

# **1. USE CASE DIAGRAM**

## **1.1 USE CASE DIAGRAM FOR ATM**

A Use Case Diagram is a vital tool in system design, it provides a visual representation of how users interact with a system. It serves as a blueprint for understanding the functional requirements of a system from a user's perspective, aiding in the communication between stakeholders and guiding the development process.

### **USE CASE DIAGRAM NOTATIONS**

#### **Actors**

Actors are external entities that interact with the system. These can include users, other systems, or hardware devices. In the context of a Use Case Diagram, actors initiate use cases and receive the outcomes. Proper identification and understanding of actors are crucial for accurately modeling system behavior.

#### **Use Case**

Use cases are like scenes in the play. They represent specific things your system can do. In the ATM system, examples of use cases could be "Inset Card", "Enter Pin", or "Check Balance". Use cases are represented by ovals.

#### **System Boundary**

The system boundary is a visual representation of the scope or limits of the system you are modeling. It defines what is inside the system and what is outside. The boundary helps to establish a clear distinction between the elements that are part of the system and those that are external to it. The system boundary is typically represented by a rectangular box that surrounds all the use cases of the system.

### **USE CASE RELATIONSHIPS**

In a Use Case Diagram, relationships play a crucial role in depicting the interactions between actors and use cases. These relationships provide a comprehensive view of the system's functionality and its various scenarios.

#### **Association Relationship**

The Association Relationship represents a communication or interaction between an actor and a use case. It is depicted by a line connecting the actor to the use case. This

relationship signifies that the actor is involved in the functionality described by the use case.

