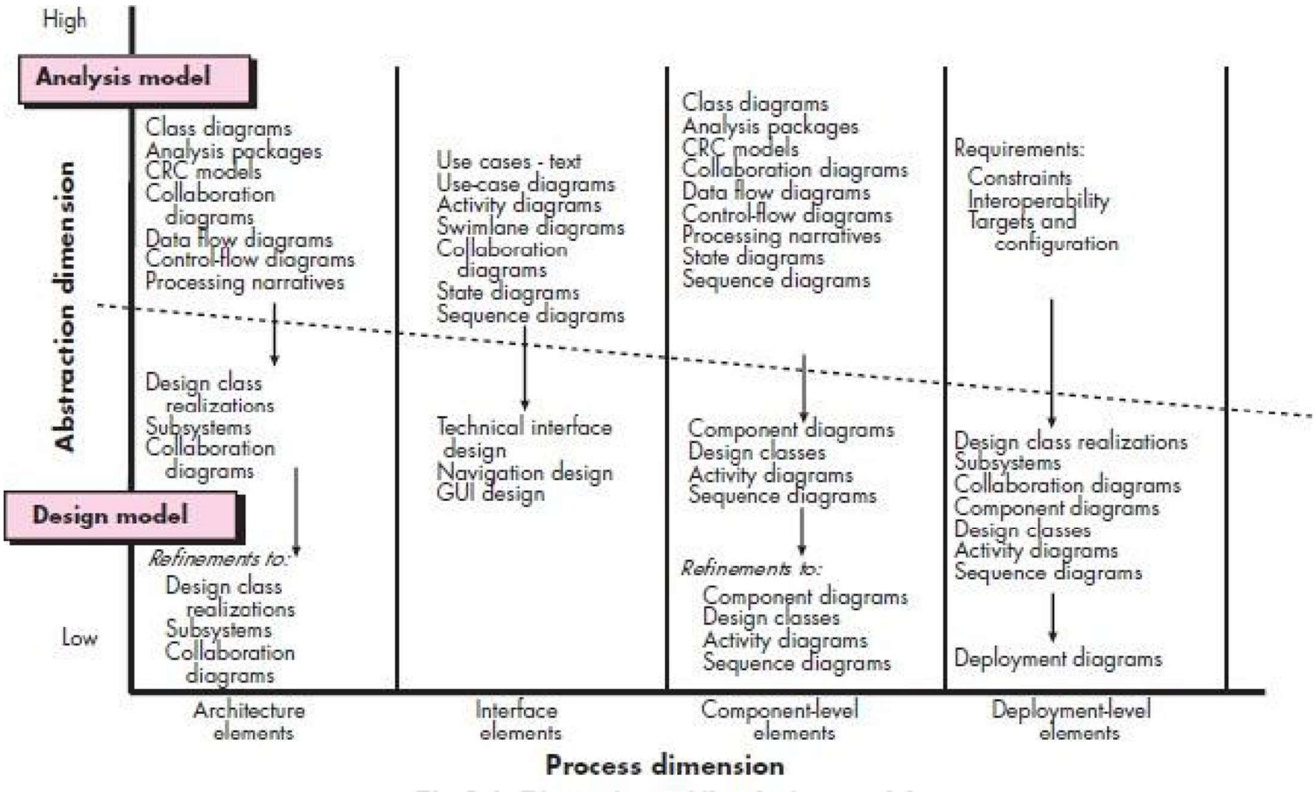
- **Horizontal Partitioning**: Divides the system into layers, like input, processing, and output.

- **Vertical Partitioning**: Organizes the system so that control flows from the top (higher-level decisions) down to the bottom (detailed tasks).




## 2. The Design Model

The design model is a detailed plan of how the software will be built, and it evolves over time. The design model uses **UML diagrams** to represent different aspects of the system.



Here are the main parts of the design model:

**Data Design Elements**

This part of the design focuses on how **data** will be organized and structured in the system. For example, a school system would need to store student data, teacher data, and course data.

**Architectural Design Elements**

These describe the **big structure** of the software. Think of it as a **floor plan** of a house showing how different parts are connected. It gives an overall view of how the software will function.

**Interface Design Elements**

This part describes how the software will interact with:

- **Users**: through a user interface (UI) like buttons, menus, and forms.
- **Other systems**: through external connections like APIs.
- **Internal components**: showing how different parts of the software will communicate with each other.

**Component-Level Design Elements**

This describes the **small parts** of the software in detail. For example, a "Login" component might handle checking usernames and passwords. Each component works independently but connects to others to form the full system.

**Deployment-Level Design Elements**

This shows how the software will be **set up in the real world**. It describes where each part of the software will run (for example, on a server or a user's computer).

## 7. UML Diagrams

**UML** diagrams are visual tools to help explain the design. Common types include:

- **Class Diagram**: Shows the structure of the system in terms of classes (e.g., a "Student" class with data like name and grades).

- **Use Case Diagram**: Shows how users will interact with the system (e.g., a student registering for a course).
- **Sequence Diagram**: Shows the sequence of steps between system parts (e.g., when a student logs in).
- **Activity Diagram**: Looks like a flowchart and shows steps in a process (e.g., registering a student).
- **State Diagram**: Shows different states an object can be in (e.g., a student registration could be "pending" or "approved").
- **Component Diagram**: Shows how the system is divided into parts that do specific jobs.
- **Deployment Diagram**: Shows how the software will be set up, like what parts will run on a server.

# SOFTWARE ENGINEERING

## UNIT - 3

## TOPIC – 3

# ARCHITECTURAL DESIGN

**Architectural design** in software engineering is the process of creating a **plan** or **blueprint** for how a software system will work. This plan defines the structure of the system, how the different parts of the software will interact, and how to make sure the system can be easily updated, scaled (grow), and maintained. It's similar to designing a house before building it, ensuring all parts like rooms, walls, and utilities work together.

## Purpose of Architectural Design

The purpose of architectural design is to ensure that:

1. **The system is well-organized** and all parts work smoothly together.
2. **It is easy to maintain** and update when needed.
3. **It can handle growth**—more users, more data, or additional features.
4. **Everyone involved** (developers, managers, clients) understands how the system is supposed to work.

## Key Elements of Architectural Design

### 1. Software Architecture

Software architecture is the **big-picture structure** of the system. It shows how the main parts of the software will be organized and how they will interact. The goal here is to make sure that all the components are in the right place and can work together effectively.

**Example**: In a **food delivery app**:

- **User Interface**: Where users browse and order food.
- **Payment System**: Where users pay for their orders.

- **Delivery Tracking**: Where users track their delivery status.

These components must connect and work together to provide a smooth user experience.

**Purpose**: To define the overall structure and ensure that all parts work together seamlessly.

## 2. Architectural Styles

Architectural styles are **different ways to structure** a software system. Each style has its own way of organizing the system's components to achieve the best performance and meet specific needs.

### Types of Architectural Styles with Examples:

- **Layered Architecture**: The system is divided into layers, with each layer handling a specific task.

  **Example**: In a **banking app**:

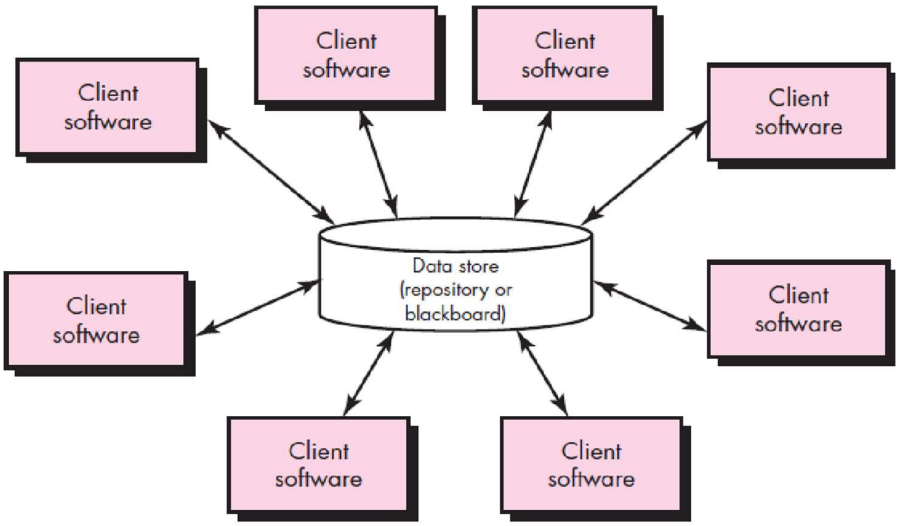
  - The **User Interface** shows account balances.
  - The **Business Logic** layer processes payments.
  - The **Data Layer** stores transaction data.

  **Purpose**: To organize tasks in layers so they can work separately but still communicate with each other.

- **Data-Centered Architecture**: The system revolves around a central **database**.

  **Example**: In a **library system**, all book and user information is stored in a central database.

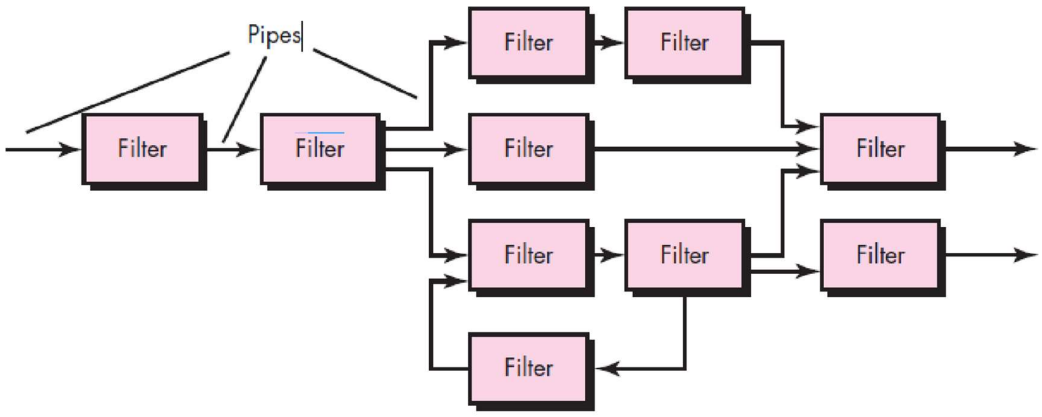
  **Purpose**: To centralize data storage and make sure all components use the same source of truth.




- **Data-Flow Architecture**: Data flows through different components and gets processed along the way.

  **Example**: In a **photo editing app**, an image passes through filters like brightness and contrast adjustments before being saved.

  **Purpose**: To process data step-by-step, allowing easy control of how data changes.

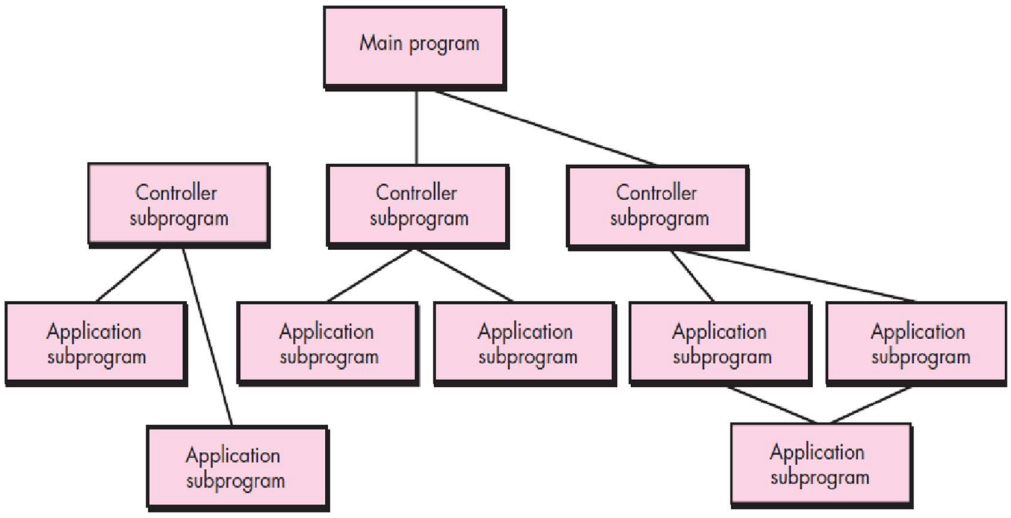



Pipes and filters

- **Call and Return Architecture**: Components call each other to perform actions.

  **Example**: In a **calculator app**, when the user clicks the "add" button, the system calls the addition function to provide a result.

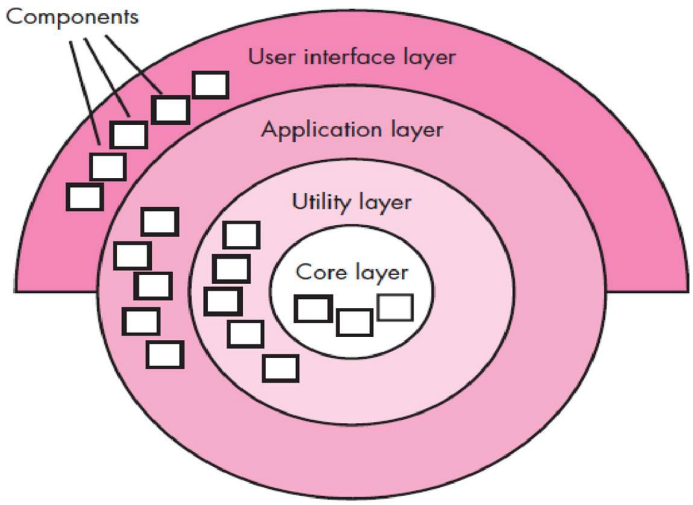
  **Purpose**: To allow components to communicate and perform tasks when requested.




- **Object-Oriented Architecture**: The system is based on **objects** that represent real-world things.

  **Example**: In a **car rental system**, there are objects like **Car**, **Customer**, and **Rental Agreement**, each with its own data and functions.

  **Purpose**: To model real-world objects in the system, making it easier to understand and manage.




## 3. Architectural Patterns

Architectural patterns are **solutions to common problems** in software design. These patterns provide proven ways to structure a system effectively.

**Types of Architectural Patterns with Examples**:

- **Model-View-Controller (MVC):** This pattern splits the system into three parts:
  - **Model**: Manages the data.
  - **View**: Displays the data.
  - **Controller**: Handles user input and updates the model and view.

  **Example**: In an **online store**:

  - The **Model** holds the product information.
  - The **View** shows the products to the users.
  - The **Controller** manages adding/removing items from the cart.

  **Purpose**: To separate the system into distinct parts, making it easier to maintain and update.

- **Microservices**: The system is divided into **small, independent services**, each handling a specific function. **Example**: In an **e-commerce site**, one service handles payments, another manages product inventory, and another manages user accounts.

  **Purpose**: To allow each part of the system to function independently, making it easier to scale and update specific parts.

- **Monolithic Architecture**: The system is built as a **single large unit** where all parts are tightly connected.

  **Example**: A **basic blog platform** where the user interface, content management, and database are all part of one system.

  **Purpose**: To build a simple system where everything is bundled together.

- **Event-Driven Architecture**: The system responds to **events** in real-time, like user actions or changes in the system.

  **Example**: In a **social media app**, a notification is sent when a new message is received.

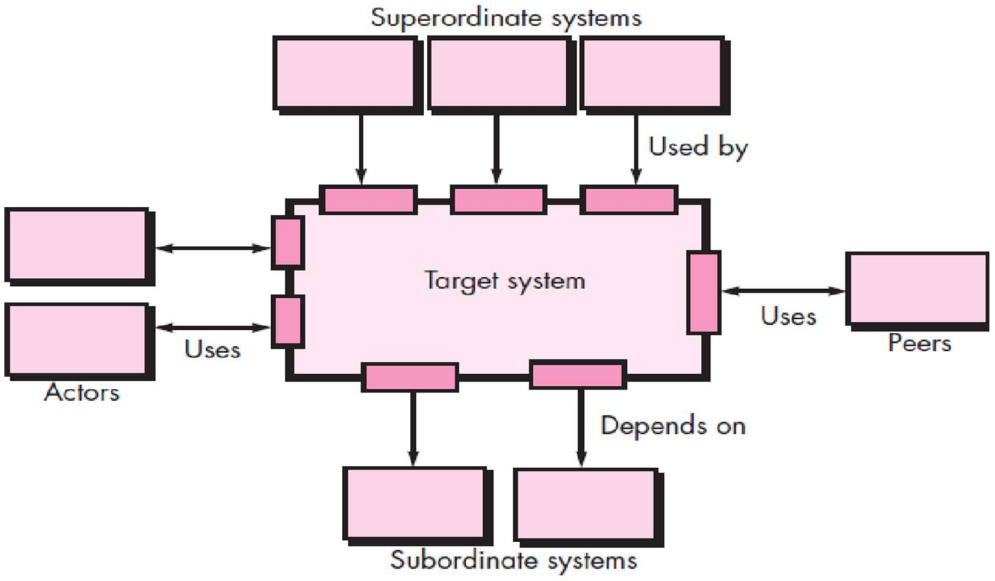
  **Purpose**: To handle real-time updates or changes in the system efficiently.

## Difference between architectural styles and architectural patterns:

| Feature | Architectural Styles | Architectural Patterns |
|---|---|---|
| **Definition** | General ways to organize and structure a system. | Specific, reusable solutions to common design problems. |
| **Purpose** | Provides an overall structure or layout for the system. | Solves specific design challenges within a system. |
| **Scope** | Broad and high-level, guiding the system's architecture. | Focused on solving particular issues or patterns of use. |
| **Example** | Layered architecture (dividing into layers like UI, logic, and database). | MVC (Model-View-Controller), separating data, display, and interaction. |
| **Analogy** | Like choosing a layout for a house (e.g., 2 floors). | Like using a specific building technique for a house. |
| **When to Use** | When deciding the overall organization of the system. | When solving a known problem in system design. |

## Context Models

A **context model** shows how a software system interacts with the **outside world**, like users, other systems, or devices. It is a simple diagram that illustrates what the system talks to and what information flows in and out.



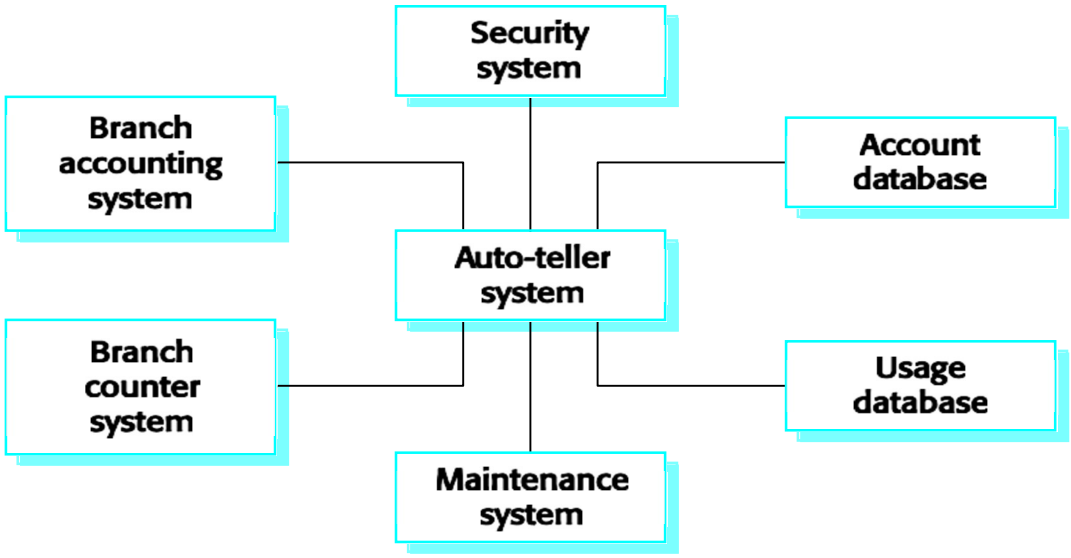
**Example**: In an **ATM system**:

- **User**: The person withdrawing or depositing money.
- **Bank Database**: Where the ATM checks account balances.
- **Card Reader**: The part that reads the user's card.
- **Cash Dispenser**: The part that gives the user money after a withdrawal.

**Purpose of a Context Model**

The purpose of a context model is to:

1. **Clarify how the system fits into a larger environment**.
2. **Show how the system interacts** with users, other systems, or devices.
3. **Identify the flow of information** between the system and external entities.




## Conclusion

The **purpose of architectural design** is to create a clear plan for how the software system will be built. It helps ensure that the system is well-organized, easy to maintain, and can handle growth. By using the right **architectural styles** and **patterns**, software engineers can build systems that are efficient, scalable, and easy to manage.

# SOFTWARE ENGINEERING

## UNIT - 3

## TOPIC – 4

# CONCEPTUAL MODEL OF UML

## UML (Unified Modeling Language)

UML, or Unified Modeling Language, is a way to draw diagrams that show how software systems work. It's like drawing a map or blueprint to help people understand what's going on inside a program. Just like how an architect uses blueprints to plan a house, software designers use UML to plan their software before they start building it with code.

## CASE Tools for UML

**CASE tools** (Computer-Aided Software Engineering tools) are special programs that help people draw these UML diagrams. They make it easy to show how the software will work. Some popular CASE tools are:

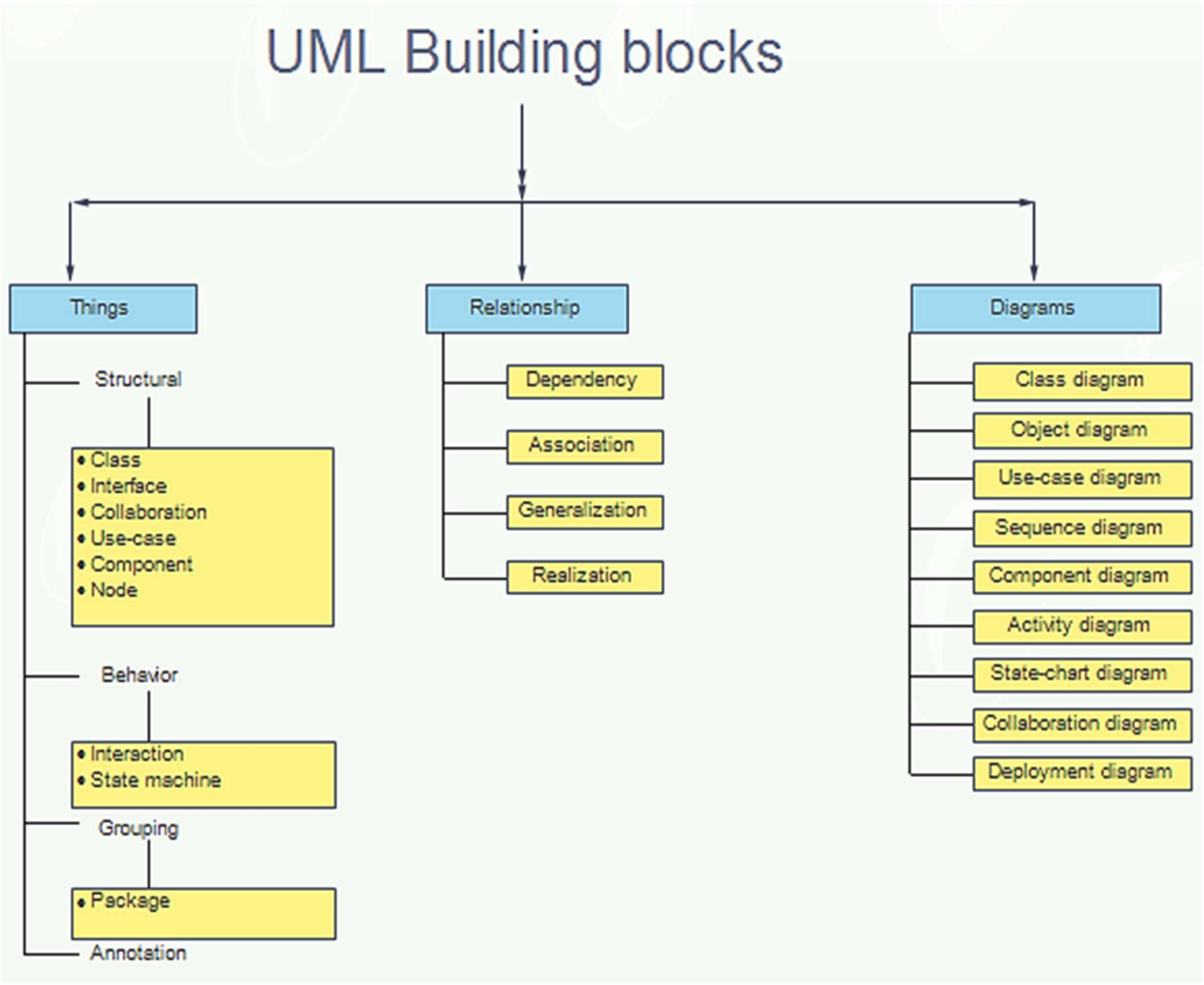
- **Rational Rose** – A well-known tool for drawing UML diagrams.
- **Enterprise Architect** – Helps create and organize diagrams.
- **Microsoft Visio** – A diagramming tool that can also make UML diagrams.
- **Lucidchart** – A web tool for drawing diagrams.
- **StarUML** and **ArgoUML** – Free tools for making UML diagrams.
- **Visual Paradigm** – Offers many features to help design software.
- **Balsamiq**, **Creately**, and **SmartDraw** – Simple tools to make diagrams quickly.
- **Umbrella** – Another tool that helps make UML diagrams.

