Software Development and Best Practices

Mahmoud Khalaf Saeed

Section (4)

DESIGN PATTERNS

Design Patterns in Java

A design patterns are **well-proved solution** for solving the specific problem/task.

design patterns are programming language **independent strategies** it represents an idea, not a particular implementation.

By using the design patterns, you can make your code more **flexible, reusable and maintainable**. java internally **follows** design patterns.

Advantage of design pattern:

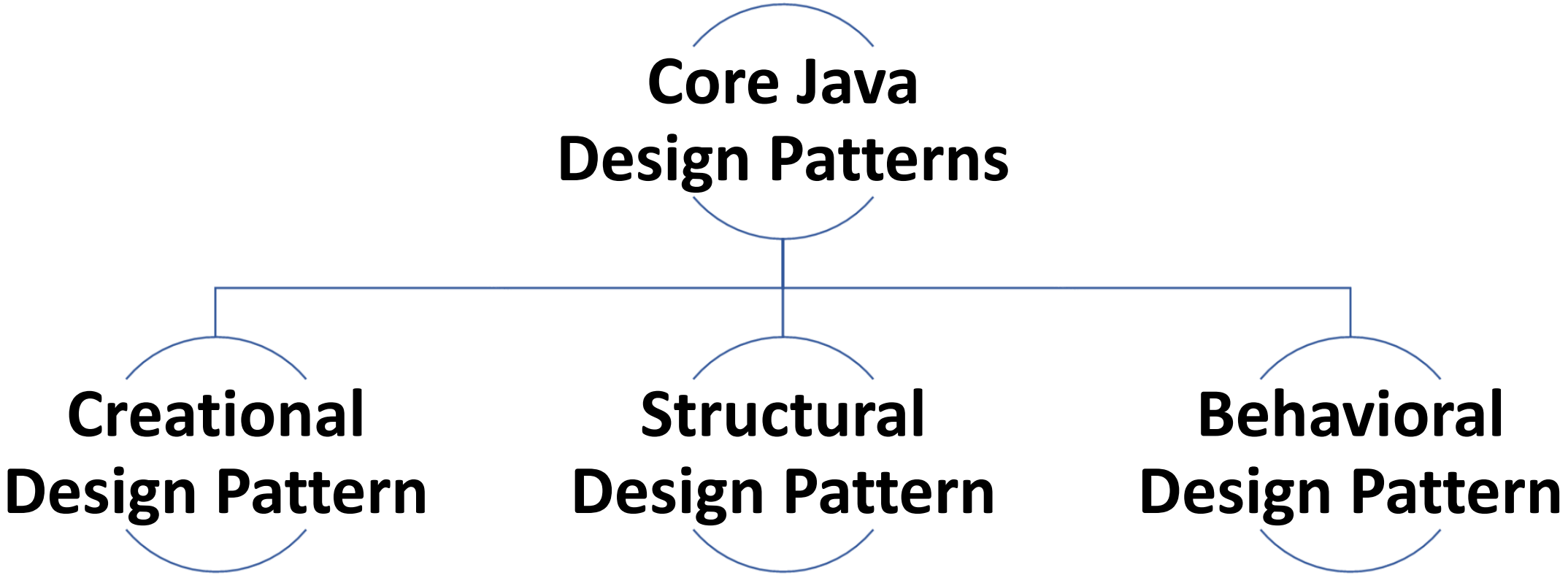
They are **reusable** in multiple projects. And provide the solutions that help to define the **system architecture.**

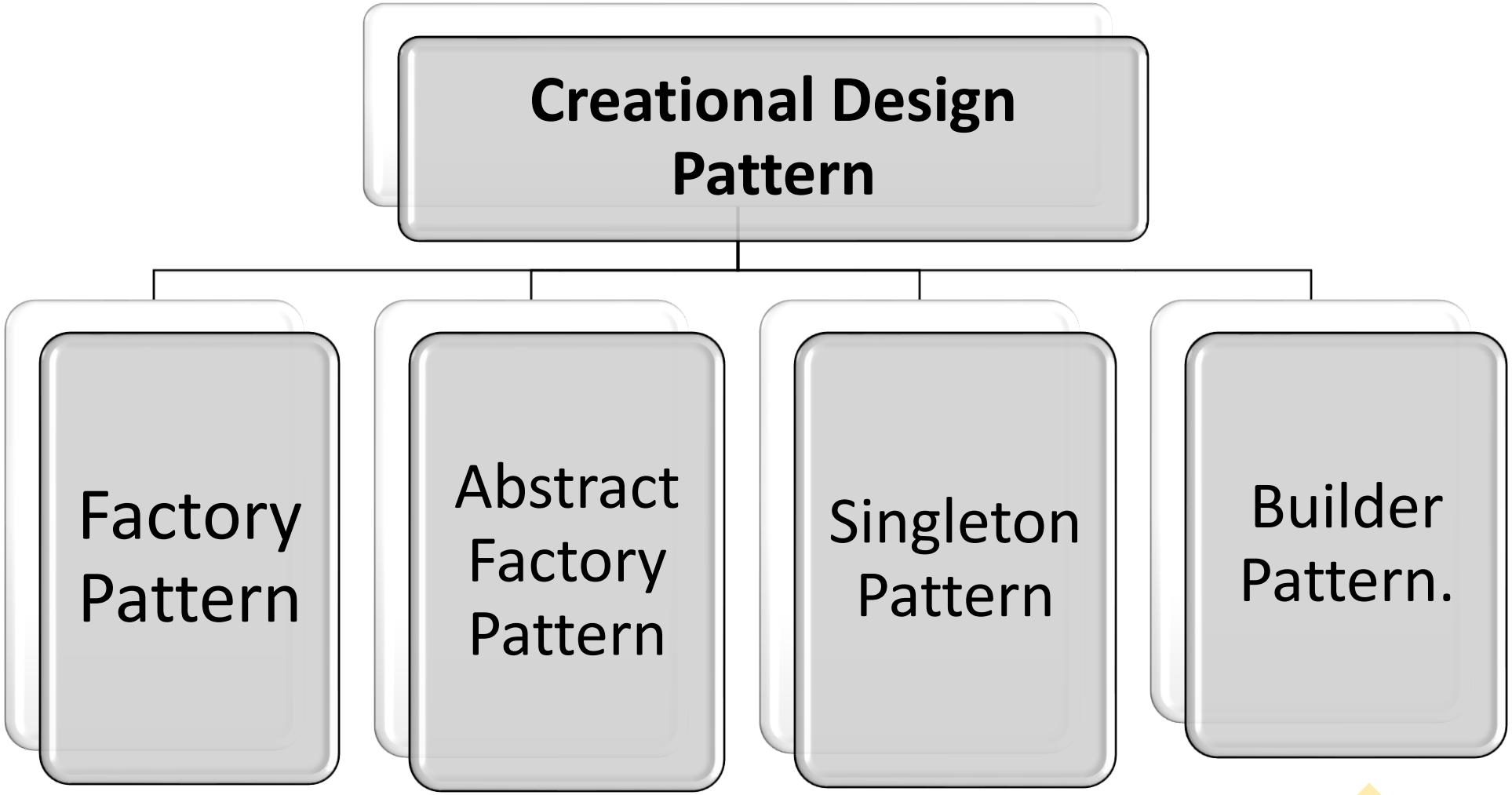
They capture the software engineering **experiences.**

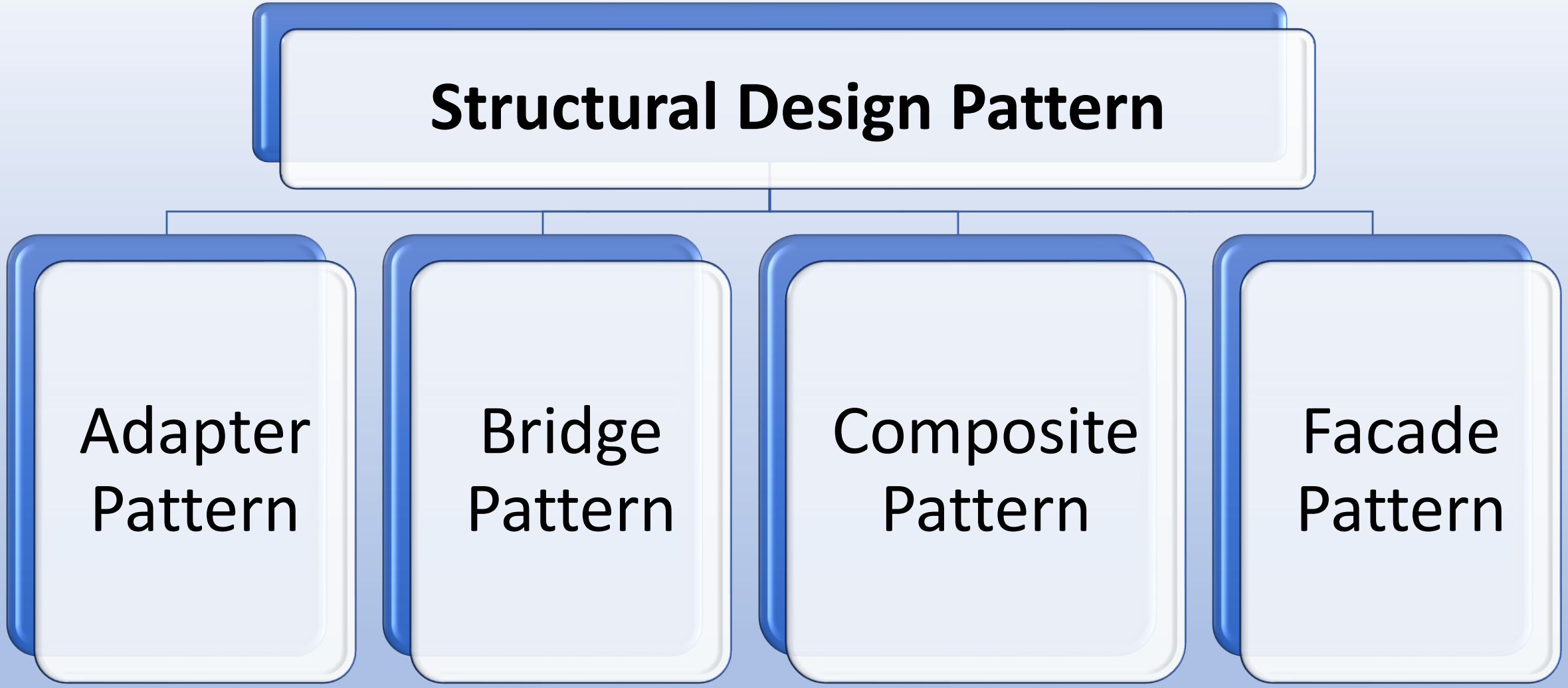
They are **well-proved** and testified solutions

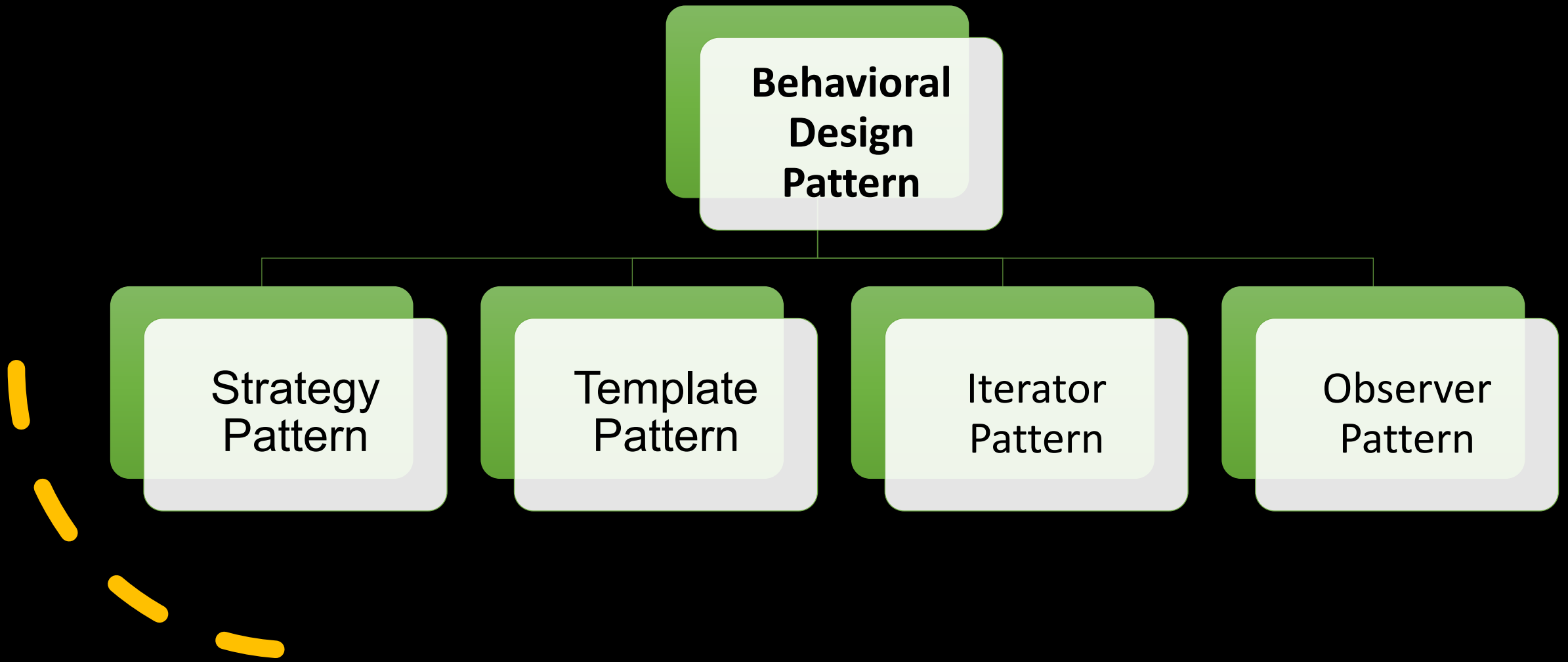
Design patterns don't guarantee an **absolute solution** They provide clarity to the system architecture

DESIGN PATTERNS TYPES

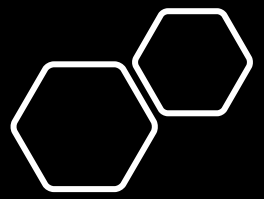

Core Java
Design Patterns
Creational
Design Pattern
Structural
Design Pattern
Behavioral
Design Pattern


Creational Design Pattern
Factory Pattern
Abstract Factory Pattern
Singleton Pattern
Builder Pattern.


