“Heaven’s Light is Our Guide”

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Lab Report

Course Code: CSE 3206

Course Title: Software Engineering Sessional

Submitted By-	Submitted To-
Name: Sajidur Rahman Tarafder Roll No: 2003154 Name: Md Abdur Rahaman Roll No: 2003155 Name: Labiba Akter Biva Roll No: 2003156	Farjana Parvin Lecturer Department of CSE, RUET

Roll no : 2 003 155

Design Pattern Name : Prototype Design Pattern.

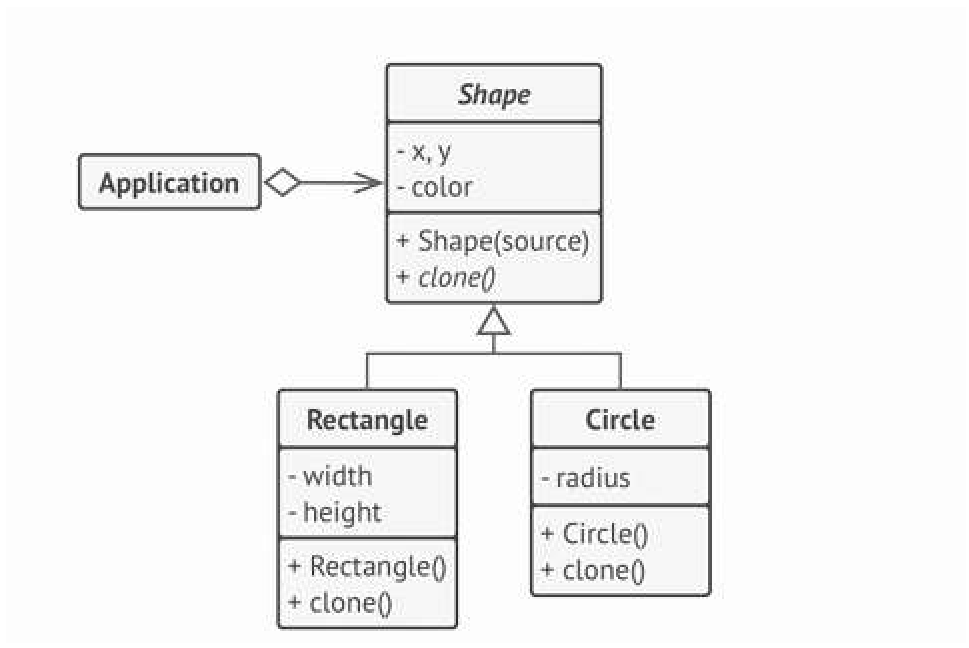
Definition :
The Prototype Pattern allows to create new objects by copying an existing object, called the prototype, instead of building the object from scratch or copying it directly.

Problems with the Direct Approach :
If the Prototype Pattern is not used, creating new objects directly can have these issues:
1. **Performance Overhead:** For complex objects, creating them from scratch can be slow.
2. **Repetition:** Code may be needed to repeat for initializing new objects, leading to redundancy and potential errors.
3. **Maintenance:** Any change in how the object is created requires updating all instances of its creation, making code harder to maintain.

Reasons to use it :
1. **Performance:** Creating a new object from scratch can be slow and resource-intensive. The Prototype Pattern saves time and resources by reusing an existing object.
2. **Complex Objects:** If an object has many settings or configurations, creating it by copying an existing object simplifies the process.
3. **Flexibility:** Slight changes can be made to the copied object without altering the original, making it easy to create variations.

Process to Use It :
1. **Create a Prototype Class:** This class should have a method like clone() that returns a copy of itself.
2. **Copy Instead of Create:** Use the clone() method to create new objects based on the prototype object.
3. **Modify as Needed:** After cloning, make any specific changes to the new object.

UML Diagram :



Explain :

This diagram represents the Prototype Design Pattern.

1. The Shape class is the base class with common properties like x, y, and color, and a clone() method for duplication.
2. Subclasses Rectangle and Circle inherit Shape and add specific properties like width/height for Rectangle and radius for Circle.
3. The Application uses the clone() method to create copies of shapes without modifying the originals.

