SOFTWARE ENGINEERING LAB

EXERCISE – 1

TOPIC – 1

INSTALLATION OF STARUML, GIT BASH AND GITHUB ACCOUNT CREATION

1. Introduction to the Lab Exercise

- **Topic:**
  This lab exercise focuses on setting up essential tools for software engineering, specifically **StarUML**, **Git Bash**, and **GitHub**. These tools are important for designing, coding, and collaborating on software projects.

NOTE - At each step of the installation process and account creation, take screenshots and save them in a Word document. Ensure the document is named with the student's roll number before uploading it to Tesselator for evaluation.

2. What is the Use of StarUML?

- **Explanation:**
  StarUML is a software tool used to create diagrams that help in designing and understanding complex systems. It is especially useful in software development.
- **Purpose:**
  It allows you to visually map out how different parts of a system or application will work together, making it easier to plan, communicate, and build the system effectively.
- **Analogy:**
  Think of StarUML as a drawing tool for software engineers to sketch out their ideas.

3. How to Install StarUML

- **Steps:**
    1. **Search for StarUML:**

       Go to Google, search for "StarUML", and click on the official link.

    2. **Download StarUML:**

       Visit the site https://staruml.io/ and download StarUML for Windows or Mac.

    3. **Installation Interface:**

       Once installed, explore the StarUML interface to get familiar with its features.

4. What is the Use of Git Bash?

- **Explanation:**
  Git Bash is an application that lets you use Git commands on your computer.
- **Git:**
  A tool for tracking changes in files, mainly used for managing code in software projects.
- **Bash:**
  A command-line tool that allows you to type in commands to perform various tasks on your computer.
- **Purpose:**
  Git Bash combines Git and Bash, making it easier to manage your code and work with files and projects. It's like a smart helper for keeping track of changes and collaborating with others.

5. How to Install Git Bash

- **Steps:**
    1. **Search for Git Bash:**

       Go to Google, search for "Git Bash", and click on the official link.

2. **Download Git Bash:**

   Visit the site https://git-scm.com/downloads and download Git for Windows or Mac.

3. **Installation Interface:**

   After installation, familiarize yourself with the Git Bash interface.

6. What is the Use of GitHub?

- **Explanation:**
  GitHub is a website where people can store and share their computer code.
- **Purpose:**
  It acts as a big online folder where you can keep your work, make changes, and collaborate with others. It helps you and your team work on a project together, track changes, and keep everything organized.

7. How to Create an Account in GitHub

- **Steps:**
  1. **Search for GitHub:**

     Go to Google, search for "GitHub", and click on the official link.

  2. **Sign Up:**

     Visit the site https://github.com/ and click on "Sign Up" to create a new account.

  3. **Complete Signup:**

     Follow the steps provided to complete the signup process.

  4. **Explore Dashboard:**

     After account creation, explore the GitHub dashboard to get familiar with its features.

SOFTWARE ENGINEERING LAB

EXERCISE – 1

TOPIC – 2

IEEE – SRS DOCUMENT

1. Introduction to SRS (Software Requirements Specification)

Note – The finalized SRS document for your project should be saved as a Word file, named with your roll number. This file must then be uploaded to Tesselator for evaluation.

- **What is an SRS Document?**
  - **SRS** stands for **Software Requirements Specification**. It's a detailed document that outlines exactly what a software product should do.
  - **Purpose:**
    Think of it as a blueprint for software development, similar to how an architect's plan guides the construction of a building.
  - **Example:**
    If you were building a house, the blueprint would show the size and layout of each room, where the doors and windows go, and what materials to use. Similarly, an SRS document lays out every detail of how the software should function, what features it should have, and how it should look.

2. Why is an SRS Document Needed?

- **Importance of an SRS Document:**

1. **Clarity and Understanding:** It ensures that everyone involved in the project—developers, designers, and stakeholders—understands what the software is supposed to do.
2. **Guides Development:** The SRS document serves as a guide for the development team, helping them build the software correctly.
3. **Communication Tool:** It acts as a communication bridge between different teams and stakeholders, making sure everyone is on the same page.

4. **Manages Changes:** If there are changes needed during development, the SRS document helps manage those changes without confusion.
5. **Helps in Testing:** The SRS document also helps testers by providing clear criteria on what the software should do, so they can check if it meets all the requirements.

3. What is IEEE - SRS?

- **Definition:**
  - **IEEE SRS** refers to the **Software Requirements Specification** as defined by the **Institute of Electrical and Electronics Engineers (IEEE)**.
  - **Purpose:**
    The IEEE SRS is a standardized format for writing an SRS document, ensuring that all important aspects of the software are covered consistently.
  - **Example:**
    Just like a recipe book has a standard format for listing ingredients and steps, the IEEE SRS template ensures that all software projects follow a standard way of documenting requirements. This makes it easier for everyone to understand and follow the document.

Software Requirements Specification (SRS) Template

1. **Introduction**
   - **Purpose**: Describes the product that the SRS covers, including the specific version or part of the system.

     Example: "This document outlines the requirements for Version 2.0 of our customer management software."

   - **Document Conventions**: Lists any standards or formatting used in the document, such as fonts, colors, or symbols that indicate priorities.

     Example: "Critical requirements are highlighted in bold and marked with an asterisk (*)."

   - **Intended Audience**: Identifies who should read the SRS, such as developers, managers, or testers, and suggests reading sequences.

Example: "This document is intended for developers, testers, and project managers to ensure a shared understanding of system requirements."

- **Product Scope**: Provides a brief description of the software, its purpose, and how it aligns with business goals.

  Example: "The software aims to automate inventory management, enhancing operational efficiency."

- **References**: Lists other documents or resources related to the SRS.

  Example: "Refer to the User Interface Style Guide, Version 1.1, for design standards."

2. **Overall Description**

- **Product Perspective**: Explains the software's background and its relationship to other systems or components.

  Example: "This tool is an upgrade to the existing analytics module, designed to integrate with our sales platform."

- **Product Functions**: Summarizes the main features the product will perform, organized in a simple list.

  Example: "Key functions include data import, user authentication, and report generation."

- **User Classes and Characteristics**: Defines different user types and their key characteristics, such as skill level or security access.

  Example: "User classes include Admins, who have full access, and Regular Users, who have limited access."

- **Operating Environment**: Details the hardware and software environments where the software will run.

  Example: "The system will operate on Linux servers with Java 11 and MySQL databases."

- **Design and Implementation Constraints**: Outlines any limitations or specific requirements that affect design choices.

Example: "The system must comply with GDPR regulations and use Python for backend development."

- **User Documentation**: Lists the documentation that will be provided to users, such as manuals or online help guides.

  Example: "A user manual and an interactive help guide will be included."

- **Assumptions and Dependencies**: Identifies assumptions that could impact the project, such as reliance on third-party components.

  Example: "The system assumes the availability of an existing LDAP server for user authentication."

3. **External Interface Requirements**

- **User Interfaces**: Describes how the software will interact with users, including screen layouts and navigation standards.

  Example: "Each screen will have a consistent layout with navigation buttons at the top."

- **Hardware Interfaces**: Details how the software will connect with hardware, such as printers or sensors.

  Example: "The software will communicate with RFID scanners via Bluetooth."

- **Software Interfaces**: Defines connections between the product and other software components, including databases and APIs.

  Example: "The software will interact with REST APIs to fetch real-time data."

- **Communications Interfaces**: Specifies requirements for any communication-related functions, like protocols or security needs.

  Example: "The system will use SSL/TLS for secure data transmission over the network."

4. **System Features**

- **Feature Description and Priority**: Provides brief descriptions of each feature and its importance level (high, medium, low).

Example: "User authentication is a high-priority feature to ensure secure access."

- **Stimulus/Response Sequences**: Lists the expected user actions and corresponding system responses.

  Example: "When the user submits a form, the system will validate the data and display a success message."

- **Functional Requirements**: Details the specific tasks the software must perform for each feature.

  Example: "REQ-1: The system must allow users to reset their passwords via email verification."

5. **Other Nonfunctional Requirements**

- **Performance Requirements**: Specifies how the software should perform under different conditions, such as speed and scalability.

  Example: "The application must process up to 500 transactions per second."

- **Safety Requirements**: Defines measures to prevent harm or data loss.

  Example: "The software must include auto-save functionality to protect user data during unexpected shutdowns."

- **Security Requirements**: Lists the security protocols that must be followed, such as data encryption or user authentication.

  Example: "User data must be encrypted using AES-256 encryption standards."

- **Software Quality Attributes**: Describes attributes like usability, reliability, and maintainability.

  Example: "The system should have a 99.9% uptime to ensure high availability."

- **Business Rules**: Outlines operational rules that influence software behaviour.

  Example: "Only users with the admin role can approve transactions over $5,000."

6. **Annexures**

- **Appendix A: Glossary**: Defines terms, abbreviations, and acronyms used throughout the SRS.

Example: "API: Application Programming Interface."

- **Appendix B: Analysis Models**: Includes relevant analysis diagrams, such as data flow diagrams or class diagrams.

Example: "See Figure B.1 for the system's state-transition diagram."

- **Appendix C: To Be Determined List**: Tracks items that are still pending decisions or additional information.

Example: "TBD-1: Decide on the final database technology."

SOFTWARE ENGINEERING LAB

EXERCISE – 3

TOPIC – 1

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:

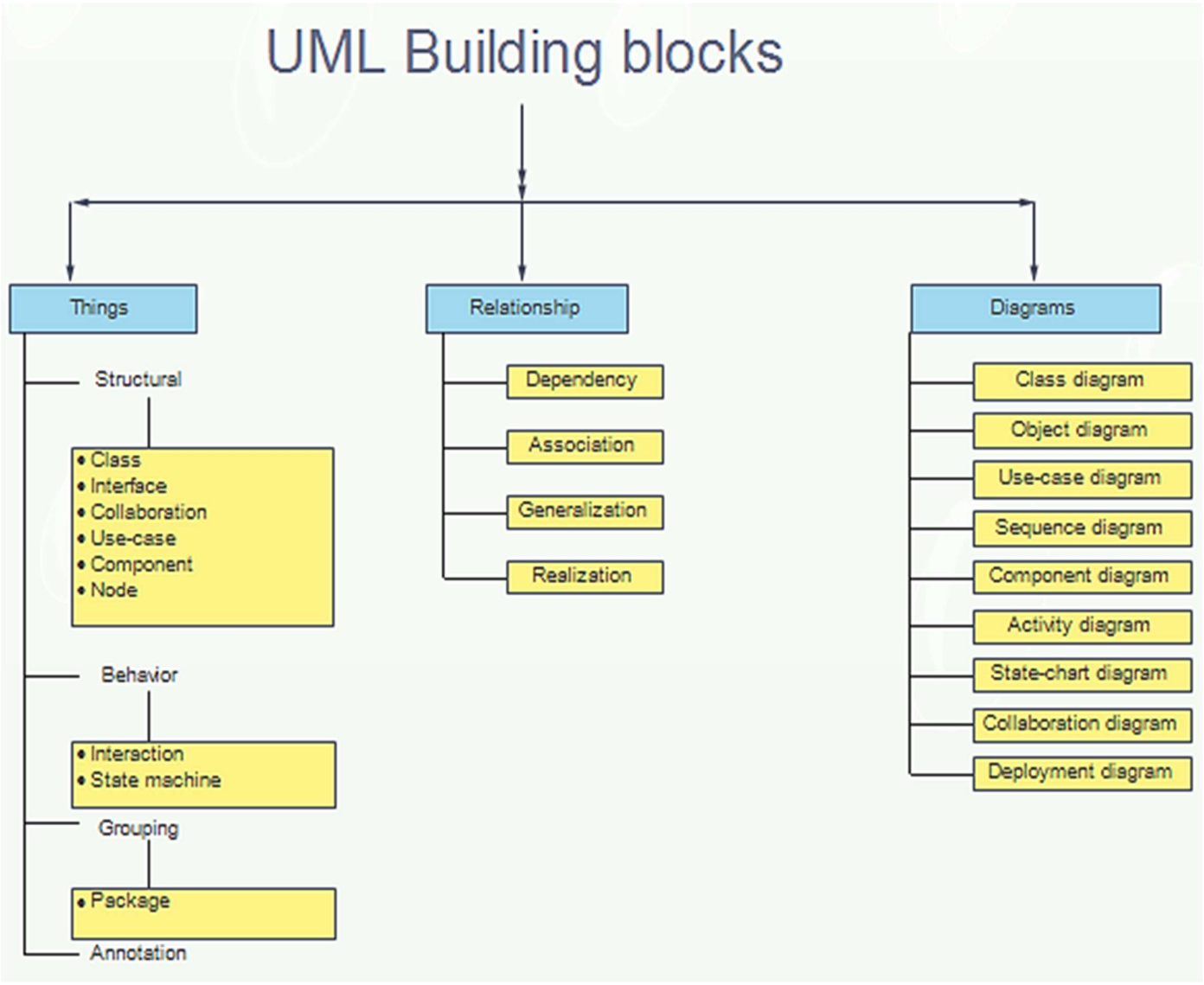
- **Rational Rose** – A well-known tool for drawing UML diagrams.
- **Enterprise Architect** – Helps create and organize diagrams.
- **Microsoft Visio** – A diagramming tool that can also make UML diagrams.
- **Lucidchart** – A web tool for drawing diagrams.
- **StarUML** and **ArgoUML** – Free tools 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.



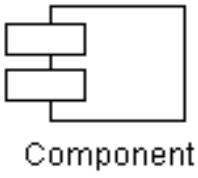
Parts of UML (Things)

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

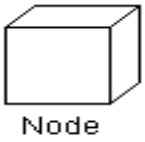
1. **Structural Things (Parts That Don't Change)** These describe the solid, unchanging parts of the system, like the structure of a building.

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

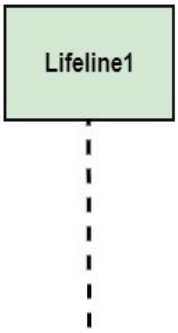
   - **Interface**: A list of rules or actions that certain objects must follow.
     - *Example: A "Drivable" interface makes sure that any class that uses it, like a Car, has a "drive" function.*
     - *Symbol:* 

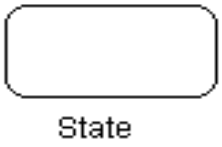
   - **Component**: A piece of the system that performs a specific job.
     - *Example: A "Payment" component handles all the payments in the system.*
     - *Symbol:* 

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

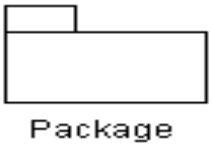
- *Symbol:* 

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.

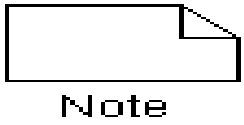
    - **Interaction**: Shows how different parts of the system talk to each other.
        - *Example: A customer asks for information from the system, and the system sends back the details.*
        - *Symbol:* 

    - **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:

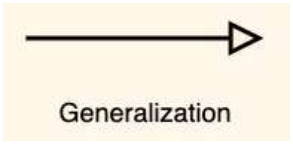
- **Dependency**: When one thing needs another thing to work.
  - *Example: A "Car" class needs an "Engine" class to function properly.*
  - *Symbol:* 

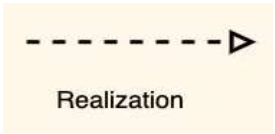
- **Association**: A simple connection between two things.
  - *Example: A teacher teaches a student, and the student learns from the teacher. They are associated with each other.*
  - *Symbol:*

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.

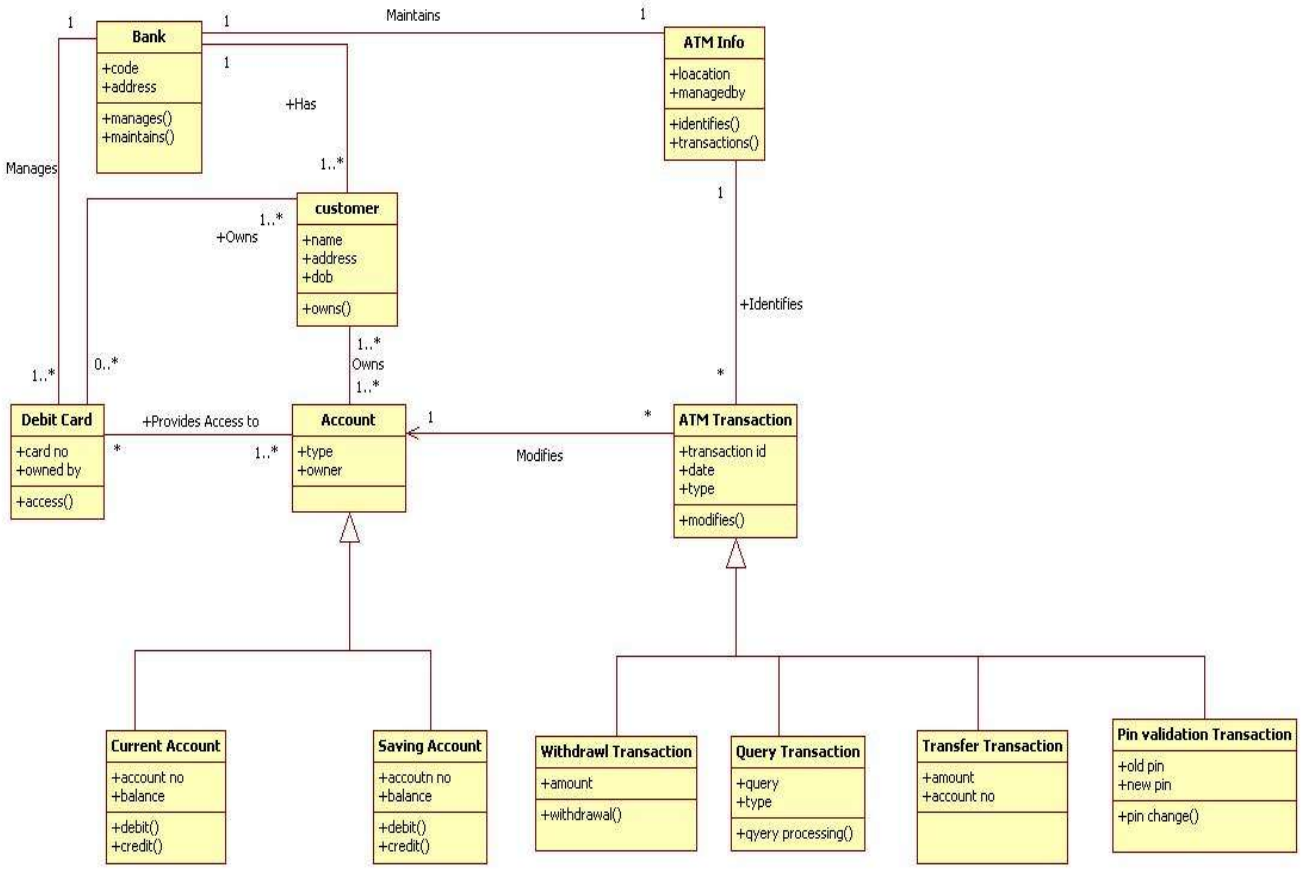
- **Generalization**: This shows inheritance, where one thing is a more general version of another.
    - *Example: A "Car" is a general category, and "SUV" and "Sedan" are more specific types of cars.*
    - *Symbol:* 