Structural Design Pattern
Adapter Pattern
Bridge Pattern
Composite Pattern
Facade Pattern


Behavioral Design Pattern
Strategy Pattern
Template Pattern
Iterator Pattern
Observer Pattern

Creational Design Pattern



Factory Method Pattern

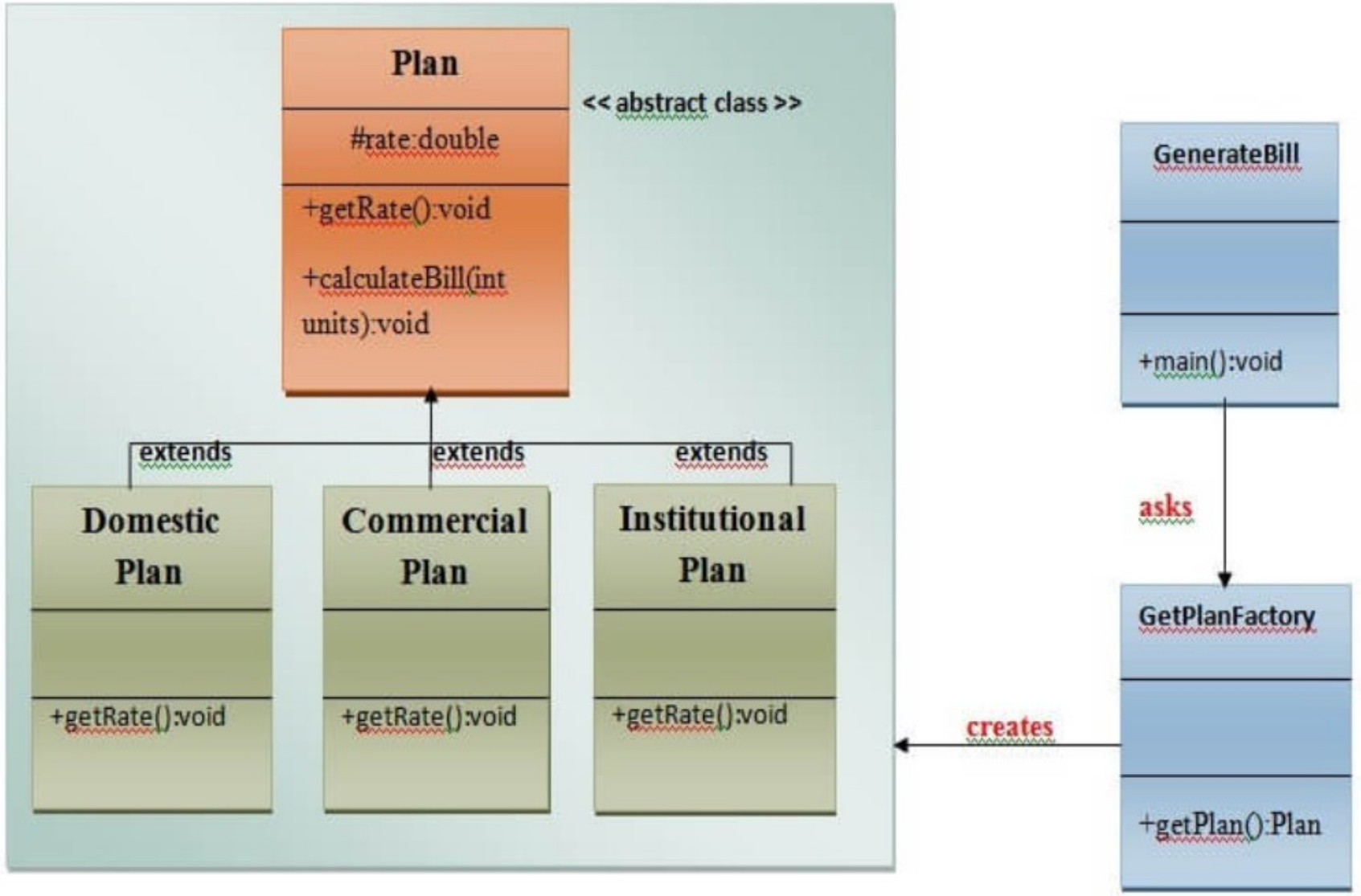
A Factory Pattern or Factory Method Pattern says that

just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.**

In other words, subclasses are responsible to create the instance of the class.

The Factory Method Pattern is also known as **Virtual Constructor.**

example


Plan
<< abstract class >>
#rate:double
+getRate():void
+calculateBill(int units):void
extends
extends
extends
Domestic Plan
Commercial Plan
Institutional Plan
+getRate():void
+getRate():void
+getRate():void
GenerateBill
+main():void
asks
GetPlanFactory
+getPlan():Plan
creates

```
package CreationalDesignPattern;
    public abstract class Plan
    {
        protected double rate;
         abstract void getRate() ;

         public void calculateBill(int units)
         {
             System.out.println(units*rate) ;
          }
    }
```

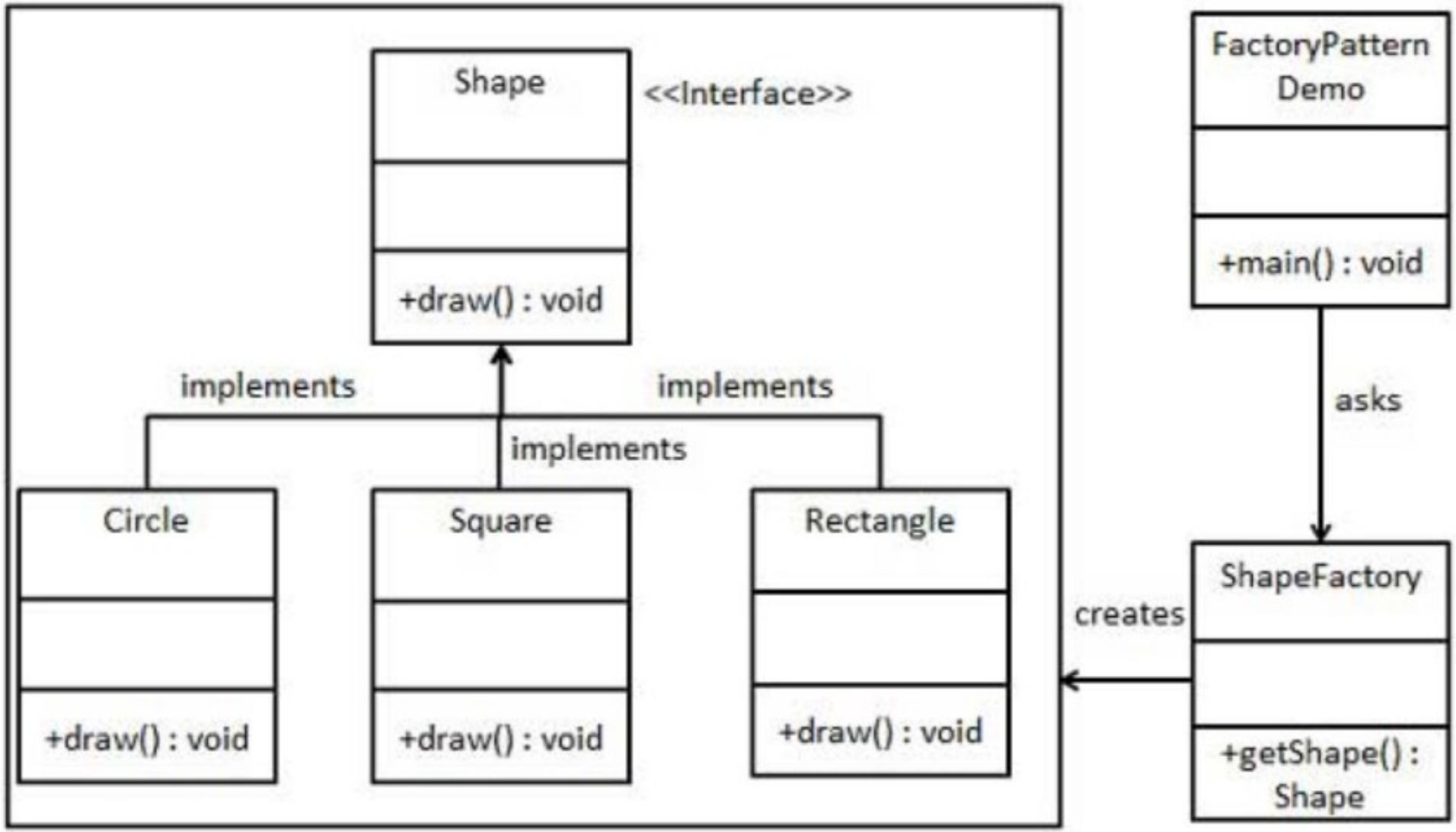
```java
package CreationalDesignPattern;
    //Step 2: Create the concrete classes that extends Plan abstract class.
    public class DomesticPlan extends Plan
        {
            void getRate()
            {
                rate=3.50;
            }
        }
```

```java
package CreationalDesignPattern;
    //Step 2: Create the concrete classes that extends Plan abstract class.
    public class CommercialPlan extends Plan
    {
        void getRate()
            {
                rate=7.50;
            }

    }
```

```
package CreationalDesignPattern;
    //Step 2: Create the concrete classes that extends Plan abstract class.
    public class InstitutionalPlan extends Plan
    {
        void getRate()
            {
                rate=5.50;
            }
    }
```

```
package CreationalDesignPattern;
    //Step 3: Create a GetPlanFactory to generate object of
    // concrete classesbased on given information..
    public class GetPlanFactory
    {
       //use getPlan method to get object of type Plan
      public Plan getPlan(String planType)
      {
          if(planType == null){
           return null;
       }
          if(planType.equalsIgnoreCase("DOMESTICPLAN"))
              {
                  return new DomesticPlan();
              }
          else if(planType.equalsIgnoreCase("COMMERCIALPLAN"))
              {
                 return new CommercialPlan();
              }
          else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN"))
               {
                  return new InstitutionalPlan();
               }
      return null;
   }

}
```

```java
package CreationalDesignPattern;
import java.util.Scanner;
public class GenerateBill
    {
     public static void main(String args[])
     {
      Scanner input = new Scanner(System.in);
      //generate object of concrete classes based on input
      GetPlanFactory planFactory = new GetPlanFactory();

      System.out.print("Enter the plan name for which the bill will be generated: ");
      String planName = input.nextLine();

      System.out.print("Enter the number of units for bill will be calculated: ");
       int  units = input.nextInt();
      // getplan method return plan type
      Plan p = planFactory.getPlan(planName);
      //call getRate() method and calculateBill()method of DomesticPaln.
       System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");
          p.getRate();
          p.calculateBill(units);
      }
    }
```


Shape
<<Interface>>
+draw() : void
FactoryPattern Demo
+main() : void
implements
implements
implements
asks
Circle
+draw() : void
Square
+draw() : void
Rectangle
+draw() : void
creates
ShapeFactory
+getShape() : Shape

```java
package FactoryMethodDP2;
    //Step 1 Create an interface.
    public interface shape
        {
            void draw ();
        }
```

```java
package FactoryMethodDP2;

    public class ShapeFactory
    {
        public shape getShape (String shapeName)
        {
           String s = shapeName.toLowerCase();
            switch (s)
            {
                case "circle":
                    return  new Circle();
                case "square":
                    return  new Square();
               case "rectangle":
                    return  new Rectangle();
               default:
                  return null;
            }
        }
    }
```

```java
package FactoryMethodDP2;
    public class Rectangle implements shape
    {
    public void draw()
        {
            System.out.println("i can draw a rectangle");
        }
    }
```

```java
package FactoryMethodDP2;
    public class Circle implements shape
    {
    public void draw()
        {
            System.out.println("i can draw a circle");
        }
    }
```

```java
package FactoryMethodDP2;
    public class Square implements shape
    {
        public void draw()
        {
            System.out.println("i can draw a Square");
        }
    }
```

```java
package FactoryMethodDP2;

import java.util.Scanner;

public class FactoryPatternDemo
{
    public static void main(String[] args)
    {
        // allow the user to choose which object to create
        Scanner input = new Scanner(System.in);
        System.out.println("plz , enter your type");
        String type = input.nextLine() ;

        ShapeFactory shfac = new ShapeFactory() ;
        shape sh = shfac.getShape(type) ;
        sh.draw();
    }
}
```