## UML Building Blocks

UML helps us think about how a software system works by using diagrams.

It focuses on three key points:

1. **Things/Objects**: The parts of the system, like people, data, or tasks.
2. **Relationships**: How these parts connect and work together.
3. **Diagrams**: Pictures that show us how everything fits together.



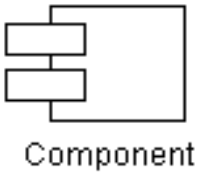
## Parts of UML (Things)

In UML, the "things" are the parts that make up a system. These parts are divided into groups:

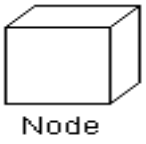
1. **Structural Things (Parts That Don't Change)** These describe the solid, unchanging parts of the system, like the structure of a building.

   - **Class**: A group of objects with similar features.
     - *Example: A "Car" class has properties like make, model, and color.*
     - *Symbol:* 

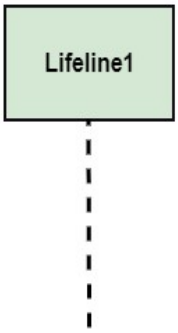
   - **Interface**: A list of rules or actions that certain objects must follow.
     - *Example: A "Drivable" interface makes sure that any class that uses it, like a Car, has a "drive" function.*
     - *Symbol:* 

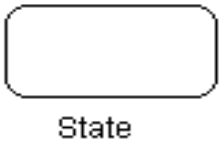
   - **Component**: A piece of the system that performs a specific job.
     - *Example: A "Payment" component handles all the payments in the system.*
     - *Symbol:* 

   - **Node**: A place where the software runs, like a server.
     - *Example: A "Database Server" is a node where customer information is stored.*

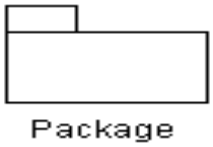
- *Symbol:* 


2. **Behavioral Things (How Things Act or Change)** These describe how the system behaves or changes over time, like how a door can open and close.

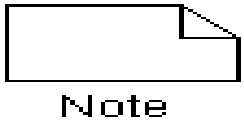
    - **Interaction**: Shows how different parts of the system talk to each other.
        - *Example: A customer asks for information from the system, and the system sends back the details.*
        - *Symbol:* 

    - **State Machine**: Shows the different states something can be in.
        - *Example: A door can be in either "Open" or "Closed" state.*
        - *Symbol:* 


3. **Grouping Things** These organize similar parts of the system into groups.

    - **Package**: Groups together related parts.
        - *Example: A "User Management" package might contain parts that deal with users, like login, registration, and permissions.*
        - *Symbol:* 

4. **Annotational Things** These are notes or comments that give extra information about the system.

- **Annotation**: A comment added to explain something in the diagram.
  - *Example: A note that explains why a particular feature is needed.*
  - *Symbol:* 


## Relationships in UML

UML diagrams also show how the different parts of a system are connected. These connections are called **relationships**.

Some common types of relationships are:

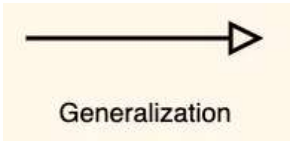
- **Dependency**: When one thing needs another thing to work.
  - *Example: A "Car" class needs an "Engine" class to function properly.*
  - *Symbol:* 

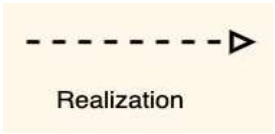
- **Association**: A simple connection between two things.
  - *Example: A teacher teaches a student, and the student learns from the teacher. They are associated with each other.*
  - *Symbol:*

*An "Order" depends on a "Product" to exist because an order is placed for products.*

- 

*A "Doctor" can treat many "Patients," and a "Patient" can see many different "Doctors." Both have a connection (doctor-patient relationship), but they can exist separately.*

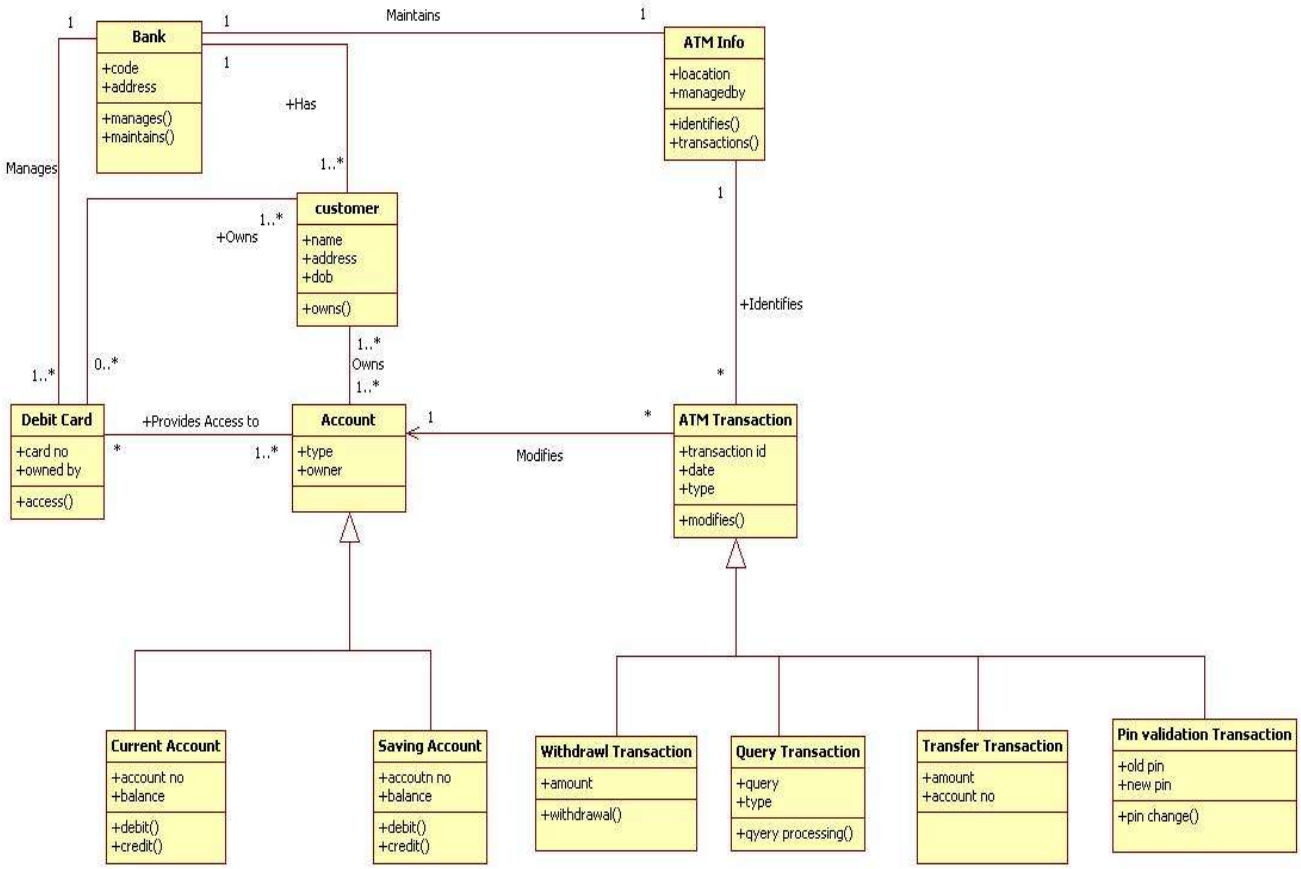
- **Generalization**: This shows inheritance, where one thing is a more general version of another.
    - *Example: A "Car" is a general category, and "SUV" and "Sedan" are more specific types of cars.*
    - *Symbol:* 

- **Realization**: When one thing follows the rules set by another (usually an interface).
    - *Example: A "Bird" class follows the "Flyable" interface by creating a "fly" function.*
    - *Symbol:* 

# Types of UML Diagrams

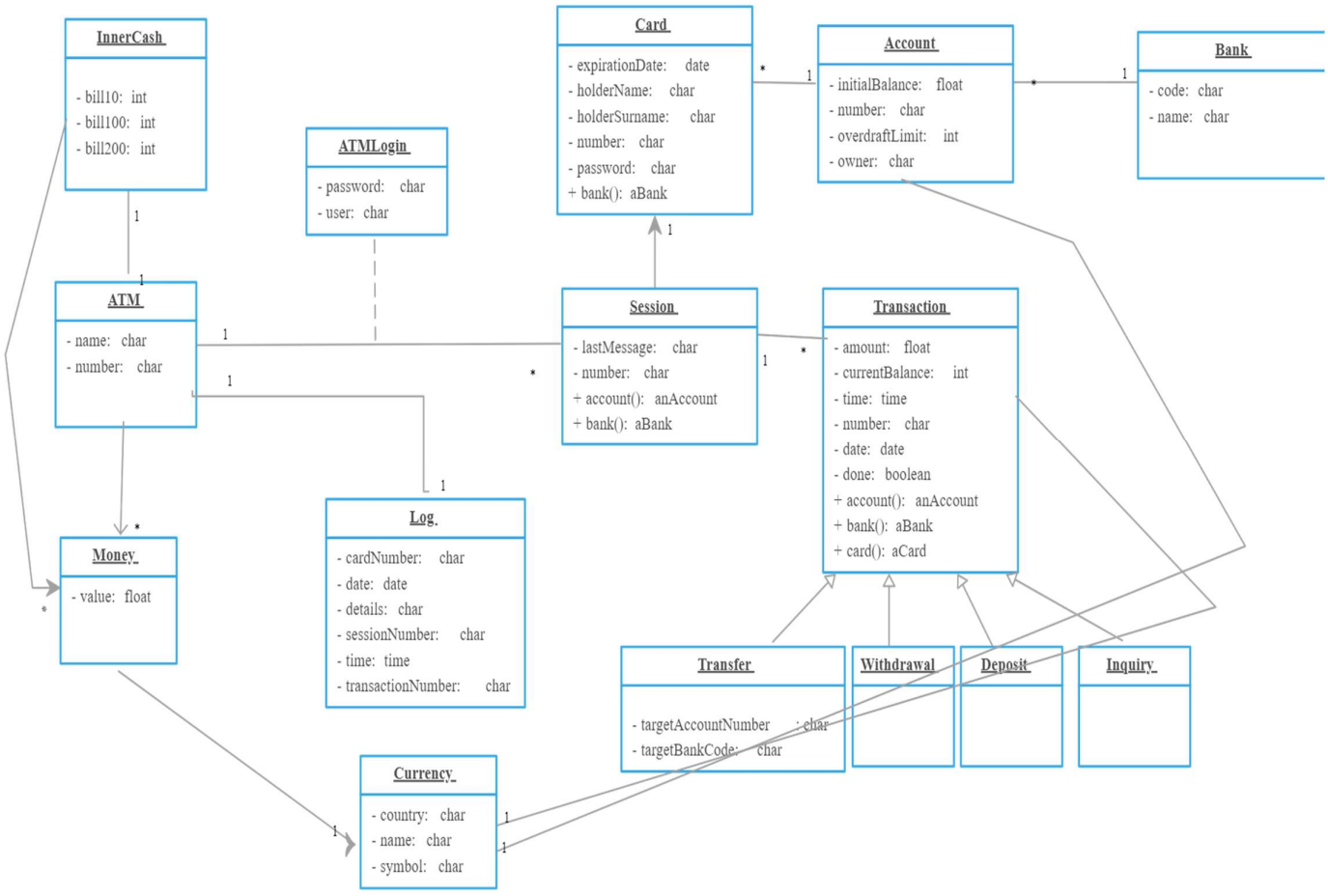
UML uses different types of diagrams to show various aspects of a system. Each diagram has its own way of explaining how the system works.

1. **Class Diagram**
   - Shows the different classes (types of objects) and how they relate to each other.
   - *Example: A "Bank Account" class connected to a "Card" class that handles money transactions.*
   - ***Example Class Diagram for an ATM system***



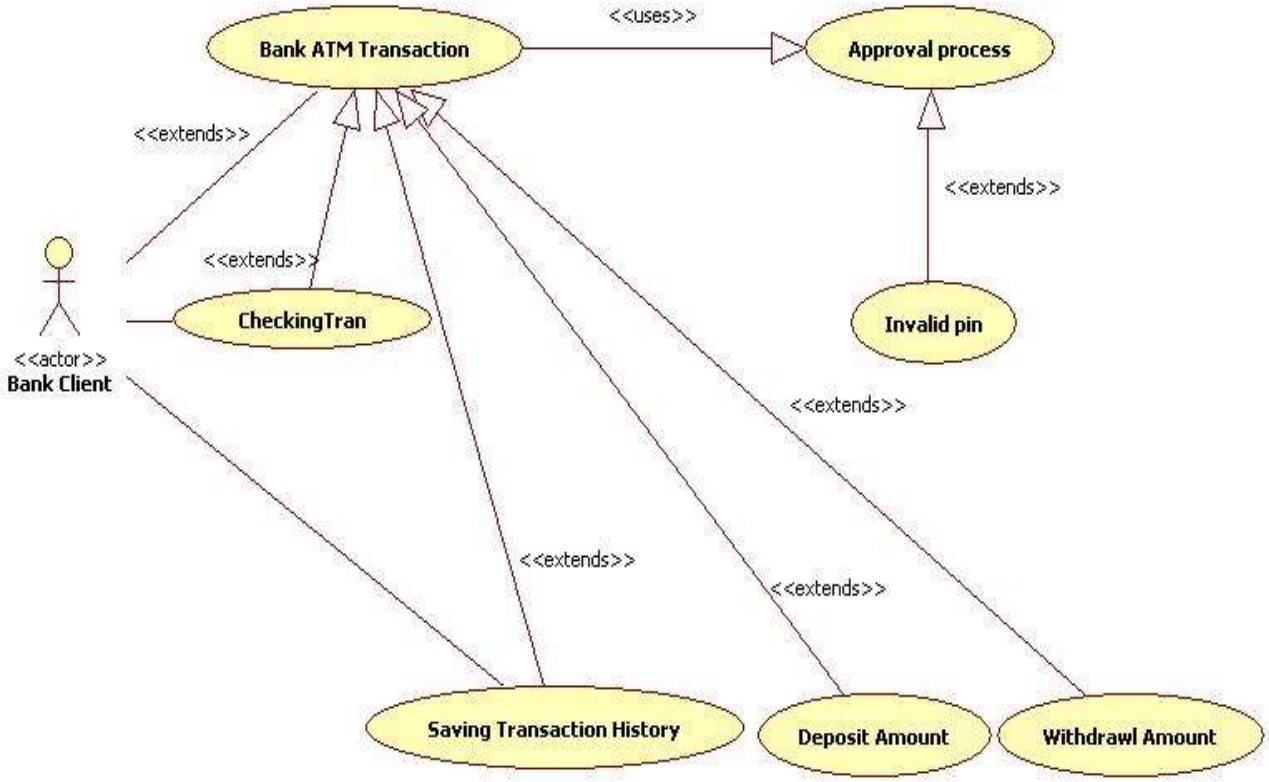
## 2. Object Diagram

- A snapshot of a specific instance in the system at a certain moment.
- *Example: A "Debit Card" showing details for one specific user, like card number and expiration date.*
- ***Example Object diagram for the ATM system***



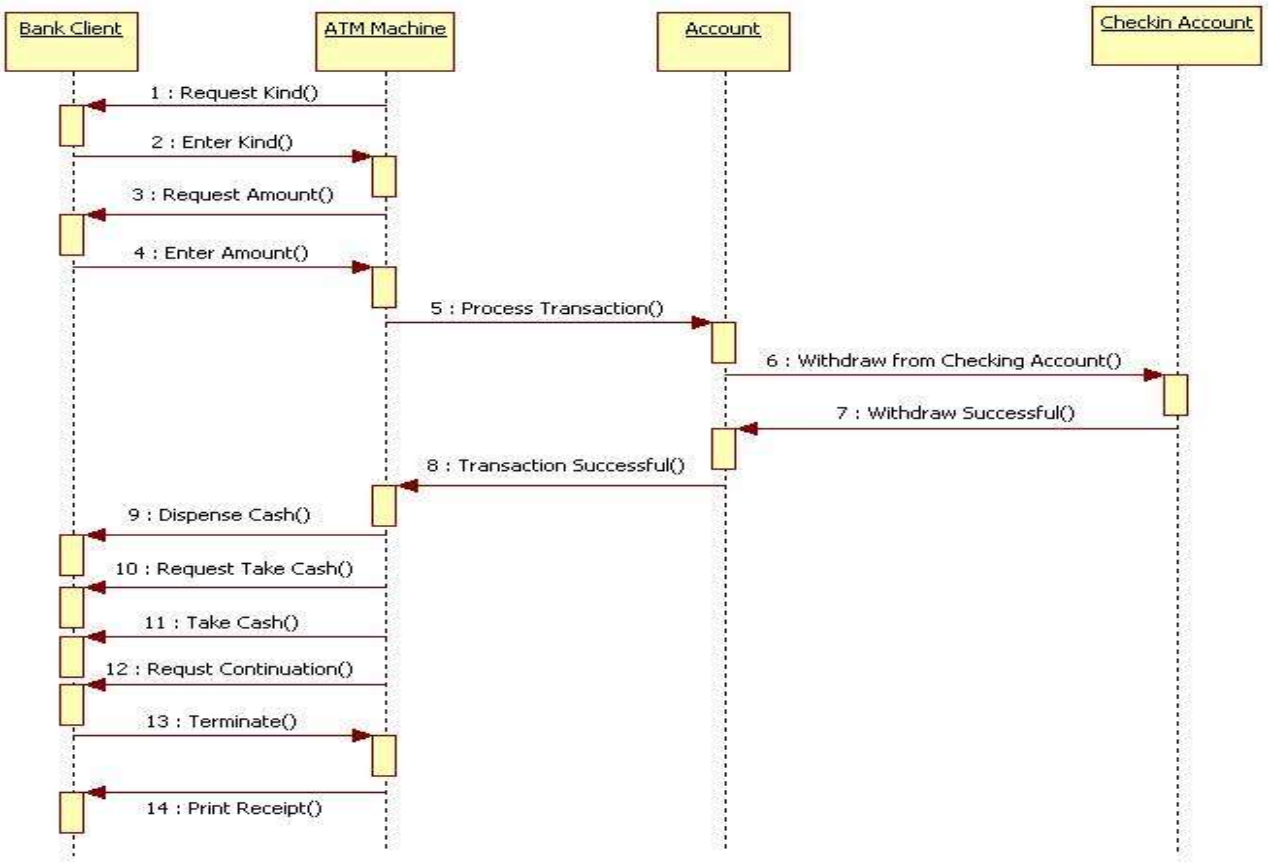
### 3. Use-Case Diagram

- Describes how users interact with the system.
- Example: A user withdrawing money from an ATM or checking their balance.
- ***Example Usecase diagram for the ATM system***



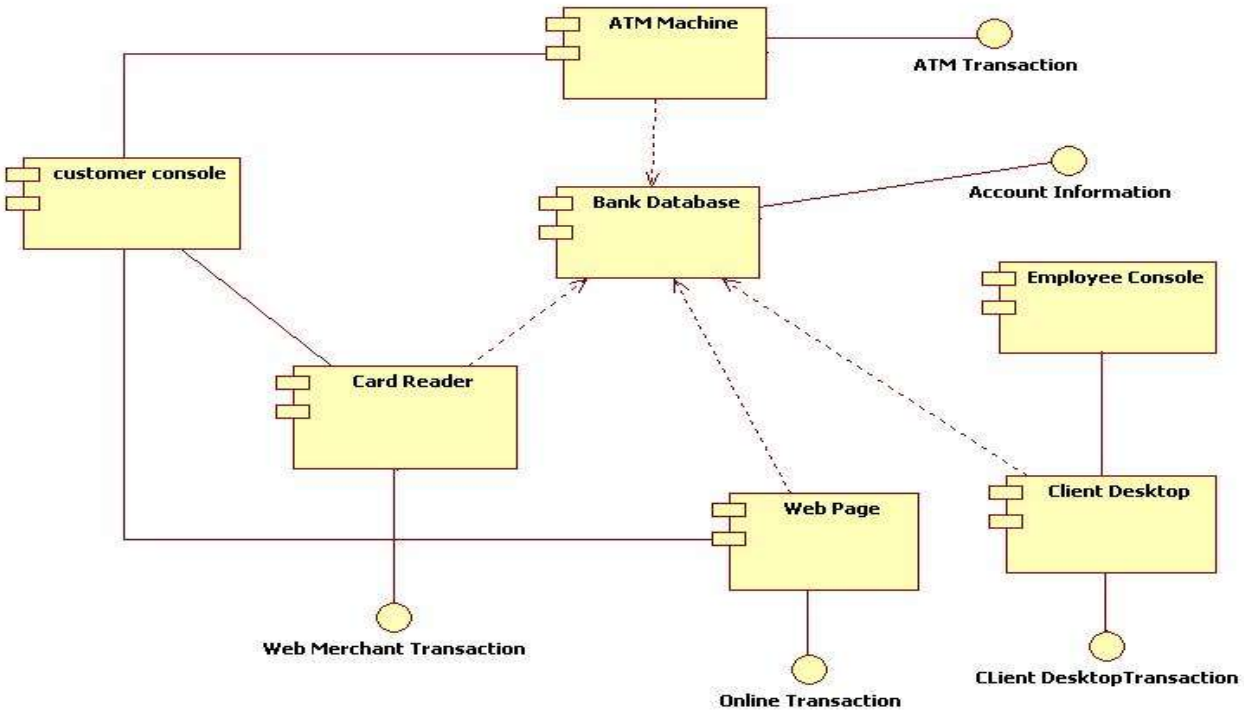
## 4. Sequence Diagram

- Shows the order in which things happen in a system.
- Example: A user requests cash, the ATM asks the bank, and money is dispensed.
- ***Example Sequence diagram for the ATM system***