- **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

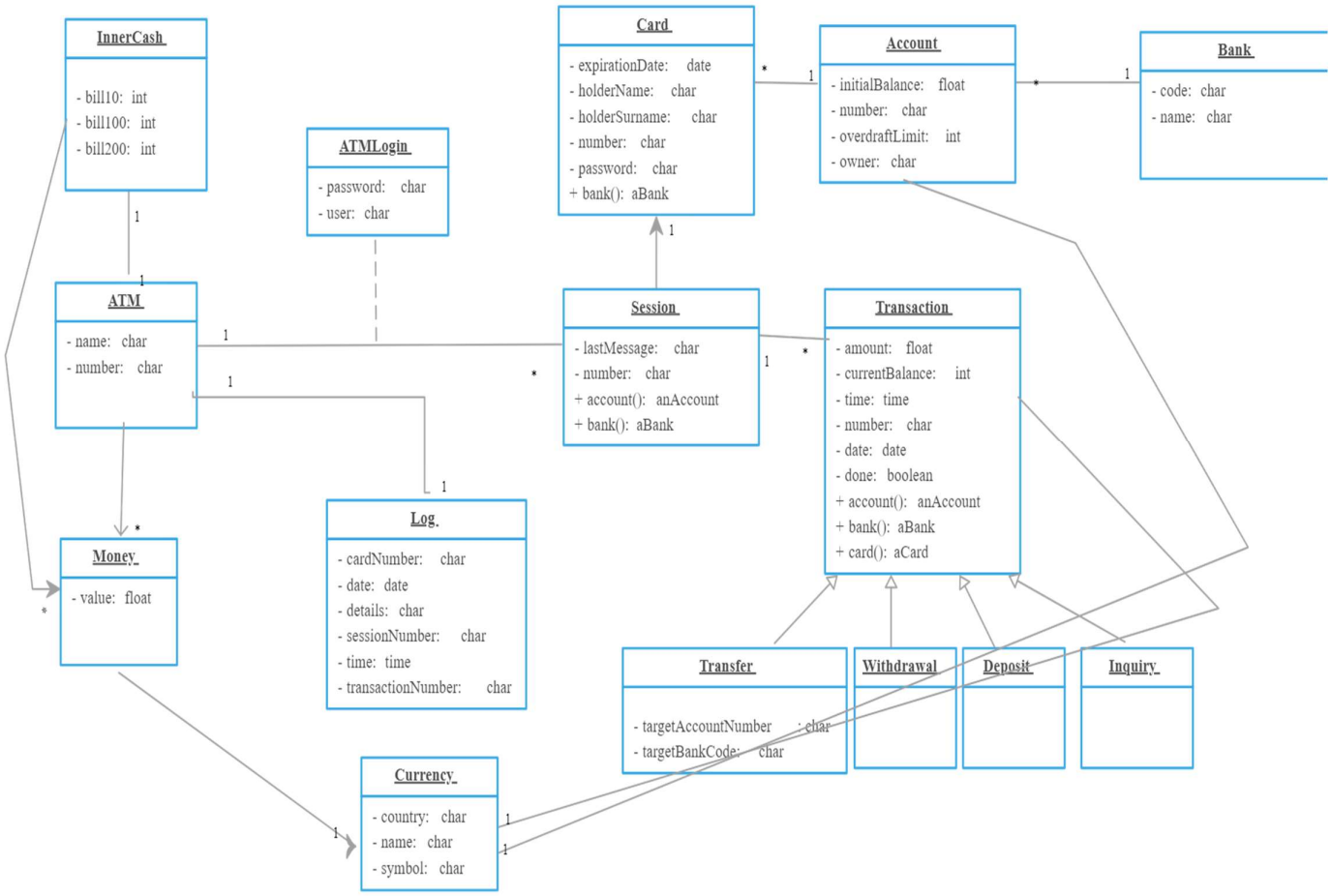
UML uses different types of diagrams to show various aspects of a system. Each diagram has its own way of explaining how the system works.