Assignment1 😀

- an interface called Staff which contain GetSalary () and getYearsOfWork() methods
- there are two classes called TeachingAssistant and Doctor can implement this interface
- A class memeberFactory allow the user to create an object of his/ her desired member ,
- create a DemoClass that simulate the previous scenario and Draw the UML diagram for it .
- You can add any appropriate methods or data fields to easily show your work

Abstract Factory Pattern

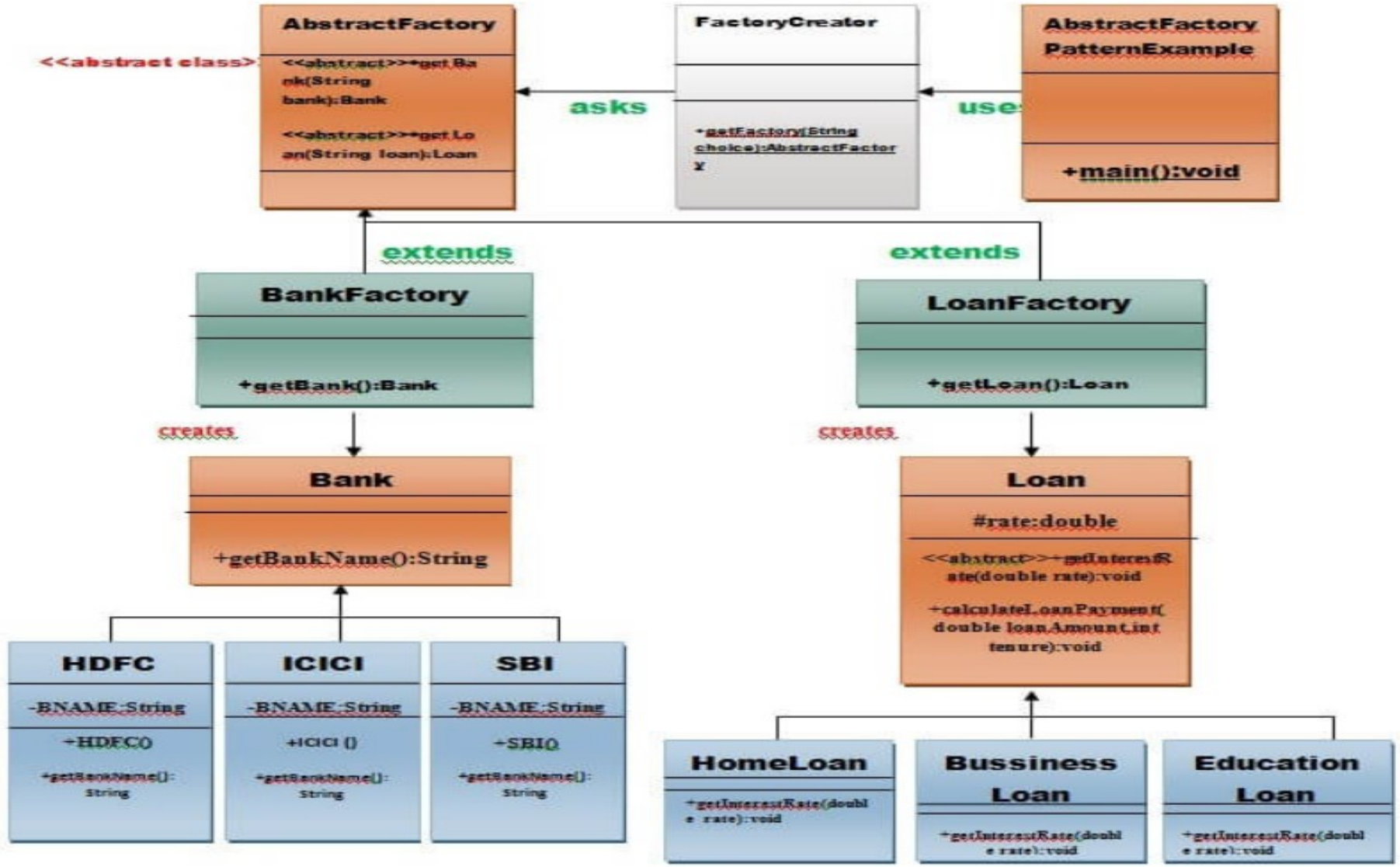
just **define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.**

That means Abstract Factory lets a class returns a factory of classes.

An Abstract Factory Pattern is also known as **Kit.**

It eases the exchanging of object families.

It promotes consistency among objects.


AbstractFactory
<<abstract class>>
<<abstract>>+getBank(String bank):Bank
<<abstract>>+getLoan(String loan):Loan
FactoryCreator
asks
+getFactory(String choice):AbstractFactory
AbstractFactory PatternExample
+main():void
extends
extends
BankFactory
+getBank():Bank
LoanFactory
+getLoan():Loan
creates
creates
Bank
+getBankName():String
Loan
#rate:double
<<abstract>>+getInterestRate(double rate):void
+calculateLoanPayment(double loanAmount,int tenure):void
HDFC
-BNAME:String
+HDFC()
+getBankName(): String
ICICI
-BNAME:String
+ICICI ()
+getBankName(): String
SBI
-BNAME:String
+SBI()
+getBankName(): String
HomeLoan
+getInterestRate(double rate):void
Bussiness Loan
+getInterestRate(double rate):void
Education Loan
+getInterestRate(double rate):void

UML for Abstract Factory Pattern

- We are going to create a **Bank interface** and a **Loan abstract class** as well as their sub-classes.
- Then we will create **AbstractFactory** class as next step.
- Then after we will create **concrete classes**, **BankFactory**, and **LoanFactory** that will extends **AbstractFactory** class
- After that, **AbstractFactoryPatternExample** class uses the **FactoryCreator** to get an object of **AbstractFactory** class.
- See the diagram carefully which is given in the next slide

```java
package AbstractFactoryPattern;

public interface Bank
    {
     String getBankName();
    }
```

```java
package AbstractFactoryPattern;
   public class ICICI implements Bank
   {
      private final String BankName ;

      public ICICI()
      {
         BankName = "ICICI Bank " ;
      }

      public String getBankName()
      {
         return  BankName ;
      }

   }
```

```java
package AbstractFactoryPattern;
   public class SBI  implements Bank
   {
      private final String BankName ;
      public SBI()
      {
         BankName = "SBI Bank " ;
      }
      public String getBankName()
      {
         return  BankName ;
      }
   }
```

```java
package AbstractFactoryPattern;
   public class HDFC implements Bank
   {
     private final String BankName ;
      public HDFC()
      {
         BankName = "HDFC Bank " ;
      }
      public String getBankName()
      {
         return  BankName ;
      }
   }
```

```
package AbstractFactoryPattern;
    public abstract class Loan
    {
        protected double rate;
        abstract void getInterestRate(double rate);
       public void calculateLoanPayment(double loanamount, int years)
        {
              double EMI;  int n;
              n=years*12;
              rate = rate/1200;
              double r = Math.pow((1+rate),n) ;
              EMI=((rate*r)/((r)-1))*loanamount;

              System.out.println("your monthly EMI is "+ EMI
                    +"\n for the amount"+loanamount+" you have borrowed");
     }
    }
```

```java
package AbstractFactoryPattern;
    public class EducationLoan extends Loan
    {
        public void getInterestRate(double r)
        {
         rate = r;
        }
    }
```

```java
package AbstractFactoryPattern;
    public class BussinessLoan extends Loan
    {
        public void getInterestRate(double r)
        {
         rate=r;
        }
    }
```

```java
package AbstractFactoryPattern;
    public class HomeLoan extends Loan
    {
        public void getInterestRate(double r)
        {
         rate=r;
        }
    }
```

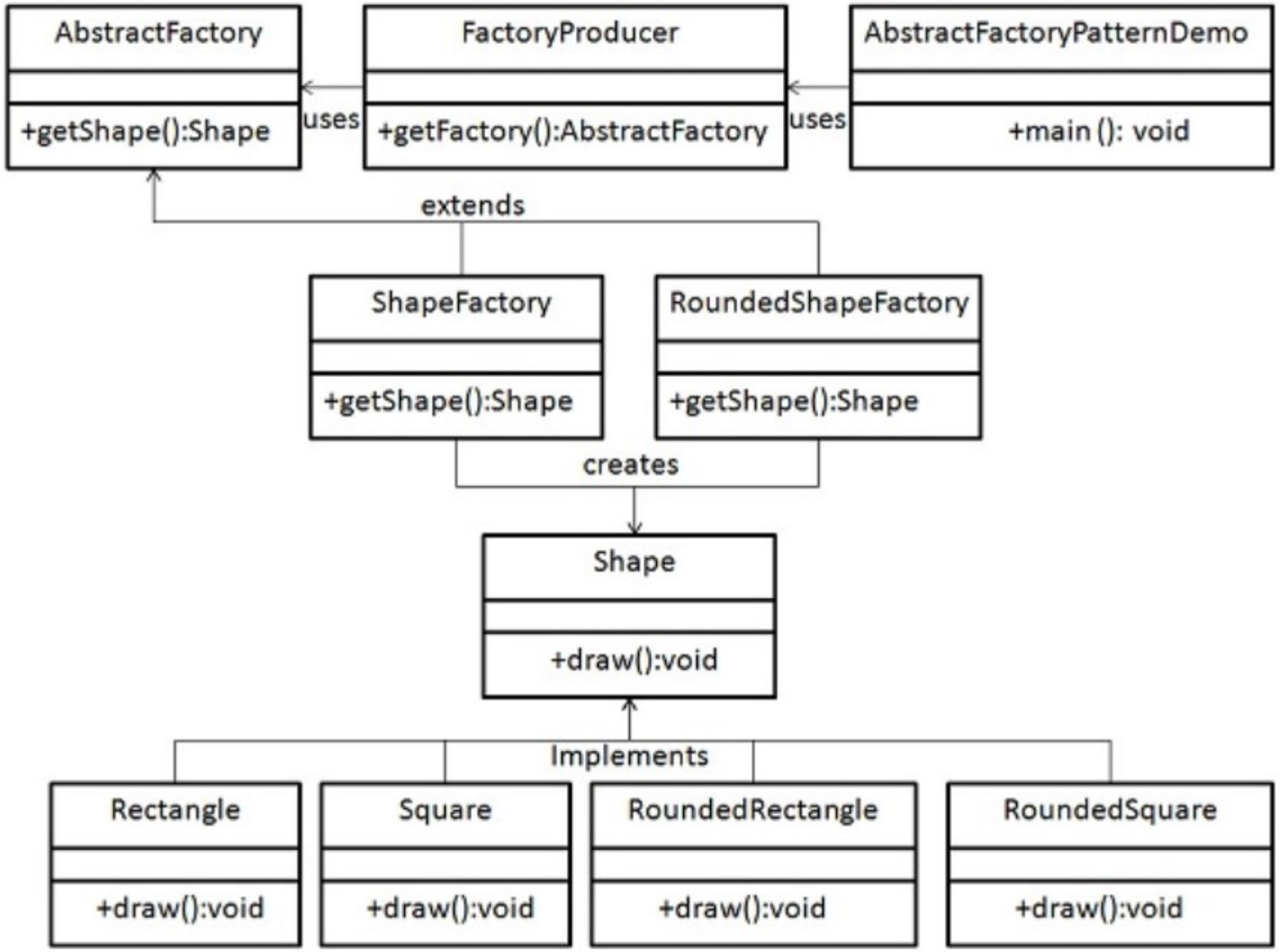
```java
package AbstractFactoryPattern;
    public abstract class AbstractFactory
    {
        //Create an abstract class (i.e AbstractFactory) to get
        //the factories for Bank and Loan Objects.
         public abstract Bank  getBank (String  Bank);
         public abstract Loan  getLoan (String  Loan);
    }
```

```
package AbstractFactoryPattern;
    public class BankFactory extends AbstractFactory
    {
        public Bank getBank(String bank)
        {
           if(bank == null)
           {
              return null;
           }
          if(bank.equalsIgnoreCase("HDFC"))
          {
            return new HDFC();
          }
          else if(bank.equalsIgnoreCase("ICICI"))
          {
            return new ICICI();
          }
          else if(bank.equalsIgnoreCase("SBI"))
          {
            return new SBI();
          }
          return null;
        }
       public Loan getLoan(String loan)
       {
        return null;
       }
  }
```

```
package AbstractFactoryPattern;
    public class LoanFactory extends AbstractFactory
    {
       public Loan getLoan(String loan)
          {
           if(loan == null)
               {
                  return null;
               }
           if(loan.equalsIgnoreCase("Home"))
               {
                  return new HomeLoan();
               }
           else if(loan.equalsIgnoreCase("Business"))
               {
                  return new BussinessLoan();
               }
           else if(loan.equalsIgnoreCase("Education"))
               {
                  return new EducationLoan();
               }
           return null;
       }
          public Bank getBank(String bank){
                        return null;
          }
    }
```

```
package AbstractFactoryPattern;
    public class FactoryCreator
    {
        public static AbstractFactory getFactory(String choice)
        {
            if(choice.equalsIgnoreCase("Bank"))
            {
                return new BankFactory();
            }
            else if(choice.equalsIgnoreCase("Loan"))
            {
                return new LoanFactory();
            }
            return null;
        }
    }
```

```
package AbstractFactoryPattern;
import java.util.Scanner;
    public class AbstractFactoryPatternExample
    {
        public static void main(String args[])
        {
            Scanner input = new Scanner (System.in);
            System.out.println("Enter the name of Bank from "
                               + "where you want to take loan amount: ");
            String bankName=input.nextLine();
            System.out.println("Enter the type of loan e.g. home loan "
                              + "or business loan or education loan : ");
            String loanName=input.nextLine();

            AbstractFactory bankFactory = FactoryCreator.getFactory("Bank");
            Bank b = bankFactory.getBank(bankName);

            System.out.println("Enter the interest rate for "+b.getBankName()+ ": ");
            double rate = input.nextDouble();
            System.out.println("Enter the loan amount you want to take: ");
            double loanAmount = input.nextDouble();
            System.out.println("Enter the number of years to pay your entire loan amount: ");
            int years = input.nextInt();
            System.out.println("you are taking the loan from "+ b.getBankName());

            AbstractFactory loanFactory = FactoryCreator.getFactory("Loan");
            Loan lo = loanFactory.getLoan(loanName);
            lo.getInterestRate(rate);
            lo.calculateLoanPayment(loanAmount,years);
        }
    }
```