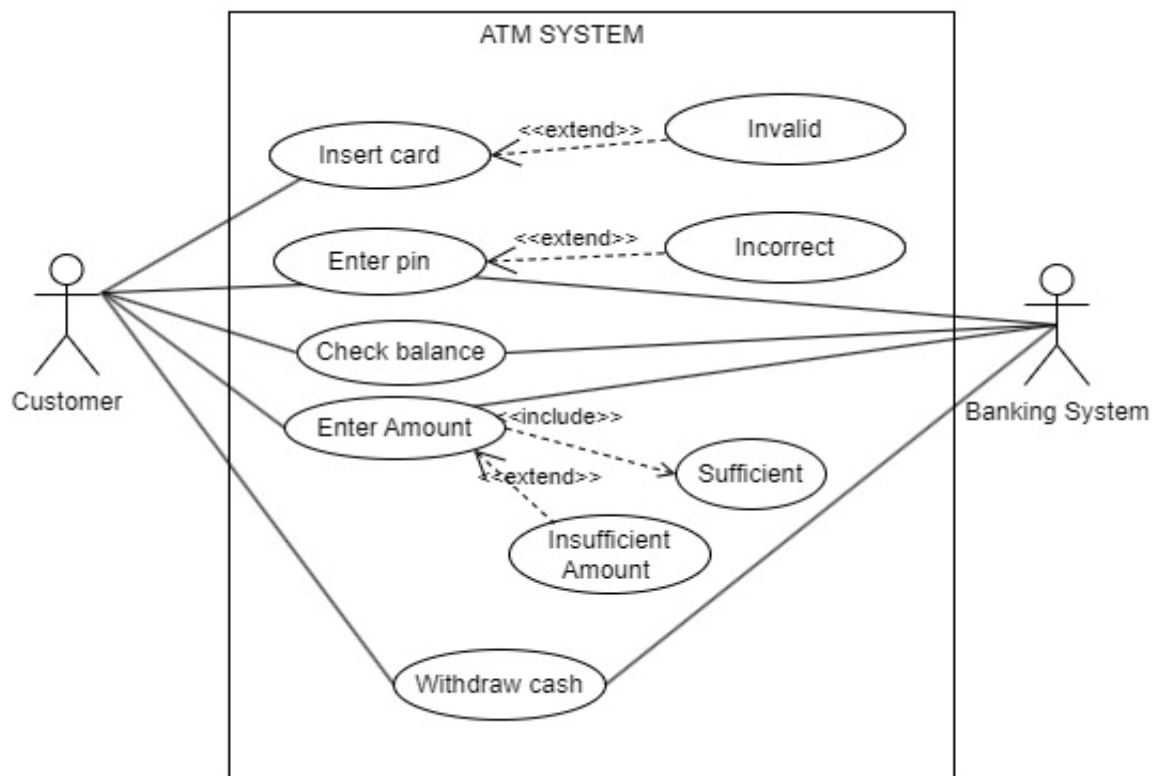
### **Include Relationship**

The Include Relationship indicates that a use case includes the functionality of another use case. It is denoted by a dashed arrow pointing from the including use case to the included use case. This relationship promotes modular and reusable design

### **Extend Relationship**

The Extend Relationship illustrates that a use case can be extended by another use case under specific conditions. It is represented by a dashed arrow with the keyword “extend.” This relationship is useful for handling optional or exceptional behavior

### **Example:**



*Figure 1.1 Use Case Diagram of ATM system*

## **1.2 USE CASE DIAGRAM FOR LIBRARY**

A Use Case Diagram is a vital tool in system design, it provides a visual representation of how users interact with a system. It serves as a blueprint for understanding the functional requirements of a system from a user's perspective, aiding in the communication between stakeholders and guiding the development process.

### **USE CASE DIAGRAM NOTATIONS**

#### **Actors**

Actors are external entities that interact with the system. These can include users, other systems, or hardware devices. In the context of a Use Case Diagram, actors initiate use cases and receive the outcomes. Proper identification and understanding of actors are crucial for accurately modeling system behavior.

#### **Use Case**

Use cases are like scenes in the play. They represent specific things your system can do. In the library system, examples of use cases could be "Search Book", "Issue Book", or "Return Book". Use cases are represented by ovals.

#### **System Boundary**

The system boundary is a visual representation of the scope or limits of the system you are modeling. It defines what is inside the system and what is outside. The boundary helps to establish a clear distinction between the elements that are part of the system and those that are external to it. The system boundary is typically represented by a rectangular box that surrounds all the use cases of the system.

### **USE CASE RELATIONSHIPS**

In a Use Case Diagram, relationships play a crucial role in depicting the interactions between actors and use cases. These relationships provide a comprehensive view of the system's functionality and its various scenarios.

#### **Association Relationship**

The Association Relationship represents a communication or interaction between an actor and a use case. It is depicted by a line connecting the actor to the use case. This relationship signifies that the actor is involved in the functionality described by the use case.

### Include Relationship

The Include Relationship indicates that a use case includes the functionality of another use case. It is denoted by a dashed arrow pointing from the including use case to the included use case. This relationship promotes modular and reusable design

### Extend Relationship

The Extend Relationship illustrates that a use case can be extended by another use case under specific conditions. It is represented by a dashed arrow with the keyword “extend.” This relationship is useful for handling optional or exceptional behavior