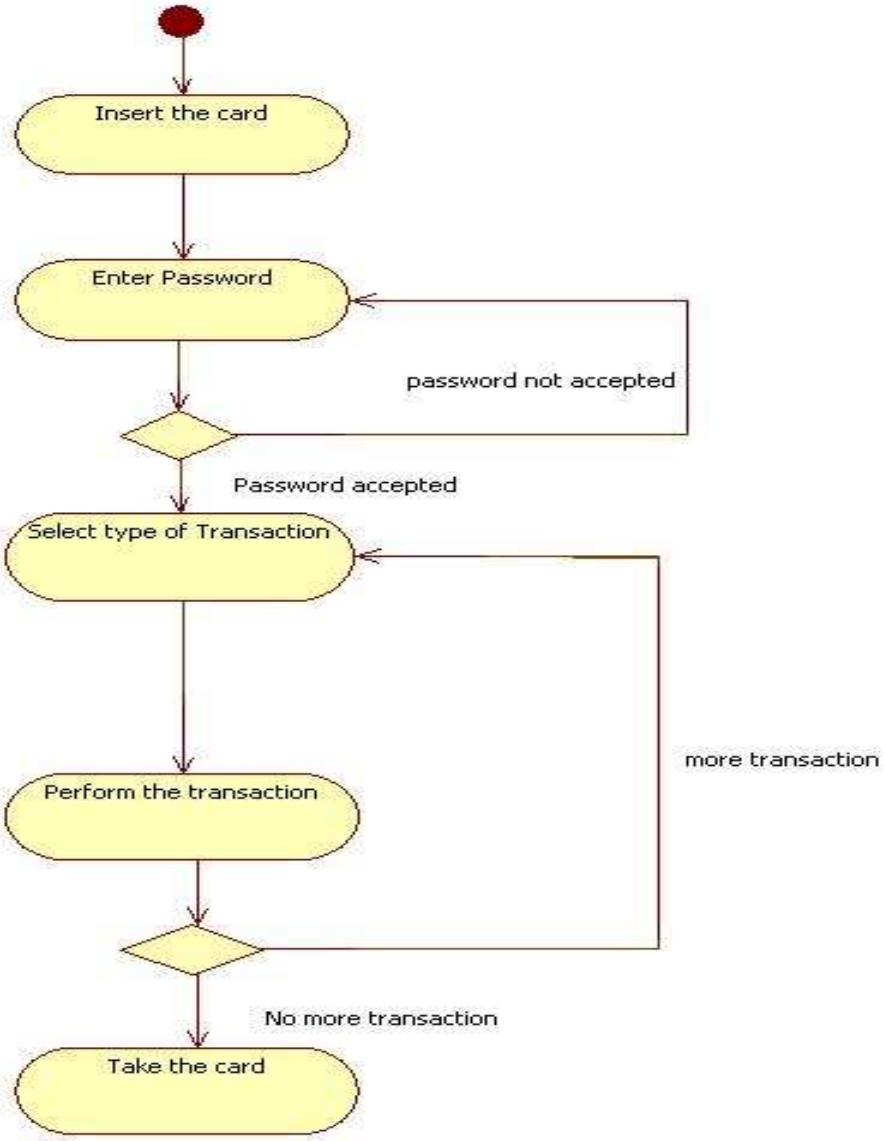
## 5. Component Diagram

- Shows how the different parts (components) of the system fit together.
- Example: An ATM machine's software connects to the bank's database to process a transaction.
- ***Example Component diagram for the ATM system***



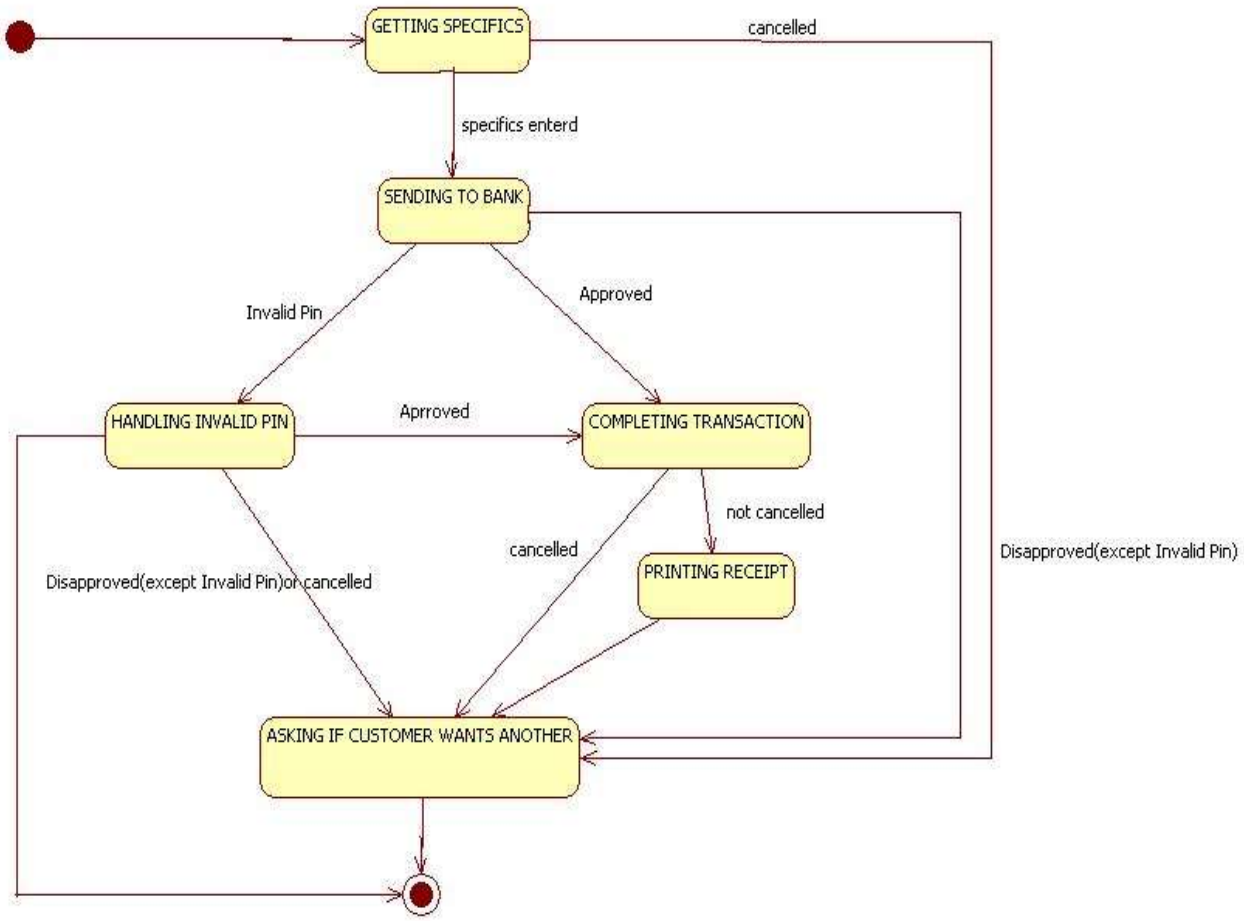
## 6. Activity Diagram

- Shows the steps in a process.
- Example: The process of withdrawing money from an ATM: insert card, enter PIN, select amount, and get cash.
- ***Example Activity diagram for the ATM system***



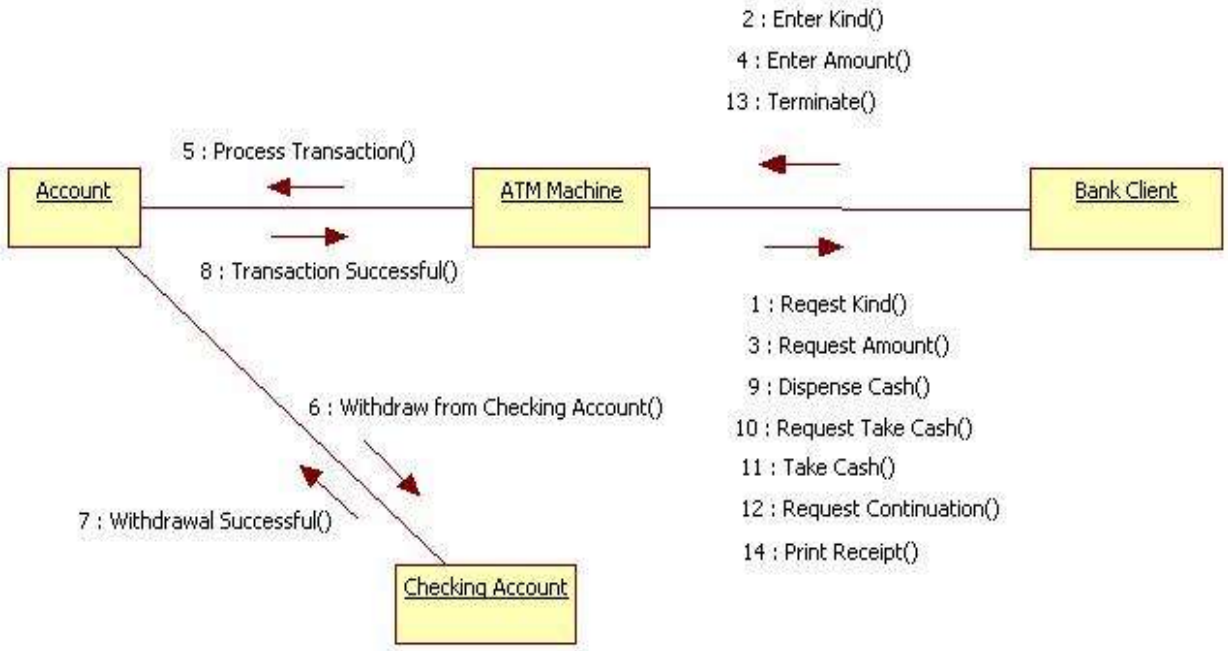
## 7. State-Chart Diagram

- Describes the different states an object can be in and how it changes between them.
- Example: An ATM machine can be "Idle" when not in use and "Dispensing Cash" when a user withdraws money.
- ***Example State-chart diagram for the ATM system***



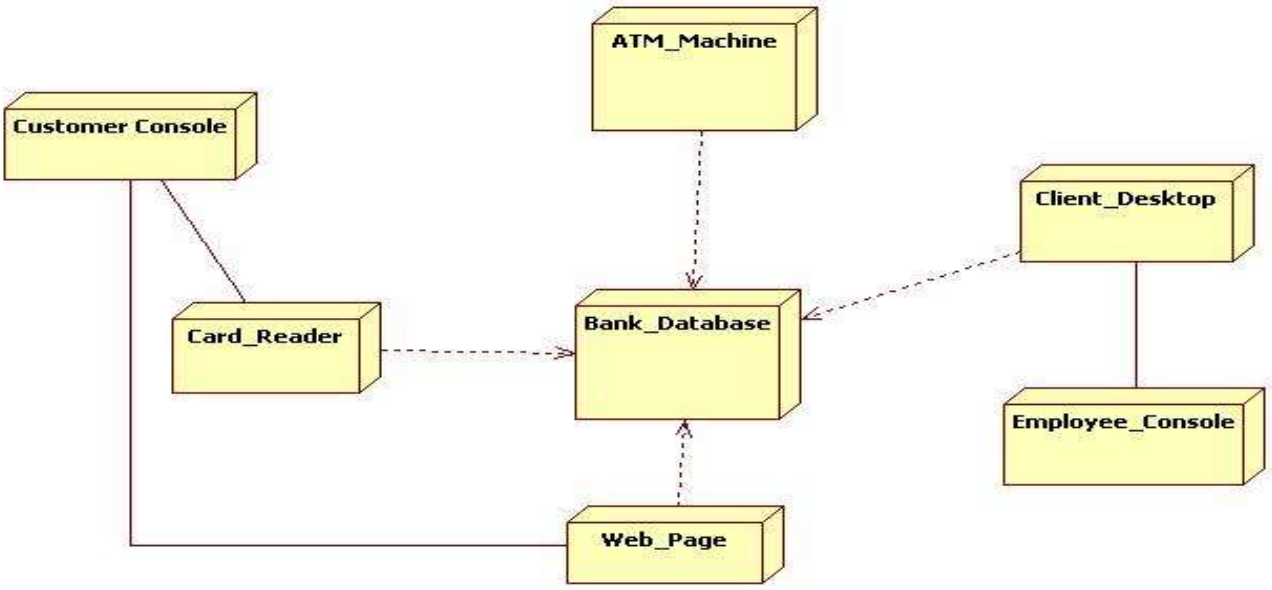
## 8. Collaboration Diagram

- Focuses on how different parts of the system work together to achieve something.
- Example: How the ATM, the user, and the bank's server work together during a cash withdrawal.
- ***Example Collaboration diagram for the ATM system***



9. **Deployment Diagram**

- Shows where the physical parts of the system are located and how they are connected.
- Example: An ATM machine at a bank connects to the bank's server, which might be in a different city.
- ***Example Deployment diagram for the ATM system***




**UML makes it easier for people working on software to understand how everything fits together. By using diagrams to show how the system will work, teams can plan, communicate, and organize their ideas before they start building the actual software. UML is a simple but powerful way to visually map out a complex system, making it much clearer for everyone involved.**

# SOFTWARE ENGINEERING

## UNIT – 3

## TOPIC – 5

# UML DIAGRAMS – USECASE

## Use Case Diagrams

A Use Case Diagram is a type of diagram that helps us understand how a software system works from the user's point of view. It shows the people (called actors) or other systems that interact with the software, and the specific actions they can perform. This diagram makes it easy to see what the system can do and how it works.

## Purpose of a Use Case Diagram

The main reason for using a Use Case Diagram is to show the functions of a system in a simple way. This diagram is very useful when you need to understand the overall picture of what the system does, without diving into too many details. It does three main things:

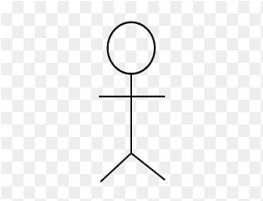
1. **Shows the Actors**: It identifies who is using the system (like a customer or another system).
2. **Describes the Actions**: It explains what actions or tasks the user can do with the system (like withdrawing money from an ATM).
3. **Defines the System's Boundaries**: It clearly shows what is **inside the system** (what the system does) and what is **outside the system** (the users or other systems).

## Key Parts of a Use Case Diagram

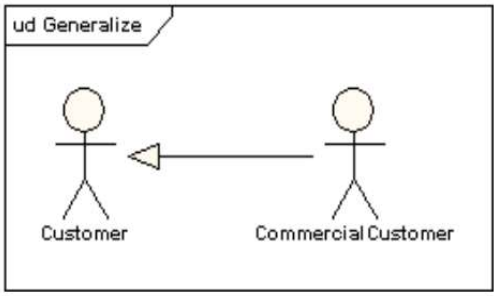
There are several important parts (symbols) in a Use Case Diagram. Each one plays a specific role in explaining how the system works.

**1. Actor:** An actor is anyone or anything that interacts with the system. This could be a person, another software system, or even a piece of hardware.

- **Example**: In an ATM system, the actor could be the "Customer" using the ATM or the "Bank" that processes the transactions.
- **Symbol**: It is represented by a **stick figure**. This makes it easy to recognize as someone or something interacting with the system.



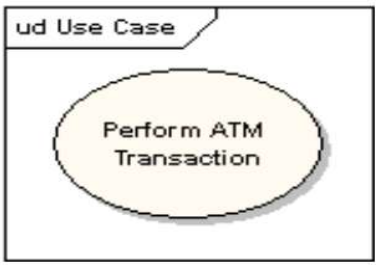
- **Generalization of Actors:** Actors can have a hierarchical relationship. One actor can "generalize" another, meaning one type of actor shares similarities with another but may also have additional roles.

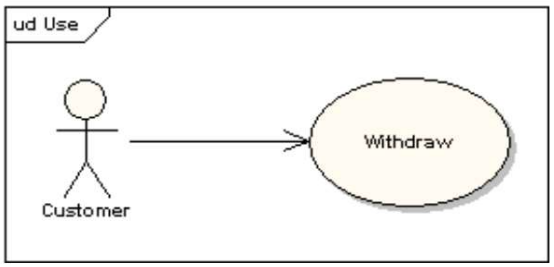



- In the image, we see a Customer Actor and a Commercial Customer actor. A Commercial Customer is a type of customer, but with more specific responsibilities. The arrow between them shows this generalization, meaning all commercial customers are customers, but not all customers are commercial customers.

## 2. Use Case

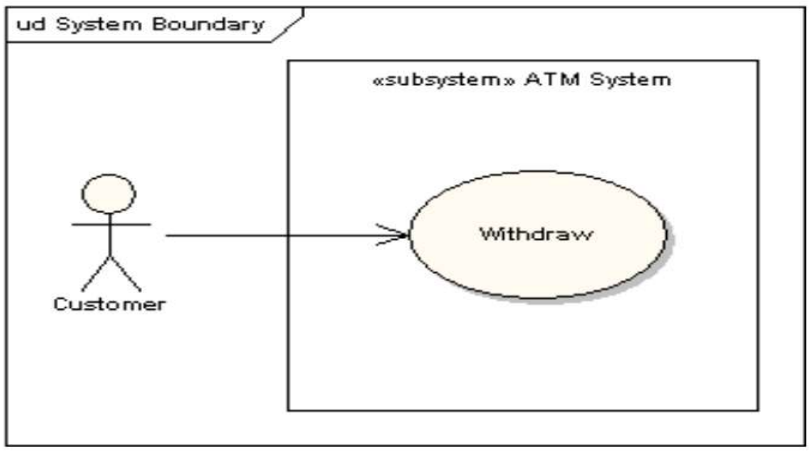
- **Definition**: A use case is an action or task that the actor can do with the system. It is like a function or activity the system provides to the user.
- **Example**: For an ATM system, use cases could include tasks like "Withdraw Cash," "Check Balance," or "Transfer Funds."
- **Symbol**: A use case is shown as an **oval** with the name of the action written inside.



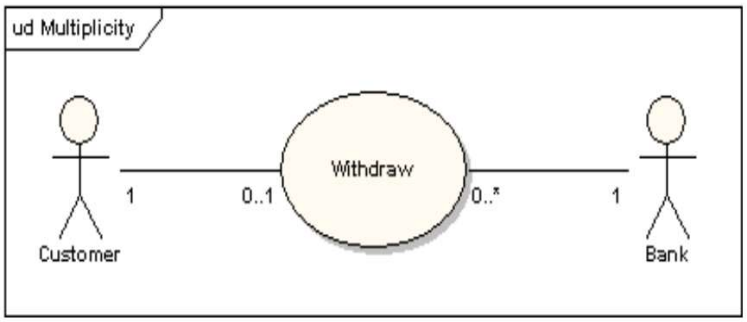




**3. System Boundary**

- **Definition**: The system boundary is a box or rectangle that surrounds the use cases. It helps to show what is part of the system and what is not. Everything inside the boundary is controlled by the system, and everything outside interacts with it.
- **Symbol**: The boundary is represented by a **rectangle** that encloses all the use cases.

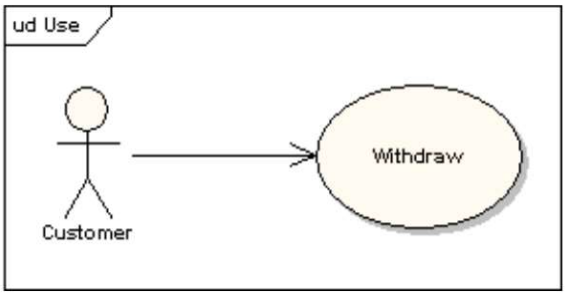



**4. Associations (Lines):** An Association is a straight line that shows a connection between an actor (a person or system) and a use case (a task or action). It means the actor is involved in the task, but it doesn't say who starts the action.



*Example:* A line between a Customer and a Login task means the customer is part of the login process, but it doesn't tell us who begins the login.

**Directed Association:** A Directed Association is like an association, but it has an arrow. The arrow shows who starts or controls the action. It points to the task, meaning the actor is the one who starts the action.




*Example:* If there's an arrow from a Customer to a Withdraw task, it means the customer starts the process of withdrawing money. Without the arrow, it just shows involvement, but we don't know who initiates the task.

**Multiplicity:** Multiplicity tells us how many times an actor can be involved in a task. Numbers next to the line show this.

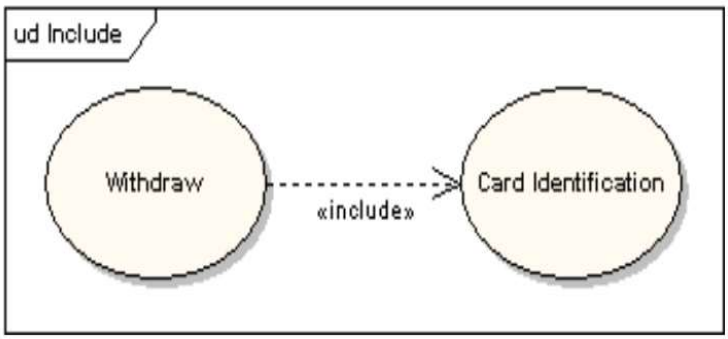
*Example:* If a Customer is linked to a Purchase task with multiplicity 0..1, it means they can either make one purchase or none at all. If it's 1..*, it means the customer can make as many purchases as they want.

Use multiplicity when you need to explain how many times a person or system can do something. For example, a customer might log in only once, but they might make many purchases.

## Special Relationships in a Use Case Diagram

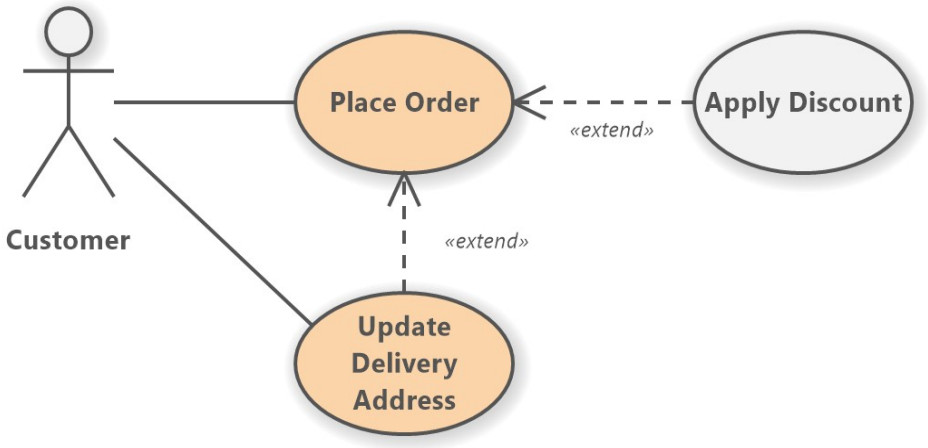
Sometimes, use cases can be related to each other in special ways. These relationships help make the diagram clearer.

**1. Include:** This means that one use case **must always include** another use case to complete its task. It is a way of saying that a certain action is part of a larger action.

- **Example**: In an ATM system, before a customer can "Withdraw Cash," they must first "Authenticate" themselves (by entering a PIN). So, "Authenticate User" is always included in "Withdraw Cash."
- **Symbol**: This relationship is shown with a **dashed arrow** labelled <<include>>.
- 


**2. Extend:** This means that one use case can **add extra steps** or **features** to another use case, but only if needed. It is like an optional action that only happens under certain conditions.

- **Example**: After withdrawing cash, the customer might want to check their balance. This means the "Check Balance" use case extends the "Withdraw Cash" use case.
- **Symbol**: This is shown with a **dashed arrow** labelled <<extend>>.



## Steps to Create a Use Case Diagram for an ATM System

Use Case Diagram for an ATM system.

### Step 1: Identify the Actors

- **Customer:** This is the person using the ATM to do tasks like withdrawing money or checking their account balance.
- **Bank:** This represents the system or organization that processes the customer's transactions.

### Step 2: Identify the Use Cases

- **Withdraw Cash:** The customer takes money out of their account using the ATM.
- **Check Balance:** The customer checks how much money is in their account.
- **Deposit Cash:** The customer puts money into their account.
- **Transfer Funds:** The customer moves money from one account to another.
- **Change PIN:** The customer changes the PIN number used to access the ATM.