AbstractFactory
+getShape():Shape
uses
FactoryProducer
+getFactory():AbstractFactory
uses
AbstractFactoryPatternDemo
+main (): void
extends
ShapeFactory
+getShape():Shape
RoundedShapeFactory
+getShape():Shape
creates
Shape
+draw():void
Implements
Rectangle
+draw():void
Square
+draw():void
RoundedRectangle
+draw():void
RoundedSquare
+draw():void

```java
package AbstractFactoryPattern2;
   //Create an interface for Shapes.
   public interface Shape
      {
       void draw ();

      }
```

```java
package AbstractFactoryPattern2;

   public class Square implements Shape
   {
      @Override
         public void draw()
         {
           System.out.println("draw method is called in Square");
         }
   }
```

```java
package AbstractFactoryPattern2;
    //Create concrete classes implementing the same interface.
    public class Rectangle implements Shape
    {
        @Override
         public void draw()
            {
              System.out.println("draw method is called in RoundedRectangle");
            }
    }
```

```java
package AbstractFactoryPattern2;
    //Create concrete classes implementing the same interface.
    public class RoundedSquare implements Shape
        {
            @Override
            public void draw()
            {
              System.out.println("draw method is called in RoundedSquare");
            }
        }
```

```java
package AbstractFactoryPattern2;

  //Create concrete classes implementing the same interface.
  public class RoundedRectangle implements Shape{

    @Override
    public void draw()
    {
      System.out.println("draw method is called in RoundedRectangle");
    }

}
```

```
package AbstractFactoryPattern2;

    //Step 4 : Create Factory classes extending AbstractFactory to
    //generate object of concrete class based on given information.

    public class ShapeFactory extends AbstractFactory
        {

        @Override
        Shape getShape(String shapeType)
            {
               if(shapeType.equalsIgnoreCase("RECTANGLE"))
               {
                 return new Rectangle();
               }
               else if(shapeType.equalsIgnoreCase("SQUARE"))
               {
                 return new Square();
               }
              return null;
            }
        }
```

```java
package AbstractFactoryPattern2;

    //Step 4 : Create Factory classes extending AbstractFactory to
    //generate object of concrete class based on given information.

    public class RoundedShapeFactory extends AbstractFactory
    {

    @Override
    Shape getShape(String shapeType)
        {
            if(shapeType.equalsIgnoreCase("RECTANGLE"))
            {
                return new RoundedRectangle();
            }
            else if(shapeType.equalsIgnoreCase("SQUARE"))
            {
                return new RoundedSquare();
            }
                return null;

        }

    }
```

```java
package AbstractFactoryPattern2;
    //Create an Abstract class to get factories for
    //Normal and Rounded Shape Objects.
    public abstract class AbstractFactory
        {
            abstract Shape getShape(String shapeType);
        }
```

```java
package AbstractFactoryPattern2;
    //Step 5 : Create a Factory generator/producer class to
    //get factories  by passing an information such as Shape
    public class FactoryProducer
     {
        public static AbstractFactory getFactory (boolean rounded)
        {
            if(rounded)
               {
                  return new RoundedShapeFactory();
               }
            else
               {
                   return new ShapeFactory();
               }
        }
     }
```

```java
package AbstractFactoryPattern2;
    public class AbstractFactoryPatternDemo
        {
            public static void main(String[] args)
            {
                //get shape factory
                AbstractFactory shapeFactory1 = FactoryProducer.getFactory(false);
                //get an object of Shape Rectangle
                Shape shape1 = shapeFactory1.getShape("RECTANGLE");
                //call draw method of Shape Rectangle
                shape1.draw();
                //get an object of Shape Square
                Shape shape2 = shapeFactory1.getShape("SQUARE");
                //call draw method of Shape Square
                shape2.draw();

                //get shape factory
                AbstractFactory shapeFactory2 = FactoryProducer.getFactory(true);
                //get an object of Shape Rectangle
                Shape shape3 = shapeFactory2.getShape("RECTANGLE");
                //call draw method of Shape Rectangle
                shape3.draw();
                //get an object of Shape Square
                Shape shape4 = shapeFactory2.getShape("SQUARE");
                //call draw method of Shape Square
                shape4.draw();
            }
        }
```

EXCEPECTED OUTPUT

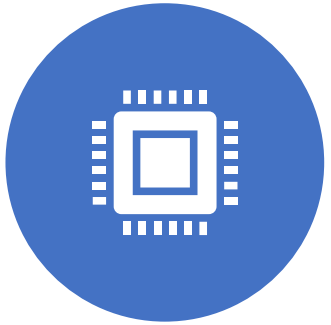
```
run:
draw method is called in RoundedRectangle
draw method is called in Square
draw method is called in RoundedRectangle
draw method is called in RoundedSquare
```

Singleton Pattern

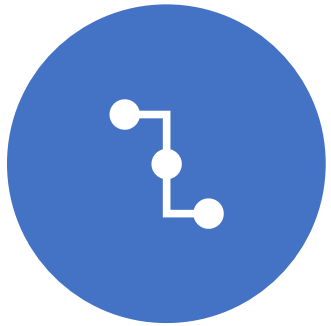
- Singleton pattern is one of the simplest design patterns in Java. this pattern provides one of the best ways to create an object.
- This pattern involves a single class which is responsible to create an object while making sure that only single object gets created.
- This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Implementation

We're going to create a *SingleObject* class.

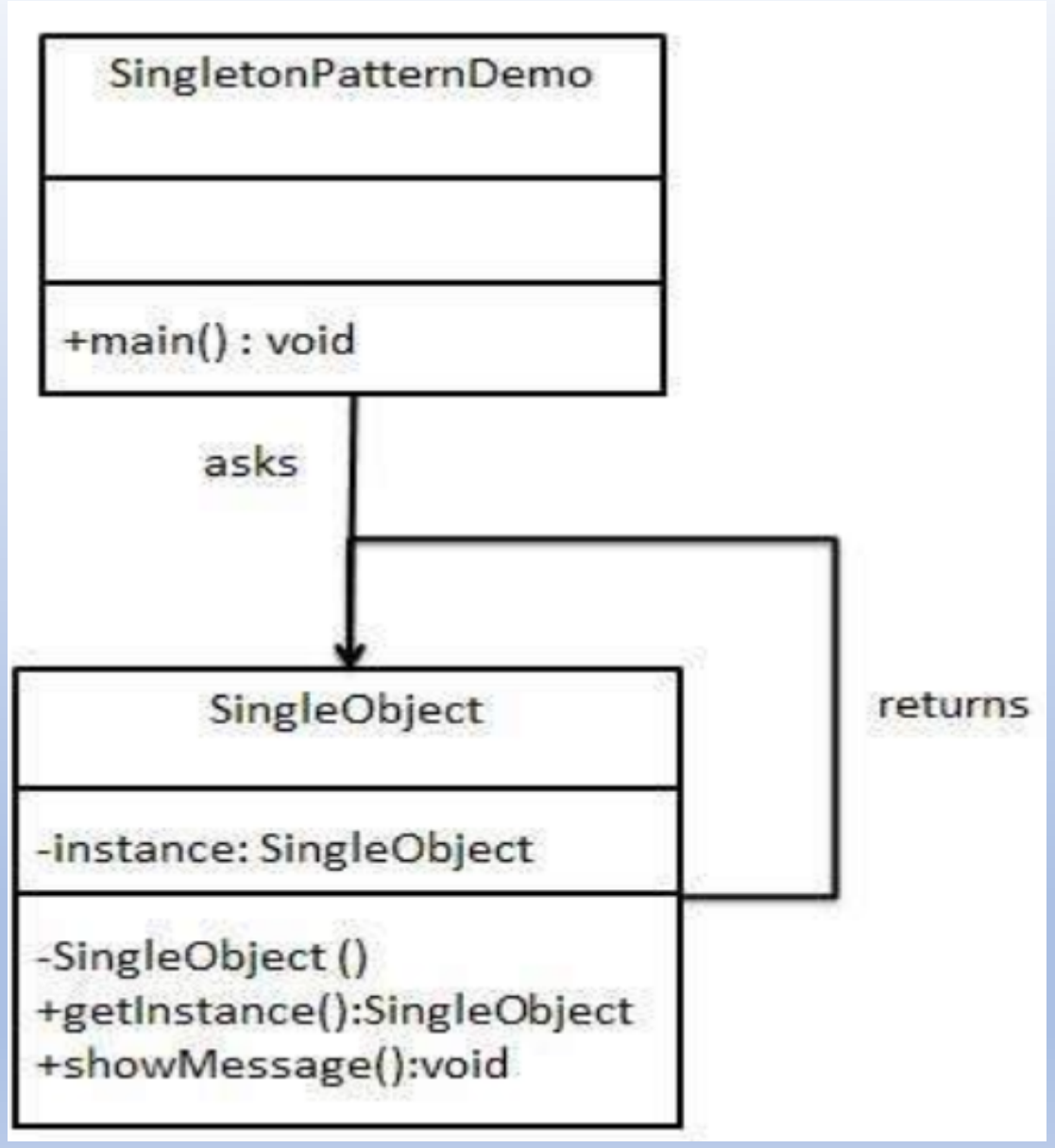


SingleObject class have its constructor as private and have a static instance of itself.



SingleObject class provides a static method to get its static instance to outside world.

SingletonPatternDemo, our demo class will use *SingleObject* class to get a *SingleObject* object.



Implementation Steps

Step 1

Create a Singleton Class.

Step 2

Get the only object from the singleton class.

Step 3

Verify the output.

```
package SingletonDP;
    public class SingleObject
        {
            //create an object of SingleObject

            private static SingleObject instance = new SingleObject();

            //make the constructor private so that this class
            // cannot be instantiated

            private SingleObject(){}

            //Get the only object available
            public static SingleObject getInstance()
            {
               return instance;
            }

            public void showMessage()
            {
               System.out.println("Hello World!");
            }
```

```
package SingletonDP;
    public class SingletonPatternDemo
    {
        public static void main(String[] args)
        {

      //SingleObject object = new SingleObject();
      //illegal construct : The constructor SingleObject() is not visible

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();

      }
    }
```

```
run:
Hello World!
```

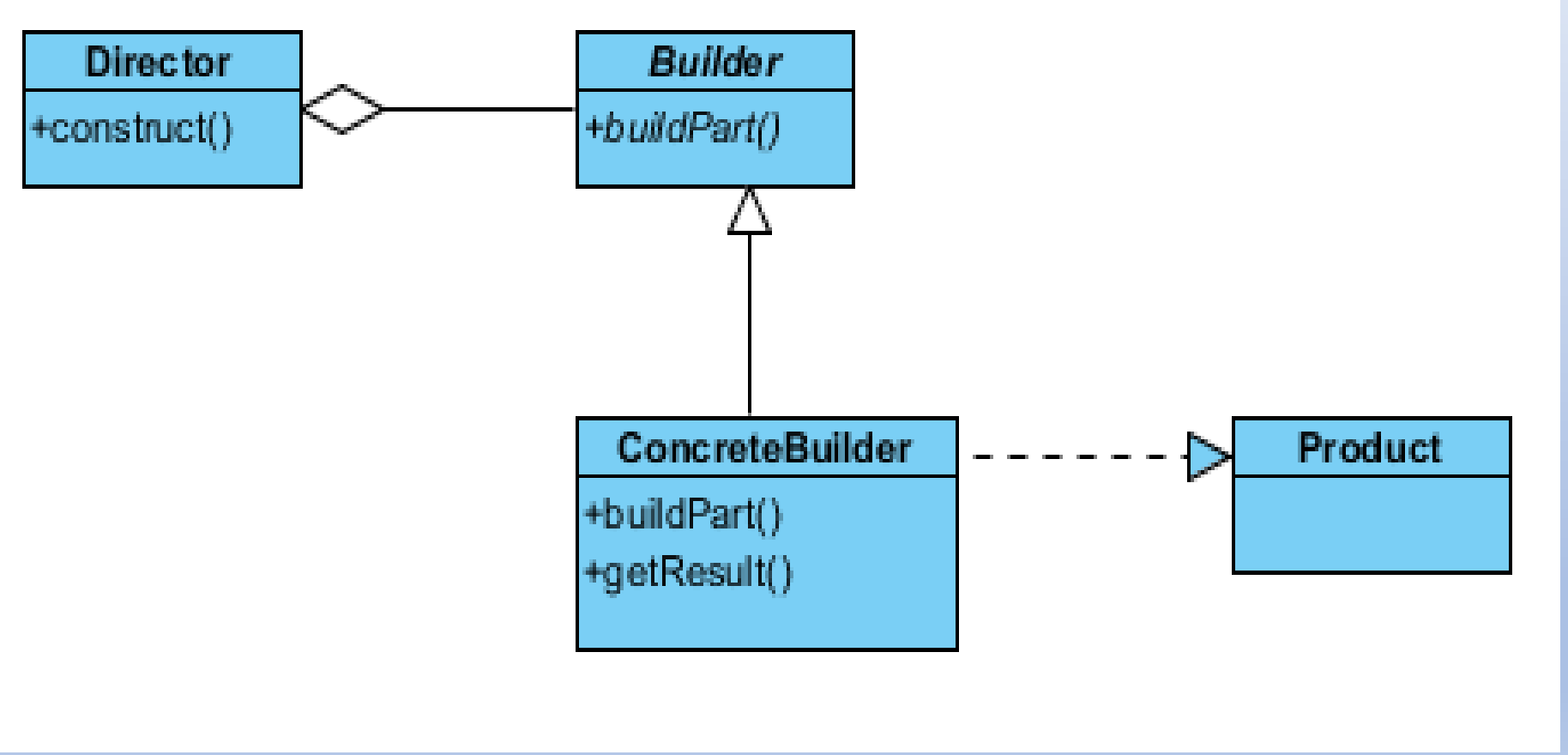
Builder Design Pattern

- Builder Pattern says that "construct a complex object from simple objects using step-by-step approach"