### Example:

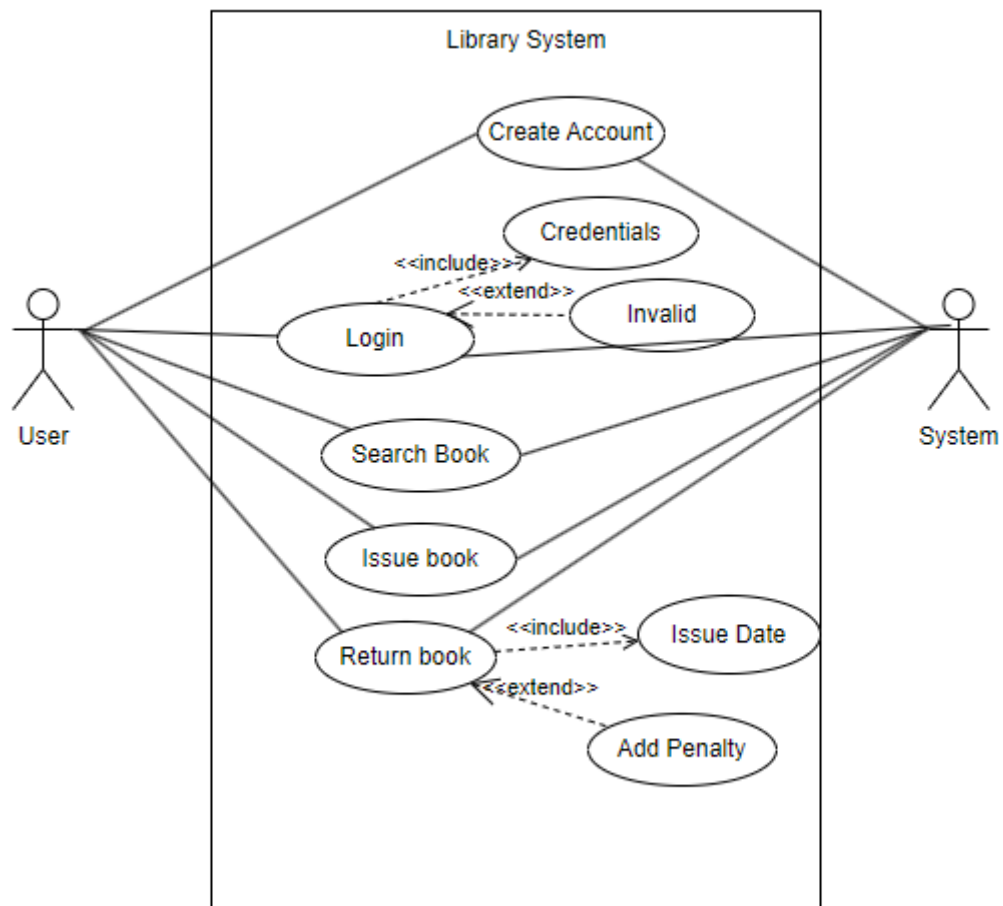


Figure 1.2 Use Case Diagram for Library System

## **2. ACTIVITY DIAGRAM**

### **2.1ACTIVITY DIAGRAM OF ATM**

Activity Diagrams are used to illustrate the flow of control in a system and refer to the steps involved in the execution of a use case. We can depict both sequential processing and concurrent processing of activities using an activity diagram i.e. an activity diagram focuses on the condition of flow and the sequence in which it happens.

#### **ACTIVITY DIAGRAM NOTATIONS**

##### **Initial State**

The starting state before an activity takes place is depicted using the initial state. A process can have only one initial state unless we are depicting nested activities. We use a black filled circle to depict the initial state of a system.

##### **Action or Activity State**

An activity represents execution of an action on objects or by objects. We represent an activity using a rectangle with rounded corners. Basically, any action or event that takes place is represented using an activity.

##### **Action Flow or Control flows**

Action flows or Control flows are also referred to as paths and edges. They are used to show the transition from one activity state to another activity state. An activity state can have multiple incoming and outgoing action flows. We use a line with an arrow head to depict a Control Flow.

##### **Decision node and Branching**

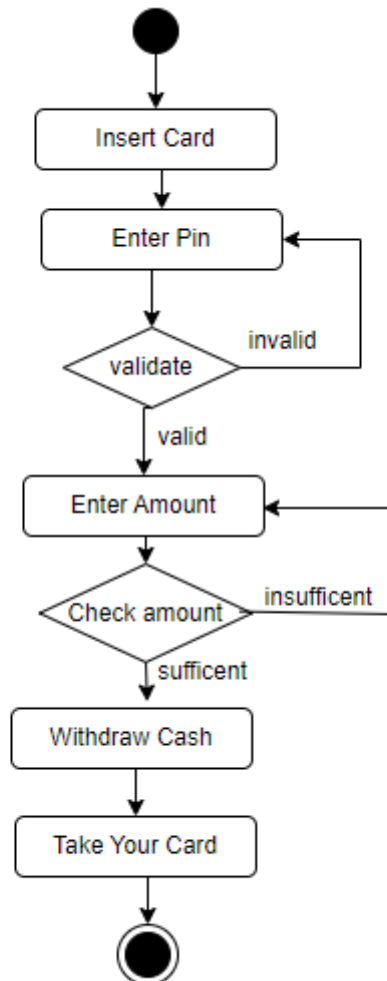
When we need to make a decision before deciding the flow of control, we use the decision node. The outgoing arrows from the decision node can be labelled with conditions or guard expressions. It always includes two or more output arrows.