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***



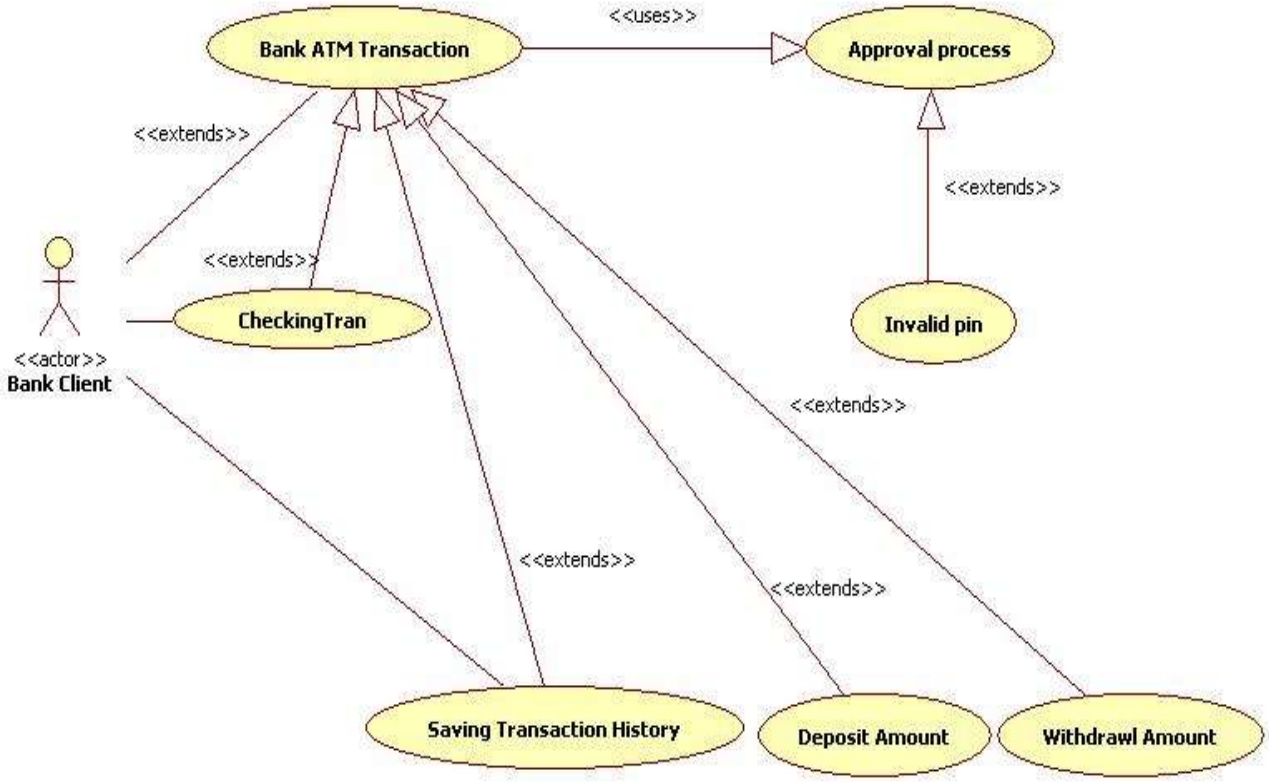
2. Object Diagram

- 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***



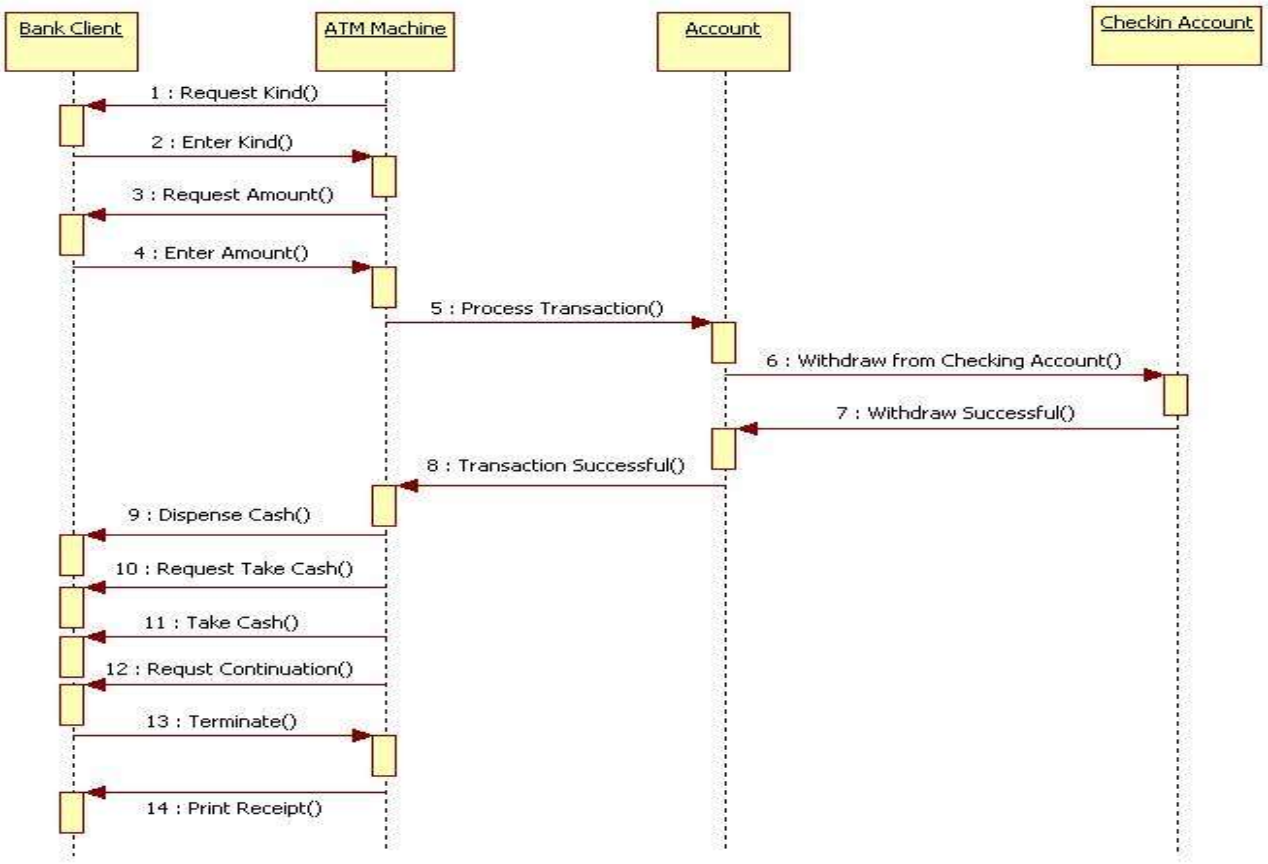
3. Use-Case Diagram

- 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***



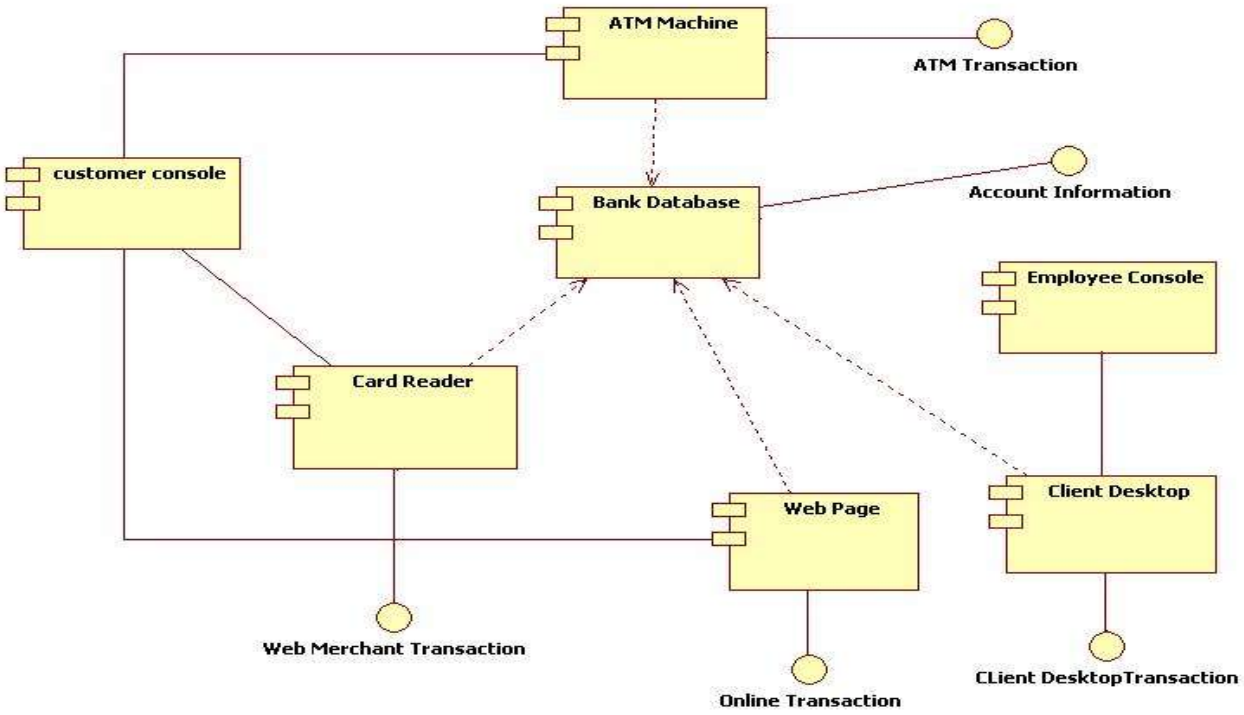
4. Sequence Diagram

- 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***



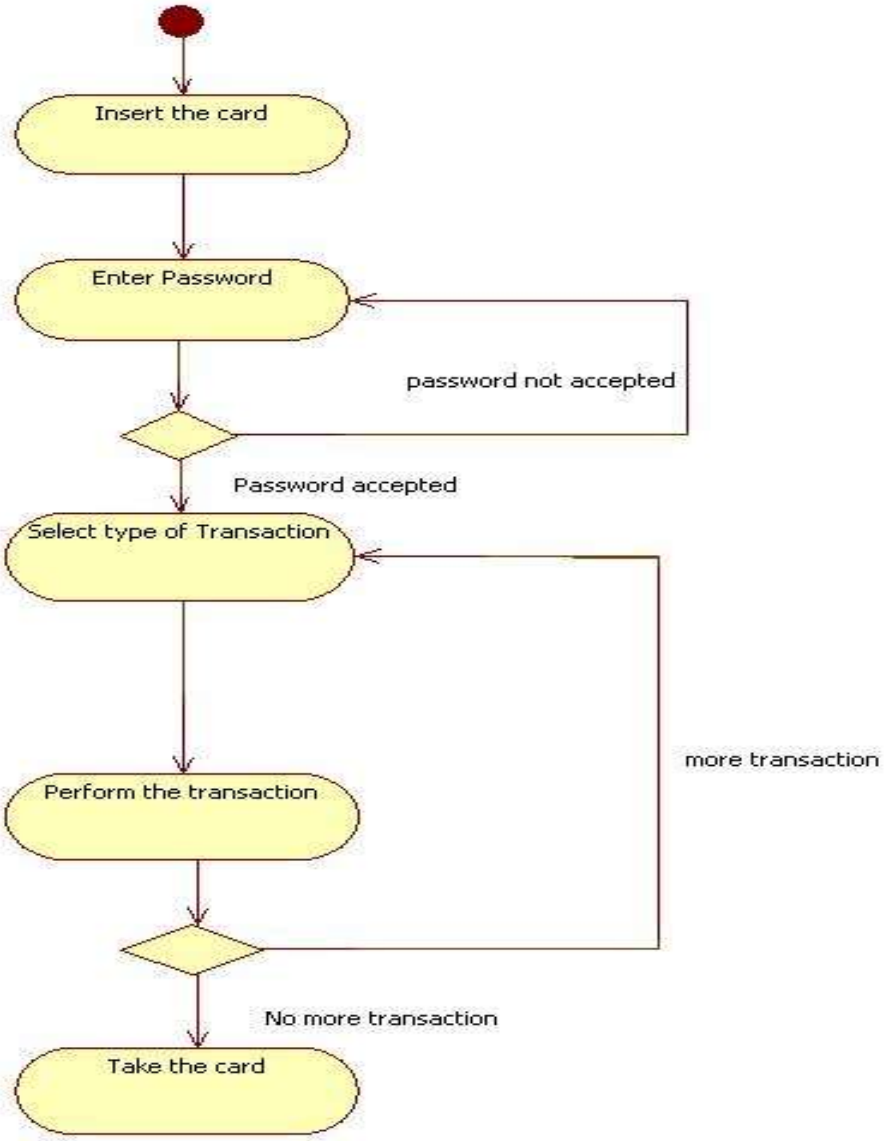
5. Component Diagram

- 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***



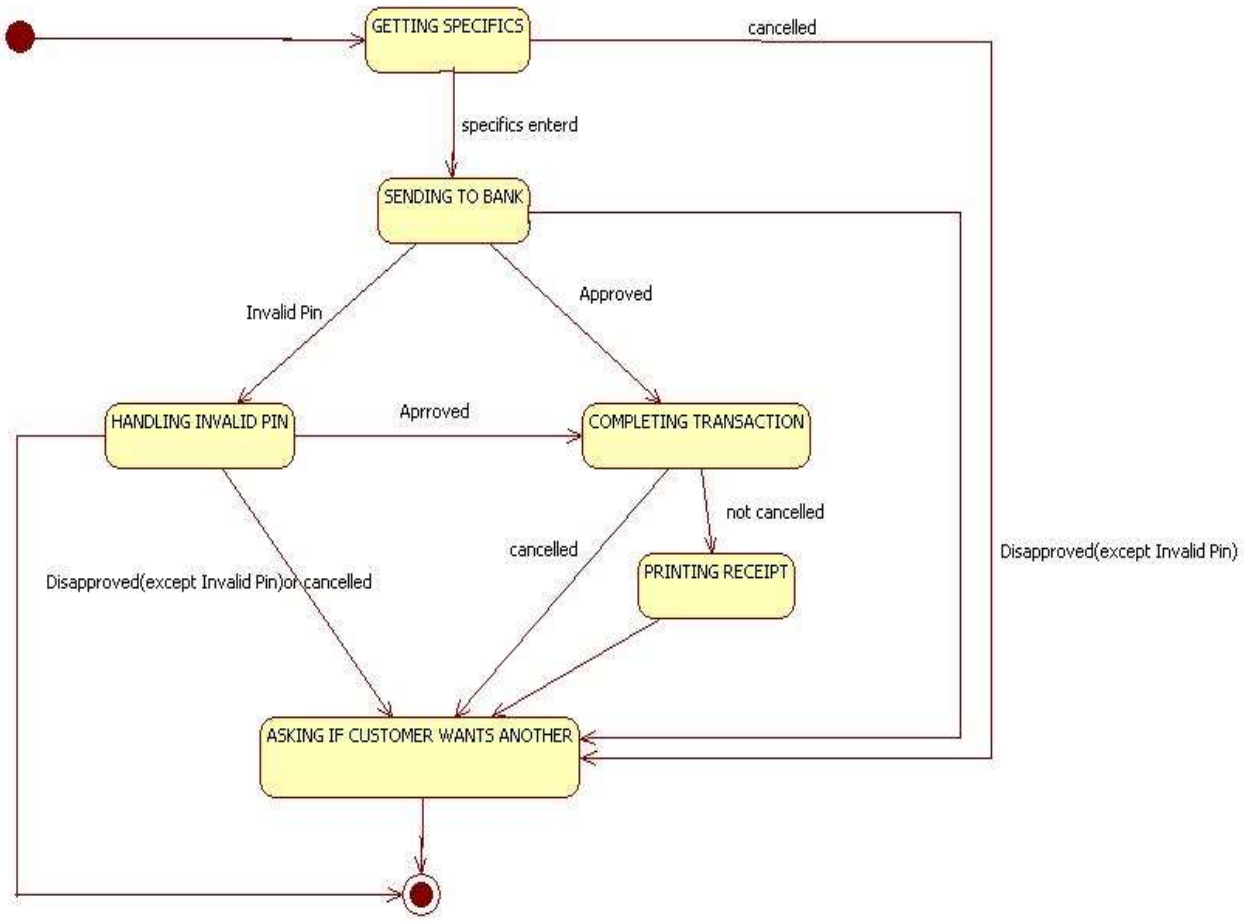
6. Activity Diagram

- 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***



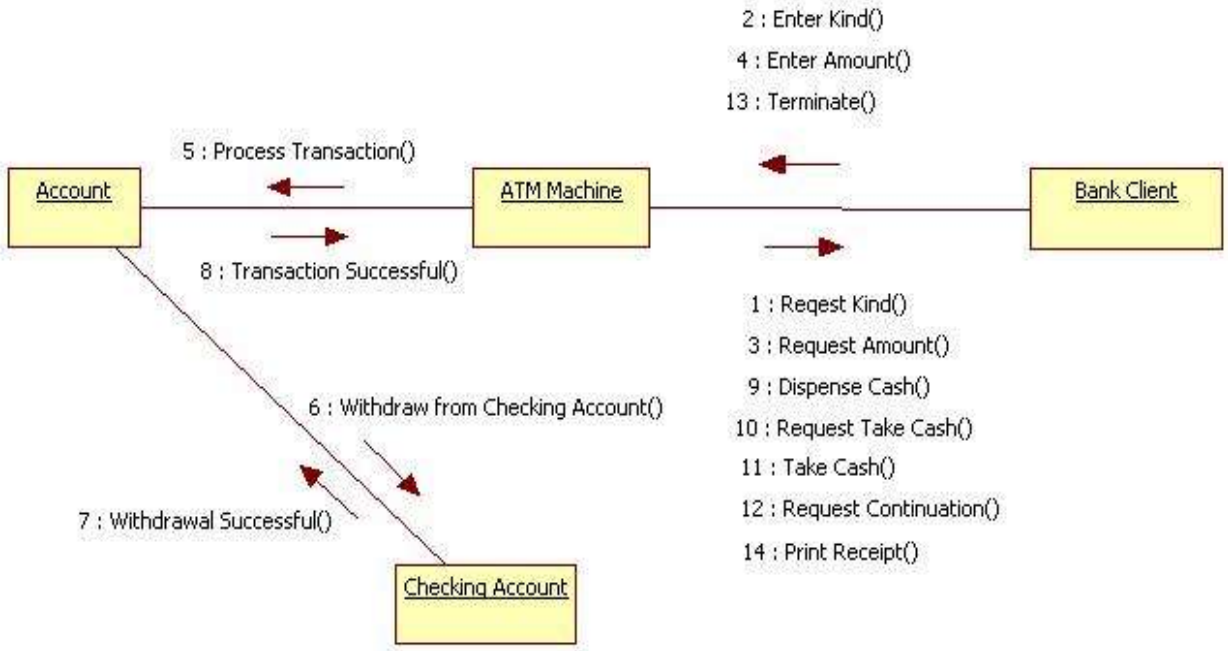
7. State-Chart Diagram

- 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***



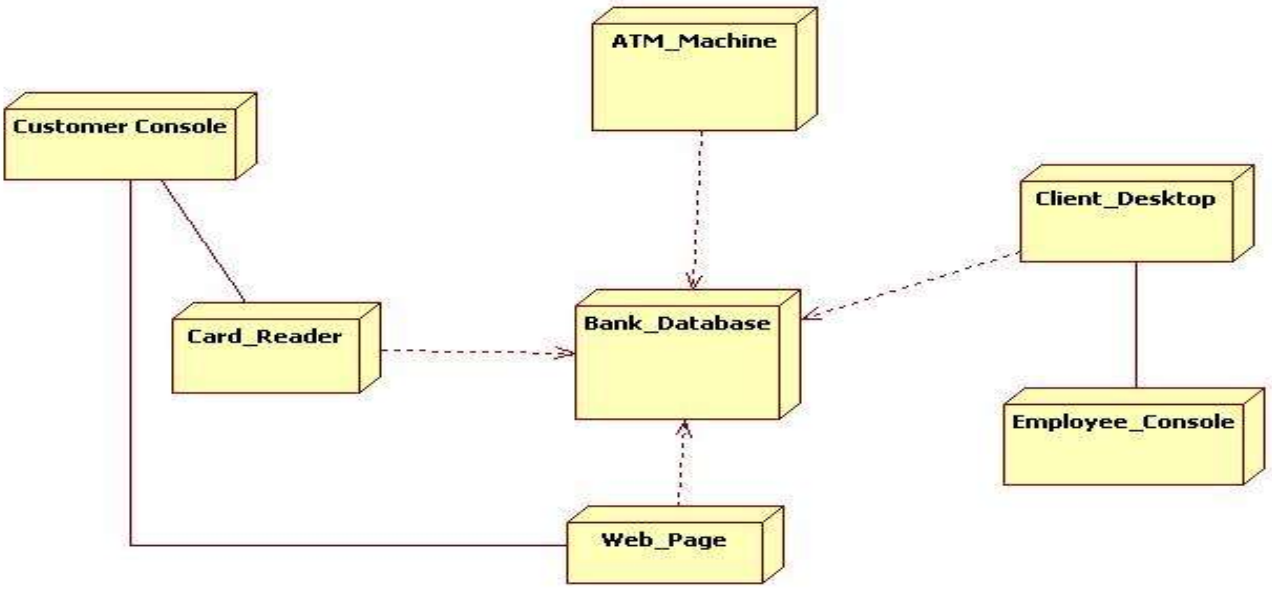
8. Collaboration Diagram

- 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 LAB

EXERCISE – 3

TOPIC – 2

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:

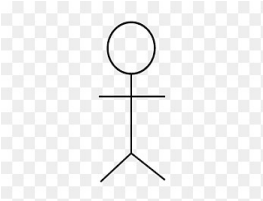
1. **Shows the Actors**: It identifies who is using the system (like a customer or another system).
2. **Describes the Actions**: It explains what actions or tasks the user can do with the system (like withdrawing money from an ATM).
3. **Defines the System's Boundaries**: It clearly shows what is **inside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

Key Parts of a Use Case Diagram

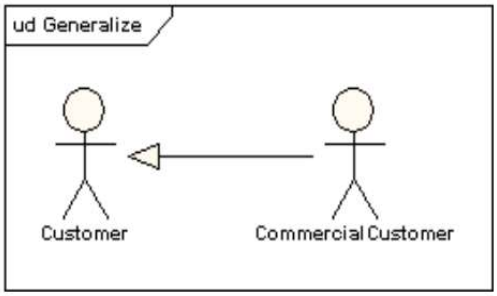
There are several important parts (symbols) in a Use Case Diagram. Each one plays a specific role in explaining how the system works.

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.



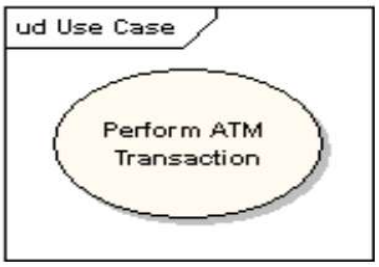
- **Generalization of Actors:** Actors can have a hierarchical relationship. One actor can "generalize" another, meaning one type of actor shares similarities with another but may also have additional roles.

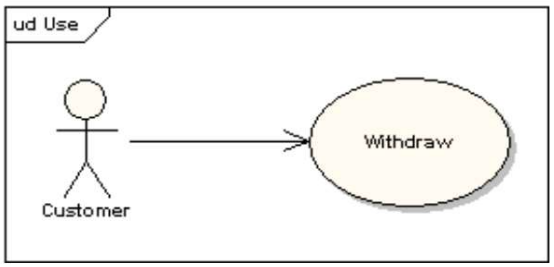


- In the image, we see a Customer Actor and a Commercial Customer actor. A Commercial Customer is a type of customer, but with more specific responsibilities. The arrow between them shows this generalization, meaning all commercial customers are customers, but not all customers are commercial customers.

2. Use Case

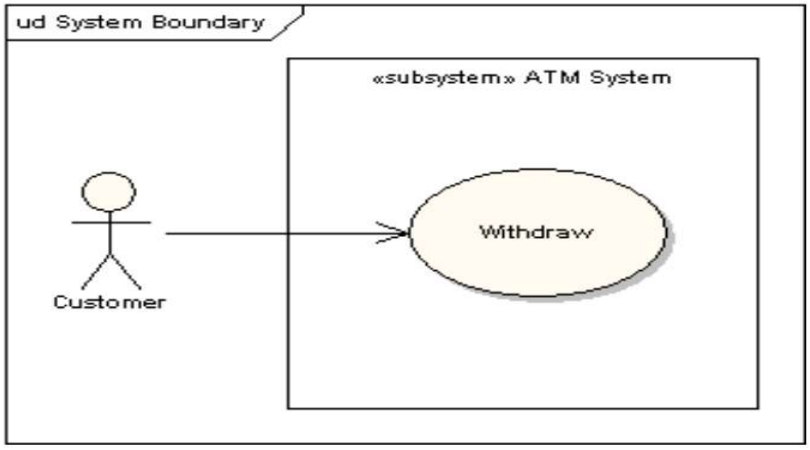
- **Definition**: A use case is an action or task that the actor can do with the system. It is like a function or activity the system provides to the user.
- **Example**: For an ATM system, use cases could include tasks like "Withdraw Cash," "Check Balance," or "Transfer Funds."
- **Symbol**: A use case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



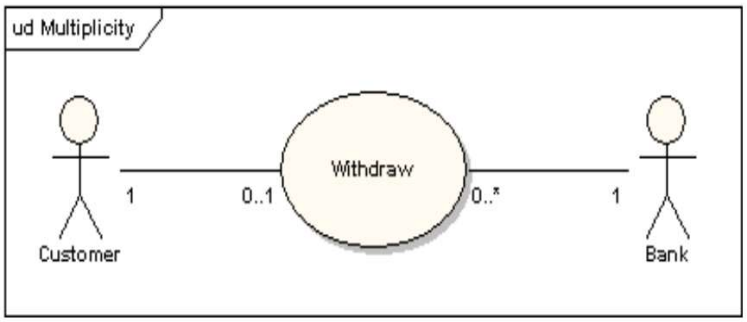


3. System Boundary

- **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.

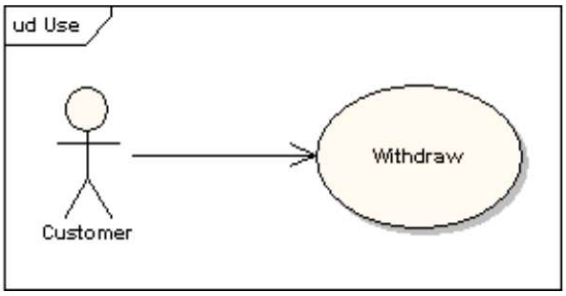


4. Associations (Lines): An Association is a straight line that shows a connection between an actor (a person or system) and a use case (a task or action). It means the actor is involved in the task, but it doesn't say who starts the action.



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.

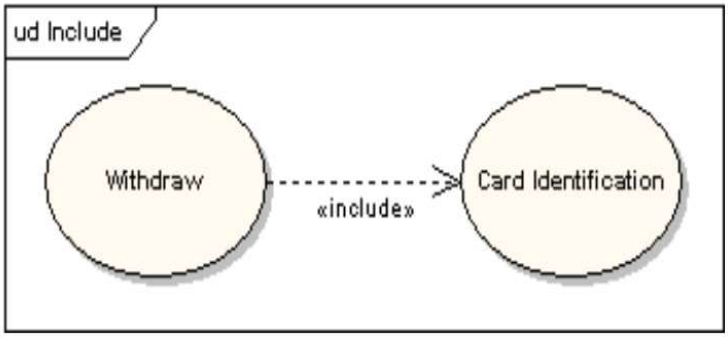
Example: If a Customer is linked to a Purchase task with multiplicity 0..1, it means they can either make one purchase or none at all. If it's 1..*, it means the customer can make as many purchases as they want.

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

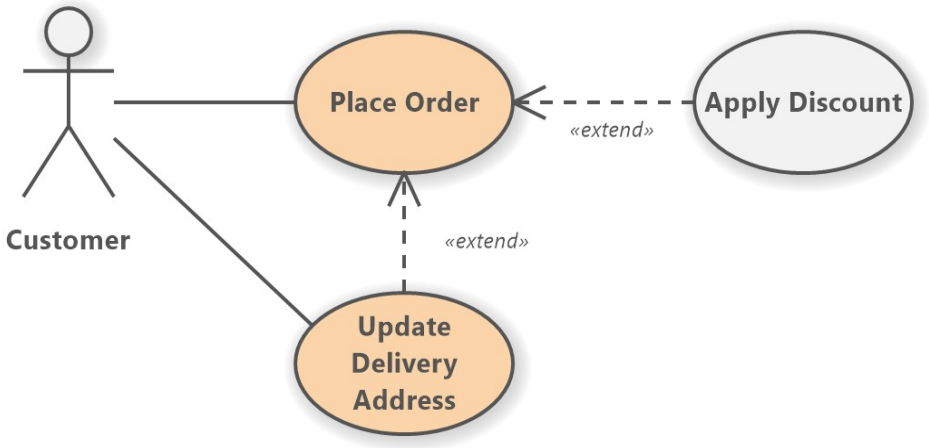
Sometimes, use cases can be related to each other in special ways. These relationships help make the diagram clearer.

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.