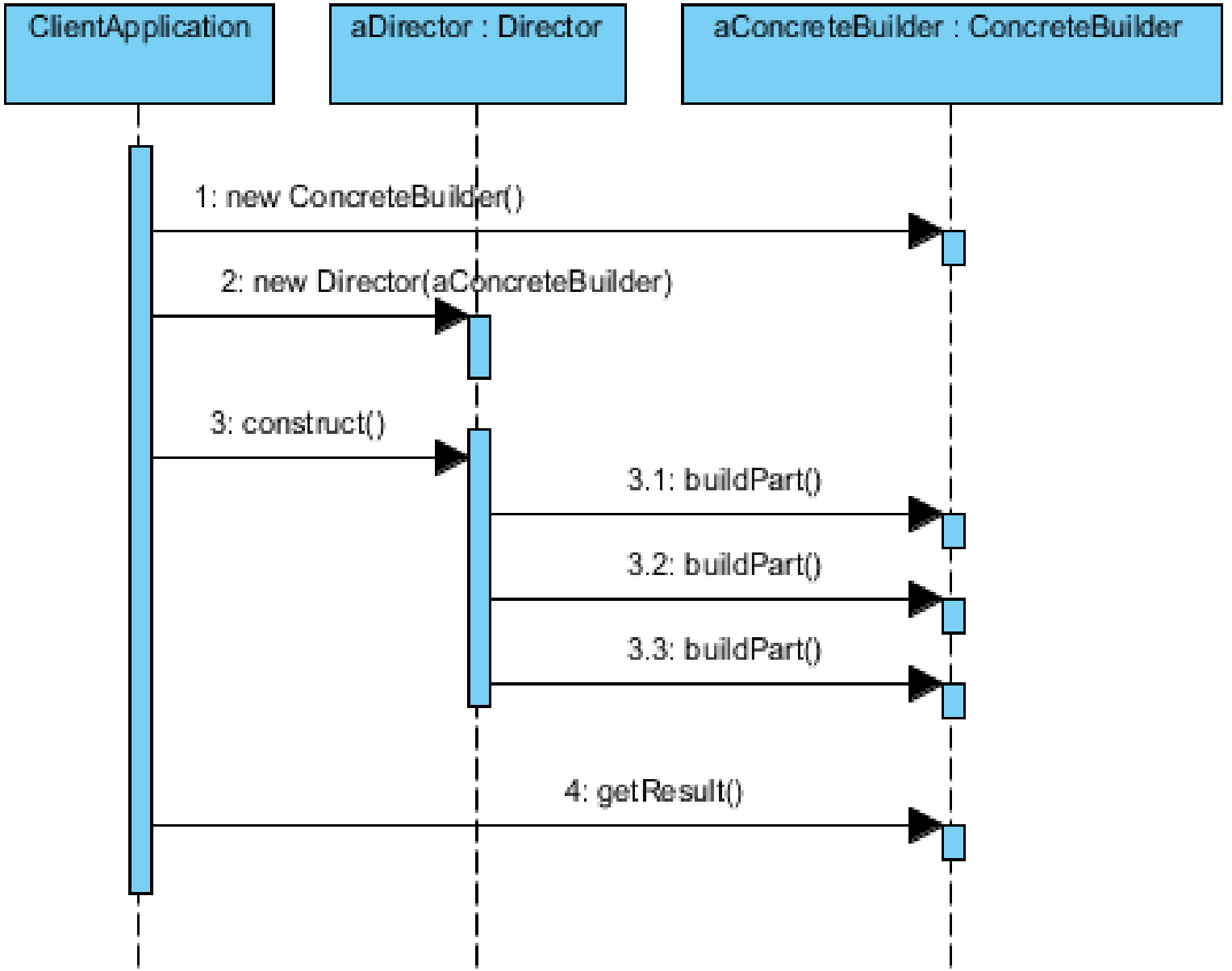
The main advantages of Builder Pattern are as follows:

- It provides clear separation between the construction and representation of an object.
- It provides better control over construction process.
- It supports to change the internal representation of objects

UML Structure



Collaborations



Example

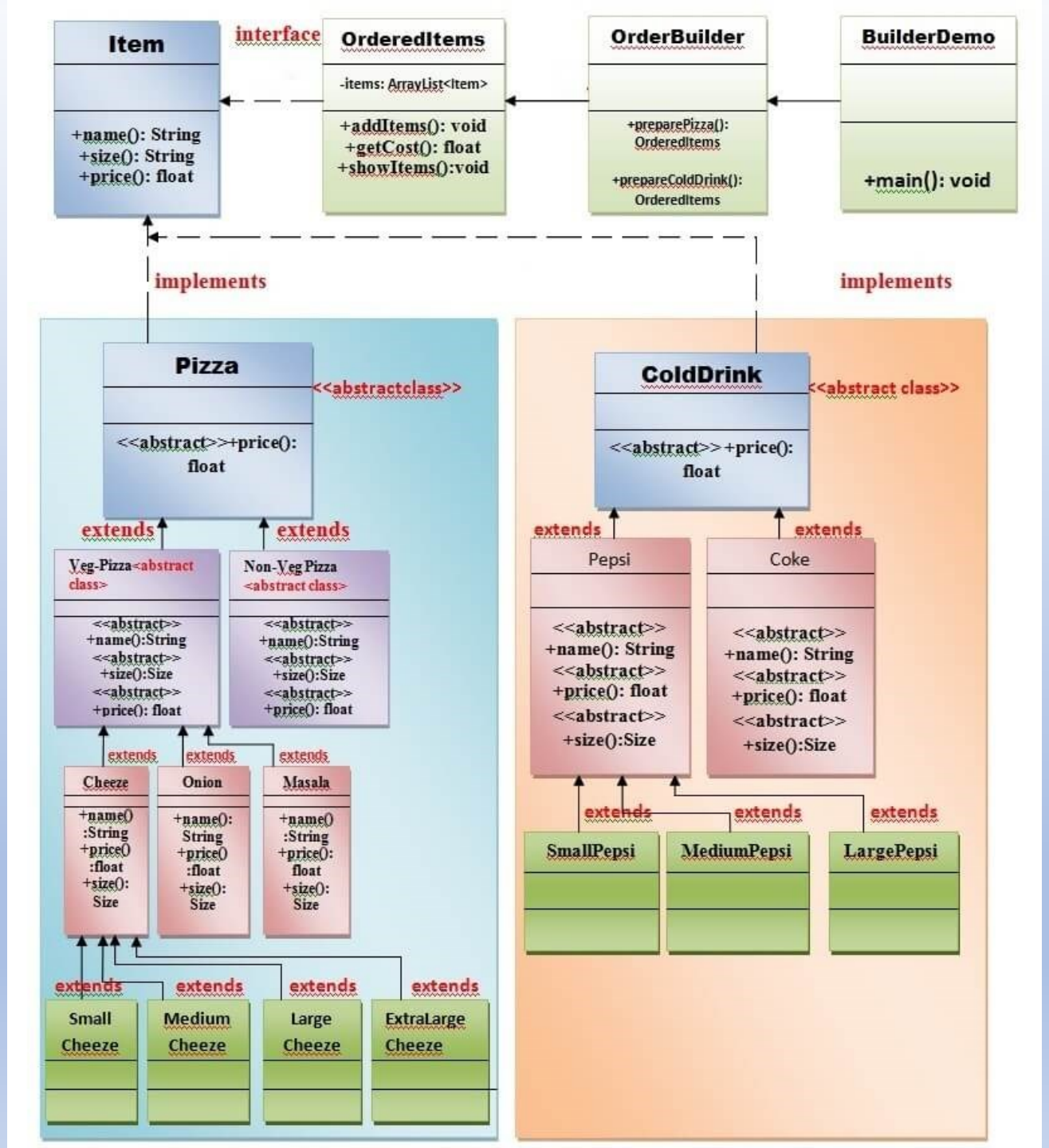
Creation of Marketplace account “Amazon, eBay”
Account attributes:
First Name
Middle Name
Last Name
Title
DOB
CC
Address
Access code
E-mail address
Alternate E-mail address
Phone number
Weekend delivery

UML for Builder Pattern Example

We are considering a business case of **pizza-hut** where we can get different varieties of pizza and cold-drink.

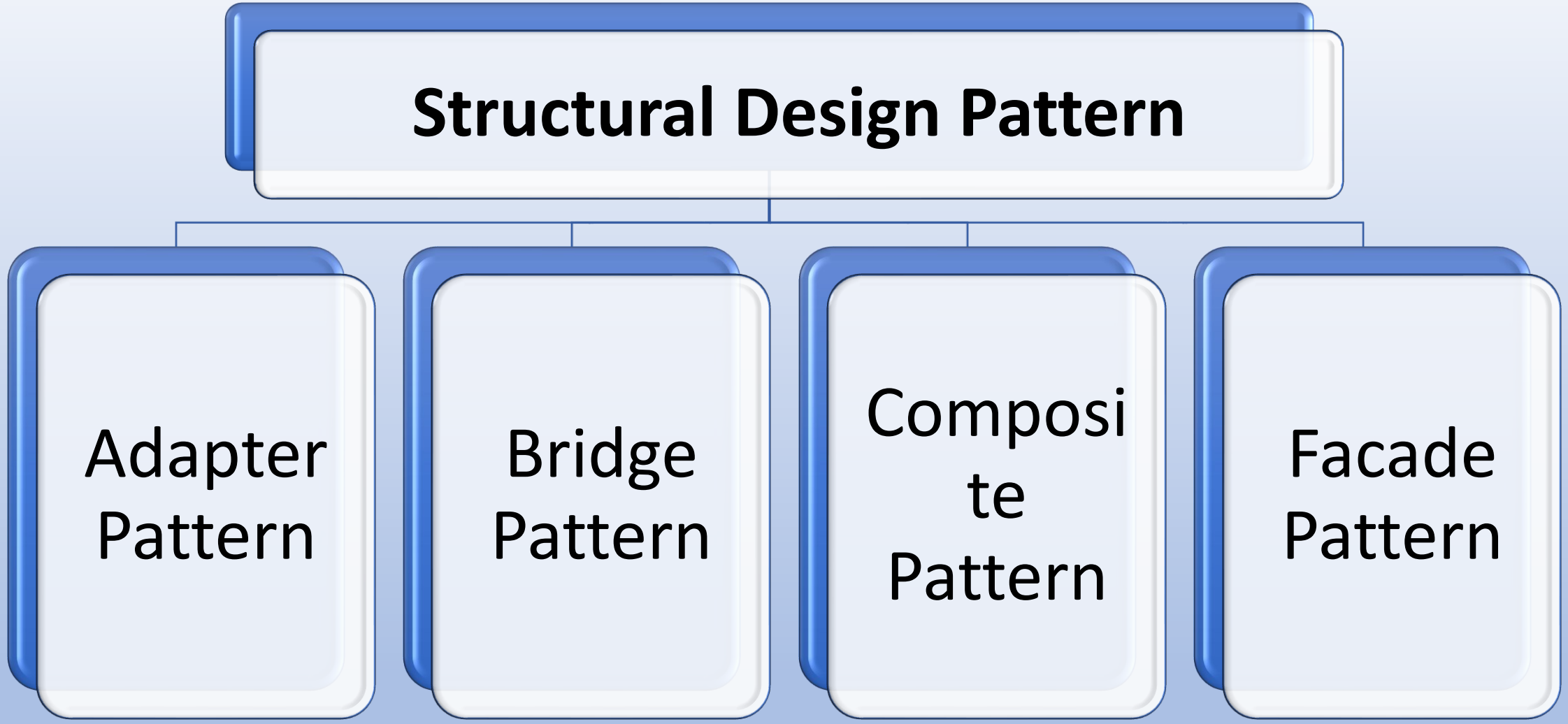
Pizza can be either a Veg pizza or Non-Veg pizza of several types (like cheese pizza, onion pizza, masala-pizza etc) and will be of 4 sizes i.e. small, medium, large, extra-large.