##### **Fork**

Fork nodes are used to support concurrent activities. When we use a fork node when both the activities get executed concurrently i.e. no decision is made before splitting the activity into two parts. Both parts need to be executed in case of a fork statement. We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent activity state and outgoing arrows towards the newly created activities. Final State or End State The state which the system reaches when a particular

process or activity ends is known as a Final State or End State. We use a filled circle within a circle notation to represent the final state in a state machine diagram. A system or a process can have multiple final states.

**Example:**



*Figure 2.1 Activity Diagram for ATM withdraw*

## **2.2 ACTIVITY DIAGRAM OF ATM**

Activity Diagrams are used to illustrate the flow of control in a system and refer to the steps involved in the execution of a use case. We can depict both sequential processing and concurrent processing of activities using an activity diagram i.e. an activity diagram focuses on the condition of flow and the sequence in which it happens.

### **ACTIVITY DIAGRAM NOTATIONS**

#### **Initial State**

The starting state before an activity takes place is depicted using the initial state. A process can have only one initial state unless we are depicting nested activities. We use a black filled circle to depict the initial state of a system.

#### **Action or Activity State**

An activity represents execution of an action on objects or by objects. We represent an activity using a rectangle with rounded corners. Basically, any action or event that takes place is represented using an activity.

#### **Action Flow or Control flows**

Action flows or Control flows are also referred to as paths and edges. They are used to show the transition from one activity state to another activity state. An activity state can have multiple incoming and outgoing action flows. We use a line with an arrow head to depict a Control Flow.

#### **Decision node and Branching**

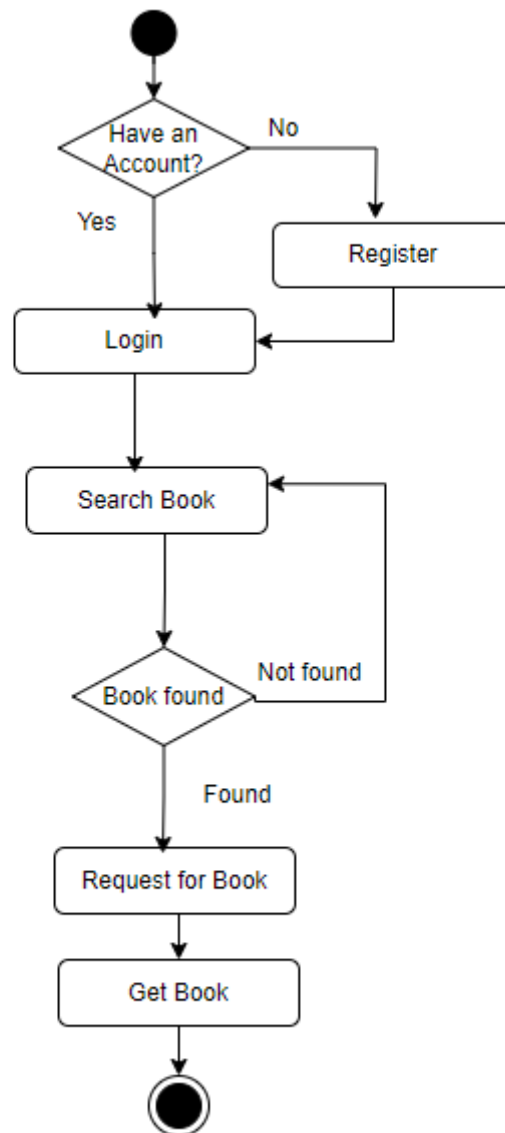
When we need to make a decision before deciding the flow of control, we use the decision node. The outgoing arrows from the decision node can be labelled with conditions or guard expressions. It always includes two or more output arrows.