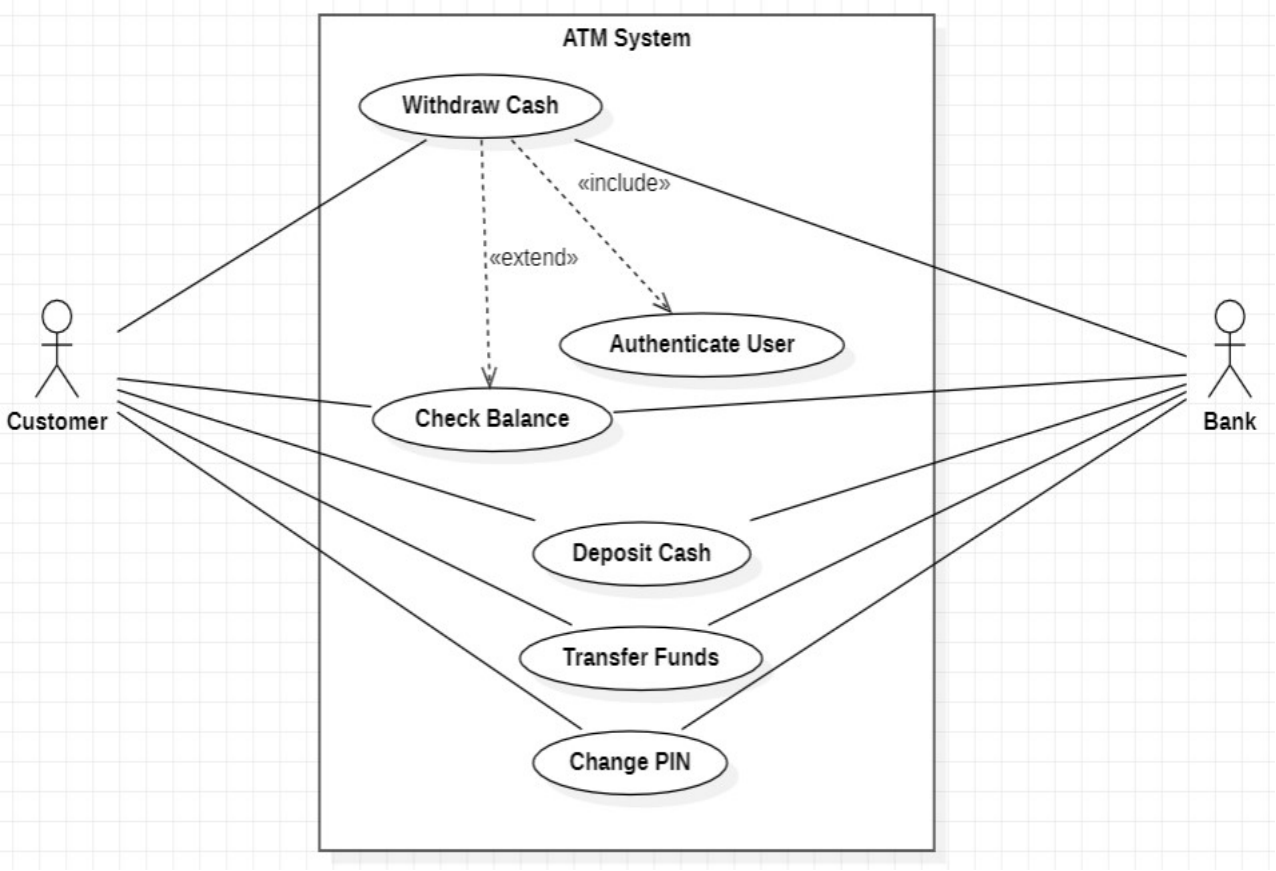
### Step 3: Draw the Diagram

- First, draw a large rectangle to represent the system boundary and label it "ATM System."
- Inside the rectangle, draw ovals for each use case (Withdraw Cash, Check Balance, etc.).
- Outside the rectangle, draw stick figures for the actors (Customer, Bank).
- Connect the actors to the use cases using straight lines to show what actions the actors can perform.

### Step 4: Use Include and Extend

- **Include:** The "Withdraw Cash" use case always includes "Authenticate User," since the customer must be authenticated before they can withdraw money. Draw a dashed arrow from "Withdraw Cash" to "Authenticate User" and label it <<include>>.

- **Extend:** The "Check Balance" use case extends "Withdraw Cash" because, after withdrawing cash, the customer may want to check their balance. Draw a dashed arrow from "Check Balance" to "Withdraw Cash" and label it <<extend>>.



# SOFTWARE ENGINEERING

## UNIT – 3

## TOPIC – 6

# UML DIAGRAMS – CLASS

## Class Diagram

A Class Diagram is a key building block in object-oriented modeling. It represents the static structure of a system by describing:

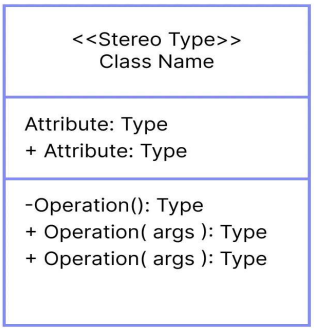
- **Classes:** The blueprint for objects.
- **Attributes:** Characteristics of the classes.
- **Methods:** The behaviors or actions associated with the class.
- **Relationships:** How the classes interact with each other.

Class diagrams are crucial in software development because they help engineers and stakeholders understand how the different components of a system fit together.

## Components of a Class Diagram:

1. **Classes:**

- A class in UML is a blueprint for objects. It is depicted as a rectangle divided into three sections:
  - **Top section:** Contains the name of the class.
  - **Middle section**: Lists the attributes (properties or data) of the class.
  - **Bottom section:** Shows the methods (functions or behaviors) that the class can perform.



### 2. Attributes:

- Attributes are variables that represent the state or properties of a class.
- They describe what an object of the class knows or possesses.
- Examples:
  - *A Car class might have attributes like color, model, and speed.*
  - *A Person class might have attributes like name, age, and height.*
- Attributes can be:
  - **Private (-):** Only accessible within the class.
  - **Public (+):** Accessible from outside the class.
  - **Protected (#):** Accessible by the class and its subclasses.

### 3. Methods:

- Methods represent the functions or behaviors that objects of the class can perform.
- They describe what actions an object of the class can take.
- Examples:
  - *A Car class might have methods like drive() and stop().*
  - *A Person class might have methods like walk(), talk(), and sleep().*

### 4. Relationships:

- The relationships between classes define how different classes are connected and interact with each other. UML supports various types of relationships, including:

**Association:**

- Represents a general connection between two classes. It implies that objects of one class interact with objects of another class.
- *Example: A Teacher and Student. The association shows that teachers teach students, and students are taught by teachers*.
- Shown by a solid line connecting the two classes.

**Inheritance (Generalization):**

- Also known as Generalization, this relationship indicates that one class inherits attributes and methods from another class.
- *Example: Dog is a subclass of Animal, meaning that all dogs inherit properties like legs and eyes from the Animal class, but dogs also have additional properties like barking.*
- Shown by a solid line with a hollow arrow pointing toward the parent class.


**5. Aggregation:**

- Represents a whole-part relationship where the part can exist independently of the whole.
- *Example: A classroom can exist without its students. Even if a student leaves, the classroom still exists.*
- Shown by a solid line with a hollow diamond near the class that represents the whole.

**6. Composition:**

- A stronger form of aggregation where the part cannot exist independently of the whole.
- *Example: A house and its rooms. If the house is destroyed, the rooms no longer exist.*
- Shown by a solid line with a filled diamond near the class that represents the whole.

**7. Dependency:**

- A weaker relationship where one class depends on another to perform its function.
- *Example: A Printer depends on a PrintJob. The printer class depends on the presence of a print job to function.*
- Shown by a dashed arrow from the dependent class to the class it depends on.

**8. Realization:**

- Represents that a class implements an interface or a contract.
- *Example: If a class Car implements an interface Vehicle, it must define all methods declared in the Vehicle interface, such as start() and stop().*
- Shown by a dashed line with a hollow arrow pointing towards the interface.

## Drawing a Class Diagram for an ATM System

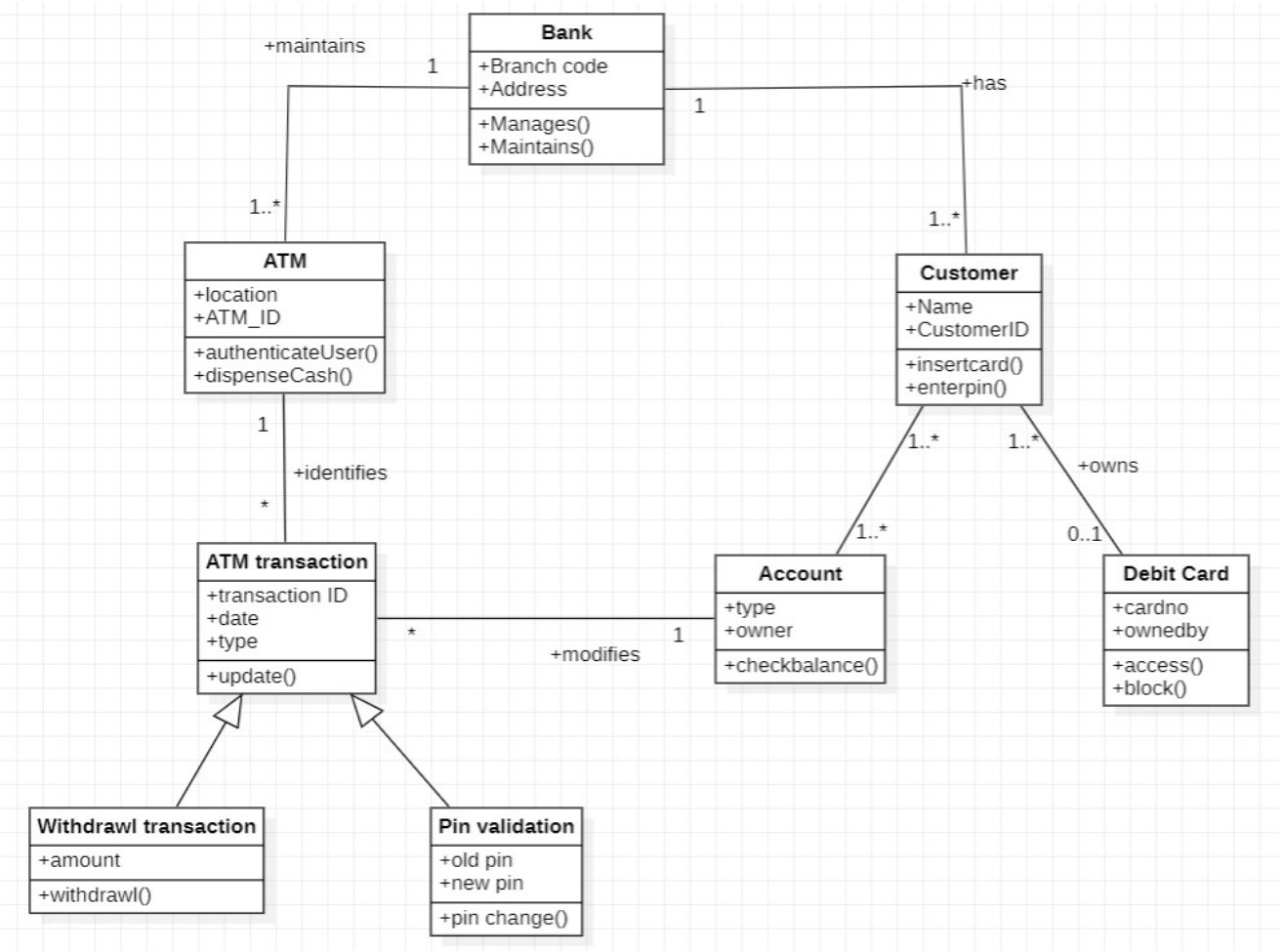
To illustrate how class diagrams are used, consider an ATM system. When modeling such a system, you break it down into classes, define their attributes and methods, and specify how they interact with each other.

**Steps to Create an ATM Class Diagram:**

1. Identify the Main Classes:
   - Think about the key components of the ATM system. These will form the classes.
   - *Examples of classes:*
     - **ATM:** The machine itself.
     - **Customer:** The person using the ATM.
     - **Bank Account:** The account that holds the customer's money.
     - **Transaction:** Actions like withdrawal, deposit, etc.
2. Define Attributes for Each Class:
   - List the important properties each class should have.
   - *Examples:*
     - **ATM:** location, ATM_ID.
     - **Customer:** name, customerID.
     - **Bank Account:** accountNumber, balance.
3. Define Methods for Each Class:
   - Methods are the actions that each class can perform.
   - *Examples:*
     - **ATM:**
     - authenticateUser(): Verify if the user is valid.
     - dispenseCash(): Give cash to the customer.
     - **Customer:**
     - insertCard(): The customer inserts their card into the ATM.
     - enterPIN(): The customer enters their PIN to authenticate.
     - **Bank Account:**
     - checkBalance(): Check the current balance in the account.
     - withdrawMoney(): Withdraw a specified amount of money.

4. Show the Relationships Between the Classes:

- **Association:** There is a relationship between the ATM and the customer, as well as between the ATM and the bank account.
- **Dependency:** The ATM depends on a Transaction to perform operations like withdrawal.
- **Composition:** An ATM contains components like a Card Reader, Cash Dispenser, and Display, and these components cannot exist without the ATM.



## Examples of Class Diagrams in the Real World

1. **E-Commerce System:**

- Classes:
  - Customer: Represents a user of the e-commerce platform.
  - Product: Represents an item that is available for purchase.
  - Order: Represents a customer's order, which includes multiple products.
- Relationships:
  - Association: A customer can place multiple orders.
  - Composition: An order consists of several products.

2. **Library System:**

- Classes:
  - Book: Represents a book in the library.
  - Member: Represents a library member.
  - Librarian: Represents the person managing the library.
- Relationships:
  - Association: A member borrows books.
  - Inheritance: A librarian and a member are both people, so they inherit from a Person class.

# SOFTWARE ENGINEERING LAB

UNIT – 3

TOPIC – 7

# UML DIAGRAMS – SEQUENCE

## Sequence Diagram

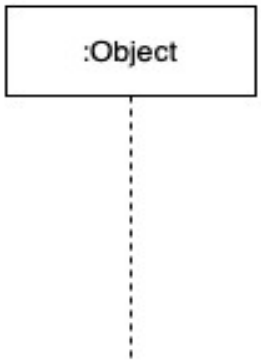
- A sequence diagram is a type of UML diagram that illustrates how objects in a system interact with each other over time.
- It shows the order in which messages are sent between objects, helping visualize the sequence of interactions in a system.
- This is particularly useful when trying to understand the flow of actions or events between different components of a system.

## 2. Components of a Sequence Diagram

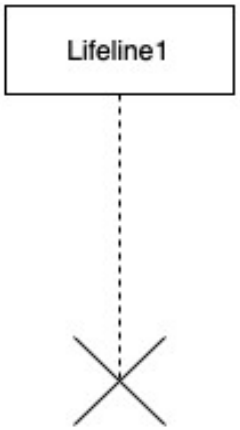
1. **Actors:**
   - Actors represent external entities interacting with the system.
   - These could be people (like users) or other systems (like a database or another software system).
   - In a sequence diagram for an ATM system, for example, the Customer would be an actor.

2. **Objects:**
   - Objects are parts of the system that perform specific actions.
   - Each object represents a class or an entity that plays a role in the interaction.

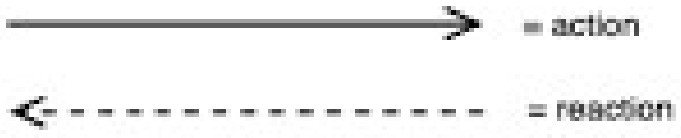
- For example, in an ATM system, objects could include:
  - The ATM machine itself.
  - The Bank Database that stores the customer's account information.




3. **Lifelines:**
   - A Lifeline is a dotted vertical line that extends below each object, representing the object's existence over time.
   - The activation box on the lifeline shows when the object is actively performing an action (e.g., processing a transaction).
   - For example, if a customer initiates a withdrawal, the ATM machine's lifeline shows the duration during which it processes the request.




4. **Messages:**
   - Messages are arrows between objects indicating the communication between them.
   - Each arrow represents a specific action or message being passed from one object to another.




- The arrow shows:
  - Direction of the message (who sends and who receives it).
  - Order of the messages.
- In the ATM system example:
  - The Customer might send a message to the ATM: insertCard().
  - The ATM then sends a message to the Bank Database: authenticateUser().

## 3. Example: Sequence Diagram for an ATM System

Let's break down how a sequence diagram might look for an ATM system, considering the interactions for a cash withdrawal process.

**Actors:**

- Customer (the person using the ATM).

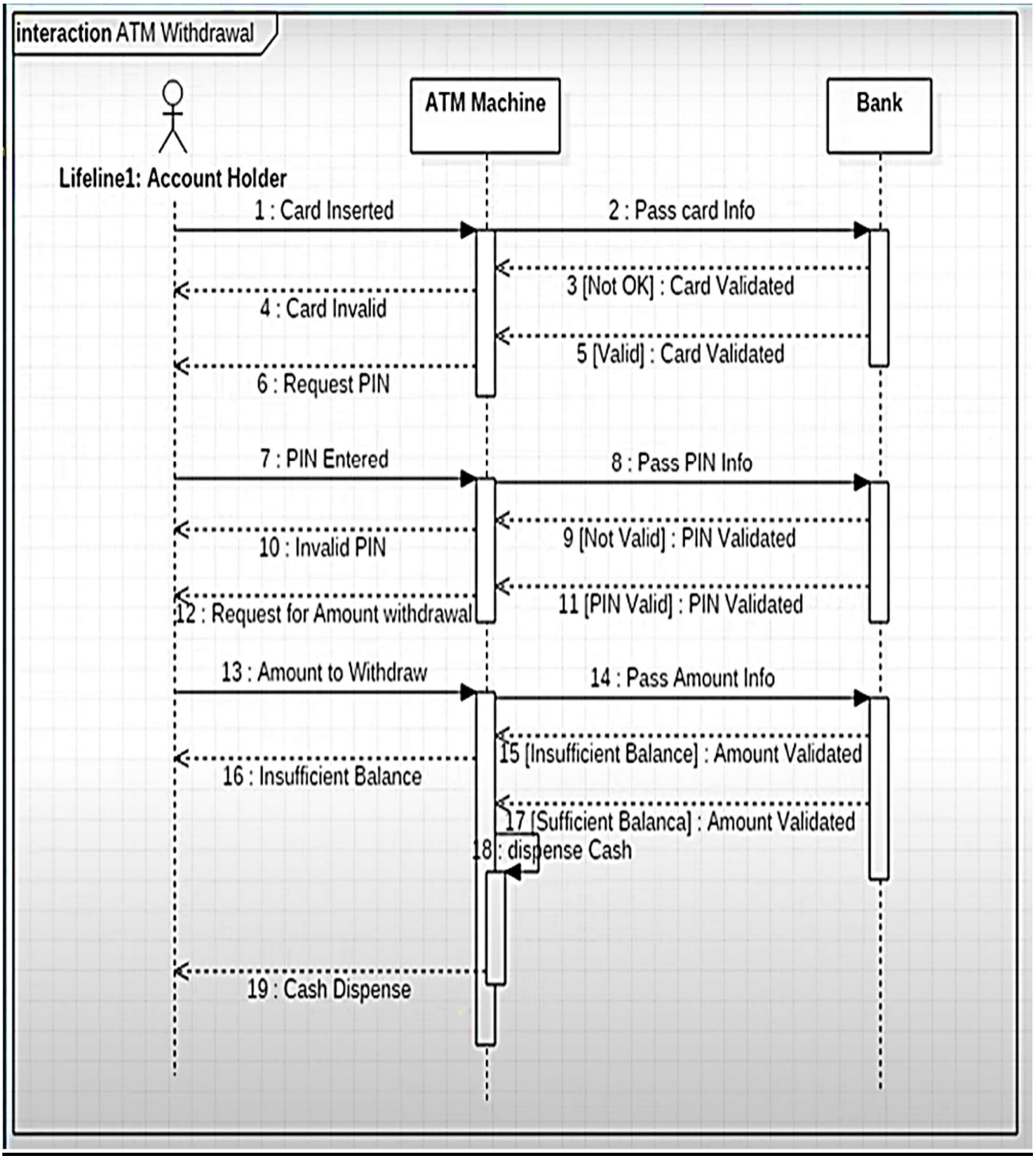
**Objects:**

- ATM (the machine the customer is interacting with).
- Bank Database (the system that holds the customer's account details).

**Messages:**

1. The Customer inserts the card into the ATM: insertCard().
2. The ATM requests the PIN from the customer: requestPIN().
3. The Customer enters the PIN: enterPIN().
4. The ATM sends the entered PIN to the Bank Database for verification: authenticateUser().
5. The Bank Database verifies the PIN and sends a response: authenticationResult().
6. If the authentication is successful, the ATM prompts the Customer to enter the withdrawal amount: requestAmount().
7. The Customer enters the withdrawal amount: enterAmount().

8. The ATM sends a request to the Bank Database to withdraw money: withdrawMoney().
9. The Bank Database confirms the transaction and updates the balance: transactionConfirmation().
10. The ATM dispenses the cash and ends the transaction: dispenseCash().



## Purpose of Sequence Diagrams

- Sequence diagrams are used to model the dynamic behavior of a system. They help in understanding:
    - How different parts of the system work together to complete a process.
    - The order in which actions or events happen in the system.
    - The flow of messages between different objects over time.

In an ATM system, for example, the sequence diagram helps visualize the step-by-step interaction between the customer, the ATM machine, and the bank's database during a transaction like withdrawing cash. It highlights how the ATM system interacts with the external user and bank database to complete the transaction process smoothly.

# SOFTWARE ENGINEERING

## UNIT – 3

## TOPIC – 8

# UML DIAGRAMS – COMPONENT

## Component Diagram

A **Component Diagram** is a type of UML (Unified Modeling Language) diagram that shows how a system is organized into **smaller parts** called **components**. These components work together to form the **whole system**. It helps to understand:

- **What parts** the system is made of.
- **How these parts connect** to each other.
- **How the system works** by looking at how each component functions.

### What are Components?

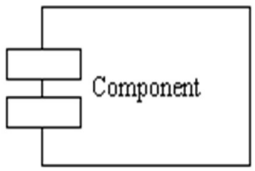
- **Components** are the **building blocks** of a system, just like parts of a machine.
- Each component is a **self-contained unit** that does a specific job.
- Components **communicate** with each other through **interfaces**, which are defined connection points that allow components to work together.
- **Example:**
- *In a **car**, components include:*
- ***Engine**: Provides power.*
- ***Brakes**: Stops the car.*
- ***Transmission**: Controls how the power is used to move the car.*
- Similarly, *in a software system, components could be:*
- ***Login system**.*
- ***Database**.*
- ***User interface**.*

## Characteristics of Components

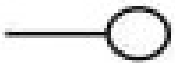
- **Reusable**: Components can often be used in different parts of a system or even in **other systems**. This reduces duplication and saves development time.
- **Replaceable**: If a component stops working or needs to be updated, it can be replaced without affecting the other parts of the system.
- **Modular**: Components are designed to work **independently**, making the system easier to maintain and update.

## Symbols Used in a Component Diagram

- **Component**:
- Represents a **part** of the system, like a software function or hardware part.
- **Symbol**: A rectangle with two smaller rectangles (tabs) sticking out from the left side.




- Example: A **login system** or a **database**.
- **Interface**:
- Shows **how a component interacts** with other parts of the system, like a connection point where other components can communicate.
- **Symbol**: A small circle (called a "lollipop") connected to the component.



- **Provided Interface**:
- This means the component **offers a way** for others to use its functionality.
- **Symbol**: A lollipop symbol (circle) connected to the component.
- **Required Interface**:
- This means the component **needs** something from another component to work properly.
- **Symbol**: A half-circle (socket) connected to the component.


- **Dependency**:
- Shows that one component **depends** on another to function properly.
- **Symbol**: A dashed arrow pointing from the dependent component to the component it relies on.


- **Connector**:
- Shows a **connection** or communication between two components.
- **Symbol**: A solid line with an arrow between components.