Cold-drink can be of several types (like Pepsi, Coke, Dew, Sprite, Fanta, Maaza, Limca, Thums-up etc.) and will be of 3 sizes small, medium, large.


Item
+name(): String
+size(): String
+price(): float
interface
OrderedItems
-items: ArrayList<Item>
+addItems(): void
+getCost(): float
+showItems():void
OrderBuilder
+preparePizza(): OrderedItems
+prepareColdDrink(): OrderedItems
BuilderDemo
+main(): void
implements
implements
Pizza
<<abstractclass>>
<<abstract>>+price(): float
extends
extends
Veg-Pizza<abstract class>
<<abstract>>
+name():String
<<abstract>>
+size():Size
<<abstract>>
+price(): float
Non-Veg Pizza
<abstract class>
<<abstract>>
+name():String
<<abstract>>
+size():Size
<<abstract>>
+price(): float
extends
extends
extends
Cheeze
+name() :String
+price() :float
+size(): Size
Onion
+name(): String
+price() :float
+size(): Size
Masala
+name() :String
+price(): float
+size(): Size
extends
extends
extends
extends
Small Cheeze
Medium Cheeze
Large Cheeze
ExtraLarge Cheeze
ColdDrink
<<abstract class>>
<<abstract>> +price(): float
extends
extends
Pepsi
<<abstract>>
+name(): String
<<abstract>>
+price(): float
<<abstract>>
+size():Size
Coke
<<abstract>>
+name(): String
<<abstract>>
+price(): float
<<abstract>>
+size():Size
extends
extends
extends
SmallPepsi
MediumPepsi
LargePepsi

- You can find code in

- https://www.javatpoint.com/builder-design-pattern

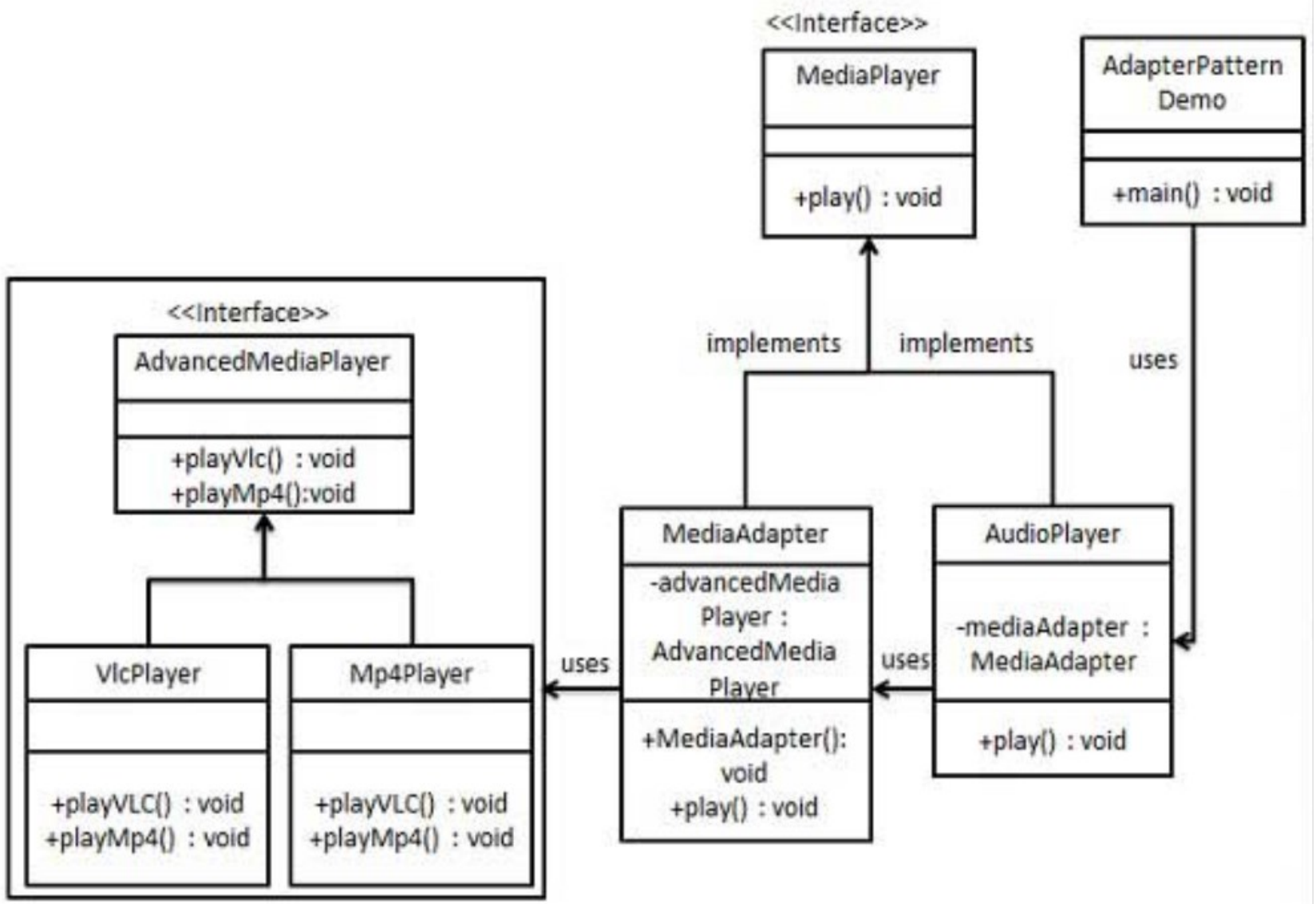

Structural Design Pattern
Adapter Pattern
Bridge Pattern
Composite Pattern
Facade Pattern

Adapter Pattern

Adapter Pattern

- **Adapter pattern Allow two incomputable interface to work together**
- **Adapter pattern works as a bridge between two incompatible interfaces.**
- **Adapter pattern combines the capability of two independent interfaces.**
- **For example, card reader which acts as an adapter between memory card and a laptop.**

Audio player to advanced Audio player



Implementation

We have a ***MediaPlayer*** interface and a concrete class ***AudioPlayer*** implementing the *MediaPlayer* interface.

AudioPlayer can play mp3 format audio files by default.

We are having another interface ***AdvancedMediaPlayer*** and

concrete classes implementing the ***AdvancedMediaPlayer*** interface. These classes can play vlc and mp4 format files.

We want to make ***AudioPlayer*** to play other formats as well.

Implementation

To attain this, we have created an adapter class *MediaAdapter* which implements the *MediaPlayer* interface and uses *AdvancedMediaPlayer* objects to play the required format.

AudioPlayer uses the adapter class *MediaAdapter* passing it the desired audio type without knowing the actual class which can play the desired format.

AdapterPatternDemo, our demo class will use *AudioPlayer* class to play various formats.

```java
package AdapterDP;

    //Step 1 : Create interfaces for Advanced Media Player.
   public interface AdvancedMediaPlayer
      {
           public void playVlc(String fileName);
           public void playMp4(String fileName);
      }
```

```java
package AdapterDP;

    //Step 1 : Create interfaces for Media Player.
  public interface MediaPlayer
      {
          public void play(String audioType, String fileName);
      }
```

```java
package AdapterDP;
//Step 2: Create concrete classes implementing the AdvancedMediaPlayer interface.
    public class VlcPlayer implements AdvancedMediaPlayer
    {

        public void playVlc(String fileName)
            {
                System.out.println("Playing vlc file with Name:"+ fileName);
            }

        public void playMp4(String fileName)
            {
                // DO NOTHING

            }
    }
```

```java
package AdapterDP;
//Step 2: Create concrete classes implementing the AdvancedMediaPlayer interface.
    public class Mp4Player implements AdvancedMediaPlayer
    {

        public void playVlc(String fileName)
            {
                // DO NOTHING
            }

        public void playMp4(String fileName)
            {
                System.out.println("Playing MP4 file with Name:"+ fileName);
            }
    }
```

```java
package AdapterDP;
//Create concrete class implementing the MediaPlayer interface.

public class AudioPlayer implements MediaPlayer
    {
       MediaAdapter  mediaAdapter;

      @Override
      public void play(String audioType, String fileName)
          {
            //inbuilt support to play mp3 music files
            if(audioType.equalsIgnoreCase("mp3"))
                {
                  System.out.println("Playing mp3 file. Name: " + fileName);
                }
            //mediaAdapter is providing support to play other file formats
            else if(audioType.equalsIgnoreCase("vlc")
                    || audioType.equalsIgnoreCase("mp4"))
                {
                  mediaAdapter = new MediaAdapter(audioType);
                  mediaAdapter.play(audioType, fileName);
                }
            else
                {
                  System.out.println("Invalid media. " + audioType + " format not supported");
                }
      }

    }
```

```java
package AdapterDP;
//Step 3 :Create adapter class implementing the MediaPlayer interface.
    public class MediaAdapter implements MediaPlayer
    {
        AdvancedMediaPlayer  advancedMusicPlayer ;

        public MediaAdapter(String audioType)
       {
          if(audioType.equalsIgnoreCase("vlc") )
              {
                  advancedMusicPlayer = new VlcPlayer() ;

              }
          else if (audioType.equalsIgnoreCase("mp4"))
              {
                  advancedMusicPlayer = new Mp4Player() ;
              }
       }

       public void play(String audioType, String fileName)
       {

          if(audioType.equalsIgnoreCase("vlc"))
              {
                  advancedMusicPlayer.playVlc(fileName) ;
              }
          else if(audioType.equalsIgnoreCase("mp4"))
              {
                  advancedMusicPlayer.playMp4(fileName) ;
              }
       }
```

```java
package AdapterDP;
    public class AdapterDemo
    {
        public static void main(String[] args)
        {
            AudioPlayer audioPlayer = new AudioPlayer();

            audioPlayer.play("mp3", "beyond the horizon.mp3");
            audioPlayer.play("mp4", "alone.mp4");
            audioPlayer.play("vlc", "far far away.vlc");
            audioPlayer.play("avi", "mind me.avi");
        }

    }
```

```
run:
Playing mp3 file. Name: beyond the horizon.mp3
Playing MP4 file with Name:alone.mp4
Playing vlc file with Name:far far away.vlc
Invalid media. avi format not supported
```

Multipliers and Dividers

The purpose of this assignment is to experiment with the Adapter pattern.

1. Build an interface called MultiplyInterface with a BigDecimal multiply(BigDecimal) method that returns a BigDecimal object containing the product.
2. Build an interface called DivideInterface with a BigDecimal divide(BigDecimal) method that returns a BigDecimal object containing the quotient.
3. Build a class called Multiplier that implements MultiplyInterface.
4. Build a class called Divider that implements DivideInterface.
5. We sometimes need Divider objects to pass as Multiplier objects, so build an adapter class call DividerAdapter for that purpose.
6. Build a main program that demonstrates that DividerAdapter objects can function like Multiplier objects.

Bridge Pattern

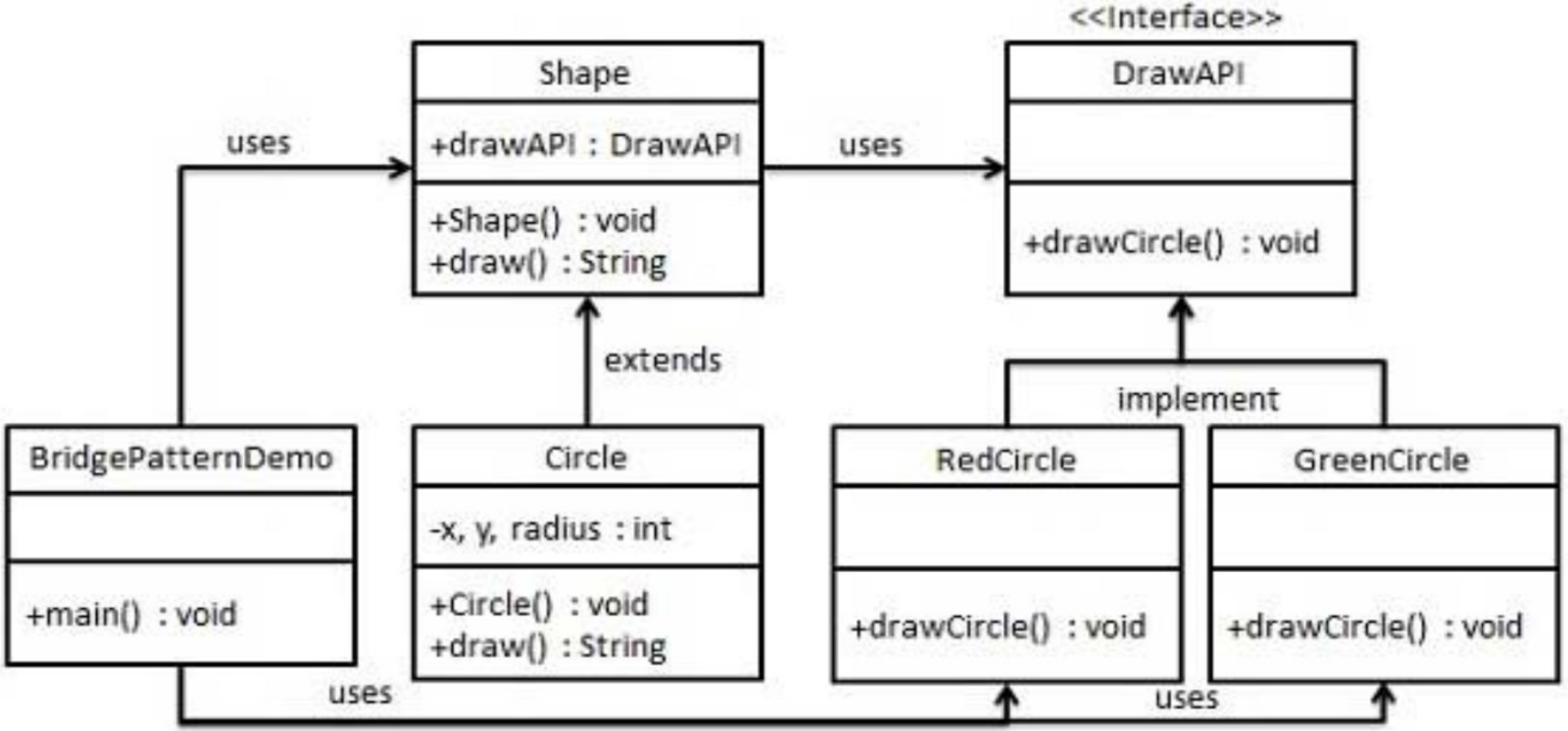
Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently.