#### **Fork**

Fork nodes are used to support concurrent activities. When we use a fork node when both the activities get executed concurrently i.e. no decision is made before splitting the activity into two parts. Both parts need to be executed in case of a fork statement. We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent activity state and outgoing arrows towards the newly created activities. Final State or End State The state which the system reaches when a particular process or activity ends is known as a Final State or End State. We use a filled circle

within a circle notation to represent the final state in a state machine diagram. A system or a process can have multiple final states.

**Example:**



*Figure 2.2 Activity Diagram for Library System*



### **3. SEQUEST DIAGRAM**

#### **3.1SEQUENCE DIAGRAM OF ATM**

A Sequence diagram is used to show the interactive behavior of a system. Since visualizing the interactions in a system can be difficult, we use different types of interaction diagrams to capture various features and aspects of interaction in a system.

#### **SEQUENCE DIAGRAM NOTATIONS**

##### **Actors**

An actor in a UML diagram represents a type of role where it interacts with the system and its objects. It is important to note here that an actor is always outside the scope of the system we aim to model using the UML diagram.

##### **Lifelines**

A lifeline is a named element which depicts an individual participant in a sequence diagram. So basically, each instance in a sequence diagram is represented by a lifeline. Lifeline elements are located at the top in a sequence diagram.

##### **Messages**

Communication between objects is depicted using messages. The messages appear in a sequential order on the lifeline.

##### **Synchronous messages**

A synchronous message waits for a reply before the interaction can move forward. The sender waits until the receiver has completed the processing of the message. The caller continues only when it knows that the receiver has processed the previous message i.e., it receives a reply message.

##### **Asynchronous Messages**

An asynchronous message does not wait for a reply from the receiver. The interaction moves forward irrespective of the receiver processing the previous message or not. We use a lined arrow head to represent an asynchronous message.

##### **Self Message**

Certain scenarios might arise where the object needs to send a message to itself. Such messages are called Self Messages and are represented with a U-shaped arrow.

## Create message

We use a Create message to instantiate a new object in the sequence diagram. There are situations when a particular message call requires the creation of an object. It is represented with a dotted arrow and create word labelled on it to specify that it is the create Message symbol. Reply Message Reply messages are used to show the message being sent from the receiver to the sender. We represent a return/reply message using an open arrow head with a dotted line. The interaction moves forward only when a reply message is sent by the receiver.