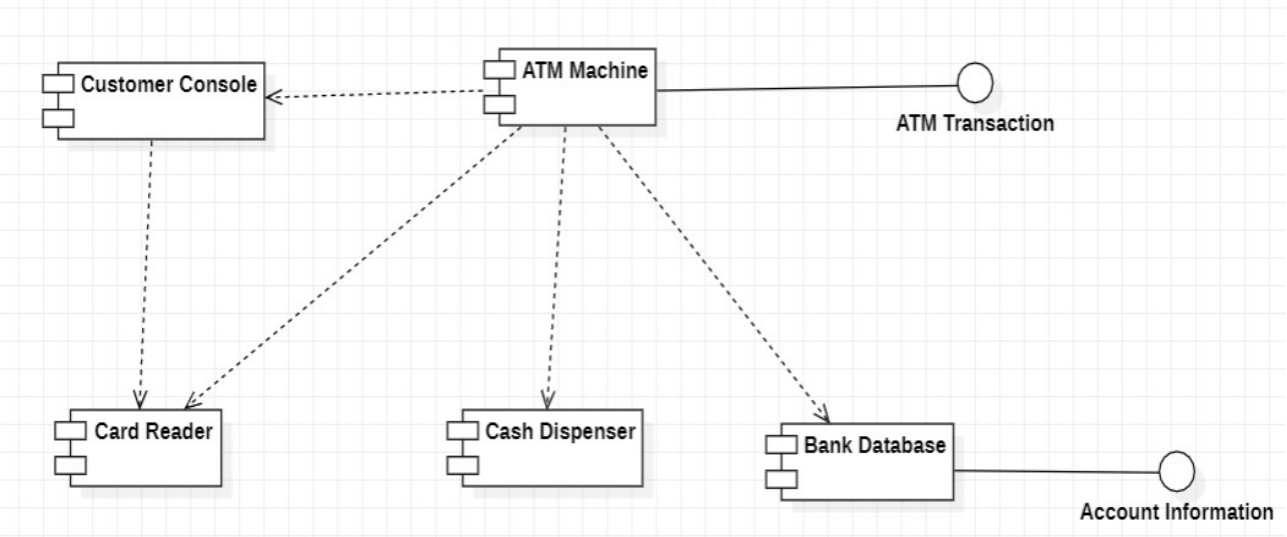

## Example: Key Components of an ATM System

The ATM is made up of several components, and each one has its own function. Here's how they work together:

1. **ATM Machine (Main Component):**
    - This is the physical machine you see and interact with.
    - It has several important sub-components:

1. User Interface: The screen where options like “Withdraw” or “Check Balance” are shown, and you interact with the machine.
2. Card Reader: The slot where you insert your card. It reads the card's information, like your account details.
3. Cash Dispenser: The part that gives you money after a successful withdrawal.

2. **Bank Database (Main Component):**
   - This is the server that stores your account information, such as your balance and transaction history.
   - It checks if you have enough money for a withdrawal and updates your account after the transaction.
3. **Network (Main Component):**
   - This is the communication line between the ATM and the bank’s database.
   - It transfers your requests (like a withdrawal) to the bank’s system and sends back confirmation (whether the transaction was successful or not).




## How Components Work Together in the ATM System

1. **Customer Interaction:**
   - The customer interacts with the ATM machine by using the User Interface and Card Reader.
2. **Communication:**

- The ATM communicates with the Bank Database through the Network to confirm account information and transaction status.

3. **Actions:**
   - If the transaction is approved, the Cash Dispenser will give the customer their money.

## Why Component Diagrams Are Useful

- Helps in understanding how a complex system is organized by showing the individual parts and how they work together.
- Makes systems easier to manage since each component can be changed, replaced, or updated without affecting the whole system.
- Ensures modularity: Systems are broken down into smaller, manageable pieces, making them easier to develop, maintain, and scale.

# SOFTWARE ENGINEERING

UNIT – 3

TOPIC – 9

# BUILD TOOLS: INTRODUCTION TO MAVEN

## 1. Understanding Maven

**Definition**: Maven is a tool used for managing and building software projects, mainly in Java. It automates key tasks such as compiling code, running tests, and packaging the final software. Maven also manages dependencies by automatically downloading and organizing external libraries that your project needs.

In simple terms, Maven helps developers by simplifying repetitive tasks and keeping projects well-organized, ensuring that all the necessary tools and libraries are in place.

## 2. The Importance of Maven in Software Development

- **Without Maven**: Developers must handle every step manually, including finding libraries, compiling code, running tests, and packaging the project.
- **With Maven**: Maven automates these tasks. It structures your project, downloads the libraries, compiles the code, runs tests, and packages everything for you. This results in faster development and fewer errors.

**Example**: If you're building a Java project that requires a library for image processing, you normally need to search for the correct library, download it, and set it up manually. **With Maven**, you just specify the library in the POM file, and Maven automatically fetches and integrates it into your project.

## 3. Maven Operations

Maven relies on a **POM file** (Project Object Model) to function. This file serves as an instruction guide for Maven, telling it what libraries are needed and how to build the project.

## 4. Exploring the POM File (Project Object Model)

The **POM file** is the backbone of any Maven project. It is written in XML and specifies how the project is structured, what dependencies are needed, and how the project should be built.

**Simple Example of a POM File:**

Think of the POM file like a recipe. It lists the ingredients (libraries) and the steps needed to create the final product.

```
xml
Copy code
<project>
    <groupId>com.example</groupId>
    <artifactId>my-project</artifactId>
    <version>1.0</version>

    <dependencies>
        <dependency>
            <groupId>org.example</groupId>
            <artifactId>image-library</artifactId>
            <version>2.1</version>
        </dependency>
    </dependencies>
</project>
```

**Breaking it down**:

- **groupId**: Identifies the organization or group that makes the project.
- **artifactId**: The name of your project (e.g., `my-project`).
- **version**: The version number of your project (e.g., `1.0`).
- **dependencies**: This section lists the libraries your project needs. Here, it specifies the **image-library** from `org.example`.
- **Without Maven**: You would need to find and download the **image-library** manually.

- **With Maven**: You just list it in the POM file, and Maven automatically downloads it for you.

**Types of POM Files:**

1. **POM for a Standalone Project**: This is a regular POM file for a single project. It defines all dependencies and build configurations for that specific project.
2. **Parent POM**: Used in multi-module projects, it defines shared settings and dependencies for all child modules.
3. **Child POM**: A POM file for a specific module that inherits settings from the parent POM.

## 5. Build Process in Maven

Maven follows a series of steps called the **build lifecycle** to create and manage your project.

- **Without Maven**: You have to manually compile your code, run tests, and package everything into a JAR file.
- **With Maven**: You can run a command like `mvn package`, and it will automatically do all these steps in the right order:
    1. **Clean**: Deletes old files from previous builds.
        - Command: `mvn clean`
    2. **Compile**: Converts your code into a usable format.
        - Command: `mvn compile`
    3. **Test**: Runs tests to check if the code works.
        - Command: `mvn test`
    4. **Package**: Bundles everything into a single file for you to use or share.
        - Command: `mvn package`

## 6. Managing Libraries with Maven Repositories

Maven uses **repositories** to store and find the libraries your project needs.

**Types of Repositories:**

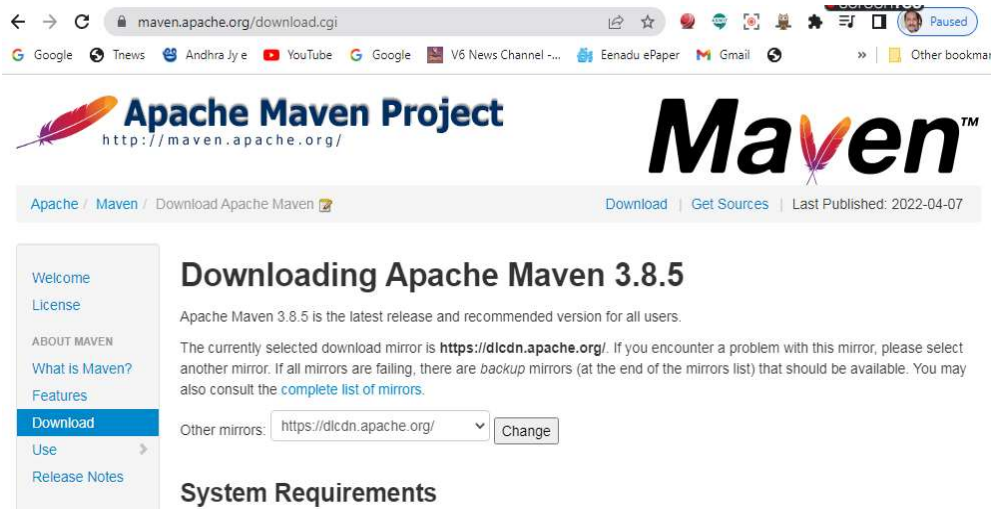
1. **Local Repository**: This is on your computer. Maven checks here first to see if you have the libraries you need.
2. **Central Repository**: This is an online store of libraries managed by the Maven community. If a library isn't in your local repository, Maven downloads it from here.
3. **Remote Repository**: If a library is not available in the central repository, you can specify an additional location for Maven to look.

**Example**: When your project needs a library, Maven will first look in your local repository. If it's not there, it will automatically fetch it from the central repository.

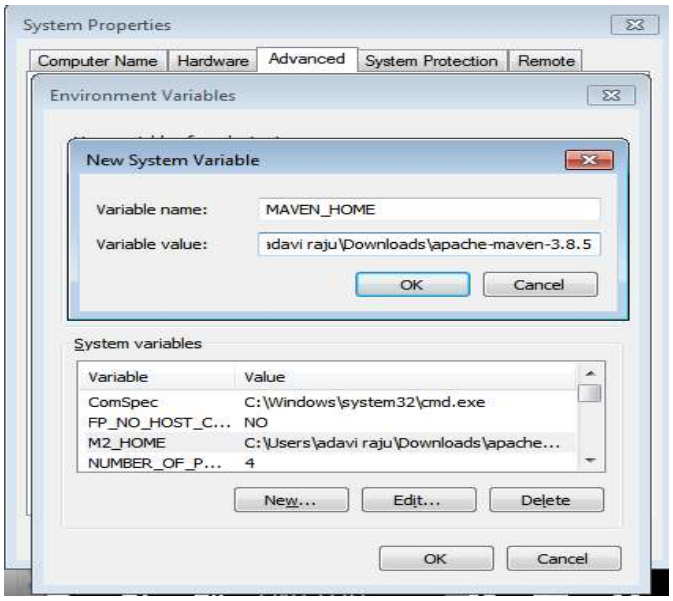
## 7. Installing and Setting Up Maven

To start using Maven, you need to install it and set it up on your computer.

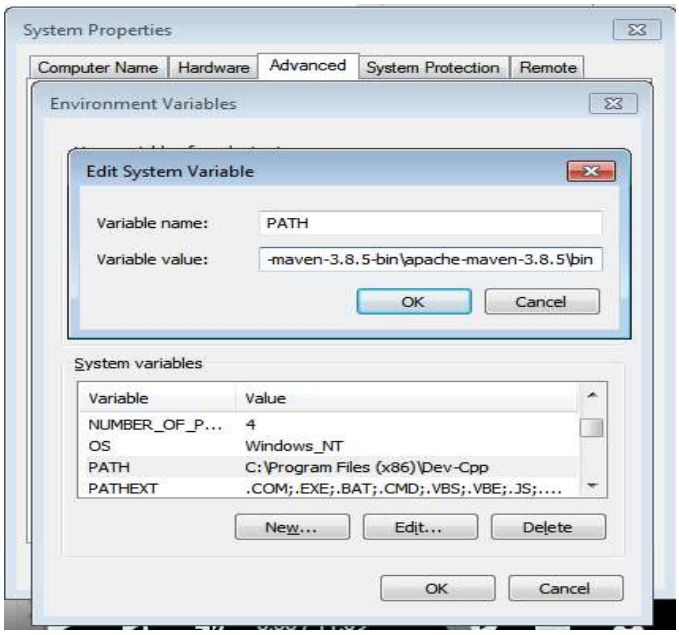
1. **Download Maven**: Get the latest version from the Apache website.
2. **Extract Files**: Unzip the downloaded files.



3. **Set Environment Variables**: You need to set up `JAVA_HOME` (where Java is installed) and `MAVEN_HOME` (where Maven is located) in your system settings.




4. **Add Maven to PATH**: This allows you to use Maven commands from any folder on your computer.



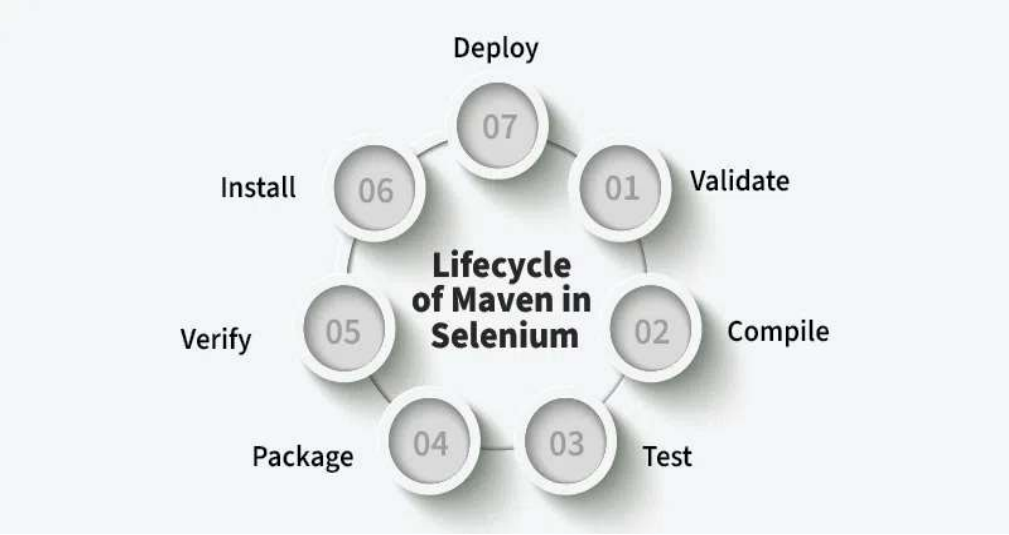

5. **Verification**: You can check if Maven is installed correctly by typing `mvn -version` in your terminal. If everything is set up right, it will show the version of Maven.

## 8. Build Lifecycle Phases in Maven

The **build lifecycle** includes several key phases that dictate what happens when you run Maven.

**Main Phases:**

1. **Validate**: Checks if the project is set up correctly.
   - Command: `mvn validate`
2. **Compile**: Compiles the source code.
   - Command: `mvn compile`
3. **Test**: Runs tests to ensure the code works.
   - Command: `mvn test`
4. **Package**: Bundles the compiled code into a JAR file.
   - Command: `mvn package`
5. **Install**: Installs the package into your local repository.
   - Command: `mvn install`
6. **Deploy**: Sends the package to a remote repository for others to use.
   - Command: `mvn deploy`




**Example**: You can use commands like `mvn clean`, `mvn compile`, or `mvn package` to perform specific actions during the build process.

## 9. Using Maven Plugins

Maven has many **plugins** that extend its capabilities. Plugins allow you to add extra features, like generating reports or performing code analysis.

**Example**: If you want to create a documentation report for your project, you can use a specific plugin that does this automatically.

## 10. Multi-Module Projects with Maven

Maven supports **multi-module projects**, which help you manage larger projects by breaking them into smaller parts. Each module can have its own POM file, making it easier to organize and maintain the project.

**Example**: If your project has a user interface and a backend service, you can create a parent project that contains these two components as separate modules.

# SOFTWARE ENGINEERING

## UNIT – 3

## TOPIC – 10

# CONTINUOUS INTEGRATION USING JENKINS

## Introduction to Jenkins

**Jenkins** is a free tool that helps automate tasks in software development. It's mainly used to build, test, and deploy code whenever there are changes. This process is called **Continuous Integration (CI)** and **Continuous Delivery (CD)**. Jenkins helps developers work faster by handling repetitive tasks and finding mistakes early. It's like an assistant that takes care of the routine steps, allowing developers to focus on more important tasks.

## Main Features of Jenkins

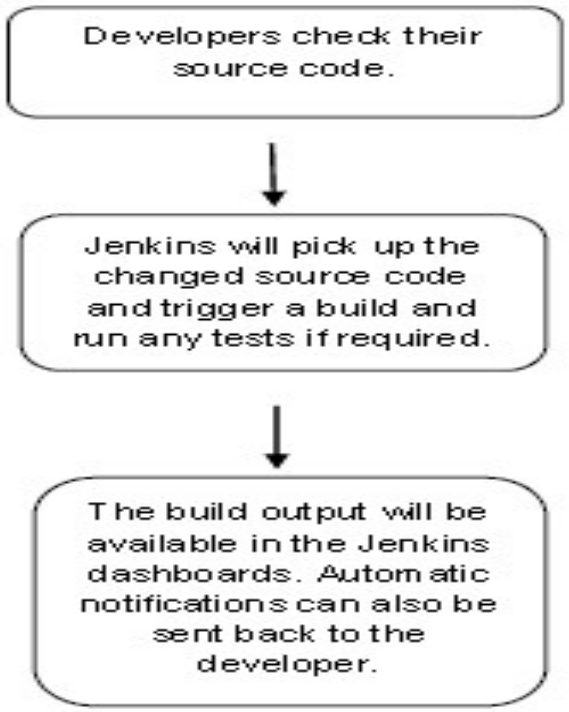
Jenkins is popular because of the following features:

- **Automation**: Jenkins automatically checks code for errors, builds the project, and runs tests, which saves developers time.
- **Integration**: It works well with tools like Git (to manage code) and Maven (to build projects).
- **Plugins**: Jenkins has many plugins that help it connect to other tools, like cloud services or testing tools.
- **Distributed Builds**: Jenkins can split tasks across multiple computers to get things done faster.
- **Customizable**: Developers can add their own plugins to make Jenkins fit their specific needs.
- **Monitoring and Reporting**: Jenkins gives updates on what's happening, so you know if something went wrong.
- **Pipeline Support**: A "pipeline" is a series of steps to build, test, and release software. Jenkins automates these steps, speeding up the whole process.

## How Jenkins Workflow Works

The **workflow** of Jenkins generally follows these steps:

1. **Code Repository**: Developers store their code in a shared place, like **Git**.
2. **Jenkins Trigger**: Jenkins watches the code repository for changes. When someone updates the code, Jenkins sees it and starts working.
3. **Code Checkout**: Jenkins pulls the updated code from the repository.
4. **Build**: Jenkins compiles the code and turns it into a usable format.
5. **Testing**: Jenkins automatically runs tests to make sure the code works as expected.
6. **Artifact Archiving**: Jenkins saves the final build (called an artifact) for later use.
7. **Notification**: Jenkins sends notifications to let developers know if the process worked or if there was a problem.




**Example**: Imagine you're building a website. Whenever a team member adds new features, Jenkins automatically checks if everything works, sends an email with the results, and saves the final product.

## Continuous Integration (CI)

**Continuous Integration (CI)** means that developers regularly add their code changes to a shared project. Each time they make changes, Jenkins tests and builds the code automatically. This helps catch issues early and ensures that everyone's code works well together.

In simple terms, CI allows teams to check their work frequently so they don't have to wait until the end of the project to find and fix problems.

## Setting Up Continuous Integration in Jenkins

Follow these steps to set up **Continuous Integration (CI)** in Jenkins, so that your code is automatically built and tested every time you make changes:

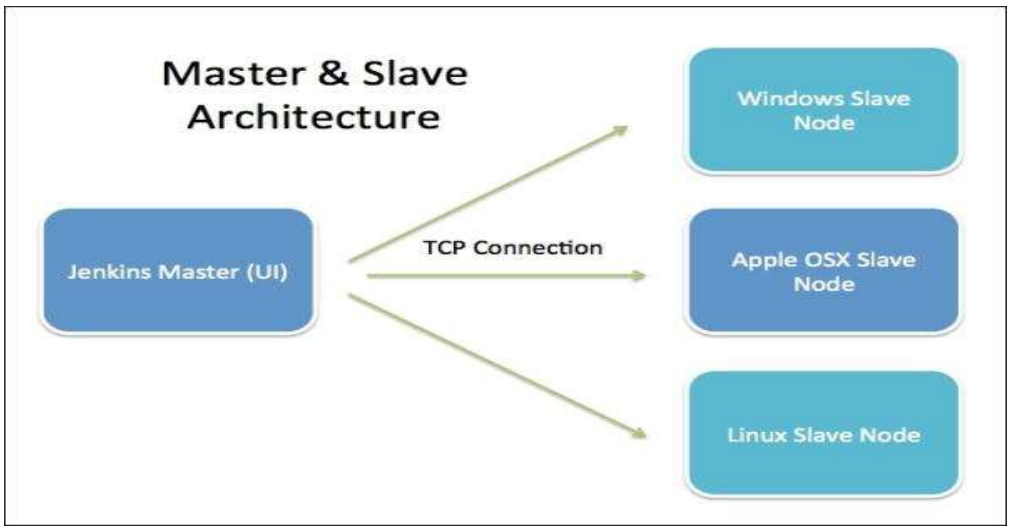
1. **Install Jenkins**: First, install Jenkins on your computer or server. This will handle all the automation.
2. **Create a New Jenkins Job**:
    - Open Jenkins in your web browser.
    - Click "New Item" and name your job (for example, "MyCIJob").
    - Select the **"Freestyle project"** option, which is a basic setup for CI tasks.
3. **Configure Source Code Management**:
    - In the job settings, go to the **"Source Code Management"** section.
    - Select **Git** or another version control system.
    - Enter the **repository URL** (the link to your code) and add credentials if needed.
4. **Set Up Build Steps**:
    - In **"Build"**, set up what Jenkins should do to build your project.
    - For example, if you're using **Maven**, choose "Invoke top-level Maven targets" and enter goals like `clean install`.
    - For **Gradle**, select "Invoke Gradle script" and specify tasks like `clean build`.
5. **Post-Build Actions**:
    - After building, you can set up additional steps in **"Post-build Actions"**.
    - You can tell Jenkins to **save artifacts** (built files) or **send notifications** about the build status.

6. **Save and Run the Job**:
    - Save your setup and run the job to make sure it works. Jenkins will pull the code, build it, and report the result.
7. **Set Up Webhooks for Automatic Builds (Optional)**:
    - If you want Jenkins to automatically build every time the code changes, set up **webhooks** in your Git repository.
    - Webhooks tell Jenkins, "A new change was made—start building!" Alternatively, Jenkins can check for changes at regular intervals.

By following these steps, Jenkins will automatically build and test your code whenever you make changes.

## Jenkins Architecture

Jenkins uses a **distributed architecture** that helps it handle many tasks at once. It consists of two main parts: the **Jenkins Master** and **Jenkins Slaves**.




**Jenkins Master**

The **Jenkins Master** is the main part of Jenkins. It does the following jobs:

- **Scheduling jobs**: The master decides when tasks (builds) should run.
- **Distributing tasks**: The master sends tasks to slave computers to do the actual work.
- **Monitoring slaves**: It keeps track of the slave machines and manages their availability.

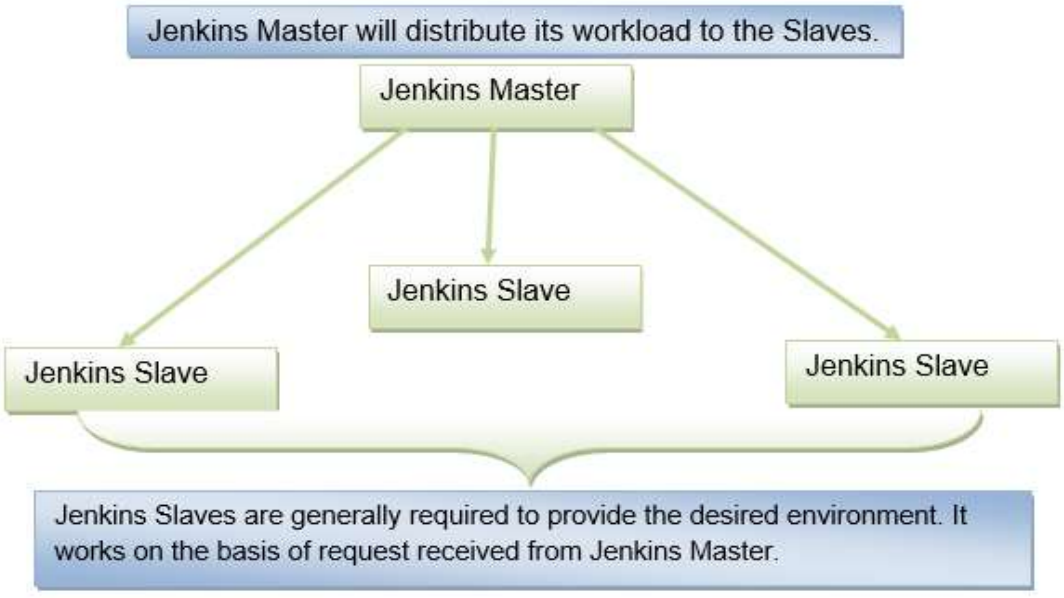
- **Displaying results**: The master shows the results of the builds, so developers know if they were successful.