This pattern decouples implementation class and abstract class by providing a bridge structure between them.

This pattern involves an **interface** which acts as a **bridge** which makes the functionality of **concrete classes** independent from **interface implementer classes**.

Both types of classes can be **altered** structurally without **affecting** each other.

a **circle** can be drawn in **different colors** using **same abstract** class method but **different bridge implementer** classes.



Implementation

We have a *DrawAPI* interface which is acting as a bridge implementer and concrete classes *RedCircle*, *GreenCircle* implementing it.

Shape is an abstract class and will use object of *DrawAPI*.

The BridgePatternDemo will use *Shape* class to draw different colored circle.

```java
package BridgePattern;
//Step 1: Create bridge implementer interface.

   public interface DrawAPI
      {
         public void drawCircle(int radius);
      }
```

```java
package BridgePattern;

    //Step 2:Create concrete bridge implementer classes
    //implementing the DrawAPI interface.

    public class GreenCircle implements DrawAPI
    {
        public void drawCircle(int radius)
        {
            System.out.println("Drawing a Green Circle with radius: " + radius);
        }
    }
```

```java
package BridgePattern;

    //Step 2:Create concrete bridge implementer classes
    //implementing the DrawAPI interface.

public class RedCircle implements DrawAPI
    {
        public void drawCircle(int radius)
        {
            System.out.println("Drawing a Red Circle with radius: " + radius);
        }
    }
```

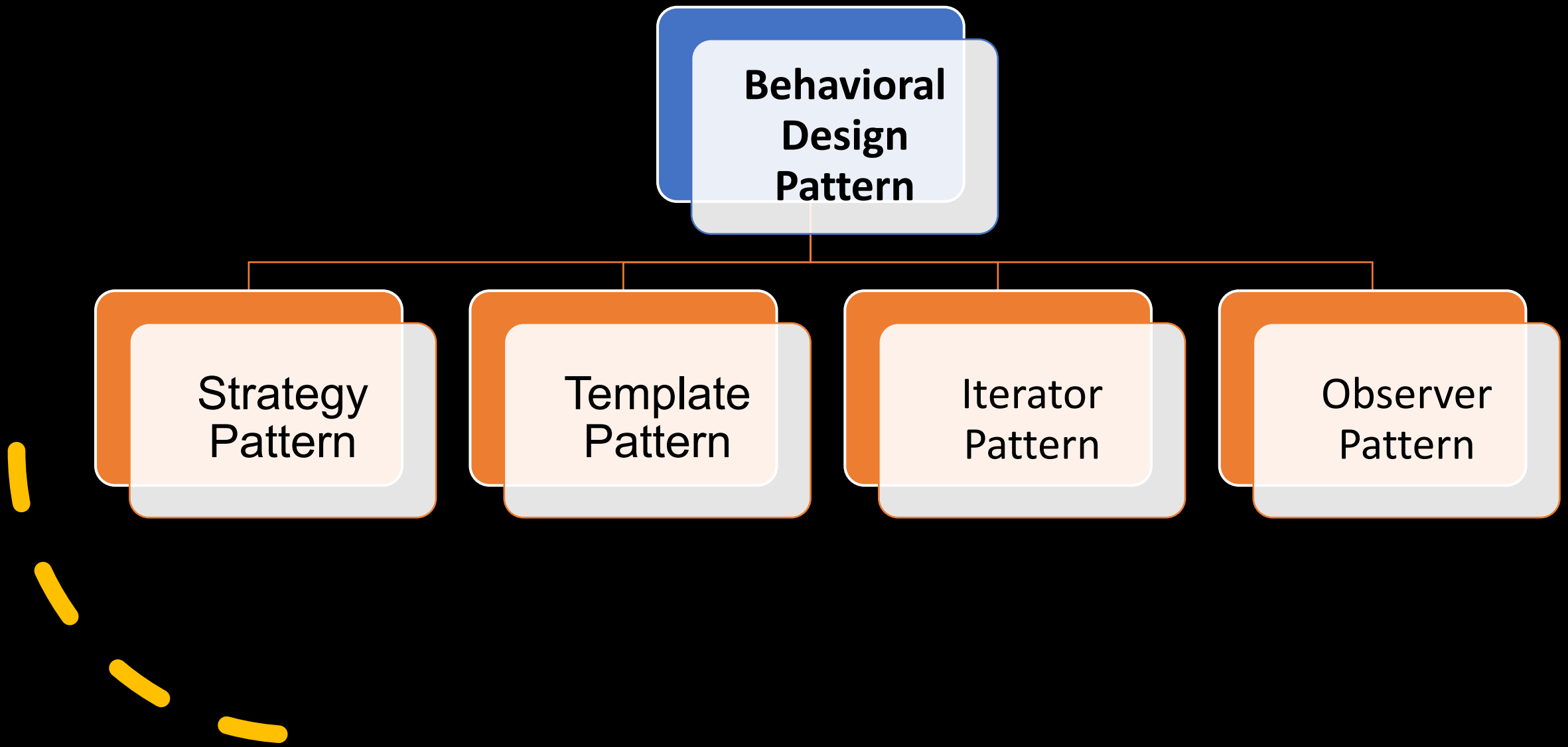
```java
package BridgePattern;

//Step 3: Create an abstract class Shape using the DrawAPI interface.

    public abstract class Shape
       {
          protected DrawAPI drawAPI;

          protected Shape(DrawAPI drawAPI)
              {
                 this.drawAPI = drawAPI;
              }
          public abstract void draw();
       }
```

```java
package BridgePattern;

//Step 4 : Create concrete class implementing the Shape interface.

    public class Circle extends Shape
        {
            private int radius;

          public Circle( int radius, DrawAPI drawAPI )
          {
              super(drawAPI);
              this.radius = radius;
          }

          public void draw()
           {
              drawAPI.drawCircle(radius);
           }
        }
```

```java
package BridgePattern;

//Step 5:Use the Shape and DrawAPI classes to draw different colored circles.

   public class BridgePatternDemo
       {
          public static void main(String[] args)
          {
               Shape redCircle    = new Circle( 10, new RedCircle());
               Shape greenCircle = new Circle( 20, new GreenCircle());

               redCircle.draw();
               greenCircle.draw();
          }
       }
```

```
run:
Drawing a Red Circle with radius: 10
Drawing a Green Circle with radius: 20
```

Behavioral Design Pattern

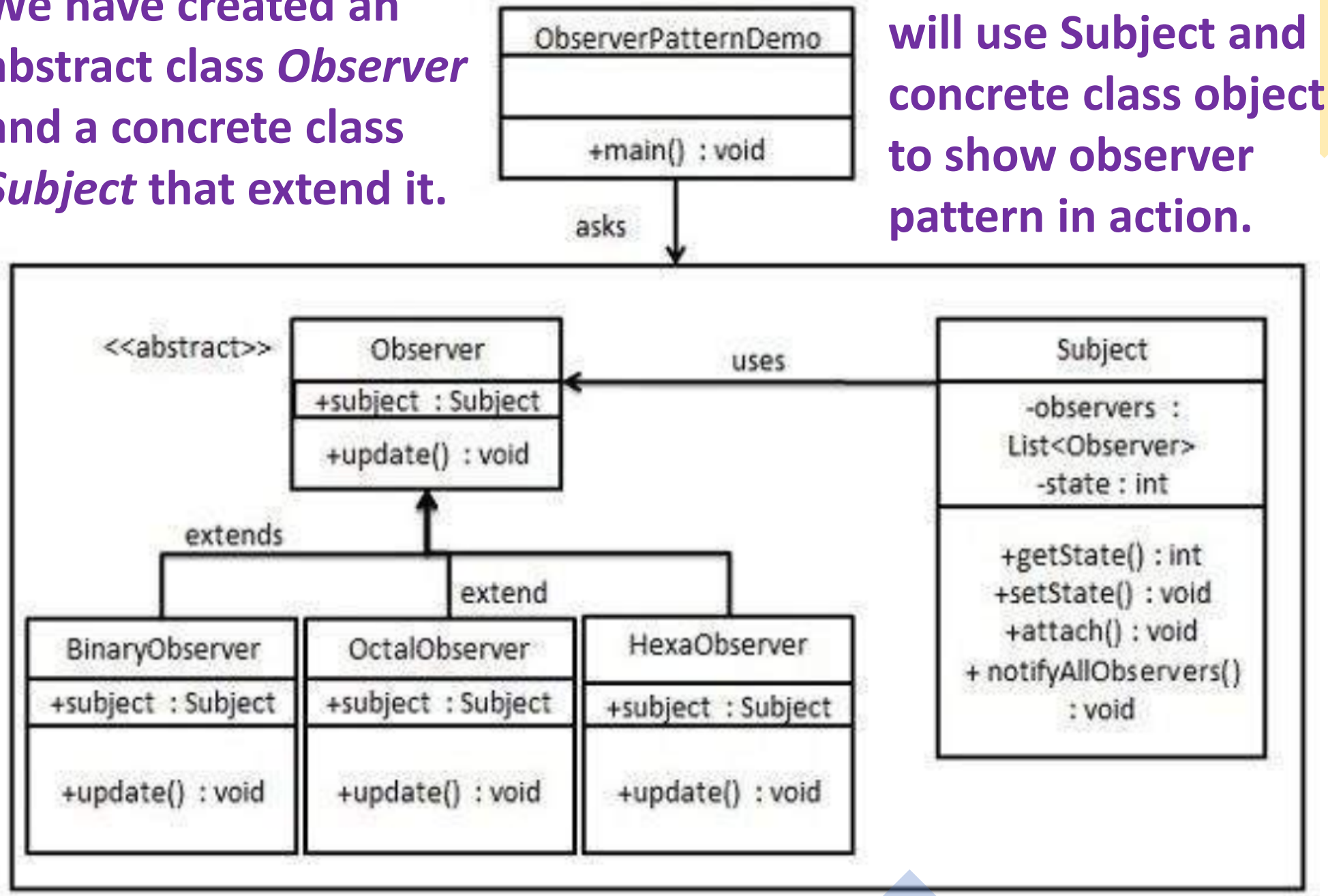

Behavioral Design Pattern
Strategy Pattern
Template Pattern
Iterator Pattern
Observer Pattern

Observer Pattern

- Observer pattern is used when there is **one-to-many** relationship among objects
- For example, if one object is modified, its dependent objects are to be notified automatically.
- Observer pattern uses three actor classes.
- **Subject**, **Observer** and **Client**.
- **Subject** is an object having methods to **attach** and **detach observers** to a **client** object.