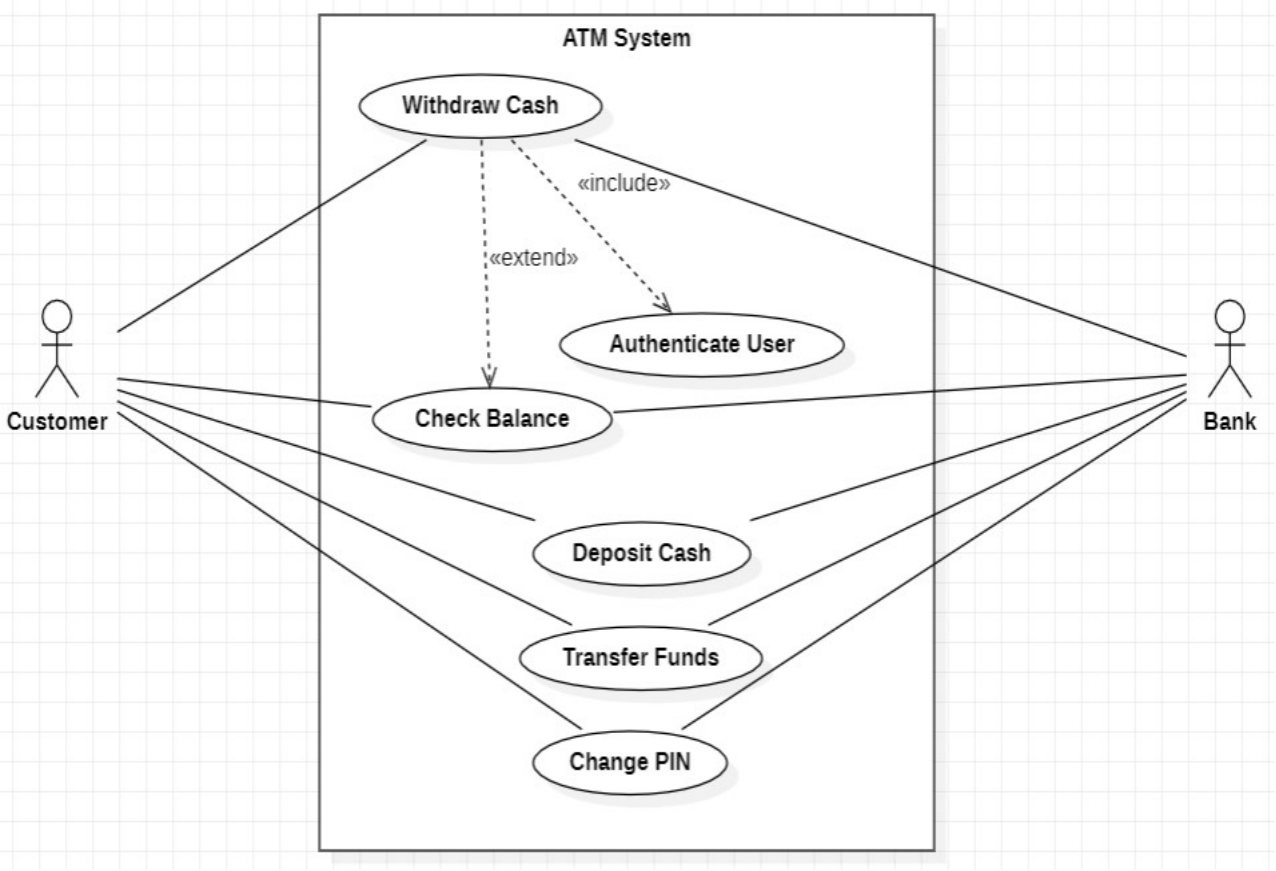
Step 3: Draw the Diagram

- 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 LAB

EXERCISE – 3

TOPIC – 3

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:

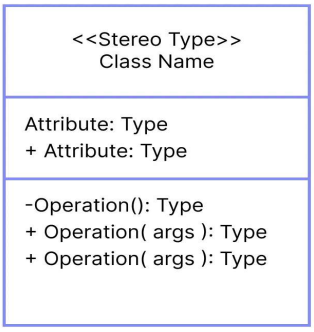
- **Classes:** The blueprint for objects.
- **Attributes:** Characteristics of the classes.
- **Methods:** The behaviors or actions associated with the class.
- **Relationships:** How the classes interact with each other.

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

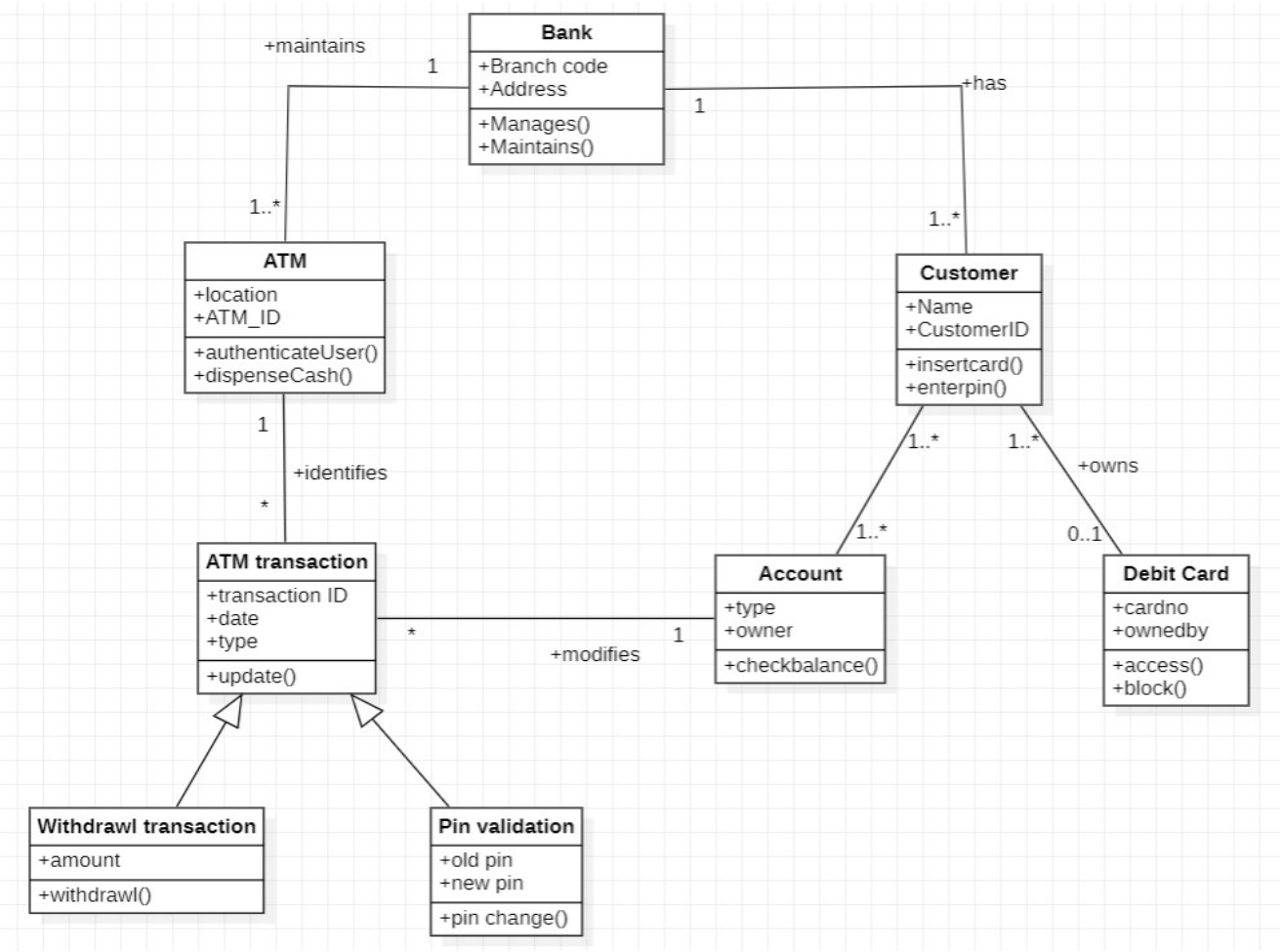
To illustrate how class diagrams are used, consider an ATM system. When modeling such a system, you break it down into classes, define their attributes and methods, and specify how they interact with each other.

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

EXERCISE – 3

TOPIC – 4

UML DIAGRAMS – SEQUENCE

Sequence Diagram

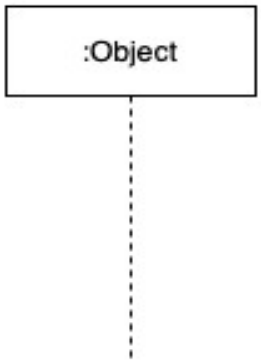
- A sequence diagram is a type of UML diagram that illustrates how objects in a system interact with each other over time.
- It shows the order in which messages are sent between objects, helping visualize the sequence of interactions in a system.
- This is particularly useful when trying to understand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

2. Components of a Sequence Diagram

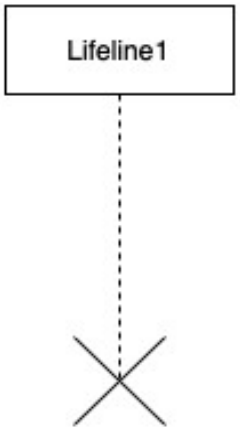
1. **Actors:**
   - Actors represent external entities interacting with the system.
   - These could be people (like users) or other systems (like a database or another software system).
   - In a sequence diagram for an ATM system, for example, the Customer would be an actor.

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.

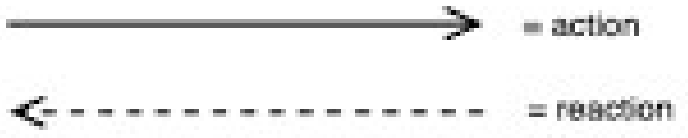
- For example, in an ATM system, objects could include:
  - The ATM machine itself.
  - The Bank Database that stores the customer's account information.



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).

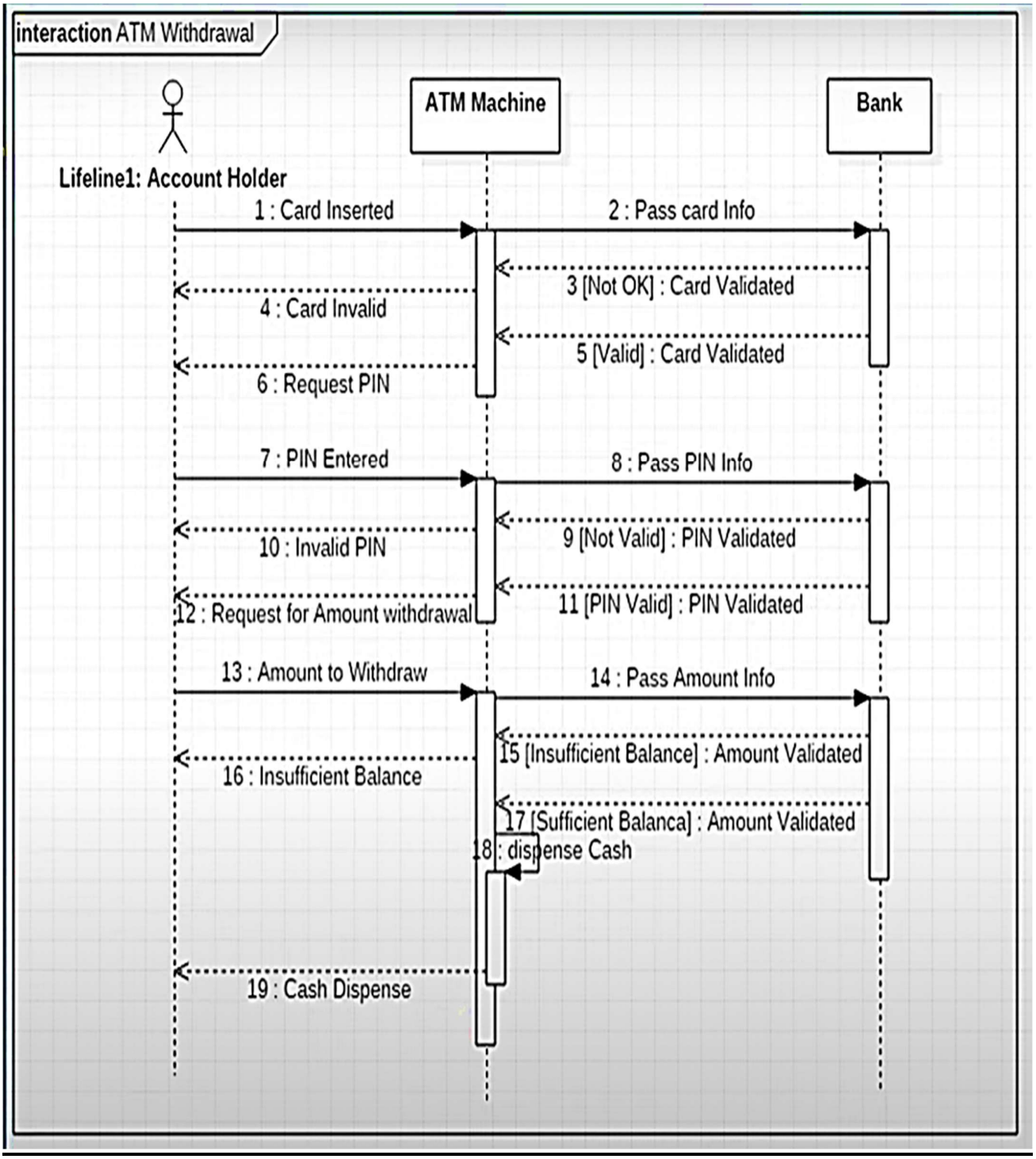
Objects:

- 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 LAB

EXERCISE – 3

TOPIC – 5

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?

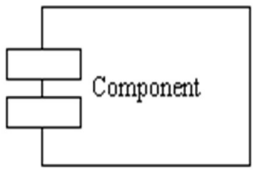
- **Components** are the **building blocks** of a system, just like parts of a machine.
- Each component is a **self-contained unit** that does a specific job.
- Components **communicate** with each other through **interfaces**, which are defined connection points that allow components to work together.
- **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

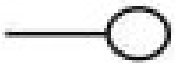
- **Reusable**: Components can often be used in different parts of a system or even in **other systems**. This reduces duplication and saves development time.
- **Replaceable**: If a component stops working or needs to be updated, it can be replaced without affecting the other parts of the system.
-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

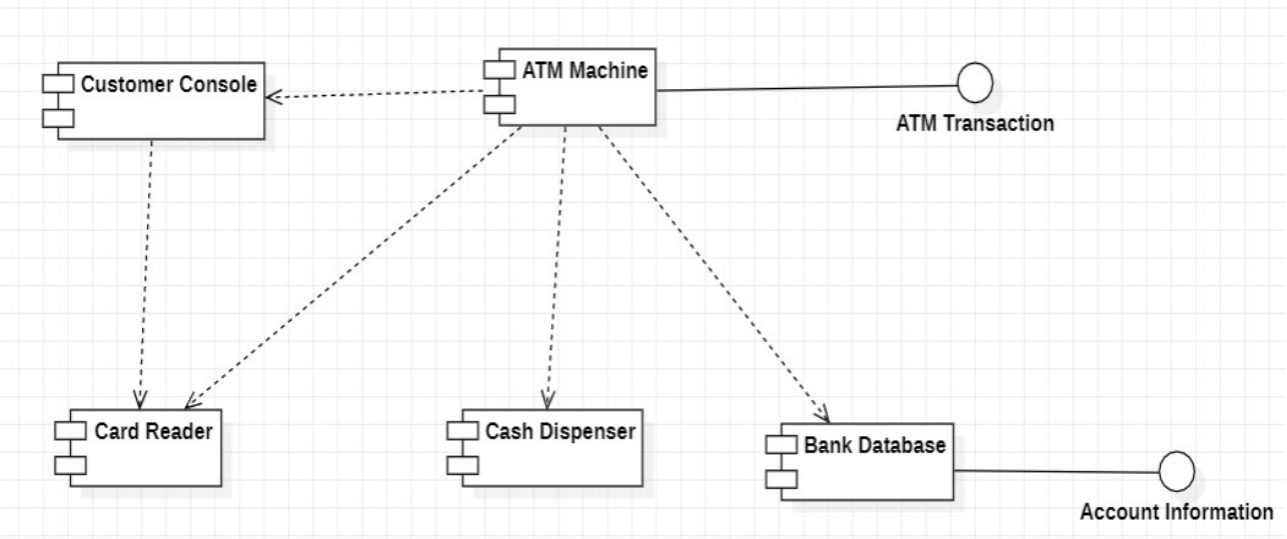
Example: Key Components of an ATM System

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 LAB

EXERCISE – 5

TOPIC – 2

CREATING MAVEN JAVA PROJECT USING ECLIPSE AND PUSH INTO TO GITHUB

Note: At every step take screenshots and save in a document

1. Introduction to Maven

What is Maven? Maven is a tool that automates tasks like managing dependencies, building, testing, and packaging Java projects. Imagine Maven as an assistant that keeps everything organized, making the project easier to work with and maintain.

Why Use Maven?

- **Handles Dependencies**: Automatically adds libraries your project needs.
- **Standard Structure**: Follows a consistent project format.
- **Automates Processes**: Streamlines tasks like compiling and testing.
- **Integration**: Works smoothly with tools like GitHub (for version control) and Jenkins (for continuous integration).

2. Setting Up a Maven Project in Eclipse

1. **Creating a New Maven Project**:
   - In Eclipse, go to **File > New > Maven Project**.
   - Choose the `quickstart` archetype under `org.apache.maven.archetypes`.
   - Set the **Group ID** and **Artifact ID** (the unique identifier for your project).
   - Click **Finish** to create the project.

2. **Understanding Reverse Domain Naming**:
    - Reverse domain naming is a way of creating unique identifiers for projects by using your website address backward.
    - **How It Works**: If your website is `mywebsite.com`, you'd write it backward as `com.mywebsite` for your **Group ID**. This helps avoid naming conflicts by making your project's name globally unique.
    - **Example**: For a project created by `example.com`, you'd use `com.example.projectname` as the **Group ID**.
    - **Why Maven Uses It**: This naming convention is widely recognized, ensuring that your project doesn't accidentally share the same name with others.
3. **Why Use the `org.apache.maven` Archetype and Quickstart?**
    - **What's an Archetype?** An archetype is a template that provides a ready-made project structure.
    - **Why `org.apache.maven.archetypes`?**: This is a standard, reliable setup provided by Maven.
    - **Why Quickstart?** The `quickstart` archetype is perfect for simple Java applications, generating:
        - A basic directory structure,
        - A sample Java file (`App.java`), and
        - Essential configuration files.
    - This setup is great for starting a new project quickly and efficiently.
4. **Understanding the Generated Files**:
    - **`pom.xml`**: The Project Object Model file, `pom.xml`, is the core of your Maven project. It defines the project's dependencies, settings, and configurations.
    - **`App.java`**: Located in `src/main/java`, this file contains a basic "Hello World" program to verify your setup.

3. Running Essential Maven Goals

Maven goals are tasks that Maven performs on your project, such as cleaning, compiling, and testing.

1. **Clean**:
    - **Purpose**: Removes any old compiled files.
    - **Why Clean?** Ensures you start fresh each time, avoiding old data that could cause errors.
    - **How to Run**: Right-click the project, choose **Run As > Maven Clean**, and check the console for "Build Success."
2. **Install**:
    - **Purpose**: Compiles and packages your code, then saves it in your local repository.
    - **Why Install?** Puts the project into a distributable format (like a `.jar` file) for use elsewhere.
    - **How to Run**: Right-click the project, select **Run As > Maven Install**, and confirm "Build Success" in the console.
3. **Test**:
    - **Purpose**: Runs tests to ensure your code behaves as expected.
    - **Why Test?** Helps catch errors early by checking that everything works correctly.
    - **How to Run**: Right-click on the project and select **Run As > Maven Test**.
4. **Running All Goals Together**:
    - Enter `Clean Install Test` in the Goals field to run all three tasks sequentially.
    - **How to Run**: Click **Apply** and **Run**. This runs clean, compile, and test in one step.