The Jenkins Master can also run jobs itself, but it usually sends them to slaves to spread the workload.

**Jenkins Slaves**

**Jenkins Slaves** are extra computers that do the actual work of building and testing. This allows Jenkins to handle more tasks at once. Each slave runs a **Jenkins agent**, which communicates with the master. The master sends tasks to the slave, and the slave does the job (like building or testing) and sends back the results.

Slaves can be set up on different machines (physical or virtual), and you can choose whether a project should always run on a specific slave or let Jenkins pick an available one.

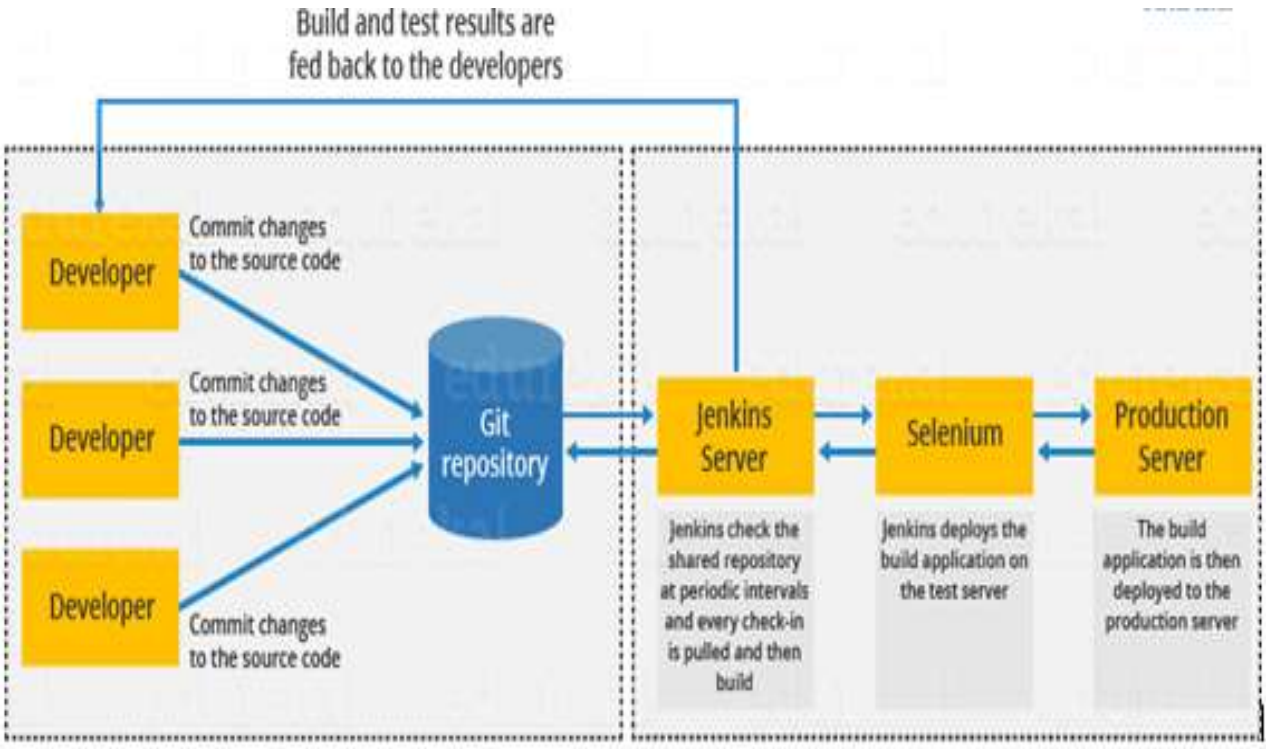



**Example**: Imagine Jenkins is like a factory. The **master** is the manager that organizes and assigns tasks, and the **slaves** are the workers that actually complete the tasks. The manager keeps track of all the workers and checks if the tasks are done correctly.

## Jenkins Flow Diagram for Continuous Integration

The Continuous Integration (CI) process with Jenkins generally works like this:

1. A developer updates the code in a **source code repository** (like Git).
2. Jenkins detects the code change and starts a new build.
3. If there's a problem, Jenkins alerts the team. If everything works, Jenkins moves on to the next steps.
4. If the build is successful, Jenkins deploys it to a test server to verify everything works.
5. Jenkins then informs developers of the results and repeats this process for each new code update.




**Example**: Imagine you're writing code for an online store. Each time you make a change, Jenkins checks if it's compatible with existing features and tells you if anything needs fixing.

## Advantages of Using Jenkins

- **Free**: Jenkins is open-source, so it's free to use.
- **Easy to Set Up**: It's simple to install and doesn't require complicated setup.
- **Customizable**: With many plugins available, Jenkins can fit different needs.

- **Works Across Platforms**: It runs on Windows, macOS, and Linux.
- **Cloud Support**: Jenkins can also run on cloud servers, making it flexible for remote use.

## Drawbacks of Jenkins

- **Outdated Interface**: Jenkins looks a bit old-fashioned and can be tricky to navigate.
- **Requires Some Technical Knowledge**: Since Jenkins runs on a server, it might require some basic server skills to manage.
- **Can Be Fragile**: Small configuration changes can sometimes cause issues, so it needs regular attention.

# SOFTWARE ENGINEERING

## UNIT – 3

## TOPIC – 11

# BUILD PIPELINE PROJECT USING JENKIN SCRIPT

## Description

Jenkins is a leading tool for automating software development processes, especially Continuous Integration (CI) and Continuous Delivery (CD). A Jenkins Pipeline defines and automates the stages needed to build, test, and deploy software projects.

## Introduction to Jenkins Script

A Jenkins script is a set of commands that automates tasks in Jenkins. These scripts are used to manage various stages of software development, from retrieving code to deploying the final application. Jenkins scripts are crucial for creating a CI/CD pipeline, which helps in automating repetitive tasks, reducing errors, and ensuring consistency.

**Common Tasks Performed by a Jenkins Script:**

1. Retrieve the latest source code from a version control system like Git.
2. Compile the code into executable files.
3. Run automated tests to ensure code quality.
4. Package the application for deployment.
5. Deploy the application to a testing or production environment.

Using scripts streamlines the software delivery process, saves time, and maintains consistent workflows.

## Types of Jenkins Scripts

There are two primary types of scripts used in Jenkins for defining pipelines:

### 1. Declarative Pipeline Syntax

- A structured and straightforward way of writing pipelines.
- Focuses on defining stages and steps in a readable format.
- **Key Features**:
  - Easy to read and understand.
  - Suitable for those new to scripting.
- **Use Case**:
  - Best for simple pipelines that don't require complex logic.