## Example:

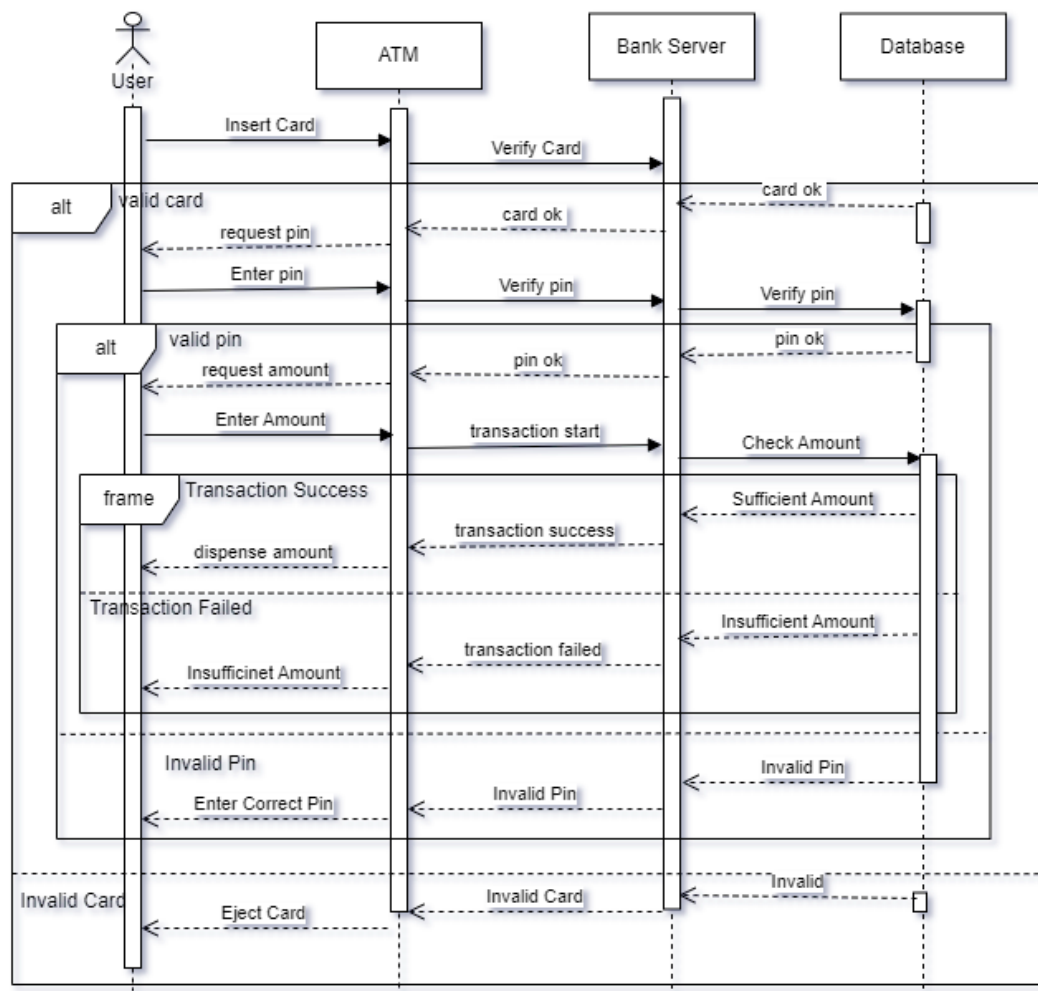


Figure 3.1 Sequence Diagram of ATM system

**A**  
**LAB REPORT**  
**ON**  
**OBJECT ORIENTED ANALYSIS**  
**AND**  
**DESIGN**

**By**  
**Roshan Nepal (12093/20)**  
**BIM 7<sup>th</sup> Semester**



**Submitted to:**

**Mr. Dinesh Deuba**

**Department of Management**

**Nepal Commerce Campus**

In partial fulfilment of the requirements for the Course

Object Oriented Analysis and Design

Min-Bhawan, Kathmandu

July 2024

## 4. Design Pattern

### Theory

Design patterns are reusable solutions to common software design problems, providing a standard terminology and a template for solving issues that frequently arise in software development. These patterns are not concrete implementations but rather a description of a solution that can be adapted to various scenarios.

### 4.1 Write a program to implement singleton pattern.

#### Source code:

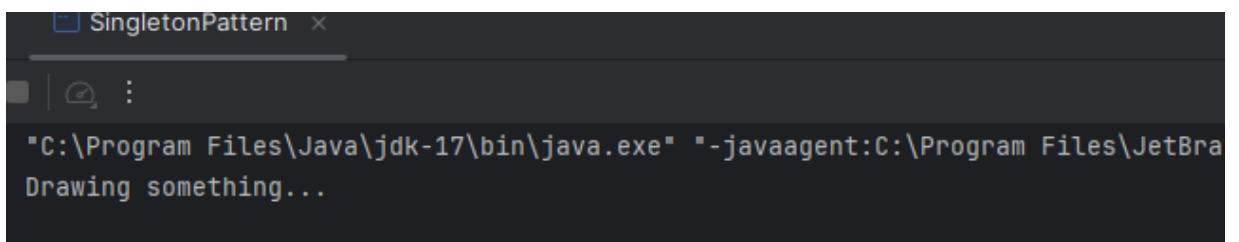
```
public class SingletonPattern {
    public static void main(String[] args) {
        Singleton singleton = Singleton.getInstance();
        singleton.draw();
    }
}

class Singleton {
    private static final Singleton singleton = new
    Singleton();

    private Singleton() {
    }

    public static Singleton getInstance() {
        return singleton;
    }
    public void draw(){
        System.out.println("Drawing something...");
    }
}
```