4. Running the Project to View Output

- Right-click the project, select **Run As > Java Application**.
- Choose the project and click **OK**.
- The console should display the output, such as "Hello World," if everything is set up correctly.

5. Pushing the Project to GitHub

Why Use GitHub? GitHub is a cloud-based platform where you can store, manage, and collaborate on your projects. It acts as a backup and allows others to work on the project with you.

Steps for Pushing the Project to GitHub:

1. **Create a Repository on GitHub**:
   - Go to GitHub and create a new repository.
   - Select `.gitignore` for Maven.
2. **Why Use `.gitignore`?**
   - **Purpose**: `.gitignore` is a file that tells Git which files or directories to ignore.
   - **Why for Maven Projects?** Maven creates many temporary files and folders, like `/target/`, which don't need to be stored in GitHub. `.gitignore` helps keep your repository clean by ignoring these unnecessary files.
3. **Copy the Repository URL**:
   - Copy the HTTPS link from GitHub to link your local project.
4. **Push the Project Using Git Bash**:
   - **Clone** the GitHub repository to your local system, creating a folder connected to your GitHub repository.
   - Copy files from your Eclipse project folder into this cloned repository.
   - Open Git Bash in the project directory and use the following commands:

```
git add .                # Adds all files
git commit -m "Initial commit"   # Commits changes with a message
git push origin main     # Pushes your project to GitHub
```

   - Refresh GitHub to verify the project has been successfully uploaded.

SOFTWARE ENGINEERING LAB

EXERCISE – 5

TOPIC – 3

CREATING MAVEN WEB PROJECT USING ECLIPSE AND PUSH INTO TO GITHUB

Note: At every step take screenshots and save in a document

1. Introduction to Maven Web Projects

What is a Maven Web Project? A Maven web project is a setup for building Java-based web applications. It's structured to support servlets, JSP (JavaServer Pages), and other web resources. Maven simplifies web development by handling project structure, dependencies, and deployment tasks.

Why Use Maven for Web Projects?

- **Dependency Management**: Automatically adds the libraries your web app needs.
- **Organized Structure**: Provides a standard layout specifically for web projects.
- **Automated Builds**: Compiles, tests, and packages the project with a few commands.
- **Smooth Deployment**: Integrates with servers like Apache Tomcat, simplifying the deployment process.

2. Setting Up a Maven Web Project in Eclipse

1. **Creating a New Maven Web Project**:
    - In Eclipse, go to **File > New > Maven Project**.
    - Choose the **maven-archetype-webapp** archetype from the list.

Why Use the `webapp` Archetype?

- **Purpose**: The `webapp` archetype provides a pre-configured structure for web applications, with essential folders like `webapp/WEB-INF`.
- **Benefits**: This structure includes:
  - A `/src/main/webapp` directory for web resources (HTML, JSP, etc.).
  - A `WEB-INF` folder for configurations and dependencies hidden from direct web access.
- Using this archetype saves setup time and ensures a consistent environment for developing web applications.
- Set the **Group ID** and **Artifact ID** (identifiers for your project).
- Click **Finish** to generate the web project.

2. **Understanding Reverse Domain Naming**:
   - Maven uses the *reverse domain naming convention* to create unique project identifiers.
   - **How It Works**: If your company's domain is `mycompany.com`, you reverse it to `com.mycompany` for the **Group ID**.
   - **Example**: For a project by `example.com`, the Group ID could be `com.example.webapp`.
   - **Purpose**: This naming method is industry-standard, ensuring unique and recognizable project names.
3. **Key Files and Folders in a Web Project**:
   - **`pom.xml`**: The main configuration file for your Maven project, defining dependencies and settings.
   - **`src/main/webapp`**: Folder for your web resources, like HTML and JSP files.
   - **`WEB-INF`**: Located under `webapp`, it contains configurations and libraries that shouldn't be accessible via URL.

3. Adding Dependencies to `pom.xml`

To create a basic web application, you'll need the `servlet-api` dependency. This dependency provides the necessary classes to create servlets, which handle web requests and responses in Java.

1. **Why Add `servlet-api` from the Central Maven Repository?**
   - **Purpose**: The `servlet-api` dependency includes essential libraries for handling HTTP requests and responses, which are fundamental to web applications.
   - **Source**: By adding this dependency to `pom.xml`, Maven automatically downloads it from the *Central Maven Repository*, saving you the effort of finding and installing it manually.
2. **How to Add the `servlet-api` Dependency**:
   - Open `pom.xml` in Eclipse.
   - Under the `<dependencies>` section, add the following:

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.5</version>
</dependency>
```

   - **Explanation**:
     - **`groupId`** and **`artifactId`**: Identify the servlet library in Maven's central repository.
     - **`version`**: Specifies the servlet API version.
     - **`scope: provided`**: Specifies that the server (e.g., Tomcat 9) provides this library at runtime, so it doesn't need to be bundled with the application.
3. **Updating the Project**:
   - Right-click on the project, select **Maven > Update Project**. This action downloads and includes the `servlet-api` from the central Maven repository into your project.

4. Running Essential Maven Goals for a Web Project

Maven's *goals* are specific tasks that help you build and prepare your web project.

1. **Clean**:
   - **Purpose**: Deletes any old compiled files, ensuring each build is fresh.
   - **How to Run**: Right-click the project, select **Run As > Maven Clean**.
2. **Install**:
   - **Purpose**: Compiles the project, packages it as a `.war` file (Web Application Archive), and adds it to your local Maven repository.
   - **How to Run**: Right-click the project, select **Run As > Maven Install**.
3. **Package**:
   - **Purpose**: Packages the project as a `.war` file, which is needed for web deployment.
   - **How to Run**: Right-click the project, select **Run As > Maven Package**.
4. **Running All Goals Together**:
   - Enter `Clean Install Package` in the Goals field to perform these tasks in sequence, then click **Apply** and **Run**.

5. Deploying the Project on Tomcat 9

Apache Tomcat 9 is a widely-used server for running Java web applications. Here's how to deploy your Maven web project on Tomcat 9.

1. **Setting Up Tomcat 9 in Eclipse**:
   - In Eclipse, go to **Servers > New > Server**.
   - Select **Apache Tomcat v9.0** and specify the location of your Tomcat installation.
   - Complete the setup to add Tomcat 9 to your Eclipse environment.
2. **Running the Project on Tomcat 9**:
   - Right-click the project, select **Run As > Run on Server**.
   - Choose **Tomcat v9.0** as the server and start it.

- Your application will be deployed on Tomcat 9, and you can access it in a browser at a URL like `http://localhost:8080/your_project_name`.

6. Pushing the Project to GitHub

Why Use GitHub for Web Projects? GitHub allows you to store your project online, manage versions, and collaborate with others.

Steps for Pushing a Web Project to GitHub:

1. **Create a Repository on GitHub**:
   - Go to GitHub, create a new repository for your project, and select **Maven** as the `.gitignore` template.
2. **Why Use `.gitignore` for Web Projects?**
   - **Purpose**: The `.gitignore` file tells Git to ignore unnecessary files and folders, like `/target/`, which are created during the build process.
   - **Benefit**: This keeps your repository clean by excluding files that aren't needed on GitHub.
3. **Copy the Repository URL**:
   - Copy the HTTPS link to link your local project to GitHub.
4. **Push the Project Using Git Bash**:
   - Open Git Bash, navigate to your project folder.
   - Use these commands to push your project:

```
git add .                  # Adds all files
git commit -m "Initial commit"   # Commits changes with a description
git push origin main     # Pushes the project to GitHub
```

   - Refresh your GitHub page to confirm the project has been successfully uploaded.

SOFTWARE ENGINEERING LAB

EXERCISE – 6

TOPIC – 1

BUILDING THE CI/CD FREESTYLE PIPELINE USING JENKINS FOR MAVEN JAVA PROJECT

Note: At every step take screenshots and save in a document

CI/CD Pipeline

Setting up a CI/CD pipeline automates the build and testing processes for software projects, ensuring continuous integration and delivery. This helps teams identify issues early, maintain code quality, and deploy updates more efficiently. Using Jenkins for this purpose provides a robust and flexible platform to automate repetitive tasks, improve productivity, and streamline software delivery.

1. Setting Up Jenkins and Creating a Freestyle Project

- **Access Jenkins**: Open Jenkins by navigating to `localhost:8080` in your web browser. This is the Jenkins dashboard where you can manage and monitor jobs.
- **Create a New Item**: Click on "New Item" to start a new project. Enter a name for your project and select **Freestyle Project**. This type of project is a basic build job in Jenkins that provides the flexibility to run custom build steps. Click "OK" to proceed.

2. Configuring the Git Repository

- **Add Git Repository URL**: Enter the URL of the Git repository that contains the project you want Jenkins to build. This step links your project source code to Jenkins for automated builds.
- **Specify the Branch**: Indicate which branch Jenkins should use, such as `main` or `master`. This ensures Jenkins builds from the correct version of your code.

3. Setting Up Build Steps

- **Invoke Maven Targets**:
  - Go to the "Build" section and choose **Invoke top-level Maven targets**. This option tells Jenkins to run Maven commands, which are essential for building and managing Java projects.
  - Ensure the Maven path is set in Jenkins under **Manage Jenkins > Global Tool Configuration**. This configuration allows Jenkins to know where Maven is installed.
- **Specify Build Goals**: Enter Maven goals like `clean install`, which clean the project and compile the code. This step replicates the manual build process in Eclipse but in an automated manner.

4. Archiving Build Artifacts

- **Select Post-Build Actions**: Navigate to the "Post-build Actions" section, which defines what happens after a build completes.
- **Archive Artifacts**:
  - Choose **Archive the artifacts** to save build outputs. This is useful for future reference or testing purposes.
  - To archive all files, type `**/*`, which means all generated files from the build process will be stored.

5. Triggering Other Projects

- **Create a Test Project**:
  - Set up a second Freestyle Project to handle testing (e.g., `test project`). For this project, skip the Git repository configuration by selecting "None," as it will use artifacts from the previous project.
- **Copy Artifacts from Another Project**:
  - In the "Build Environment" section, check the option to discard old builds to save space.
  - Use **Copy artifacts from another project** and enter the name of the build project. This ensures the test project has access to the build outputs.

- Select "Stable build only" to ensure only successful builds are used, and type `**/*` to copy all files.

- **Add Build Steps for Testing**:
  - Choose **Invoke top-level Maven targets** and set the goal to `test`. This command runs unit tests defined in the project.

6. Finalizing Post-Build Actions

- **Archive Test Artifacts**: Repeat the artifact archiving step by typing `**/*` to save test results.
- **Apply and Save**: Click "Apply" and "Save" to store all project configurations.

7. Creating the Build Pipeline

- **Add a Pipeline View**:
  - On the Jenkins dashboard, click the "+" symbol to create a new view.
  - Select **Build Pipeline View**, which allows you to visualize and manage the sequence of jobs.
  - Provide a name for the pipeline and choose the initial build project as the starting point.
- **Run the Pipeline**:
  - Click "Run" to start the pipeline. This will trigger the sequence of jobs configured.
  - Click on the small black box in the pipeline view to open the console and monitor the build progress.
- **Verify Pipeline Status**:
  - A successful pipeline run appears in green, indicating that all linked jobs completed successfully.
  - Check the console output to confirm that both the build and test stages ran correctly and that artifacts were archived.

SOFTWARE ENGINEERING LAB

EXERCISE – 6

TOPIC – 2

BUILDING THE CI/CD FREESTYLE PIPELINE USING JENKINS FOR MAVEN WEB PROJECT WITH POLL SCM

Note: At every step take screenshots and save in a document

A CI/CD (Continuous Integration and Continuous Deployment) pipeline helps automate the building, testing, and deployment of a web project. Using Jenkins, you can monitor changes in the source code, automatically trigger builds, and deploy the project to a server. This guide details each step for setting up a Jenkins CI/CD pipeline for a Maven project, leveraging Poll SCM for automation and deploying the output to a Tomcat 9 server.

1. Setting Up Jenkins for Your Maven Web Project

Accessing Jenkins

1. **Open Jenkins:** In your browser, type `localhost:8080` to access Jenkins. This URL opens the Jenkins dashboard, where you manage projects and jobs.

Creating a New Project

1. **Start a New Project:** Click on "New Item" on the Jenkins dashboard.
2. **Enter a Project Name:** Choose a meaningful name for the project.
3. **Select Freestyle Project:** Freestyle is a flexible job type for configuring various build steps. Choose "Freestyle Project" and click "OK" to proceed.

2. Configuring Source Code Management (SCM)

SCM links Jenkins to your project's source code repository (such as Git). Jenkins can then fetch the code and build the project automatically whenever there are changes.

1. **Choose Git under SCM:** In the "Source Code Management" section, select "Git."
2. **Enter the Repository URL:** Paste the URL of your Git repository containing the Maven project.
    - **Example:** For a GitHub project, the URL may look like `https://github.com/username/projectname.git`.
3. **Branch Selection:** Specify the branch to build from, such as `main` or `master`, to ensure Jenkins monitors the correct code version.

3. Enabling Poll SCM for Automated Builds

Poll SCM in Jenkins allows you to set up an automated schedule for checking code changes.

1. **Enable Poll SCM:** In the "Build Triggers" section, check the "Poll SCM" option.
2. **Define the Schedule:** Enter a cron-like schedule to determine when Jenkins should check for changes.
    - ****Example Schedule (H/5 * * *):*** This schedule checks for updates every 5 minutes and triggers a build if changes are detected.
3. **Purpose of Poll SCM:** Poll SCM supports continuous integration by initiating builds automatically based on code updates.

4. Configuring the Build Environment and Steps

To compile and package the project, you need to set up a Maven build step.

1. **Add a Maven Build Step:** In the "Build" section, add a step called "Invoke top-level Maven targets."

2. **Enter Maven Goals:** Use `clean package` to clean old files and compile the project into a deployable WAR file.
    - **Why These Goals?**
        - `clean` deletes old build outputs to avoid conflicts.
        - `package` compiles the code into a WAR file, ready for deployment.
3. **Maven Configuration in Jenkins:** Ensure that Jenkins is configured with a Maven installation.
    - **Path:** Go to "Manage Jenkins > Global Tool Configuration" to add or verify the Maven path.
    - **Purpose:** This allows Jenkins to execute Maven commands for the build process.

5. Configuring Deployment to Tomcat 9 Server

After building the project, set up Jenkins to deploy the generated WAR file to a Tomcat 9 server.

1. **Install the Deploy to Container Plugin:**
    - Go to "Manage Jenkins > Manage Plugins" and install the "Deploy to Container" plugin. This plugin supports deployment to various servers, including Tomcat.
2. **Configure Deployment:**
    - In the "Post-build Actions" section, select "Deploy war/ear to a container."
3. **WAR File Location:** Specify the path to the WAR file, typically something like `**/*.war`.
4. **Configure the Container:**
    - Choose "Tomcat 9.x Remote" as the container.
    - Enter the Tomcat server details:
        - **Tomcat URL:** `http://yourserver:8080` (replace `yourserver` with the actual server IP or hostname).
        - **Credentials:** Provide a username and password for a Tomcat user with deployment privileges (usually configured in `tomcat-users.xml`).
5. **Why Deploy to Tomcat?**

- Tomcat is widely used for deploying Java web applications, and integrating it with Jenkins helps automate the deployment process after each successful build.

6. Setting Post-Build Actions

Post-build actions allow Jenkins to handle additional steps, like archiving artifacts or notifying team members after a successful build.

1. **Archive Build Artifacts:** In the "Post-build Actions" section, choose "Archive the artifacts" to store the generated WAR file.
   - **Pattern:** Use `**/*.war` to archive all WAR files produced by the build.
2. **Notification or Additional Steps (Optional):** Depending on your requirements, you can configure Jenkins to send notifications or trigger other jobs.
   - **Example:** Notify the team upon successful deployment or start another job for post-deployment tests.

7. Monitoring and Running the Pipeline

Once the configuration is complete, you can run the build process and monitor it.

1. **Start the Build Process:** Save your project settings and click "Build Now" on the dashboard.
   - **With Poll SCM:** Jenkins will automatically initiate a build based on the configured polling schedule whenever code changes are detected.
2. **Viewing Build Console Output:** During the build, click on the build number (in "Build History" on the left panel) and select "Console Output" to view real-time logs.
3. **Success Indicators:**
   - **Green Ball:** Indicates a successful build.
   - **Tomcat Deployment Confirmation:** Verify that the application is deployed by checking Tomcat's management console or by accessing the application URL.
4. **Failed Build Indicator:** A red ball means the build failed. Check the console output to troubleshoot and resolve issues.

Creating a Pipeline View for Monitoring

Setting up a pipeline view in Jenkins helps visualize and monitor the stages of the CI/CD pipeline.

1. **Install the Pipeline Plugin (if not already installed):** Go to "Manage Jenkins > Manage Plugins" and install the "Pipeline" and "Build Pipeline" plugins.
2. **Create a New View:**
    - From the Jenkins dashboard, select "New View."
    - Name the view (e.g., "Maven Project Pipeline") and choose "Build Pipeline View."
3. **Configure the Pipeline View:**
    - **Initial Job:** Select the initial job you created for this project.
    - **Display Options:** Customize how you want to view each stage, such as displaying build numbers, statuses, or logs.
4. **Save the View:** This new view provides a graphical representation of the pipeline, showing each stage (build, deploy) and their status.
    - **Purpose of the Pipeline View:** Makes it easier to track progress, spot issues, and access logs or details for each stage in the CI/CD process.

SOFTWARE ENGINEERING LAB

EXERCISE – 6

TOPIC – 3

BUILDING THE CI/CD SCRIPTED PIPELINE USING JENKINS FOR MAVEN JAVA PROJECT WITH POLL SCM

Note: At every step take screenshots and save in a document

1. Introduction to Jenkins for CI/CD Pipelines

Jenkins is an open-source automation server that helps in building, testing, and deploying code through continuous integration (CI) and continuous delivery (CD). For software development teams, Jenkins ensures that changes in code are automatically built and tested, facilitating a streamlined workflow.

2. Creating a New Pipeline in Jenkins

To begin, navigate to the Jenkins dashboard:

- **Step 1:** Click on "New Item".
- **Step 2:** Enter a project name.
- **Step 3:** Select the third option, "Pipeline", and click "OK".

Why choose "Pipeline"?

Jenkins Pipeline is a suite of plugins that supports implementing and integrating continuous delivery pipelines. It provides a robust and flexible framework for modelling complex workflows.

3. Configuring Build Triggers in Jenkins

Scroll to the "Build Triggers" section:

- **Select "Build Periodically"**: This option allows Jenkins to trigger builds at specified intervals, even without any code changes.
- **Why "Build Periodically"?**

  Automating builds at regular intervals helps ensure that scheduled tasks such as code compilation, tests, or deployments occur consistently. This keeps the project up-to-date and highlights issues promptly.

4. Understanding the Scheduling Syntax

The scheduling syntax uses a crontab format:

- Example: ****H/15 * * * ***** - Triggers a build every 15 minutes.
- Example: ****H * * 3 ***** - Triggers a build on the first day of every third month.

Explanation of the Script Syntax:

- The "H" symbol represents a hashed value to distribute load evenly.
- Numbers separated by spaces represent: minute, hour, day of the month, month, and day of the week.

5. Adding the Pipeline Script

Navigate to the "Pipeline" section and paste the following script:

```
pipeline {
    agent any
    tools {
        maven 'MAVEN_HOME'
    }
    stages {
        stage('git repo & clean') {
            steps {
```

```
                bat "rmdir  /s /q SampleMavenJavaProject"
                bat "git clone https://github.com/budarajumadhurika/SampleMavenJavaProject.git"
                bat "mvn clean -f SampleMavenJavaProject"
            }
        }
        stage('install') {
            steps {
                bat "mvn install -f SampleMavenJavaProject"
            }
        }
        stage('test') {
            steps {
                bat "mvn test -f SampleMavenJavaProject"
            }
        }
        stage('package') {
            steps {
                bat "mvn package -f SampleMavenJavaProject"
            }
        }
    }
}
```

Explanation of the Script:

- **'pipeline' block**: Declares the start of the Jenkins pipeline.
- **'agent any'**: Specifies that Jenkins can run the pipeline on any available agent.
- **'tools' block**: Ensures the specified Maven version is available.
- **Stages**:
    - **Git Repo & Clean Stage**: Deletes any existing project directory using `rmdir` and clones the Git repository. Runs `mvn clean` to clean the project.

- **Install Stage**: Runs `mvn install` to compile, test, and install the package into the local repository.
- **Test Stage**: Executes `mvn test` to run unit tests and validate code functionality.
- **Package Stage**: Runs `mvn package` to bundle the compiled code into a deployable format (e.g., a JAR or WAR file).

6. Applying Changes and Running the Pipeline

- **Click "Apply" and "Save"**: Save your pipeline configuration.
- **Click "Build Now"**: Initiates the pipeline run.
- **Successful Build Verification**: Confirm a successful build by checking the console output for completion messages.

7. Poll SCM for Automatic Triggers

Polling the Source Code Management (SCM) ensures that the pipeline triggers when new changes are committed:

- **Enable Poll SCM** under "Build Triggers".
- **Why use Poll SCM?**

  It automates the process of detecting changes in the repository, making builds more efficient by triggering them only when updates are made.

- **Example Schedule:** `H/1 * * * *` triggers a build check every minute.

Conclusion:

Using Jenkins with Poll SCM and pipelines streamlines CI/CD processes, automates builds, and enhances overall project efficiency. This approach minimizes manual intervention, supports consistent testing, and ensures prompt feedback.

SOFTWARE ENGINEERING LAB

EXERCISE – 7

TOPIC – 1

DOCKER CLI COMMANDS-PART-1

Note: At every step take screenshots and save in a document

Understanding Docker and Redis

What is Docker?

Docker is a tool that makes running applications easy by packaging everything (code, libraries, tools) into containers.

- Containers are like **lightweight virtual machines** but more efficient because they share the host's system resources.

What is Redis?

Redis (Remote Dictionary Server) is:

- A **super-fast database** that stores data in memory (not on a disk).
- Commonly used for:
  - **Caching**: Storing temporary data for quick access.
  - **Real-time applications**: Like live chat, analytics, or leaderboards.
  - **Data structures**: Redis supports lists, hashes, sets, and more.

Example:

- Save data: Use the key "`name`" and the value "`Alice`".
- Retrieve data: Ask Redis for "`name`", and it will give you "`Alice`" instantly.

Setting Up Docker

Step 1: Choose the Right Terminal

- **Windows**: Use **Git Bash** or **PowerShell** (Git Bash is preferred for Docker commands).
- **Mac/Linux/Ubuntu**: Use the built-in **Terminal**.

Step 2: Verify Docker Installation

Run this command to check if Docker is installed:

```
docker --version
```

What It Does:

- Displays the installed Docker version to ensure everything is ready.

Docker CLI Commands with `hello-world`

Why Use `hello-world`?

The **`hello-world`** image is a basic test to ensure Docker is working correctly.

Step 1: Pull the `hello-world` Image

Command:

```
docker pull hello-world
```

What It Does:

- Downloads the **`hello-world`** image from Docker Hub (Docker's app store).

Where to Run:

- Open your terminal (Git Bash for Windows or Terminal for Mac/Linux).
- Run the command from any folder.

Step 2: Run the `hello-world` Image

Command:

```
docker run hello-world
```

What It Does:

- Creates and runs a **container** from the `hello-world` image.
- Displays a message to confirm that Docker is installed and working.

Output Example:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

Step 3: View All Containers

Command:

```
docker ps -a
```

What It Does:

- Lists all containers (running and stopped).
- The `hello-world` container will show as "Exited" because it stops after displaying the message.

Step 4: Remove the `hello-world` Container

Command:

```
docker rm [container-id]
```

What It Does:

- Deletes the container to free up space.
- Replace `[container-id]` with the actual ID from `docker ps -a`.

Step 5: Remove the `hello-world` Image

Command:

```
docker rmi hello-world
```

What It Does:

- Deletes the `hello-world` image if you no longer need it.

Docker CLI Commands with `redis`

Why Use `redis`?

Redis is a powerful, real-world example of a service often run using Docker.

Step 1: Pull the `redis` Image

Command:

```
docker pull redis
```

What It Does:

- Downloads the official `redis` image from Docker Hub to your system.

Step 2: Run a Redis Container

Command:

```
docker run --name my-redis -d redis
```

What It Does:

- Creates and starts a container named `my-redis` from the `redis` image.
- The `-d` flag runs the container in the background.

Step 3: Check Running Containers

Command:

```
docker ps
```

What It Does:

- Lists all running containers.
- You should see the Redis container (`my-redis`) in the list.

Step 4: Access Redis

Command:

```
docker exec -it my-redis redis-cli
or
winpty docker exec -it myredis redis-cli
```

What It Does:

- Opens the Redis command-line tool (`redis-cli`) inside the container.
- You can now send commands directly to the Redis server.
- winpty: This command makes Git Bash handle the terminal interaction correctly, allowing you to run commands that require user input.
- docker exec -it myredis redis-cli: This runs the Redis command-line interface (redis-cli) inside the running myredis container.

Example Redis Commands:

```
127.0.0.1:6379> SET name "Alice"
OK
127.0.0.1:6379> GET name
"Alice"
```

Step 5: Stop the Redis Container

Command:

```
docker stop my-redis
```

What It Does:

- Stops the Redis container but doesn't delete it.

Step 6: Restart the Redis Container

Command:

```
docker start my-redis
```

What It Does:

- Restarts the stopped container.
- **Command**:

```
docker stop my-redis
```

Step 7: Remove the Redis Container

Command:

```
docker rm my-redis
```

What It Does:

- Deletes the container permanently.

Step 8: Remove the Redis Image

Command:

```
docker rmi redis
```

What It Does:

- Deletes the Redis image from your local system.

SOFTWARE ENGINEERING LAB

EXERCISE – 7

TOPIC – 1

DOCKER CLI COMMANDS-PART-2

Note: At every step take screenshots and save in a document

Using a `Dockerfile`

What is a Dockerfile?

A `Dockerfile` is a text file with instructions to create a custom Docker image.

Step 1: Set Up Your Folder

1. **Windows**:
   - Create a folder like `C:\DockerProjects\Redis.`
   - Open Git Bash and navigate to the folder:

     ```
     cd /c/DockerProjects/Redis
     ```

2. **Mac/Linux**:
   - Create a folder:

     ```
     mkdir ~/DockerProjects/Redis
     cd ~/DockerProjects/Redis
     ```

Step 2: Write the Dockerfile

1. Inside the folder, create a file named `Dockerfile` (no extension).
2. Add the following content:

```
FROM redis:latest
CMD ["redis-server"]
```

What It Does:

- Starts with the official Redis image.
- Configures the container to run a Redis server.
- FROM redis:latest
- Think of "Redis" as a ready-made base (like instant noodles). Instead of making everything from scratch, you're starting with a Redis image (software) that someone else already made.
- latest means you're using the newest version of Redis.
- CMD ["redis-server"]
- This tells Docker to start the Redis program (like clicking "Run" on a software) whenever the container is started.

Docker Commands (Step-by-step):

1. `docker build -t redisnew .`

   What it does:

   - This creates (builds) a Docker image using the recipe (Dockerfile) in the current folder (.).
   - -t redisnew: Gives the image a name/tag ("redisnew"), so you can find it easily.

2. `docker run --name myredisnew -d redisnew`

What it does:

- Starts a new container (mini computer) from the redisnew image.
- --name myredisnew: Names the container "myredisnew" so it's easy to identify.
- -d: Runs the container in the background.

3. `docker ps`

What it does:

- Shows a list of containers that are running right now.

4. `docker stop myredisnew`

What it does:

- Stops the container named "myredisnew" (like turning off a computer).

5. `docker login`

What it does:

- Logs you into your Docker Hub account, so you can upload images.

6. `docker ps -a`

What it does:

- Shows a list of all containers, including stopped ones.

7. `docker commit 0e993d2009a1 budarajumadhurika/redis1`

What it does:

- Takes a snapshot (saves changes) of the container with ID 0e993d2009a1 and creates a new image called budarajumadhurika/redis1.

8. `docker images`

What it does:

- Lists all images saved on your system.

9. `docker push budarajumadhurika/redis1`

What it does:

- Uploads the image budarajumadhurika/redis1 to Docker Hub, so others can download it.

10. `docker rm 0e993d2009a1`

What it does:

- Deletes the container with ID 0e993d2009a1.

11. `docker rmi budarajumadhurika/redis1`

What it does:

- Deletes the image budarajumadhurika/redis1 from your system.

12. `docker ps -a`

What it does:

Shows all containers again to confirm changes.

13. `docker logout`

What it does:

- Logs you out of Docker Hub.

14. `docker pull budarajumadhurika/redis1`

What it does:

- Downloads the image budarajumadhurika/redis1 from Docker Hub.

15. `docker run --name myredis -d budarajumadhurika/redis1`

What it does:

- Starts a new container using the image budarajumadhurika/redis1.

16. `docker exec -it myredis redis-cli`

What it does:

- Opens the Redis command-line interface (like a terminal) inside the running container myredis.

17. `SET name "Abcdef"`

What it does:

- Saves a key-value pair in Redis (key = name, value = Abcdef).

18. `GET name`

What it does:

- Retrieves the value of the key name from Redis (it will return "Abcdef").

19. `exit`

What it does:

- Exits the Redis CLI.

20. `docker ps -a`

What it does:

- Shows all containers again to check their status.

21. `docker stop myredis`

What it does:

- Stops the container myredis.

22. `docker rm 50a6e4a9c326`

What it does:

- Deletes the container with ID 50a6e4a9c326.

23. `docker images`

What it does:

- Lists all images again to confirm which ones remain.

24. `docker rmi budarajumadhurika/redis1`

What it does: Deletes the image budarajumadhurika/redis1 again.

Step 4: Remove Login Credentials (Optional)

If you no longer need to be logged in, you can log out:

```
docker logout
```

What It Does: Logs you out from Docker Hub and removes your stored credentials.

SOFTWARE ENGINEERING LAB

EXERCISE – 7

TOPIC – 2

MODIFY AND PUSH DOCKER IMAGE

By following these Commands, you will learn how to:

- Create and modify a container (E.g. Ubuntu).
- Save the changes to a custom image.
- Push the image to Docker Hub.
- Pull and reuse the image. This workflow is useful for creating reusable and shareable container environments.

- **Note: At every step take screenshots and save in a document**

1. Pull the Ubuntu image

```
docker pull ubuntu
```

•What it does: Downloads the official Ubuntu base image from Docker Hub to your local system. This image is like a minimal operating system ready to run inside a Docker container.

2. Run a container from the Ubuntu image

```
docker run -it --name newubuntu -d ubuntu
```

•What it does: Creates and starts a new container from the Ubuntu image.

•-it: Allows you to interact with the container (interactive terminal mode).

•--name newubuntu: Names the container "newubuntu" for easy identification.

•-d: Runs the container in the background (detached mode).

3. List all running containers

```
docker ps
```

•What it does: Displays a list of all currently running containers, showing details like the container ID, name, image used, and uptime.

4. Access the running container

```
docker exec -it 885a01bcdbe0 bash
```

•What it does: Opens a shell (terminal) inside the running container.

•exec: Executes a command in a running container.

•-it: Allows interactive access.

•885a01bcdbe0: The unique container ID of the running container.

•bash: Opens the bash shell inside the container.

5. Check if Git is installed

```
git --version
```

•What it does: Checks the version of Git installed in the container.

•Why it failed: The error bash: git: command not found means Git is not installed in the container.

6. Update the package list

```
apt update
```

•What it does: Updates the list of available software packages in the container. It prepares the system for installing new software by fetching the latest versions from online repositories.

7. Install Git

```
apt install git -y
```

•What it does: Installs Git inside the container.

•-y: Automatically confirms the installation (avoids asking for "yes/no").

8. Verify Git installation

```
git --version
```

•What it does: Checks if Git is installed correctly and displays its version (e.g., git version 2.43.0).

9. Exit the container

```
exit
```

•What it does: Closes the shell session inside the container and returns to your host system.

10. Stop the running container

```
docker stop 885a01bcdbe0
```

•What it does: Stops the running container. It doesn't delete the container, but it halts its operation.

11. Save the container as an image

```
docker commit 885a01bcdbe0 budarajumadhurika/newubuntu2024
```

•What it does: Creates a new image from the stopped container with all the changes (like the Git installation).

•budarajumadhurika/newubuntu2024: Names the new image with a custom name and tag.

12. List all local images

```
docker images
```

•What it does: Shows all the Docker images stored on your system, including the newly created image budarajumadhurika/newubuntu2024.

13. Log in to Docker Hub

```
docker login
```

•What it does: Logs you into your Docker Hub account so you can upload (push) your image.

•It will prompt for your Docker Hub username and password.

14. Push the image to Docker Hub

```
docker push budarajumadhurika/newubuntu2024
```

•What it does: Uploads the newly created image to your Docker Hub account so it can be accessed from anywhere.

15. Log out of Docker Hub

```
docker logout
```

•What it does: Logs you out of Docker Hub for security.

16. Remove the container

```
docker rm 885a01bcdbe0
```

•What it does: Deletes the stopped container permanently from your system.

17. Remove the local image

```
docker rmi budarajumadhurika/newubuntu2024
```

•What it does: Deletes the custom image from your local system, freeing up space. (The image is still available on Docker Hub.)

18. Pull the image from Docker Hub

```
docker pull budarajumadhurika/newubuntu2024
```

•What it does: Downloads the custom image budarajumadhurika/newubuntu2024 from Docker Hub to your local system.

19. Run a container from the custom image

```
docker run --name newubuntu2024 -it budarajumadhurika/newubuntu2024
```

•What it does: Starts a new container from the custom image.

•--name newubuntu2024: Assigns a name to the container for easy reference.

•-it: Allows interactive access to the container.

20. Check Git version

```
git --version
```

•What it does: Verifies that Git is still installed in the new container, confirming that the custom image retains the installed software.

21. Exit the container

```
exit
```

•What it does: Closes the shell session inside the container.

22. List all containers (including stopped ones)

```
docker ps -a
```

•What it does: Displays all containers on your system, including those that are stopped.

23. Remove the container

```
docker rm 28aee36085cb
```

•What it does: Deletes the container created from the custom image.

24. Remove the custom image

```
docker rmi budarajumadhurika/newubuntu2024
```

•What it does: Deletes the custom image from your local system again.

SOFTWARE ENGINEERING LAB

EXERCISE – 7

TOPIC – 3

CREATE AND PUSH DOCKER FILE IMAGE

By following these Commands, you will learn how to:

- Creating a JavaScript calculator program.
- Building and running a Docker container.
- Pushing the container image to Docker Hub for sharing.
- Pulling and reusing the image on another system.

- **Note: At every step take screenshots and save in a document**

Step 1: Create the JavaScript File

Create a file named calculator.js in your project directory.

Add the following code to define simple calculator functions:

```
// calculator.js

function add(a, b) {

    return a + b;

}

function subtract(a, b) {

    return a - b;

}

function multiply(a, b) {

    return a * b;

}
```

```
function divide(a, b) {
    if (b === 0) {
        return "Cannot divide by zero!";
    }
    return a / b;
}
// Print the calculated values
console.log("Addition (2 + 3):", add(2, 3));
console.log("Subtraction (5 - 2):", subtract(5, 2));
console.log("Multiplication (4 * 3):", multiply(4, 3));
console.log("Division (10 / 2):", divide(10, 2));
```

Purpose: This script performs basic arithmetic operations and logs the results to the console.

Output: When run, this program prints the results of the calculations.

Step 2: Create a Dockerfile

The Dockerfile contains instructions for building the Docker image.

1. Create a file named Dockerfile (no file extension).
2. Add the following content:

```
FROM node:16-alpine
WORKDIR /app
COPY calculator.js /app
CMD ["node", "calculator.js"]
```

Explanation of the Dockerfile

- FROM node:16-alpine

This uses the Alpine version of Node.js 16, which is a minimal, lightweight image.

- WORKDIR /app

Sets the working directory inside the container to /app, where your app files will go.

- COPY calculator.js /app

Copies the calculator.js file into the container.

- CMD ["node", "calculator.js"]

Tells Docker to run the calculator.js file using Node.js when the container starts.

Step 3: Build the Docker Image

1. Open a terminal in the directory containing your Dockerfile and calculator.js.
2. Run the following command:

```
docker build -t simple-calculator .
```

- docker build: This command builds an image from the Dockerfile.
- -t simple-calculator: Tags the image with the name simple-calculator.
- .: Refers to the current directory where the Dockerfile is located.

Step 4: Run the Docker Container

Run the container using the image you just created:

```
docker run simple-calculator
```

- docker run: Starts a new container from the image.
- simple-calculator: The name of the image to use.

Expected Output: The console will display the results of the calculations.

Step 6: Push the Image to Docker Hub

Ensure you have a Docker Hub account. Log in using the following command:

```
docker login
```

Enter your **Docker Hub username** and **password** if prompted

Tag the Image for Docker Hub:

Docker images need to be tagged with your Docker Hub username before they can be uploaded:

```
docker tag simple-calculator your-dockerhub-username/simple-calculator
```

Replace your-dockerhub-username with your actual Docker Hub username.

Push the Image to Docker Hub:

Push the tagged image to Docker Hub:

```
docker push your-dockerhub-username/simple-calculator
```

Once complete, your image will be available in your Docker Hub repository.

Step 7: Pull the Image from Docker Hub

To test the reusability, first, remove the existing image and container:

```
# List all containers (even stopped ones)
docker ps -a
# Remove the container by ID
docker rm <container-id>
# Remove the local image
docker rmi your-dockerhub-username/simple-calculator
```

On another machine or after deleting the local image and container, pull the image from Docker Hub:

```
docker pull your-dockerhub-username/simple-calculator
```

Step 8: Run the Pulled Image

Run the container from the pulled image:

```
docker run your-dockerhub-username/simple-calculator
```

You'll see the same calculation results printed to the console.