Problem Statement :
Create an app where users can make and copy shapes like circles and rectangles. Each shape has details like color, size, and dimensions. Use the Prototype Design Pattern to copy shapes easily, so changes to the copy don't affect the original. Show this by creating, cloning, and changing shapes while keeping the original unchanged.

Code :

Shape.java :-
```
public abstract class Shape {
    String color;

    // Constructor
    public Shape(String color) {
        this.color = color;
    }

    // Clone method
    public abstract Shape clone();

    // Display details
    public abstract void display();
}
```

Circle.java :
```
public class Circle extends Shape {
    int radius;

    // Constructor
    public Circle(String color, int radius) {
        super(color);
        this.radius = radius;
    }
    // Clone method
    @Override
    public Circle clone() {
        return new Circle(this.color, this.radius);
    }

    // Display details
    @Override
    public void display() {
        System.out.println("Circle [Color: " + color + ", Radius: " + radius + "]");
    }
}
```

Rectangle.java :

```
public class Rectangle extends Shape {
    int width, height;

    // Constructor
    public Rectangle(String color, int width, int height) {
        super(color);
        this.width = width;
        this.height = height;
    }

    // Clone method
    @Override
    public Rectangle clone() {
        return new Rectangle(this.color, this.width, this.height);
    }

    // Display details
    @Override
    public void display() {
        System.out.println("Rectangle [Color: " + color + ", Width: " + width + ", Height: " + height + "]");
    }
}
```

Main.java :

```
public class Main {
    public static void main(String[] args) {

        // Create and display the original Circle

        Circle originalCircle = new Circle("Red", 10);
        System.out.println("\nOriginal Circle:");
        originalCircle.display();
        // Clone the Circle

        Circle clonedCircle = originalCircle.clone();
        System.out.println("Cloned Circle:");
        clonedCircle.display();


        // Modify the cloned Circle

        clonedCircle.color = "Blue";
        clonedCircle.radius = 15;

        // Display both the modified cloned Circle and the original Circle

        System.out.println("\nAfter modifying the cloned Circle:");
        System.out.println("Modified Cloned Circle:");
        clonedCircle.display();
        System.out.println("Original Circle (unchanged):");
        originalCircle.display();
    }
}
```

Output :

```
[Running] cd "c:\Users\abdur\OneDrive\Documents\Java\" && javac Main.java && java Main

Original Circle:
Circle [Color: Red, Radius: 10]
Cloned Circle:
Circle [Color: Red, Radius: 10]

After modifying the cloned Circle:

Modified Cloned Circle:
Circle [Color: Blue, Radius: 15]

Original Circle (unchanged):
Circle [Color: Red, Radius: 10]

[Done] exited with code=0 in 1.266 seconds
```

Roll: 2003154

Design Pattern Name: Singleton Design Pattern

Introduction:

Singleton is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. It is one of the simplest design patterns.

This pattern ensures a class has only one instance during the application's lifecycle. It prevents creating new instances by always returning the same one. This pattern allows easy access to the instance from anywhere in the program and protects it from being overwritten. It also supports extending functionality through subclassing and helps manage shared resources like logging and database connections.

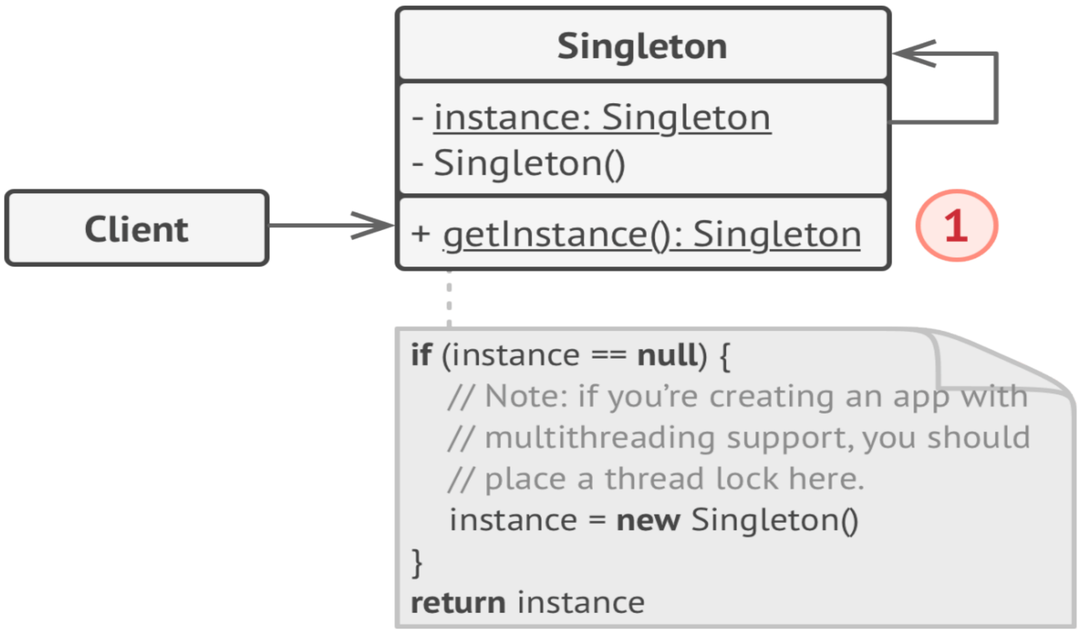
Components of Singleton Pattern:

Private Constructor: A regular constructor always returns a new object by design. This pattern solves this issue by making the default constructor as private constructor. It prevents creating fresh objects by using ‘new’ operator of that class.

Static Instance Variable: A static variable that holds the single instance of the class.

Public Static Method: Provides a global access point to the instance. It calls the private constructor internally and returns the same instance for all subsequent calls.

UML Diagram:



```
if (instance == null) {
   // Note: if you're creating an app with
   // multithreading support, you should
   // place a thread lock here.
   instance = new Singleton()
}
return instance
```

Explanation:

The Singleton class declares the static method getInstance that returns the same instance of its own class. The Singleton's constructor should be hidden from the client code. Calling the getInstance method should be the only way of getting the Singleton object.

Problem Statement:

Think of a country's government. There is only one central government at any time, no matter who is in charge. Everyone in the country refers to it as "The Government of X", making it a single, centralized point of control and access for managing the nation.

Implement this using Singleton pattern, where only one instance of a class exists, and everyone accesses it through a global point.

<u>Code</u>:

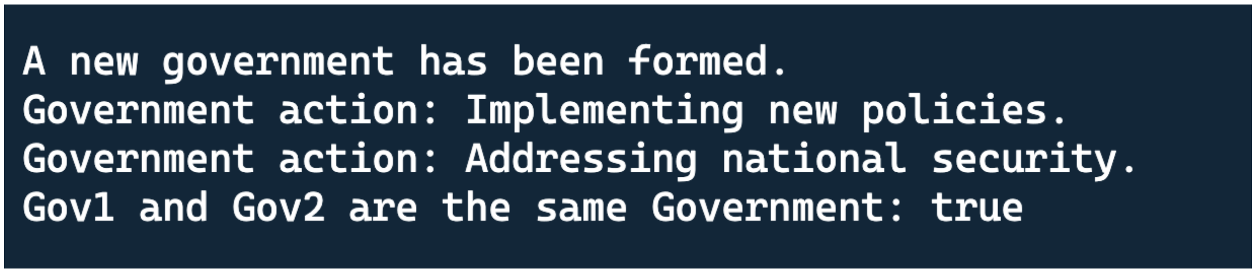
Government.java

```java
public class Government {
   private static Government instance;

   private Government()
  {
       System.out.println("\nA new government has been formed.");
   }



 public static Government getInstance() {
       if (instance == null) {
          instance = new Government();
       }
       return instance;
   }

   public void govern(String message) {
       System.out.println("Government action: " + message);
   }
}
```

Main.java

```java
public class Main
{
    public static void main(String[] args)
    {
        Government Gov1 = Government.getInstance();
        Government Gov2 = Government.getInstance();

        Gov1.govern("Implementing new policies.");
        Gov2.govern("Addressing national security.");

        System.out.println("Gov1 and Gov2 are the same Government: " + (Gov1 == Gov2) + '\n');
    }
}
```

<u>Terminal Output</u>:



```
A new government has been formed.
Government action: Implementing new policies.
Government action: Addressing national security.
Gov1 and Gov2 are the same Government: true
```

Roll:2003156

Adapter Design Pattern:

A structural design pattern is a blueprint in software development that focuses on how to organize and connect classes or objects to create larger, more efficient, and flexible systems. It ensures that components work well together while remaining easy to modify and reuse

- **Target Interface (AppleCharger):**
- Defines the chargePhone() method which Iphone13 uses.
- **Adaptee (AndroidCharger):**
- Defines the method chargerAndroidPhone(), which doesn't match the expected method of AppleCharger.
- **Adapter (AdapterCharger):**
- Implements the AppleCharger interface and translates the method calls to the appropriate methods of AndroidCharger.
- **Client Code (Iphone13):**
- The Iphone13 class is modified to use the AppleCharger interface, and it calls the chargePhone() method without worrying about the underlying charger implementation.

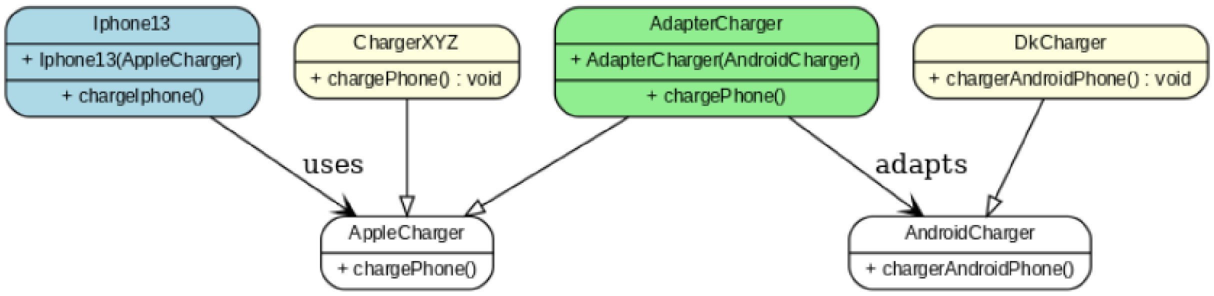
Problem Description:

1.The Iphone13 class is designed to work with AppleCharger, but the DkCharger is an AndroidCharger.
2. The AppleCharger interface doesn't support charging Android phones, and vice
Versa.

Solution Description:

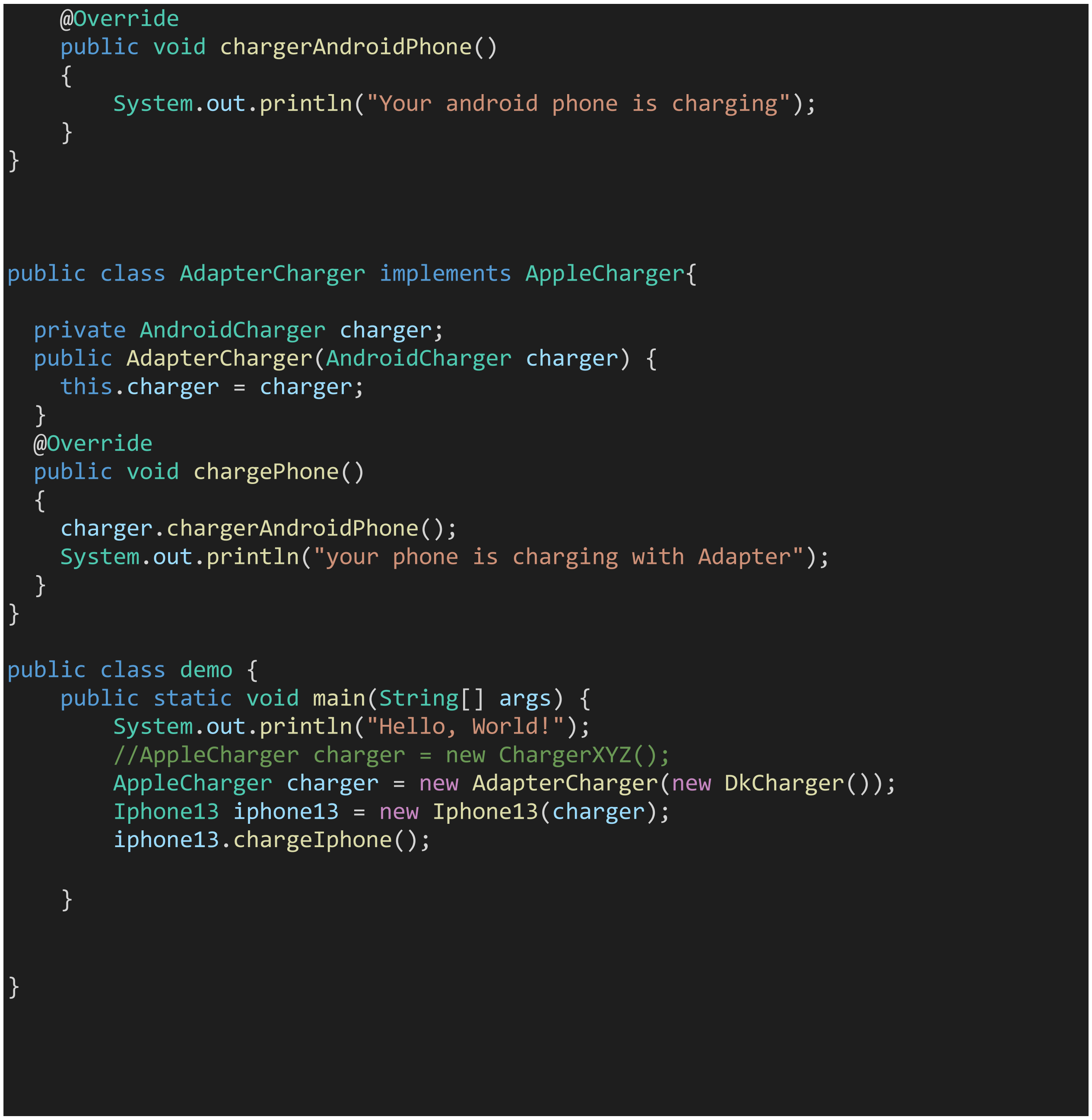
Create an AdapterCharger that implements the AppleCharger interface and adapts the AndroidCharger (DkCharger) to be compatible with the Iphone13.