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building the application'
            }
        }
        stage('Test') {
            steps {
                echo 'Running tests'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying the application'
            }
        }
    }
}
```

- **Example**:
  - Configuration-style scripting that defines what needs to happen in each stage.

## 2. Scripted Pipeline Syntax

- A more flexible and powerful way of writing pipelines using a Groovy-based scripting language.
- Offers advanced control over pipeline behavior.
- **Key Features**:
  - Flexibility and extensive customization.
  - Suitable for users who need detailed control over the pipeline.
- **Use Case**:
  - Ideal for complex projects requiring advanced CI/CD features.

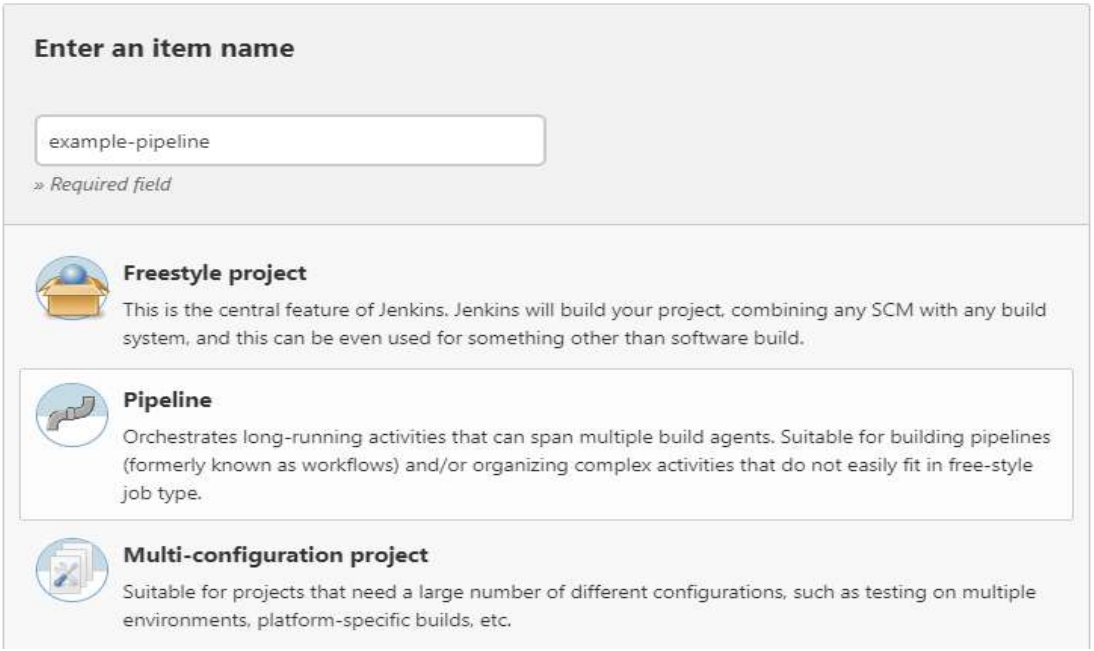
```
node {
    stage('Build') {
        echo 'Building the application'
        // Additional build steps here
    }
    stage('Test') {
        echo 'Running tests'
        // Additional test steps here
    }
    stage('Deploy') {
        echo 'Deploying the application'
        // Additional deployment steps here
    }
}
```

- **Example**:
  - Code-like scripting that allows conditional logic and custom workflows.

The choice between **Declarative** and **Scripted** syntax depends on the project's complexity and the level of customization needed.

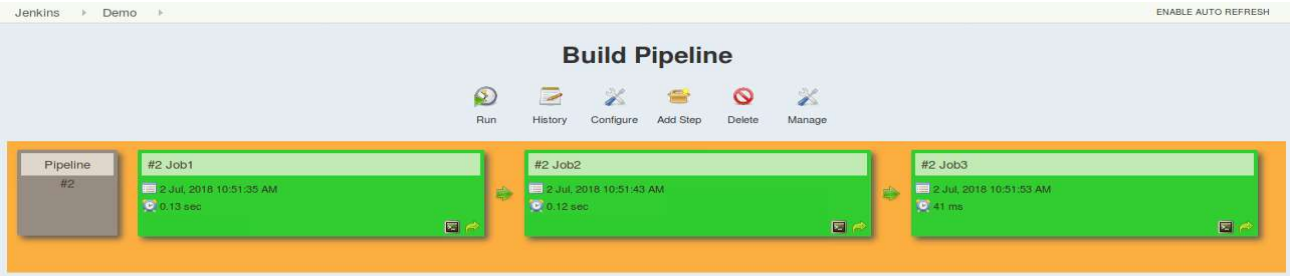
## Types of Jenkins Pipelines

Jenkins supports two major types of pipelines, which manage the CI/CD process:




### 1. Freestyle Projects

- The traditional and simpler way to create jobs in Jenkins using a graphical interface.
- **Key Features**:
  - Easy to set up using the web interface.
  - Suitable for simple build and deployment tasks.
- **Use Case**:
  - Ideal for basic CI/CD projects with straightforward requirements.




- **Example**:
  - Creating a job using a visual interface without writing code.

**2. Pipeline Projects**

- A more advanced and flexible method introduced with the Pipeline plugin.
- **Key Features**:
  - Uses Groovy-based scripts to define the pipeline.
  - Supports version control and complex workflows.
- **Use Case**:
  - Perfect for advanced projects that require dynamic and code-based pipeline configurations.


- **Example**:
  - Writing Groovy scripts to define each step of the build and deployment process.

**Freestyle Projects** are intuitive and visual, while **Pipeline Projects** provide greater control and automation through scripting. The choice between them depends on project complexity.

## Building a Jenkins Pipeline Using a Script

Detailed instructions on creating a Jenkins Pipeline with scripting, a crucial process in software development for automating Continuous Integration (CI) and Continuous Delivery (CD).

# 1. Preparing the Environment

### Checking Tool Versions

Before starting with Jenkins, it's essential to verify that the required tools are installed and working:

- **Maven**: Use `mvn -version` to confirm Maven's installation.
- **Java**: Run `java -version` to check if the Java Development Kit (JDK) is correctly set up.
- **Git**: Verify Git's availability with `git --version`.

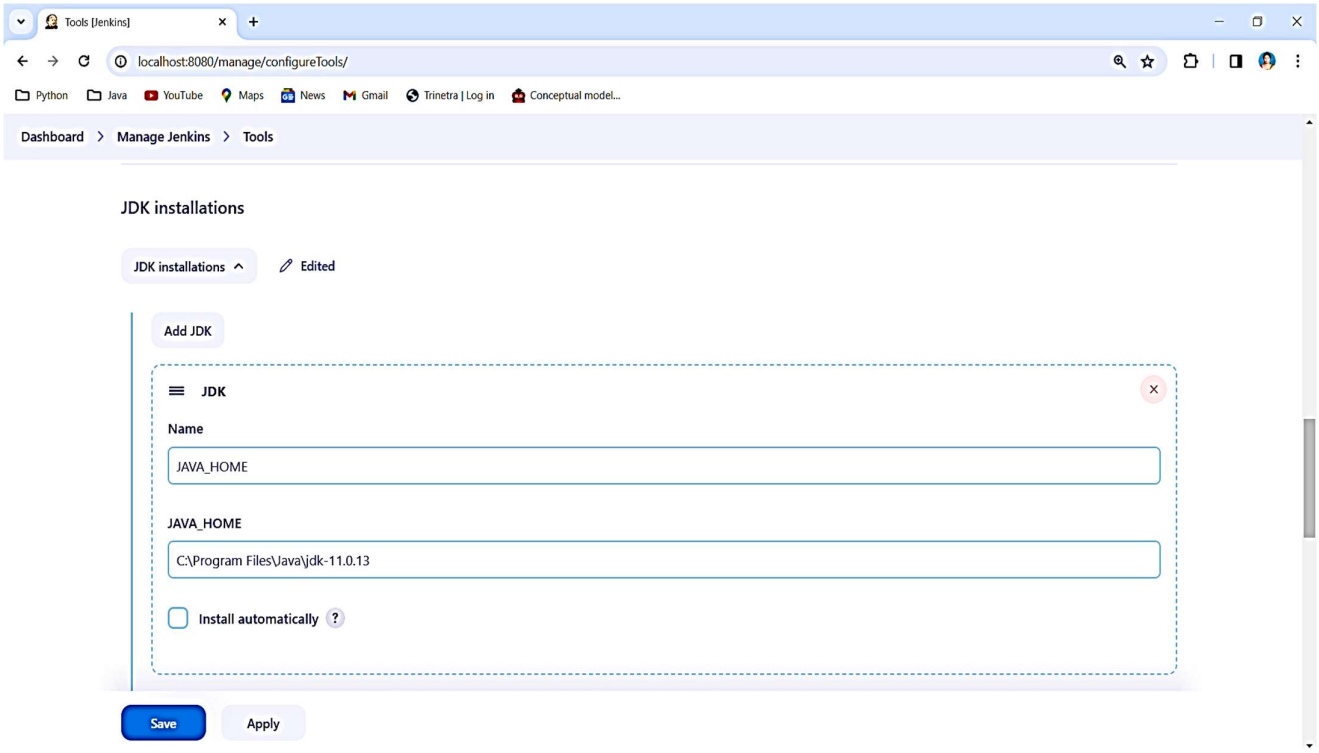

Confirming these tools are correctly installed and accessible helps ensure that Jenkins will function properly.

## Configuring Jenkins with Required Tools

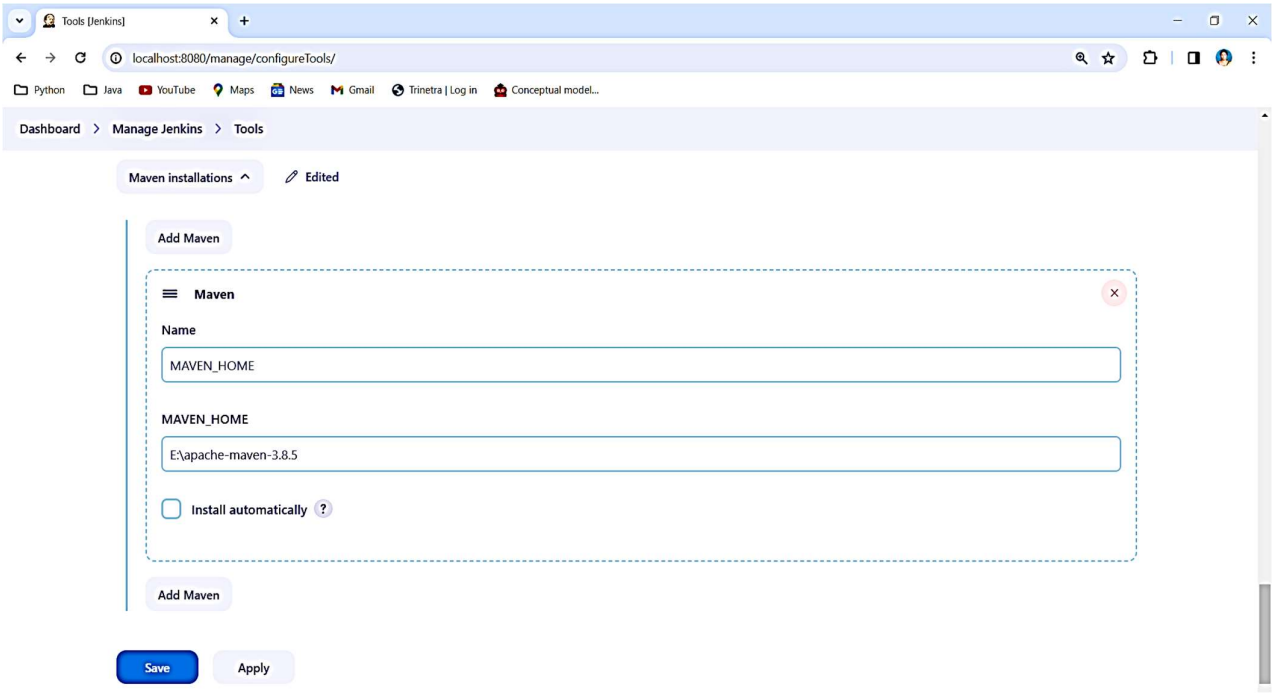
Once the tools are verified, configure Jenkins to recognize them:

1. Log into Jenkins and navigate to **Manage Jenkins → Global Tool Configuration**.

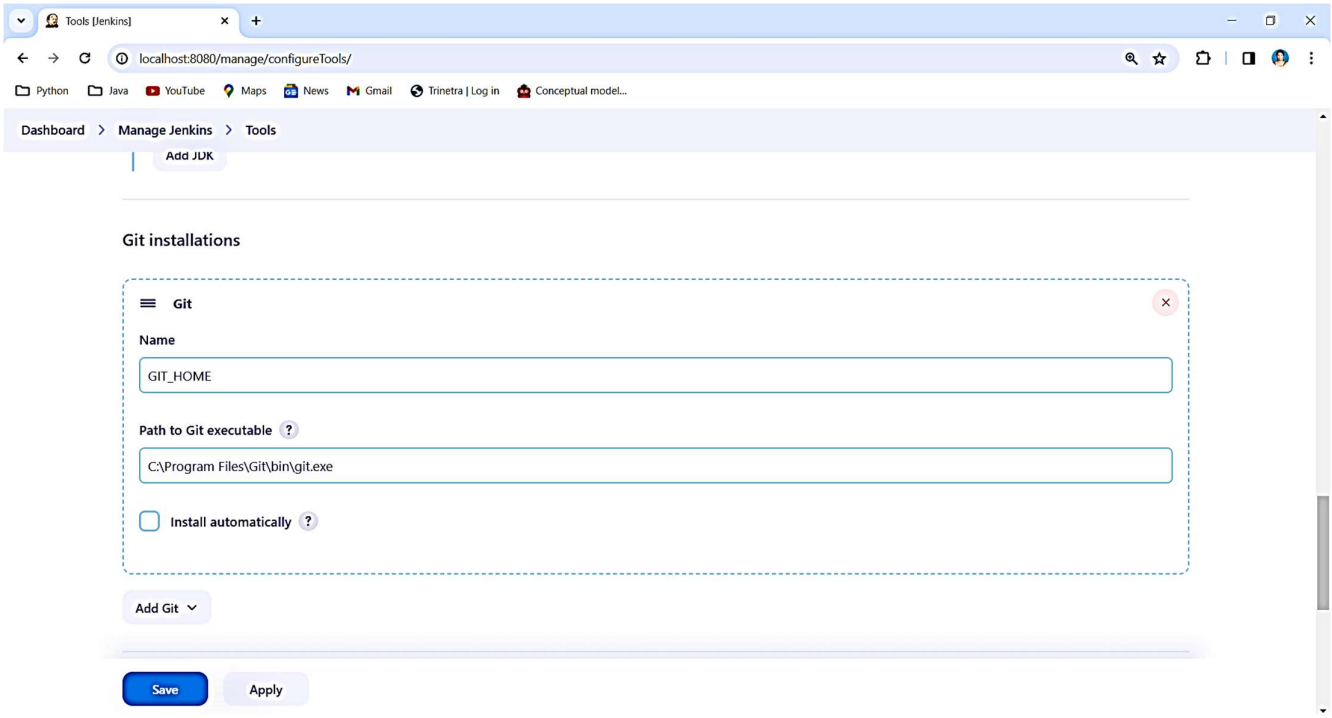
2. Specify paths for:
    - **Java**: Set the JDK path so Jenkins can compile Java projects.




- **Maven**: Define the Maven installation path for managing project builds.



- **Git**: Configure the Git path for source control integration.



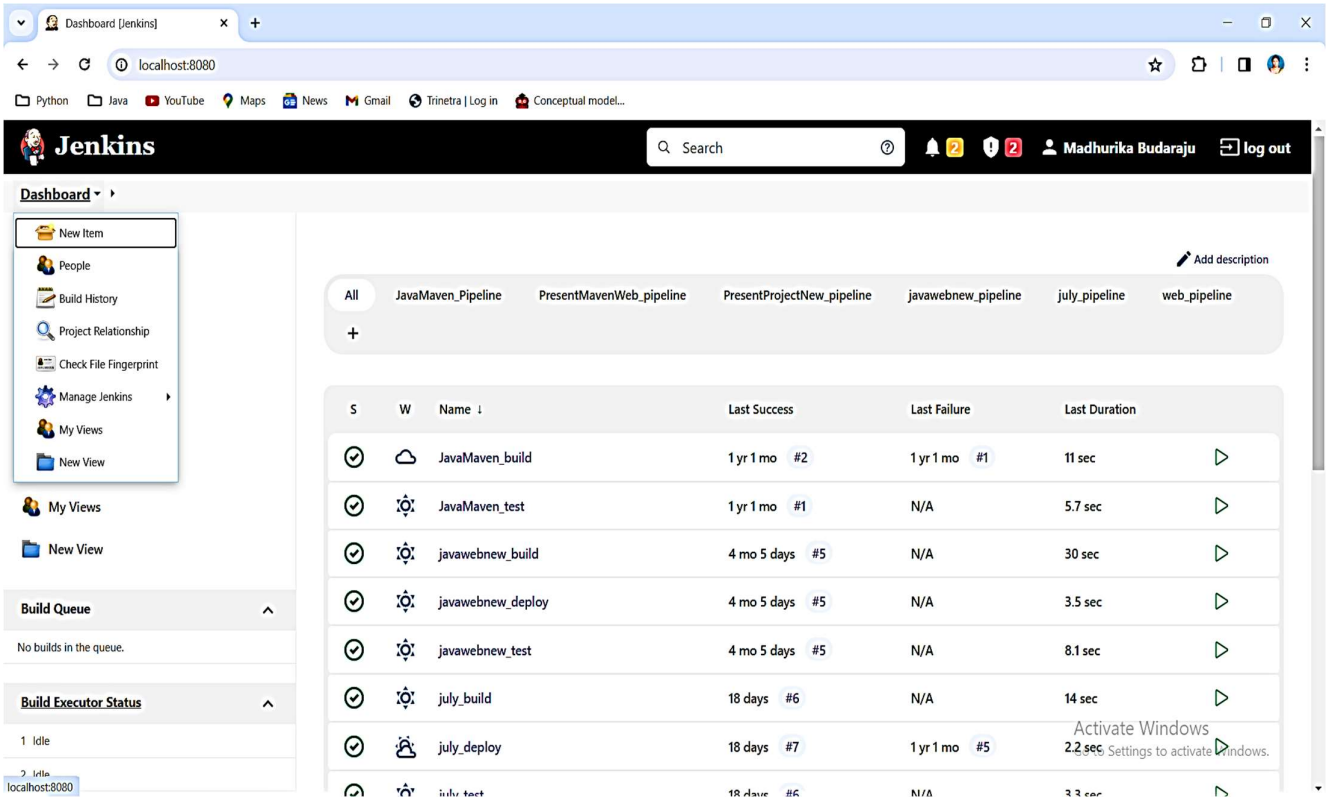

Correct configuration ensures that Jenkins can build, test, and deploy projects without errors.

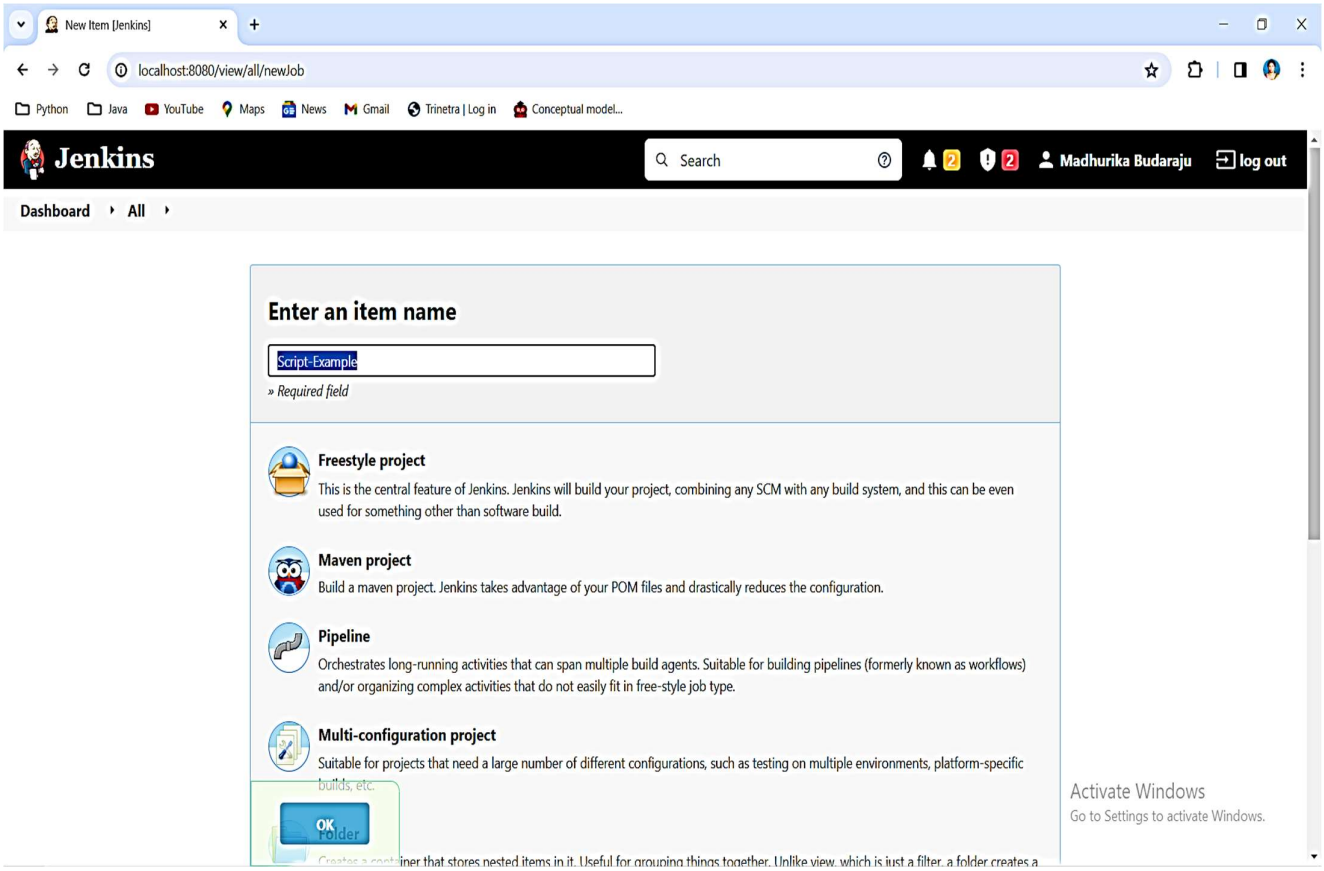
## 2. Creating a New Pipeline in Jenkins

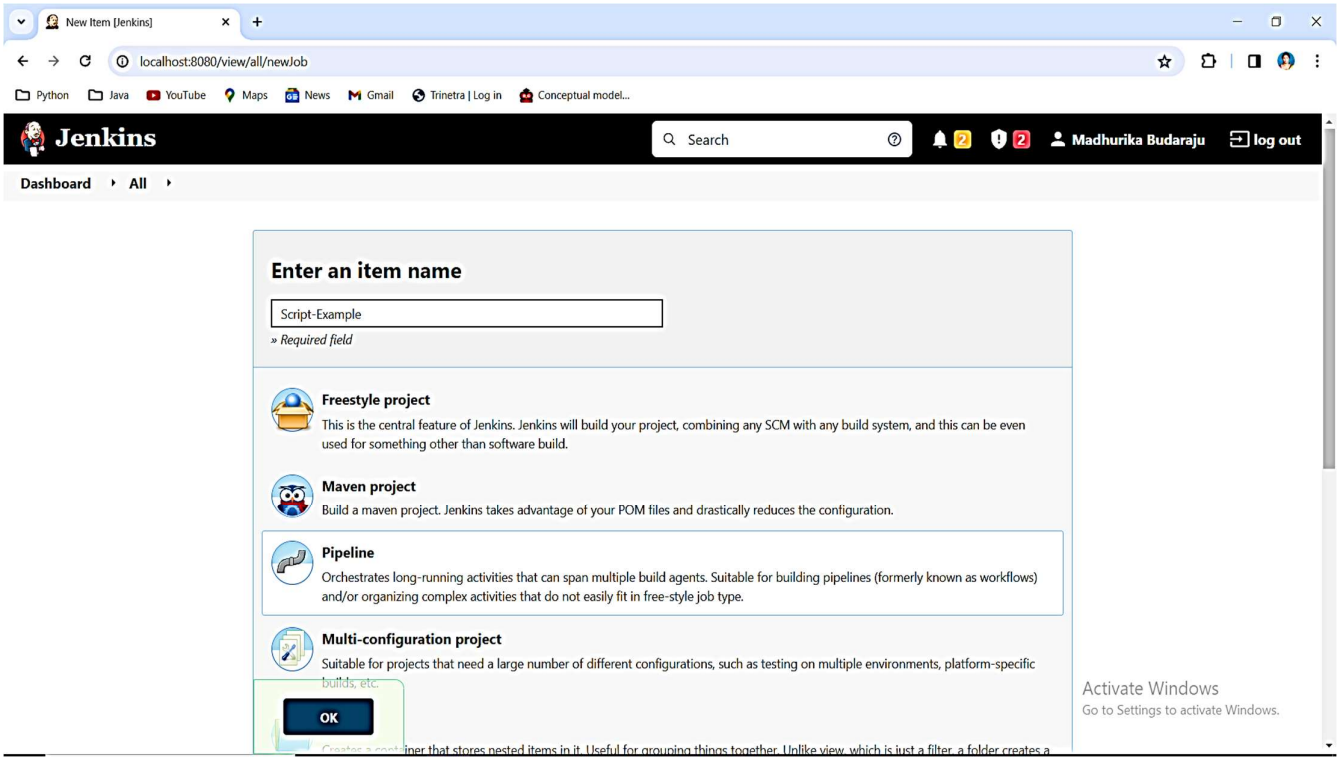
### Setting Up a New Pipeline

With tools configured, start setting up a new Jenkins Pipeline:

1. Access the Jenkins dashboard.
2. Click on **New Item** in the main menu.
3. Enter a descriptive name for the pipeline.
4. Choose the **Pipeline** option and click **OK** to create the new pipeline structure.


Dashboard [Jenkins]
localhost:8080
Python
Java
YouTube
Maps
News
Gmail
Trinetra | Log in
Conceptual model...
Jenkins
Search
Madhurika Budaraju
log out
Dashboard
New Item
People
Build History
Project Relationship
Check File Fingerprint
Manage Jenkins
My Views
New View
Add description
All
JavaMaven_Pipeline
PresentMavenWeb_pipeline
PresentProjectNew_pipeline
javawebnew_pipeline
july_pipeline
web_pipeline
S
W
Name ↓
Last Success
Last Failure
Last Duration
JavaMaven_build 1 yr 1 mo #2 1 yr 1 mo #1 11 sec
JavaMaven_test 1 yr 1 mo #1 N/A 5.7 sec
javawebnew_build 4 mo 5 days #5 N/A 30 sec
javawebnew_deploy 4 mo 5 days #5 N/A 3.5 sec
javawebnew_test 4 mo 5 days #5 N/A 8.1 sec
july_build 18 days #6 N/A 14 sec
july_deploy 18 days #7 1 yr 1 mo #5 2.2 sec
Build Queue
No builds in the queue.
Build Executor Status
1 Idle
2 Idle
localhost:8080
Activate Windows
Go to Settings to activate Windows.


New Item [Jenkins]
localhost:8080/view/all/newJob
Python
Java
YouTube
Maps
News
Gmail
Trinetra | Log in
Conceptual model...
Jenkins
Search
Madhurika Budaraju
log out
Dashboard
All
Enter an item name
Script-Example
» Required field
Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.
Maven project
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.
Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.
Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.
OK
Folder
Activate Windows
Go to Settings to activate Windows.



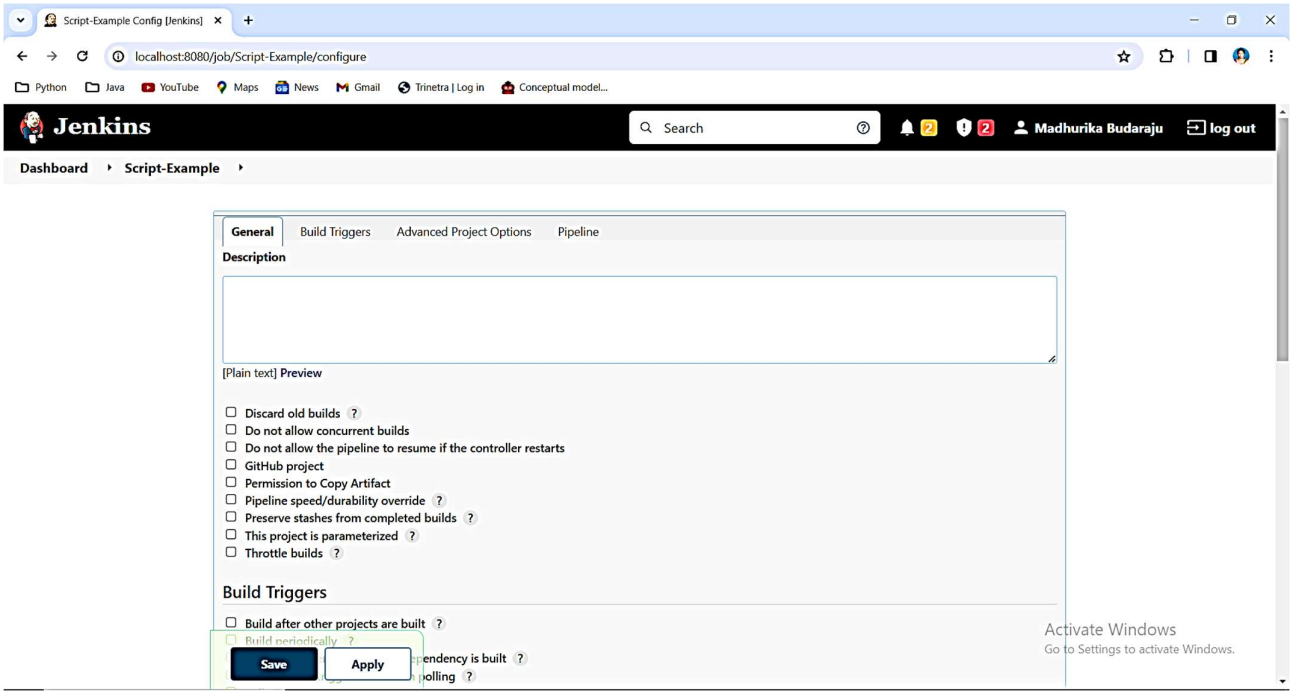

This setup forms the foundation of the pipeline, ready for script input.