#### Output:

A screenshot of a Java IDE window titled "SingletonPattern". The console output shows the command "C:\Program Files\Java\jdk-17\bin\java.exe" followed by the output "Drawing something...".

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBra
Drawing something..."
```

## 4.2 Write a program to implement factory pattern.

### Source Code

```
public interface Shape {
    void draw();
}

public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}

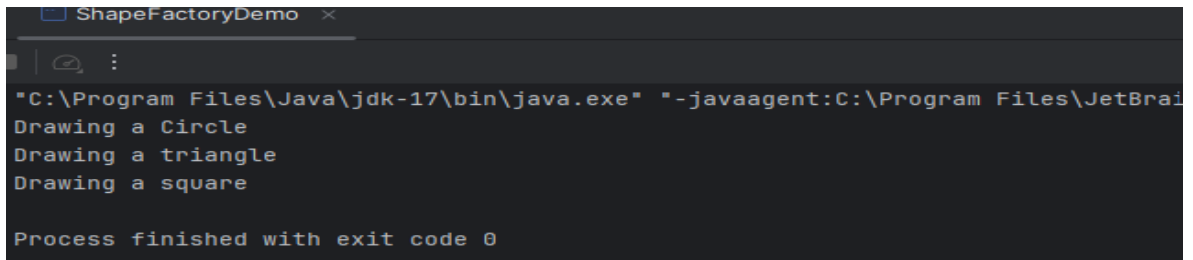
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a square");
    }
}

public class Triangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a triangle");
    }
}

public class ShapeFactory {
    public static Shape getShape(String shapeType) {
        if(shapeType == null) {
            return null;
        }
        if (shapeType.equalsIgnoreCase("Circle")) {
            return new Circle();
        }
        if (shapeType.equalsIgnoreCase("Triangle")) {
            return new Triangle();
        }
        if(shapeType.equalsIgnoreCase("Square")) {
            return new Square();
        }
        return null;
    }
}

public class ShapeFactoryDemo {
    public static void main(String[] args) {
        Shape shape = ShapeFactory.getShape("Circle");
        shape.draw();
        shape = ShapeFactory.getShape("Triangle");
        shape.draw();
        shape = ShapeFactory.getShape("Square");
        shape.draw();
    }
}
```

## Output:



```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrai
Drawing a Circle
Drawing a triangle
Drawing a square
Process finished with exit code 0
```

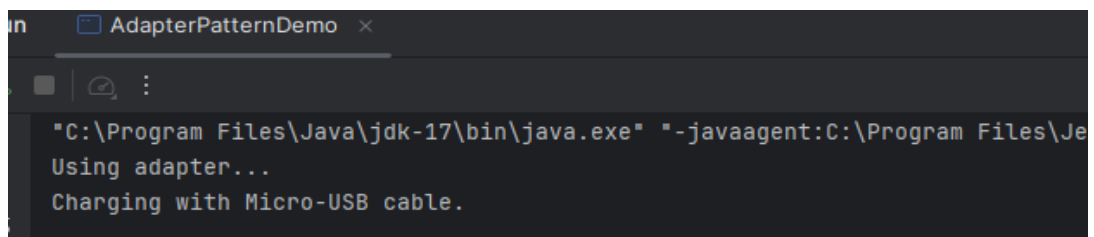
## 4.3 Write a program to implement adapter pattern

### Source code