```
#List all the containers

docker ps -a

#Delete the container by its ID

docker rm <container-id>

# List all images

docker images

# Remove unused images by ID

docker rmi <image-id>

# Log out of Docker Hub.

docker logout
```

SOFTWARE ENGINEERING LAB

EXERCISE – 7

TOPIC – 4

RUNNING MULTIPLE CONTAINERS USING DOCKER COMPOSE

By following the steps in the Docker Compose exercise, you will learn how to:

- **Set up a WordPress and MySQL multi-container environment** using a simple configuration file (docker-compose.yml).
- **Create and configure Docker services** for a web application and a database, including setting environment variables and persistent data storage.
- **Run multiple containers together** with a single command, simplifying deployment.
- **Access the WordPress application locally** through a web browser and link it to the MySQL database for data storage.
- **Use Docker Compose commands** to start, stop, and remove containers easily.
- **Persist and share data** between containers using Docker volumes.
- **Understand basic networking in Docker Compose** to enable communication between containers.
- **Reuse the setup on any system** by copying the docker-compose.yml file and running it.

- **Note: At every step take screenshots and save in a document**

1. What is a docker-compose.yml File?

A docker-compose.yml file is a configuration file that defines:

What containers (services) to create.

How they should work together (like connecting WordPress to MySQL).

Settings for each container (such as ports, environment variables, etc.).

This file is written in YAML format, which is a simple, human-readable way to structure data.

2. Where to Find Sample Configurations

You can find examples of docker-compose.yml files:

Docker Documentation: The official Docker website has guides and sample configurations.

GitHub: Many developers share sample files. For example, you can visit Brad Traversy's Docker Example and scroll to version 3 for a beginner-friendly example.

- https://gist.github.com/bradtraversy/faa8de544c62eef3f31de406982f1d42

Docker Hub: Each image (like MySQL or WordPress) often includes example configurations.

For simplicity, let's build a basic docker-compose.yml file for WordPress and MySQL from scratch.

3. Writing a Basic docker-compose.yml File

Step 1: Create a Folder

Go to your desktop or any folder you prefer.

Right-click and select New Folder.

Name the folder my_docker_project.

Step 2: Open a Text Editor

Open Visual Studio Code, Notepad, or any other text editor.

Create a new file.

Step 3: Write the YAML Configuration

Here's a simple example of a docker-compose.yml file for WordPress and MySQL:

```
version: '3.1'

services:

  db:
```

```
    image: mysql:5.7
    container_name: mysql_container
    environment:
      MYSQL_ROOT_PASSWORD: rootpassword
      MYSQL_DATABASE: wordpress_db
      MYSQL_USER: wordpress_user
      MYSQL_PASSWORD: wordpress_pass
    volumes:
      - db_data:/var/lib/mysql

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    container_name: wordpress_container
    ports:
      - "8000:80"
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress_user
      WORDPRESS_DB_PASSWORD: wordpress_pass
      WORDPRESS_DB_NAME: wordpress_db
    volumes:
      - ./wordpress_data:/var/www/html

volumes:
```

```
  db_data:
```

4. Step-by-Step Explanation

1. File Format

version: '3.1': This specifies the Docker Compose file version. Version 3 is widely supported.

2. Services

Under services, we define the containers we want to run:

- db (MySQL Container):

image: Specifies the MySQL version (here, 5.7).

container_name: Gives the container a name (mysql_container).

environment: Sets environment variables for the database:

MYSQL_ROOT_PASSWORD: Root user password.

MYSQL_DATABASE: Name of the database WordPress will use.

MYSQL_USER and MYSQL_PASSWORD: Credentials WordPress will use to connect.

volumes: Persists database data to the db_data volume.

- wordpress (WordPress Container):

depends_on: Ensures the MySQL container starts first.

image: Specifies the WordPress version (here, latest).

container_name: Gives the container a name (wordpress_container).

ports: Maps port 8000 on your computer to port 80 in the container. You'll access WordPress at http://localhost:8000.

environment: Provides settings to connect WordPress to the MySQL database.

volumes: Connects your local directory (./wordpress_data) to the container for file storage.

3. Volumes

volumes help save data even if the container stops. For example:

db_data: Stores MySQL database data.

5. Saving the File

Save the file as docker-compose.yml.

Place it in the my_docker_project folder.

6. Running the Setup

Step 1: Open Command Line

Open PowerShell or Command Prompt.

Navigate to the my_docker_project folder:

cd path_to_my_docker_project

Step 2: Start the Containers

Run:

```
docker-compose up -d
```

This command reads the docker-compose.yml file and creates both the WordPress and MySQL containers.

-d runs the containers in the background.

7. Accessing the Application

Open your web browser.

Go to `http://localhost:8000`.

Follow the WordPress setup wizard to complete the installation:

Site Name.

Admin Username and Password.

Email Address.

8. Managing Containers

Stop the Containers

To stop the containers without removing them:

```
docker-compose stop
```

Start Again

To restart the containers:

```
docker-compose start
```

Remove Containers

To stop and remove everything:

```
docker-compose down
```

SOFTWARE ENGINEERING LAB

EXERCISE – 7

TOPIC – 5

DEPLOYING AND SCALING APPLICATIONS USING MINIKUBE

How to Set Up, Scale, and Stop Nginx Using Kubernetes (Minikube)

What Are We Doing?

In this guide, we will:

1. **Set up a mini-Kubernetes system on your computer** using a tool called Minikube. This is like creating a small virtual computer where we can run apps.
2. **Create a Nginx app** (Nginx is a simple web server that shows websites).
3. **Make the Nginx app accessible** from your browser so you can see it on your screen.
4. **Create more copies of the Nginx app** (this helps it handle more users or traffic).
5. **Clean up everything** when we're done so nothing is left running.

This process will work the same on **Windows, macOS, and Linux**, with only a few small differences in how Minikube is set up on each system.

Step 1: Setting Up Minikube and Kubernetes

1.1 Start Minikube

Run this command to start the Minikube cluster:

```
minikube start
```

- **Minikube** creates a **local Kubernetes cluster**. This cluster helps us manage and run applications on our machine. Think of it as a small **virtual computer** running in the background that Kubernetes will use to manage apps.

Step 2: Create and manage the Nginx Deployment

2.1 Create the Nginx Deployment

Run the following command to create a new deployment (which is like creating an app) for **Nginx**:

```
kubectl create deployment mynginx --image=nginx
```

- **Explanation of the command**:
  - `kubectl`: This is the tool we use to talk to Kubernetes.
  - `create deployment`: This tells Kubernetes to create a **Deployment**, which is a way of managing and running apps.
  - `mynginx`: This is the name we give to our app. You can call it anything you want, but we are calling it **mynginx**.
  - `--image=nginx`: This tells Kubernetes to use the **nginx image** (a ready-made version of Nginx) to create the app. An **image** is like a blueprint for creating an app.

You can verify if the Nginx deployment was created correctly by running this command:

```
kubectl get deployments
```

You should see **mynginx** listed as a deployment.

Step 3: Expose the Nginx Deployment to the Outside World

3.1 Expose the Service

We will use the following command to **expose** the Nginx app to the outside world:

```
kubectl expose deployment mynginx --type=NodePort --port=80 --target-port=80
```

- **Explanation of the command**:
  - `kubectl expose`: This tells Kubernetes to create a **Service**. A Service is a way to expose your app to the outside world.

- `deployment mynginx`: This specifies which app (Deployment) we want to expose.
- `--type=NodePort`: This makes the service accessible **outside the Kubernetes cluster** by opening a specific port on your computer.
- `--port=80`: This is the port we'll use to access Nginx from outside the cluster (port 80 is the default for web servers).
- `--target-port=80`: This is the port inside the pod where Nginx is running (also port 80).

Step 4: Scale the Deployment (Make More Copies)

4.1 Scale the Deployment

To scale Nginx to **4 replicas** (running 4 copies of the app), run the following command:

```
kubectl scale deployment mynginx --replicas=4
```

- **Explanation of the command**:
  - `kubectl scale`: This tells Kubernetes to **increase or decrease** the number of **pods** running.
  - `--replicas=4`: This specifies that we want **4 copies** (pods) of the Nginx app running at the same time.

Step 5: Accessing the Nginx App

5.1 What is Port Forwarding?

Port forwarding is a way to **forward traffic** from a port on your computer (like **8081**) to a port inside the Kubernetes cluster (like **port 80** in the Nginx pod).

We need **port forwarding** because **Jenkins is already using port 8080** on your computer. That's why we forward traffic from **8081** to **port 80** on Nginx.

5.2 Run Port Forwarding

Run this command to forward port **8081** on your local machine to port **80** inside the Nginx container:

```
kubectl port-forward svc/mynginx 8081:80
```

- **Explanation of the command**:
  - `kubectl port-forward`: This command forwards traffic from one port to another.
  - `svc/mynginx`: This refers to the **mynginx service** we created earlier.
  - `8081:80`: This forwards **port 8081** on your local machine to **port 80** inside the Nginx container.

- Open your browser and go to:

```
http://localhost:8081
```

You should see the **Nginx welcome page**, confirming that the app is running!

Step 6: Alternative Method – Minikube Tunnel

If **port forwarding** doesn't work for you, or if you prefer an alternative method, you can use **Minikube Tunnel**.

6.1 Start Minikube Tunnel

Run this command to create a tunnel that lets you access the service directly:

```
minikube tunnel
```

- **Explanation**: This command creates a **tunnel** between your local machine and the Kubernetes cluster, allowing you to access the service.

6.2 Get the Minikube Service URL

Once the tunnel is running, get the URL of the service using:

```
minikube service mynginx --url
```

- **What happens**: This command gives you a URL (like `http://192.168.x.x:80`) that you can open in your browser to access Nginx.

6.3 Open the URL in Your Browser

Copy and paste the provided URL into your browser to access Nginx directly.

Step 7: Stopping and Cleaning Up Everything

7.1 Stop the Nginx Deployment and Service

To delete the **Nginx deployment** and **service**, run:

```
kubectl delete deployment mynginx
kubectl delete service mynginx
```

7.2 Stop Minikube (Optional)

If you're done with Minikube, stop it to free up system resources:

```
minikube stop
```

7.3 Delete the Minikube Cluster (Optional)

To delete the entire Minikube cluster, run:

```
minikube delete
```

SOFTWARE ENGINEERING LAB

EXERCISE – 7

TOPIC – 6

DEPLOYING AND MANAGING MONITORING SYSTEMS USING NAGIOS IN DOCKER

In this exercise, we will be:

- Setting up and running Nagios using Docker for quick and simple installation.
- Accessing the Nagios Dashboard to explore its features.
- Monitoring the health and status of systems and services effectively.
- Learning how to manage checks, notifications, and downtime settings.
- Understanding how to stop and remove the Nagios Docker container when finished.
- **Note: At every step take screenshots and save in a document**

Step 1: Pulling the Nagios Image

1. **Command to Pull Nagios**

   Open a terminal and type this command to download the Nagios image:

   ```
   docker pull jasonrivers/nagios:latest
   ```

   - This command tells Docker to download the **latest version** of Nagios from a specific repository (`jasonrivers`).

Step 2: Running Nagios

1. **Command to Run Nagios**