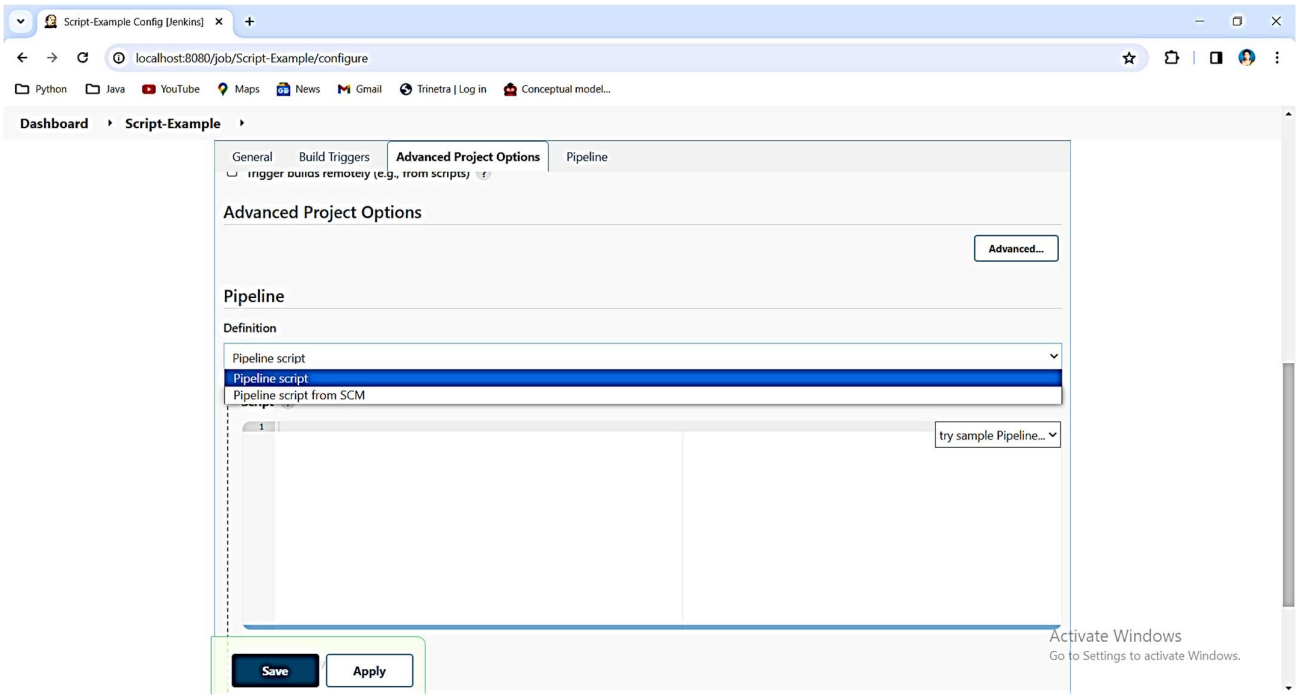
# 3. Configuring the Pipeline Script

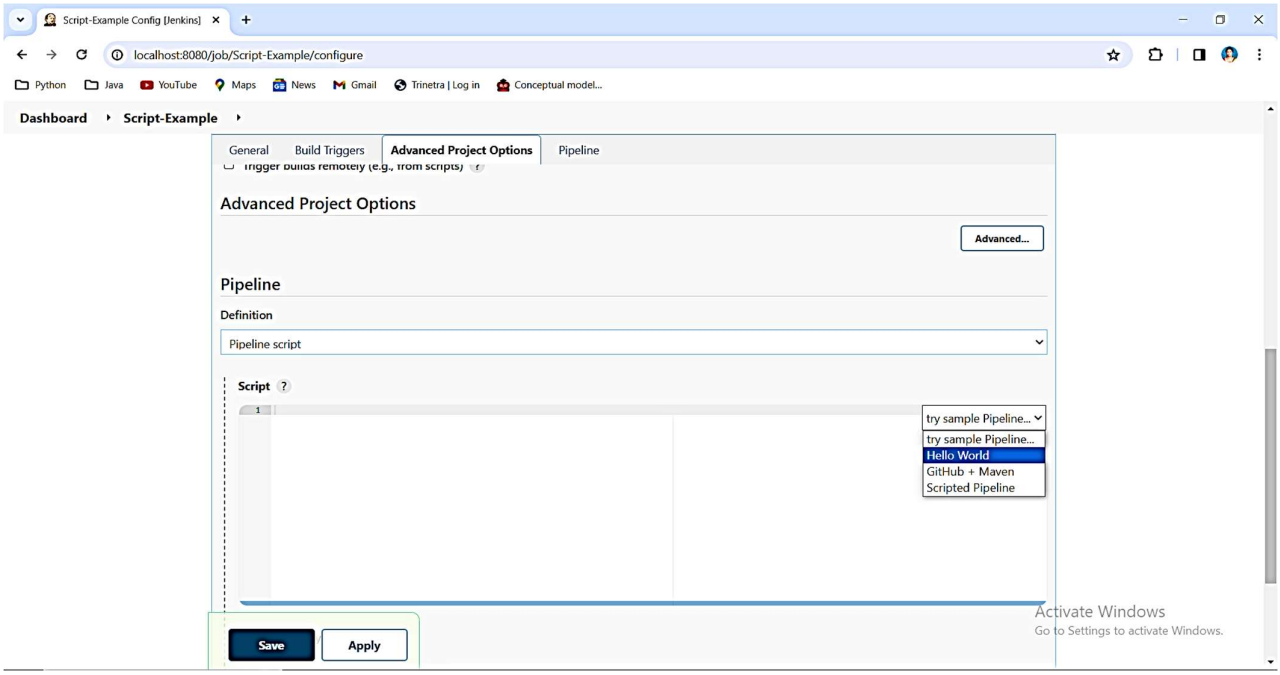
## Accessing the Pipeline Script Section

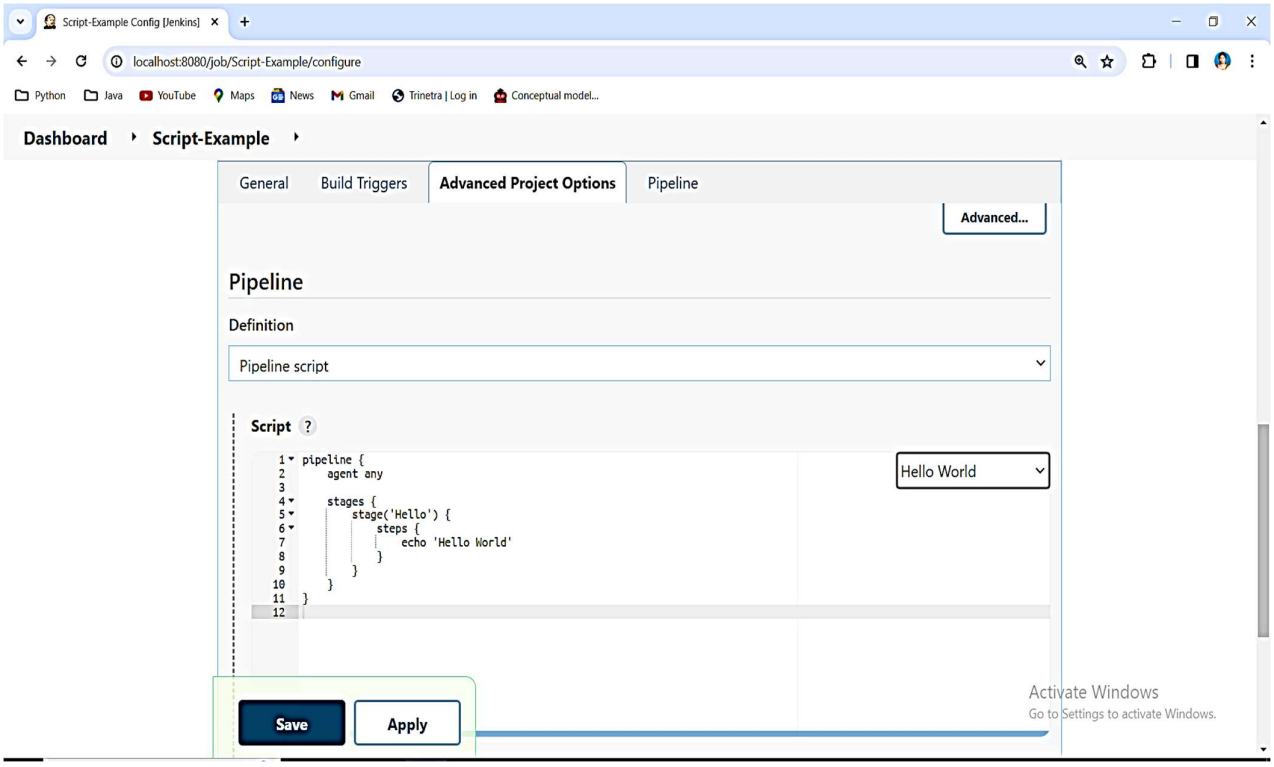
After creating the pipeline, it's time to define the workflow:

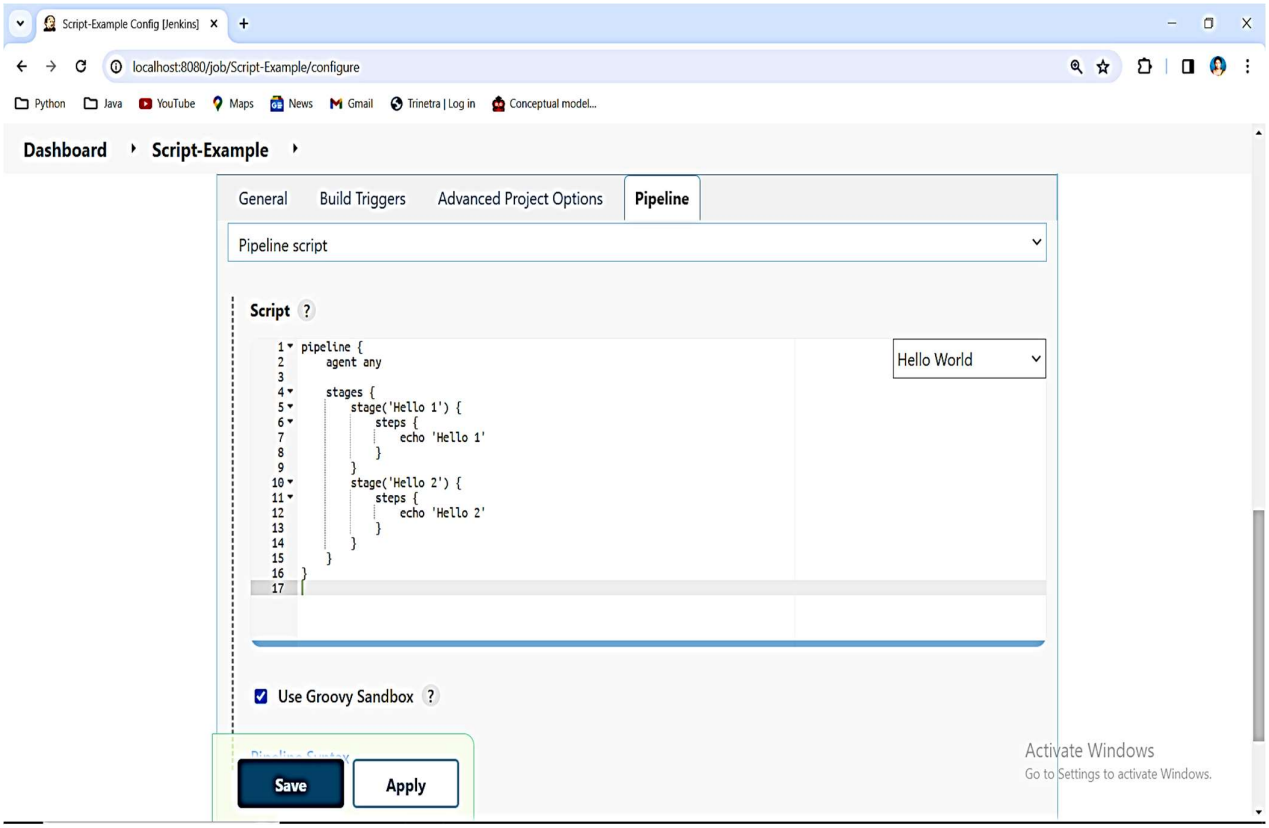
1. Scroll to the **Pipeline** section in the project configuration screen.
2. Input or paste the script that will dictate the steps of the pipeline.
3. Utilize templates like the "Hello World" example to quickly start if needed. Templates offer a simple and accessible way to get familiar with Jenkins scripts.

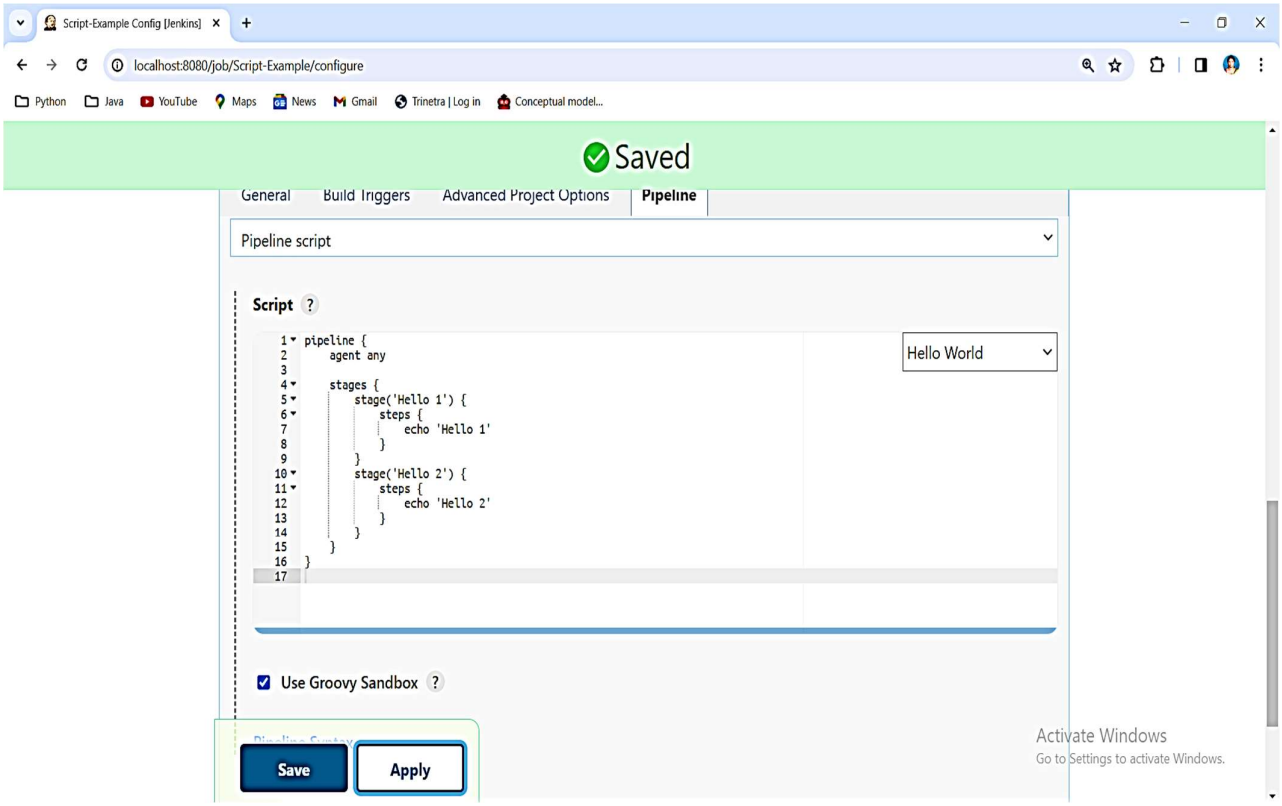

Script-Example Config [Jenkins]
localhost:8080/job/Script-Example/configure
Python
Java
YouTube
Maps
News
Gmail
Trinetra | Log in
Conceptual model...
Jenkins
Search
Madhurika Budaraju
log out
Dashboard
Script-Example
General
Build Triggers
Advanced Project Options
Pipeline
Description
[Plain text] Preview
Discard old builds
Do not allow concurrent builds
Do not allow the pipeline to resume if the controller restarts
GitHub project
Permission to Copy Artifact
Pipeline speed/durability override
Preserve stashes from completed builds
This project is parameterized
Throttle builds
Build Triggers
Build after other projects are built
Save
Apply
Activate Windows
Go to Settings to activate Windows.


Script-Example Config [Jenkins]
localhost:8080/job/Script-Example/configure
Python
Java
YouTube
Maps
News
Gmail
Trinetra | Log in
Conceptual model...
Dashboard
Script-Example
General
Build Triggers
Advanced Project Options
Pipeline
Advanced Project Options
Advanced...
Pipeline
Definition
Pipeline script
Pipeline script
Pipeline script from SCM
try sample Pipeline...
Save
Apply
Activate Windows
Go to Settings to activate Windows.


Script-Example Config [Jenkins]
localhost:8080/job/Script-Example/configure
Dashboard
Script-Example
General
Build Triggers
Advanced Project Options
Pipeline
Advanced Project Options
Advanced...
Pipeline
Definition
Pipeline script
Script
try sample Pipeline...
try sample Pipeline...
Hello World
GitHub + Maven
Scripted Pipeline
Save
Apply
Activate Windows
Go to Settings to activate Windows.


Script-Example Config [Jenkins]
localhost:8080/job/Script-Example/configure
Dashboard
Script-Example
General
Build Triggers
Advanced Project Options
Pipeline
Advanced...
Pipeline
Definition
Pipeline script
Script
pipeline {
agent any
stages {
stage('Hello') {
steps {
echo 'Hello World'
}
}
}
}
Hello World
Save
Apply
Activate Windows
Go to Settings to activate Windows.


Script-Example Config [Jenkins]
localhost:8080/job/Script-Example/configure
Dashboard
Script-Example
General
Build Triggers
Advanced Project Options
Pipeline
Pipeline script
Script
Hello World
pipeline {
agent any
stages {
stage('Hello 1') {
steps {
echo 'Hello 1'
}
}
stage('Hello 2') {
steps {
echo 'Hello 2'
}
}
}
}
Use Groovy Sandbox
Save
Apply
Activate Windows
Go to Settings to activate Windows.


Script-Example Config [Jenkins]
localhost:8080/job/Script-Example/configure
Saved
General
Build Triggers
Advanced Project Options
Pipeline
Pipeline script
Script
Hello World
pipeline {
agent any
stages {
stage('Hello 1') {
steps {
echo 'Hello 1'
}
}
stage('Hello 2') {
steps {
echo 'Hello 2'
}
}
}
}
Use Groovy Sandbox
Save
Apply
Activate Windows
Go to Settings to activate Windows.

# 4. Understanding the Pipeline Script Structure

## Key Elements of the Sample Script

The basic pipeline script has a few key components:

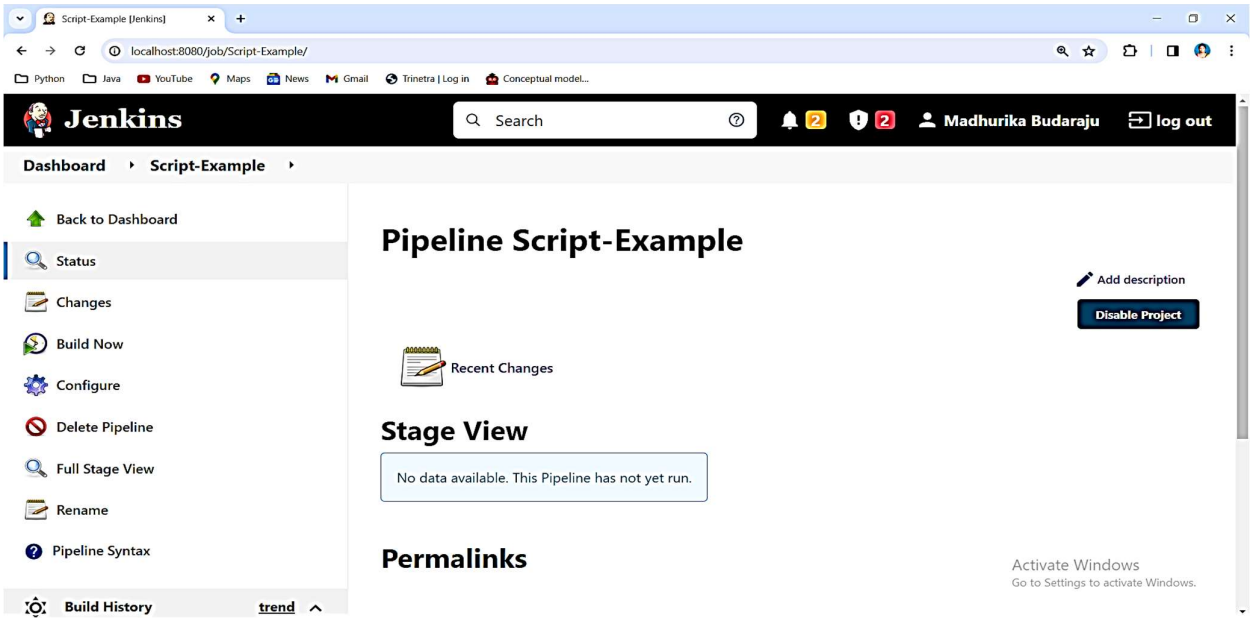
- **Agent**: Defines the machine (or slave) that runs the tasks. In a single-machine setup, the local machine acts as both the master and the slave. In more complex environments, specify the slave's ID in the script.
- **Stages**: The script is divided into stages, each representing a specific step, like building, testing, or deploying the code. Proper indentation is important to maintain clarity and script functionality.

# 5. Expanding the Pipeline with Additional Stages

## Creating Multiple Stages for Workflow Flexibility

For detailed workflows, add multiple stages:

- Define additional stages like "Hello 1" and "Hello 2" to handle different tasks. Each stage can perform a unique function, aiding in the management of complex pipelines.
- Save changes by clicking **Apply** and **Save** to ensure the updated configuration is stored.

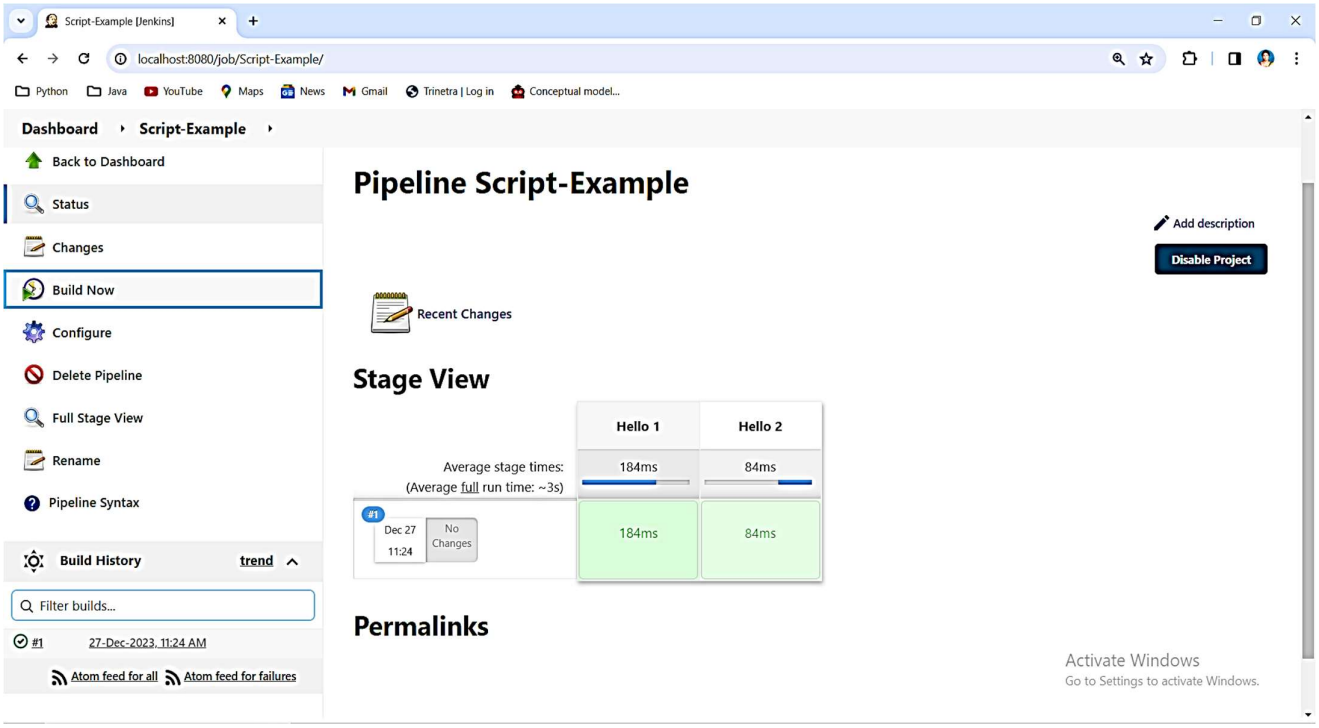


# 6. Running the Pipeline

## Executing the Scripted Pipeline

Once the pipeline script is set:

- Return to the Jenkins dashboard.
- Click **Build Now** to trigger the execution. Jenkins will follow the sequence outlined in the script, processing each stage step-by-step.




# 7. Monitoring Pipeline Progress

## Tracking Execution in Real-Time

Jenkins provides a visual display of the pipeline's execution:

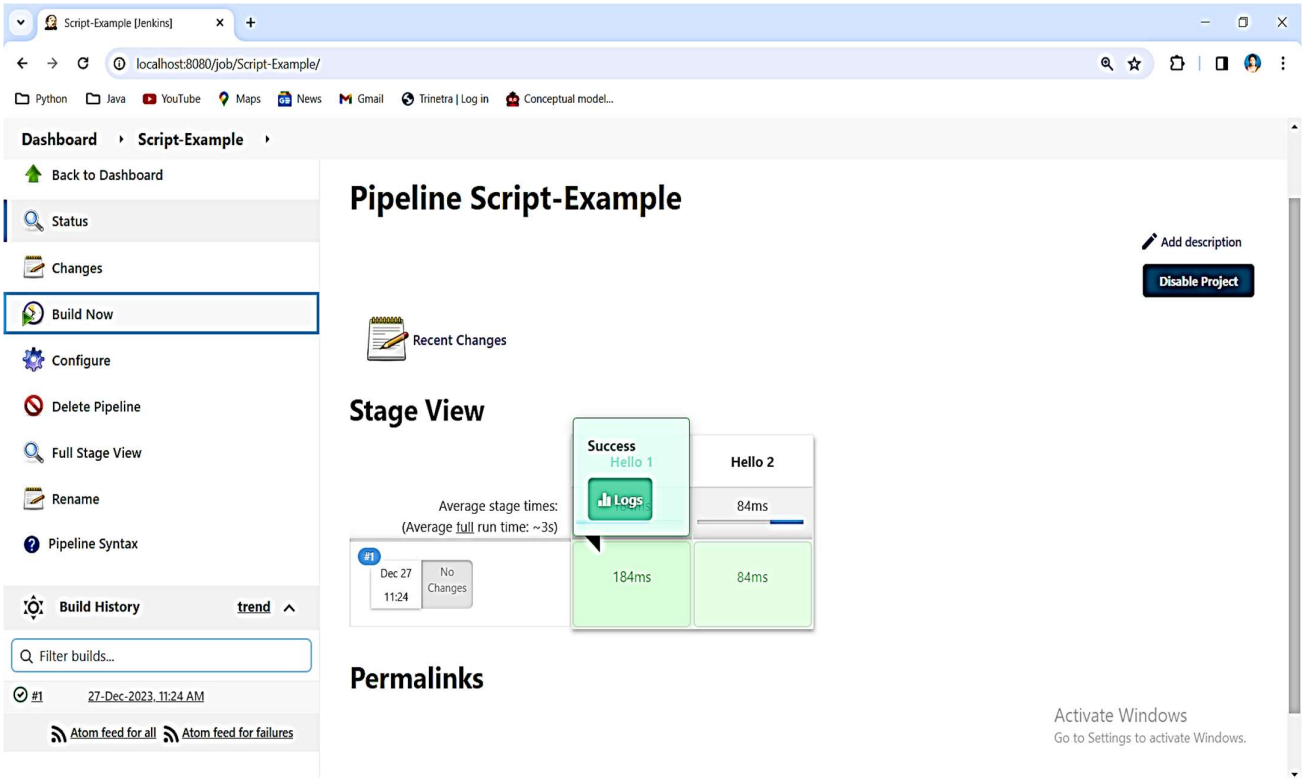
- Each stage's status is clearly shown, indicating whether it's in progress, completed, or has encountered an error.
- Hover over any stage to access more detailed information about its execution.

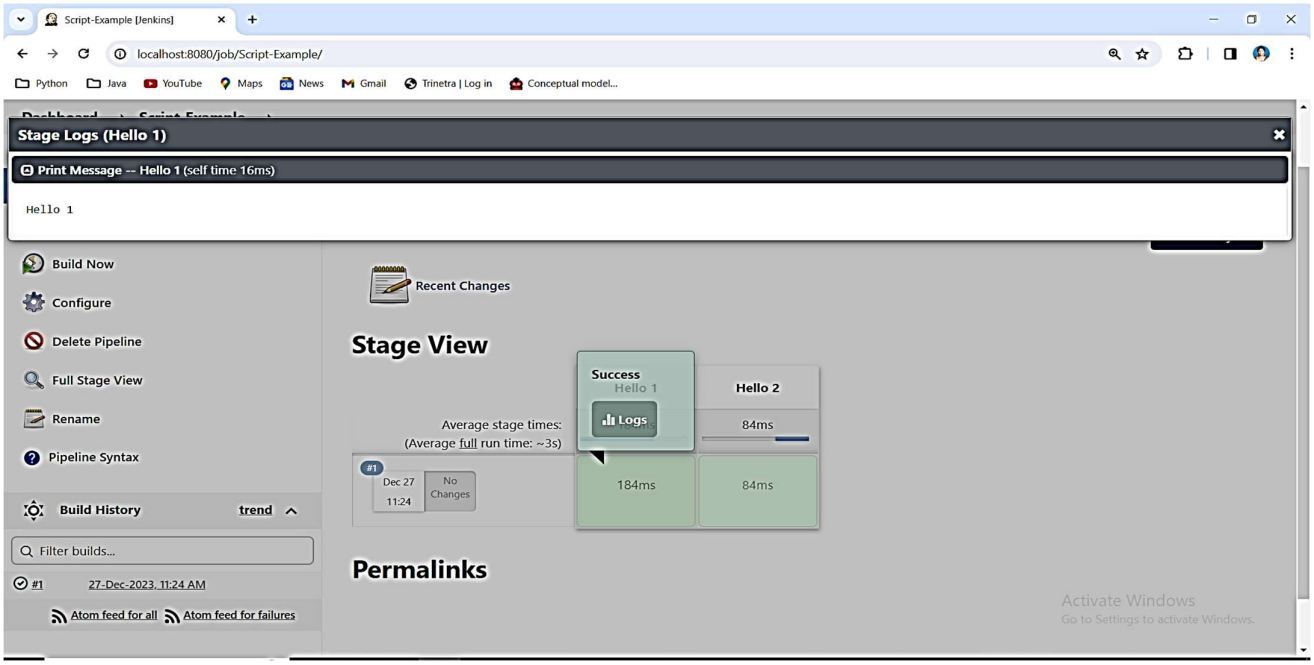
# 8. Reviewing Logs for Each Stage

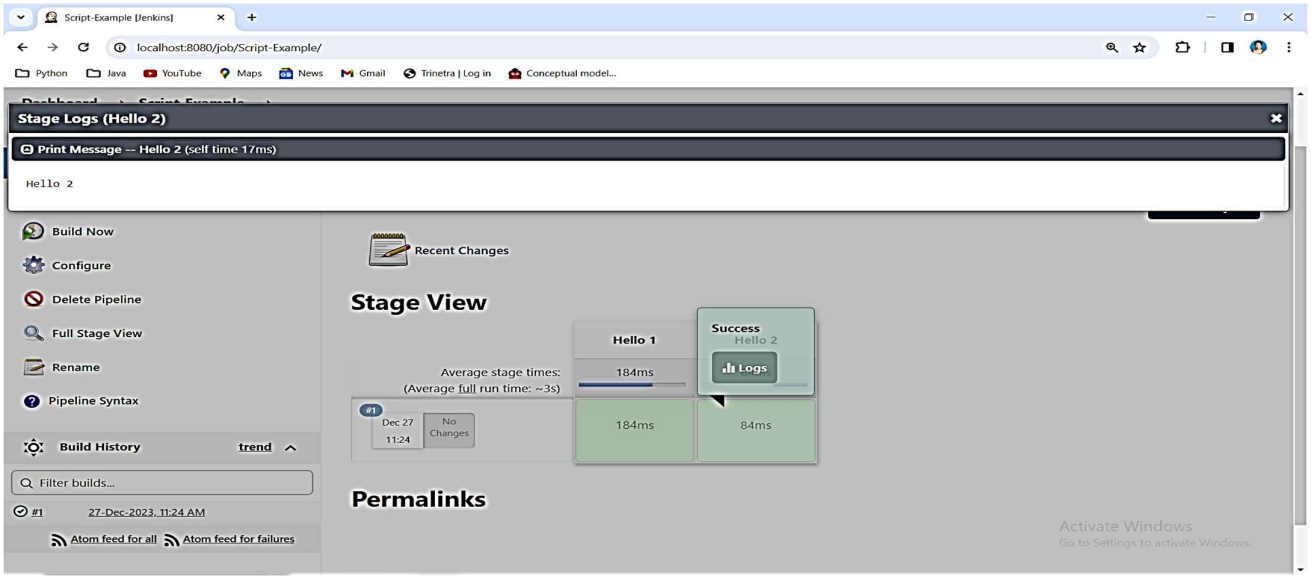
## Accessing Detailed Logs for Troubleshooting

Every stage generates logs that provide a breakdown of actions taken:

- Early stages might show information on environment setup or code compilation.
- Later stages might include the results from testing or packaging processes.
- Analyzing these logs helps to ensure that each part of the pipeline functions correctly and assists in pinpointing any issues.






**Note: If this topic comes as a Long Answer Question, instead of using the basic "Hello World" script, write a scripted pipeline for a Maven project. For a detailed example, refer to the video of LAB EXERCISE - 6-C: BUILDING THE CI/CD SCRIPTED PIPELINE USING JENKINS FOR A MAVEN JAVA PROJECT WITH POLL SCM. This example will help explain the Jenkins scripted pipeline in more detail.**