```
public class TypeCPhone {
    public void chargewithTypeC() {
        System.out.println("Charging phone with Type-C
cable.");
    }
}
public class MicroUSBCharger {
    public void chargewithMicroUSB() {
        System.out.println("Charging with Micro-USB
cable.");
    }
}
public class MicroUSBToTypeCAdapter extends TypeCPhone
{
    private MicroUSBCharger microUSBCharger;
    public MicroUSBToTypeCAdapter(MicroUSBCharger
microUSBCharger) {
        this.microUSBCharger = microUSBCharger;
    }

    @Override
    public void chargewithTypeC() {
        System.out.println("Using adapter...");
        microUSBCharger.chargewithMicroUSB();
    }
}
public class AdapterPatternDemo {
    public static void main(String[] args) {
        MicroUSBCharger microUSBCharger = new
MicroUSBCharger();
        TypeCPhone adapter = new
MicroUSBToTypeCAdapter(microUSBCharger);
        adapter.chargewithTypeC();
    }
}
```

## Output:



```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\Je
Using adapter...
Charging with Micro-USB cable.
```

#### 4.4 Write a program to implement observer pattern

##### Source code

```
public interface Observer {
    void update(float temperature);
}

import java.util.ArrayList;
import java.util.List;

public class WeatherStation {
    private final List<Observer> observers = new
    ArrayList<>();
    private float temperature;

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    public void setTemperature(float temperature) {
        this.temperature = temperature;
        notifyObservers();
    }

    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature);
        }
    }
}

public class DisplayDevice implements Observer {
    private final String name;
    public DisplayDevice(String name) {
        this.name = name;
    }

    @Override
    public void update(float temperature) {
        System.out.println(name + " received
temperature update: " + temperature + "°C");
    }
}

public class ObserverPatternDemo {
    public static void main(String[] args) {
        WeatherStation weatherStation = new
        WeatherStation();

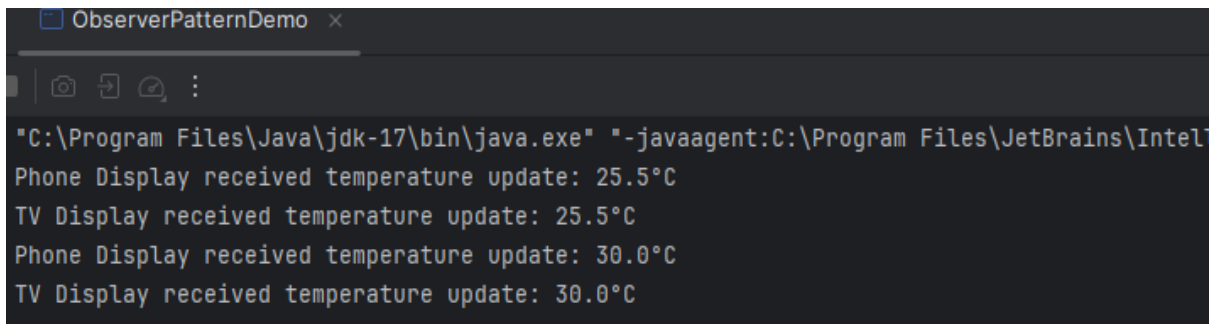
        DisplayDevice phoneDisplay = new
        DisplayDevice("Phone Display");
```

```
        DisplayDevice tvDisplay = new
DisplayDevice("TV Display");

        weatherStation.addObserver(phoneDisplay);
        weatherStation.addObserver(tvDisplay);

        weatherStation.setTemperature(25.5f);
        weatherStation.setTemperature(30.0f);
    }
}
```

**Output:**



```
ObserverPatternDemo x
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\Intel
Phone Display received temperature update: 25.5°C
TV Display received temperature update: 25.5°C
Phone Display received temperature update: 30.0°C
TV Display received temperature update: 30.0°C
```