```
docker run --name nagiosdemo -p 8888:80 jasonrivers/nagios:latest
```

Explanation of Each Part:

- `--name nagiosdemo`: Names the container **nagiosdemo** so you can easily identify it later.
- `-p 8888:80`: Maps your computer's **port 8888** to the container's **port 80** (port 80 is used for web access).
- `jasonrivers/nagios:latest`: Specifies the software image (Nagios) and its version.

Step 3: Accessing Nagios

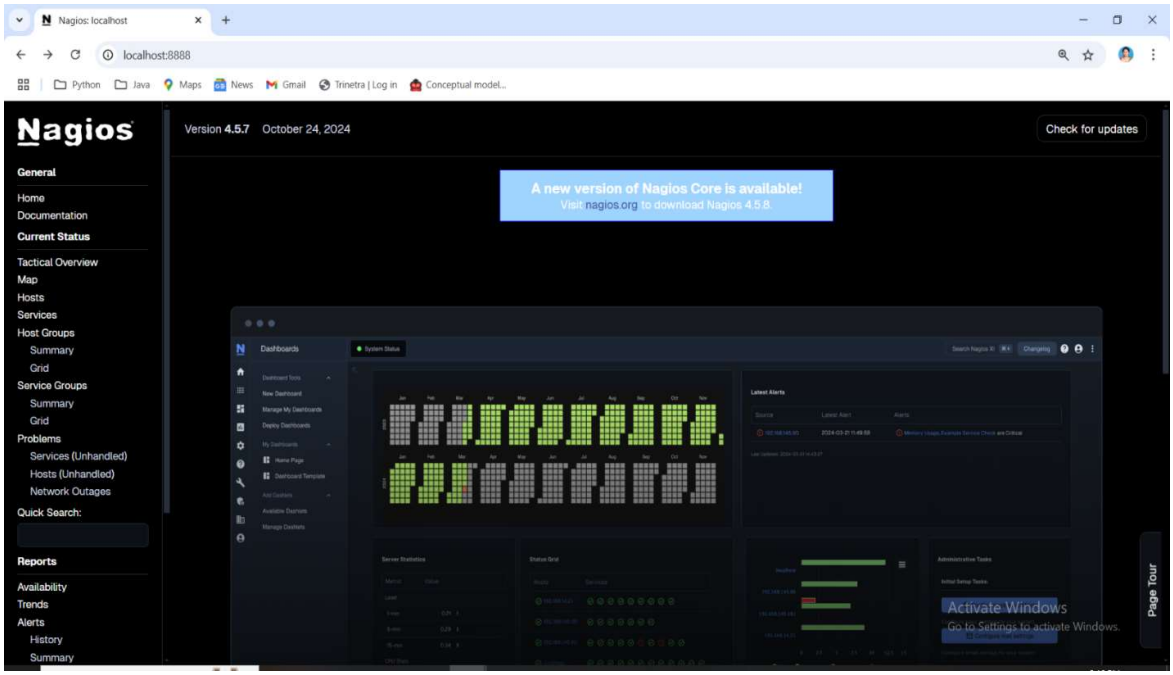
1. Open a browser and type:

   `localhost:8888`

   This will open the Nagios web interface.

2. **Login Credentials**:
   - **Username**: `nagiosadmin`
   - **Password**: `nagios`

Understanding the Nagios Dashboard



1. General Information

- **Nagios Version**: The top-left corner shows the version of Nagios you're using (e.g., Version 4.5.7).

2. Navigation Panel (Left-Side Menu)

- **Home**: Returns you to the main dashboard.
- **Hosts**: Displays the list of systems (hosts) being monitored.
- **Services**: Shows activities or tasks being monitored, such as CPU usage, disk space, or network status.
- **Reports**: Provides historical data and trends about your systems.

3. Dashboard Content

a. System Status (Top Section)

- **Green Boxes**: Indicate periods when everything was working fine.
- **Red or Yellow Boxes**: Indicate problems or warnings.

b. Latest Alerts

- **Source**: Shows which system (e.g., IP address) had the issue.
- **Alert**: Describes the problem (e.g., "Memory usage critical").

c. Server Statistics

- **Load**: How much work the server is handling.
- **CPU Stats**: How busy the processor is.

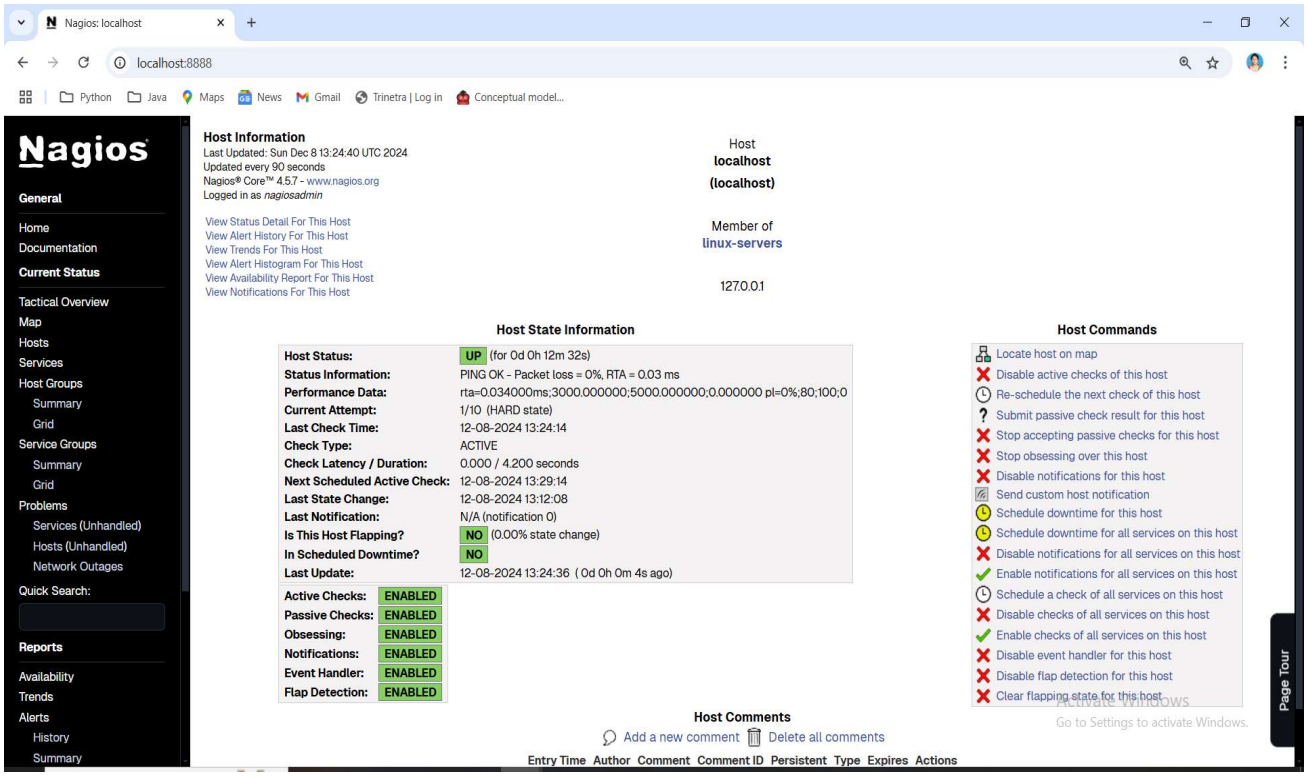
d. Status Grid

- **Green Circles**: Everything is working fine.
- **Red or Yellow Circles**: Indicate issues.

4. Monitoring Specific Hosts and Services

1. **Monitor Hosts**:
    - Go to the **Hosts** menu on the left.
    - You'll see a list of systems being monitored.

2. **Monitor Services**:
    - Click on a host to see details about the services being monitored, such as CPU usage or memory status.



Step 4: Exploring the Host Information Page

a. Top Section (Basic Host Details)

- **Host**: The system being monitored is named `localhost` (your own computer).
- **Member of**: This host belongs to the `linux-servers` group.
- **IP Address**: The host's IP address is `127.0.0.1` (this means it's your local system).

b. Middle Section (Host State Information)

This section shows the **current health and performance** of the host:

- **Host Status**: **UP** (green color) means the system is working fine.
- **Status Information**: Nagios checks if the system is alive using a "ping" command.
    - **No Packet Loss**: The system is responding properly.

  - **Round-Trip Time (RTA)**: The time it takes for a message to go to the system and back is very fast.
- **Last Check Time**: The last time Nagios checked the system.
- **Next Check**: When the next check will happen.
- **Flapping**: Flapping occurs when a system repeatedly goes up and down. Here, it's stable (NO).
- **Active Checks**: Monitoring features like active checks and notifications are all **ENABLED** (green boxes).

c. Right Section (Host Commands)

You can perform actions for this host using these commands:

- **Locate Host on Map**: See the host's location on the network map.
- **Disable Active Checks**: Stop Nagios from checking this host automatically.
- **Re-schedule Next Check**: Force Nagios to check the host immediately.
- **Disable Notifications**: Turn off alerts for this host.
- **Schedule Downtime**: If you plan to shut down the host, use this to avoid unnecessary alerts.

Stopping Nagios

To stop Nagios, we need to stop the Docker container running it.

1. **Find the Container Name:** If you used the earlier command to start Nagios, the container name is **nagiosdemo**. If you're unsure about the name, follow these steps:
   - Open your terminal.
   - Type this command to see all running containers:

     ```
     docker ps
     ```

   - You'll see a list of containers. Look for the **CONTAINER NAME** column to find the container running Nagios (e.g., **nagiosdemo**).
2. **Stop the Container:** Use the following command to stop the Nagios container:

   ```
   docker stop nagiosdemo
   ```

- `nagiosdemo`: This is the name of the container. Replace it with the actual container name if it's different.

Step 2: Deleting the Nagios Container

If you no longer need Nagios, you can delete the container.

1. **Remove the Container:** Use this command to delete the container:

   ```
   docker rm nagiosdemo
   ```

   - This permanently deletes the Nagios container.
   - If the container is still running, you'll see an error. In that case, ensure you've stopped it first (Step 1).

Step 3: Deleting the Nagios Image

1. **Find the Image Name:** To see all Docker images on your system, type:

   ```
   docker images
   ```

   - Look for the **IMAGE NAME** column. You should see something like `jasonrivers/nagios`.
2. **Delete the Image:** Use this command to remove the image:

   ```
   docker rmi jasonrivers/nagios:latest
   ```

   - If the image is still being used by a container, you'll get an error. Make sure you've already deleted the container (Step 2).

SOFTWARE ENGINEERING LAB

EXERCISE – 8

TOPIC – 1

AWS ACADEMY LEARNING ACCOUNT CREATION

In this exercise, we will be:

- Set up an AWS Academy account to access cloud services for learning purposes.
- Navigate the AWS Academy Learner Lab to activate and explore its features.
- Create a free AWS account by completing email, billing, and phone verification steps.
- Access the AWS Management Console to explore services like EC2 for cloud-based tasks.
- Learn to start, manage, and terminate AWS lab sessions while adhering to usage limits.
- **Note: At every step take screenshots and save in a document**

AWS Academy Account Creation

1. **Check Your Email**
   - Look for an AWS Academy course invitation email.
   - Click on **Get Started**.
2. **Create Your Account**
   - Click **Create My Account**.
   - Set up a **password**.
   - Choose an Indian time zone (e.g., IST).
   - Check all the provided boxes.
   - Scroll down and click **Register**.
3. **Access the Account**
   - After account creation, click on **Account** in the window.
4. **Logout and Relogin Instructions**
   - To log out: Click **Logout**.

- To log back in:
  - Go to Google and search for **AWS Academy Login**.
  - From the search results, look for **Untitled** and click on it.
  - Select **Student Login**.
  - Enter your username and password, then click **Login**.

5. **Navigating the AWS Academy Learner Lab**
   - After logging in, click on **AWS Academy Learner Lab**.
   - Click on **Modules** and scroll down.
   - Select **AWS Academy Learner Lab**.
6. **Agree to Terms & Conditions**
   - Read through the terms and conditions provided.
   - Scroll down and click on **I Agree**.
7. **Start the Sandbox Environment**
   - If there's a **red dot** beside AWS, it indicates that the lab environment is in a stopped state.
   - Click **Start Lab** to activate the environment.
8. **Lab Environment Details**
   - Once the lab starts, you can use it for **4 hours**.
   - If additional time is needed, restart the lab to reset the timer.
   - You are provided with **$50 in free credits**:
     - **Important:** Exceeding this limit will block further AWS access using the same email ID.
   - Once AWS status turns **green**, click on it to access the AWS dashboard.
9. **Perform Exercises**
   - From the AWS dashboard, click on **EC2** (visible under the **Recently Visited** section).
   - Begin your lab exercises from this point.
10. **Ending the Lab**
    - After finishing your tasks, click **End Lab**.
    - Confirm by clicking **Yes**.
    - A **red dot** beside AWS indicates the lab environment has stopped.

The below is for your information, no need to take screenshots from here

Steps to Create a Free Tier AWS Account

1. **Visit the AWS Website**
   - Open aws.amazon.com.
   - Click **Create AWS Account**.
2. **Provide Account Details**
   - Enter your **email ID** and **name**.
   - A verification code will be sent to your email.
   - Enter the verification code on the website.
3. **Set Up Password**
   - Create and confirm a password for your account.
4. **Provide Contact Details**
   - Enter your contact information and agree to the **Terms and Conditions**.
   - Click **Create Account**.
5. **Billing Details**
   - Add your billing information for account verification.
   - AWS will temporarily charge **₹2** for verification purposes, which will be refunded once verification is complete.
6. **Phone Verification**
   - After billing verification, enter your **working contact number**.
   - Choose your **country** and click **Send SMS**.
   - Enter the verification code sent to your phone.
7. **Select a Plan**
   - Choose the **Basic Plan** (free).
   - Click **Complete Sign-Up**.
8. **Access the AWS Management Console**
   - After completing the sign-up, navigate to the **AWS Management Console**.
   - Select your role as **Academic/Researcher** and your interest as **DevOps**.
   - Click **Submit**.

Using the AWS Management Console

- Sign into the AWS console using your **Root User** credentials.
- The console dashboard will appear, where you can access and explore various AWS services.

SOFTWARE ENGINEERING LAB

EXERCISE – 8

TOPIC – 2

PROJECT DEPLOYMENT IN THE AWS CLOUD USING EC2 INSTANCE

In this exercise, we will be:

- Launch a virtual server (EC2 instance) on AWS.
- Install essential tools like Docker, Git, and Nano.
- Create and deploy a simple web application using Docker.
- Access the application online.
- Clean up resources to avoid unnecessary charges.

Note: At every step take screenshots and save in a document

Step 1: Log in to AWS and Go to EC2

In this step, we will log in to our AWS account and access the EC2 service.

1. Log in to your AWS account.
2. On the AWS homepage, click **Services**, then choose **EC2** under **Compute**.

Step 2: Launch an EC2 Instance

Here, we will set up a virtual server to host our web application.

1. Click **Launch Instance**.
2. Configure the settings as follows:
    - **Name**: Enter a name like "MyWebServer" to identify your server.
    - **Application and OS**: Choose **Ubuntu (Free Tier Eligible)**.

- **Instance Type**: Select **t2.micro** (1 CPU, 1 GB RAM).
- **Key Pair**: Create a new key pair, download the `.pem` file, and save it securely.
- **Network**: Enable **Allow HTTP/HTTPS traffic** to make your website accessible.
- **Storage**: Use the default 8 GB.

3. Click **Launch Instance** and wait until the status changes to "Running."

Step 3: Connect to the EC2 Instance

In this step, we will connect to our virtual server.

1. Select your instance, click **Connect**, and copy the **SSH command**.
2. Open **PowerShell** (Windows) or **Terminal** (Mac/Linux) on your computer.
3. Navigate to the folder where your `.pem` file is saved using the `cd` command.
4. Paste the SSH command and press Enter. Type "yes" if prompted.

Step 4: Prepare the Instance

Now, we will prepare the server by installing required tools.

1. **Update the system** to ensure all software is up to date:

```
sudo apt update
```

2. **Install Docker** to package and run our web application:

```
sudo apt-get install docker.io
```

3. **Install Git** to manage and download code:

```
sudo apt install git
```

4. **Install Nano** for editing files directly on the server:

```
sudo apt install nano
```

Step 5: Create Your Web Application

In this step, we will build a simple web page and upload it to GitHub.

1. On your computer, create a file named `index.html` and add the following content:

```
<html>
<head><title>My Webpage</title></head>
<body><h1>Hello from AWS!</h1></body>
</html>
```

2. Initialize Git in the file's folder:

```
git init
git add .
git commit -m "First commit"
```

3. Create a GitHub repository, copy its HTTPS URL, and upload your file:

```
git remote add origin <Your_Repo_URL>
git push -u origin main
```

Step 6: Deploy the Web Application Using Docker

Here, we will deploy the web application to the EC2 instance.

1. On the EC2 instance, clone your GitHub repository:

```
git clone <Your_Repo_URL>
```

2. Create a `Dockerfile` in the project folder using Nano:

```
nano Dockerfile
```

Add the following content:

```
FROM nginx:alpine
COPY . /usr/share/nginx/html
```

Save the file by pressing **Ctrl + O**, then Enter, and exit Nano with **Ctrl + X.**

3. Build and run the Docker container to serve the web application:

```
sudo docker build -t my-web-app .
sudo docker run -d -p 80:80 my-web-app
```

Step 7: Access Your Web Application

In this step, we will view the deployed web page online.

1. Copy the **Public IP Address** of your EC2 instance from the AWS console.
2. Paste it into your browser (e.g., **http://<Public_IP>).**
3. You'll see your web page with the message "Hello from AWS!" displayed.

Step 8: Clean Up

Finally, we will clean up resources to avoid any charges.

1. Stop the running Docker container:

```
sudo docker ps
sudo docker stop <Container_ID>
```

2. Terminate the EC2 instance in the AWS console by selecting it, clicking **Instance State**, and choosing **Terminate Instance**.

SOFTWARE ENGINEERING LAB

EXERCISE – 8

TOPIC – 3

MAVEN WEB PROJECT DEPLOYMENT IN THE AWS CLOUD USING EC2 INSTANCE

In this exercise, we will be:

- Launch an EC2 instance on AWS.
- Install Docker, Git, and Nano.
- Deploy a Maven web project using Docker and expose it on port 9090.
- Access the application online.
- Clean up resources to avoid unnecessary charges.

Note:

1. At every step take screenshots and save in a document.

2. This guide uses JDK 11, which was used for developing the project. If your project was developed with JDK 21, adjust the Dockerfile as instructed in Step 5.

Step 1: Launch an EC2 Instance

In this step, we will set up a virtual server to host our Maven web project.

1. **Log in to AWS**: Access your AWS account and navigate to **Services > Compute > EC2**.
2. **Launch the instance**:
   - **Name**: Enter a descriptive name, e.g., `MavenWebProjectServer`.
   - **AMI**: Select **Ubuntu Server (Free Tier Eligible)**.
   - **Instance Type**: Choose **t2.micro**.
   - **Key Pair**: Create a key pair or use an existing one. Save the `.pem` file securely.
   - **Network Settings**: Enable **Allow HTTP/HTTPS traffic**.

- **Storage**: Use the default size (8 GB).

3. Click **Launch Instance** and wait for the status to change to "Running."
4. **Note down the Public IP Address** from the EC2 dashboard.

Step 2: Connect to the EC2 Instance

In this step, we will connect to the server.

1. Open **PowerShell** (Windows) or **Terminal** (Mac/Linux) and navigate to the folder with the `.pem` file using the `cd` command.
2. Use SSH to connect to the instance:

   ```
   ssh -i "<KeyFile>.pem" ubuntu@<Public_IP>
   ```

   Replace `<KeyFile>` with the `.pem` file name and `<Public_IP>` with your instance's public IP.

3. If prompted, type "yes" to confirm the connection.

Step 3: Prepare the EC2 Server

Now, we will install the necessary tools.

1. **Update the system**:

   ```
   sudo apt update
   ```

2. **Install Docker**:

   ```
   sudo apt-get install docker.io -y
   ```

3. **Install Git**:

   ```
   sudo apt install git -y
   ```

4. **Install Nano** (text editor):

   ```
   sudo apt install nano -y
   ```

Step 4: Clone Your Maven Web Project

In this step, we will download the Maven project from GitHub.

1. Go to your GitHub repository, click **Code > HTTPS**, and copy the URL.
2. Clone the repository:

   ```
   git clone <Your_Repo_URL>
   ```

   Replace `<Your_Repo_URL>` with the copied URL.

Step 5: Create a Dockerfile

We will create a Dockerfile to containerize the Maven project.

1. Navigate to the project folder:

   ```
   cd <Your_Project_Folder>
   ```

   Replace `<Your_Project_Folder>` with the folder name.

2. Open Nano to create the Dockerfile:

   ```
   nano Dockerfile
   ```

3. Add the following content based on the JDK version used during development:
   - For **JDK 11** (used in this guide):

     ```
     FROM tomcat:9-jdk11
     COPY target/*.war /usr/local/tomcat/webapps/
     ```

   - For **JDK 21**:

     ```
     FROM tomcat:9-jdk21
     COPY target/*.war /usr/local/tomcat/webapps/
     ```

4. Save and exit Nano: Press `Ctrl + O`, then `Enter`, and `Ctrl + X`.

Explanation:

- `FROM tomcat:9-jdk11` or `FROM tomcat:9-jdk21` specifies the Tomcat base image with the appropriate JDK version.
- `COPY target/*.war /usr/local/tomcat/webapps/` copies the `.war` file into the webapps directory of Tomcat for deployment.

Step 6: Build and Run the Docker Container

1. **Build the Docker image**:

   ```
   sudo docker build -t maven-web-project .
   ```

2. **Run the container**:

   ```
   sudo docker run -d -p 9090:8080 maven-web-project
   ```

   - `-d:` Runs the container in the background.
   - `-p 9090:8080:` Maps port 9090 on your instance to port 8080 in the container.

Step 7: Configure Security Group for Port 9090

We will ensure the EC2 instance allows traffic on port 9090.

1. In the AWS EC2 dashboard, go to **Security** and click the **Security Group ID**.
2. Add an inbound rule:
   - **Type**: Custom TCP
   - **Port Range**: 9090
   - **Source**: Anywhere (0.0.0.0/0) or your IP.
3. Save the changes.

Step 8: Access the Web Application

We will now test the deployment.

1. Open a browser and navigate to:

```
http://<Public_IP>:9090/<Your_Project_Name>
```

Replace **<Public_IP>** with the instance's public IP and **<Your_Project_Name>** with your Maven project name.

Step 9: Clean Up

Finally, we will stop the container and terminate the instance.

1. **Stop the Docker container**:

   ```
   sudo docker ps
   sudo docker stop <Container_ID>
   ```

   Replace **<Container_ID>** with the container ID.

2. **Terminate the EC2 instance** In the EC2 dashboard, go to **Instance State** and select **Terminate Instance**.