UML Diagram:



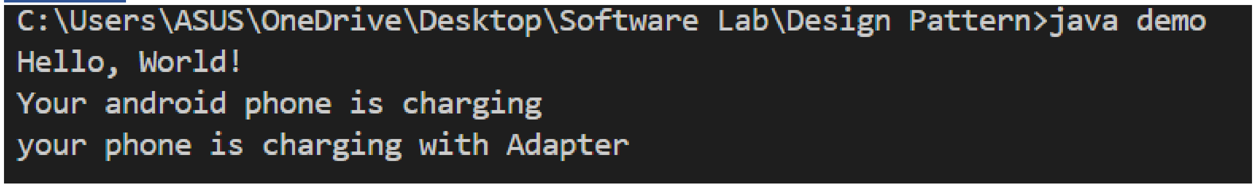
Code:

```
public class Iphone13
{
    private AppleCharger appleCharger;

    public Iphone13(AppleCharger appleCharger){
         this.appleCharger = appleCharger;
     }
    public void chargeIphone(){
        appleCharger.chargePhone();
    }
}

interface AppleCharger{


    void chargePhone();
}

public class ChargerXYZ implements AppleCharger {

    @Override
    public void chargePhone() {
        System.out.println("Your iphone is charging");
    }
}


interface AndroidCharger{
    void chargerAndroidPhone();
}


public class DkCharger implements AndroidCharger{
```



```java
    @Override
    public void chargerAndroidPhone()
    {
        System.out.println("Your android phone is charging");
    }
}



public class AdapterCharger implements AppleCharger{

  private AndroidCharger charger;
  public AdapterCharger(AndroidCharger charger) {
    this.charger = charger;
  }
  @Override
  public void chargePhone()
  {
    charger.chargerAndroidPhone();
    System.out.println("your phone is charging with Adapter");
  }
}

public class demo {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
        //AppleCharger charger = new ChargerXYZ();
        AppleCharger charger = new AdapterCharger(new DkCharger());
        Iphone13 iphone13 = new Iphone13(charger);
        iphone13.chargeIphone();

    }


}
```

Output:



```
C:\Users\ASUS\OneDrive\Desktop\Software Lab\Design Pattern>java demo
Hello, World!
Your android phone is charging
your phone is charging with Adapter
```