Implementation

We have created an abstract class *Observer* and a concrete class *Subject* that extend it.



will use Subject and concrete class object to show observer pattern in action.

```
package ObserverDesignPattern;

//Step 1 : Create Observer abstract class.

public abstract class Observer
    {
        protected Subject subject;
        public abstract void update();

    }
```

```java
package ObserverDesignPattern;
    import java.util.ArrayList;
    import java.util.List;
    //Step 2 : Create Subject class.
public class Subject
    {
        private List<Observer> observers = new ArrayList<Observer>();
        private int state;

        public int getState(){return state;}

        public void setState(int state) {
           this.state = state;
           notifyAllObservers();
        }

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

        public void notifyAllObservers()
        {
            for (Observer observer : observers)
            {
               observer.update();
            }
        }
```

```java
package ObserverDesignPattern;
    //Step 3 : Create concrete observer classes
    public class BinaryObserver extends Observer
    {
        public BinaryObserver(Subject subject)
        {
          this.subject = subject;
          this.subject.attach(this);  //passing reference
        }

       @Override
       public void update()
       {
          System.out.println( "Binary String: " +
                                Integer.toBinaryString( subject.getState()));
       }
    }
```

```java
package ObserverDesignPattern;

    //Step 3 : Create concrete observer classes
    public class OctalObserver  extends Observer
    {
        public OctalObserver (Subject subject)
        {
          this.subject = subject;
          this.subject.attach(this);
        }

       @Override
       public void update()
       {
          System.out.println( "Octal String: " +
                                Integer.toOctalString(subject.getState() ) );
       }
    }
```

```java
package ObserverDesignPattern;

    //Step 3 : Create concrete observer classes
    public class HexaObserver  extends Observer
        {
            public HexaObserver (Subject subject)
            {
              this.subject = subject;
              this.subject.attach(this);
            }

           @Override
           public void update()
           {
             System.out.println( "HexaDecimal String: " +
                                  Integer.toHexString(subject.getState() ) );
           }
        }
```

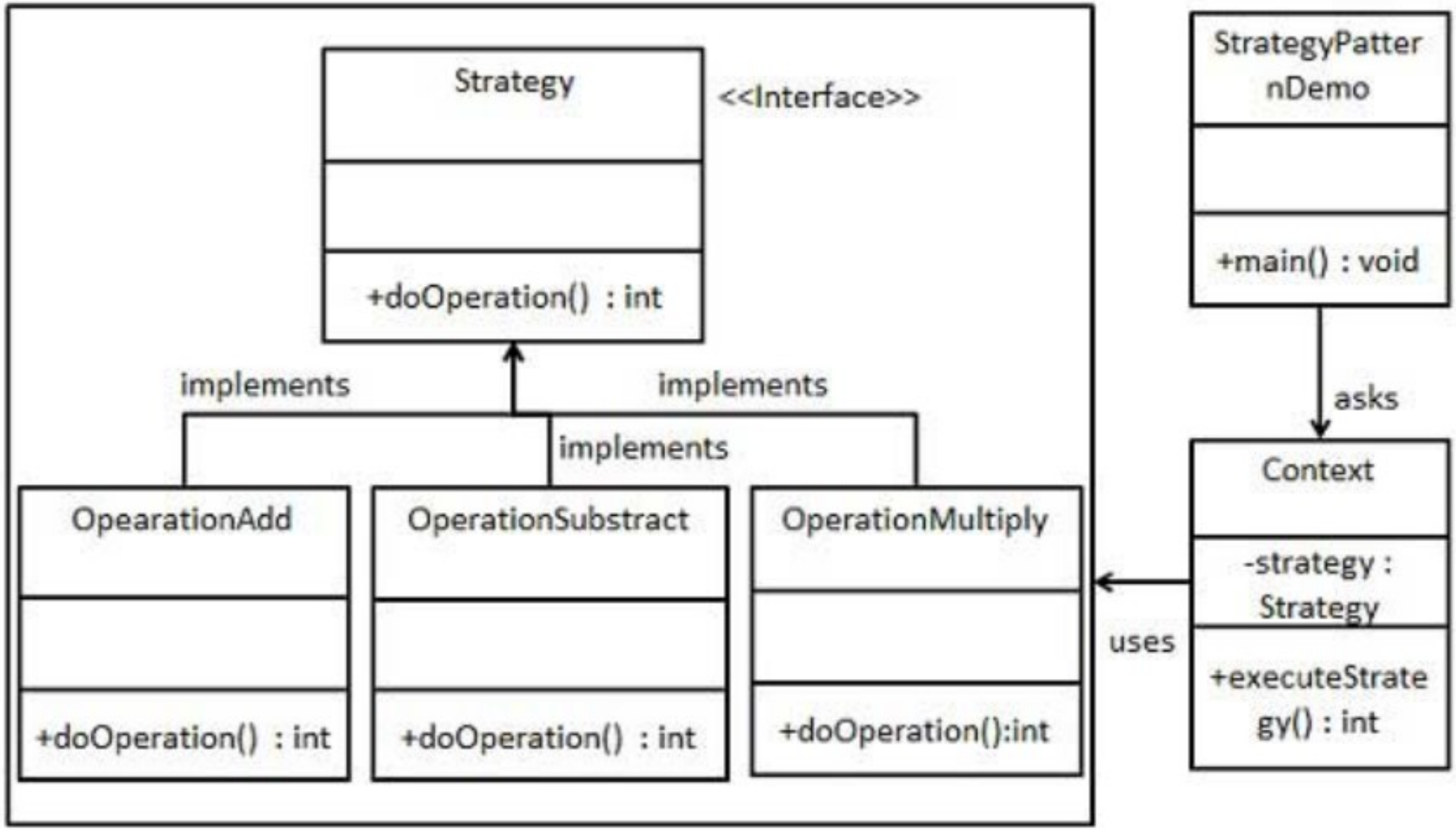
```
package ObserverDesignPattern;

    //Use Subject and concrete observer objects.
   public class ObserverPatternDemo
   {
      public static void main(String[] args)
      {
      Subject subject = new Subject();

      new HexaObserver(subject);
      new OctalObserver(subject);
      new BinaryObserver(subject);

      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("\nSecond state change: 10");
      subject.setState(10);
      }
   }
```

Strategy Pattern

In Strategy pattern, a class behavior or its algorithm can be changed at run time.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object.

The strategy object changes the executing algorithm of the context object.


Strategy
<<Interface>>
+doOperation() : int
implements
implements
implements
OpearationAdd
+doOperation() : int
OperationSubstract
+doOperation() : int
OperationMultiply
+doOperation():int
StrategyPatter nDemo
+main() : void
asks
Context
-strategy : Strategy
+executeStrate gy() : int
uses

Implementation

We are going to create a ***Strategy*** **interface** defining an action and

Concrete strategy classes implementing the *Strategy* interface.

Context is a class which uses a Strategy.

StrategyPatternDemo, will use ***Context*** and **strategy** objects to demonstrate **change** in Context behavior based on strategy it uses.

```java
package StrategyDesignPattern;

    //Step 1: Create a Strategy interface
   public interface Strategy
      {
          public int doOperation(int num1, int num2);
      }
```

```java
package StrategyDesignPattern;

    //Step 2 :Create concrete classes implementing the same interface.
   public class OperationAdd implements Strategy
      {
          @Override
          public int doOperation(int num1, int num2)
          {
              return num1 + num2;
          }
      }
```

```java
package StrategyDesignPattern;

 //Step 2 :Create concrete classes implementing the same interface.
public class OperationMultiply implements Strategy
    {
        @Override
        public int doOperation(int num1, int num2)
        {
            return num1 * num2;
        }
    }
```

```java
package StrategyDesignPattern;

 //Step 2 :Create concrete classes implementing the same interface.
public class OperationSubstract implements Strategy
    {
        @Override
        public int doOperation(int num1, int num2)
        {
            return num1 - num2;
        }
    }
```

```java
package StrategyDesignPattern;

//Step 3 : Create Context Class.
    public class Context
        {
            private Strategy strategy;

            public Context(Strategy strategy)
            {
                this.strategy = strategy;
            }

            public int executeStrategy(int num1, int num2)
            {
                return strategy.doOperation(num1, num2);
            }
        }
```

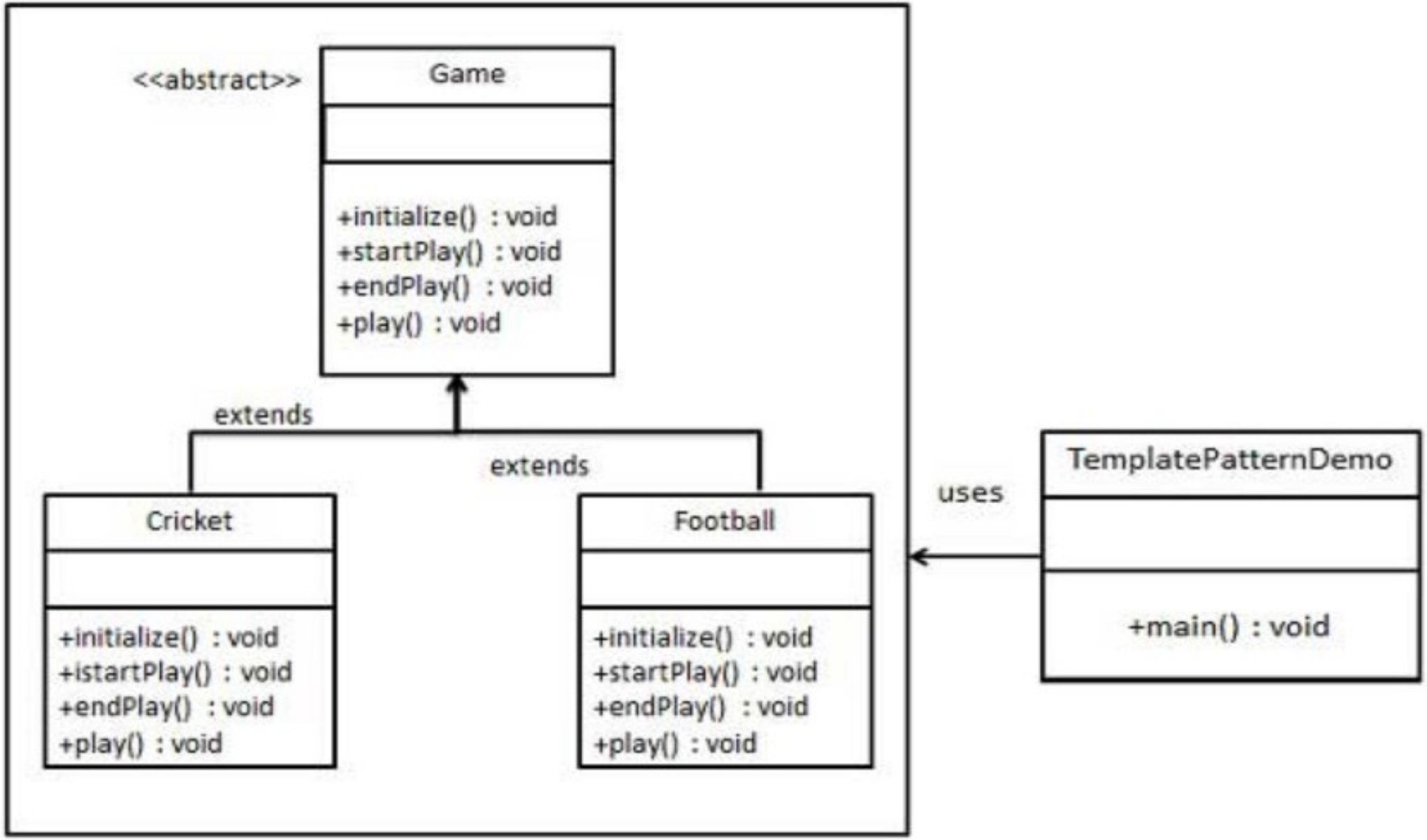
```java
package StrategyDesignPattern;

//Step4:Use the Context to see change in behaviour when it changes its Strategy.
    public class StrategyPatternDemo
        {
            public static void main(String[] args)
                {
                    Context context = new Context(new OperationAdd());
                    System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

                    context = new Context(new OperationSubstract());
                    System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

                    context = new Context(new OperationMultiply());
                    System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
                }
        }
```

Template Pattern

in Template pattern, an abstract class exposes defined way(s)/template(s) to execute its methods.

Its subclasses can override the method implementation as per need, but the invocation is to be in the same way as defined by an abstract class.

This pattern comes under behavior pattern category.


<<abstract>>
Game
+initialize() : void
+startPlay() : void
+endPlay() : void
+play() : void
extends
extends
Cricket
+initialize() : void
+istartPlay() : void
+endPlay() : void
+play() : void
Football
+initialize() : void
+startPlay() : void
+endPlay() : void
+play() : void
uses
TemplatePatternDemo
+main() : void

Implementation

We are going to create a ***Game*** abstract class defining operations with a template method set to be final so that it cannot be overridden

Cricket and ***Football*** are concrete classes that **extend** *Game* and override its methods.

TemplatePatternDemo, our demo class, will use *Game* to demonstrate use of template pattern.

```java
package TempleteDesignPattern;

//Step 1: Create an abstract class with a template method being final.
public abstract class Game
    {
       abstract void initialize();
       abstract void startPlay();
       abstract void endPlay();
       //template method
       public final void play()
       {
           initialize();
           startPlay();
           endPlay();
       }
    }
```

```java
package TempleteDesignPattern;

//Step 2 :Create concrete classes extending the Game class.
    public class Football extends Game
    {
        @Override
        void initialize() {
            System.out.println("Football Game Initialized! Start playing.");
        }
        @Override
        void startPlay() {
            System.out.println("Football Game Started. Enjoy the game!");
        }
        @Override
        void endPlay() {
            System.out.println("Football Game Finished!\n");
        }
    }
```

```java
package TempleteDesignPattern;

//Step 2 :Create concrete classes extending the Game class.
public class Cricket extends Game
{
    @Override
    void initialize() {
        System.out.println("Cricket Game Initialized! Start playing.");
    }
    @Override
    void startPlay() {
        System.out.println("Cricket Game Started. Enjoy the game!");
    }
    @Override
    void endPlay() {
        System.out.println("Cricket Game Finished!\n");
    }
}
```

```
package TempleteDesignPattern;

//Step 3:Use the Game's template method play() to
//demonstrate a defined way of playing game.

public class TemplatePatternDemo
{
    public static void main(String[] args)
    {
            Game game;

            game = new Cricket();
            game.play();

            game= new Football();
            game.play();
    }
}
```

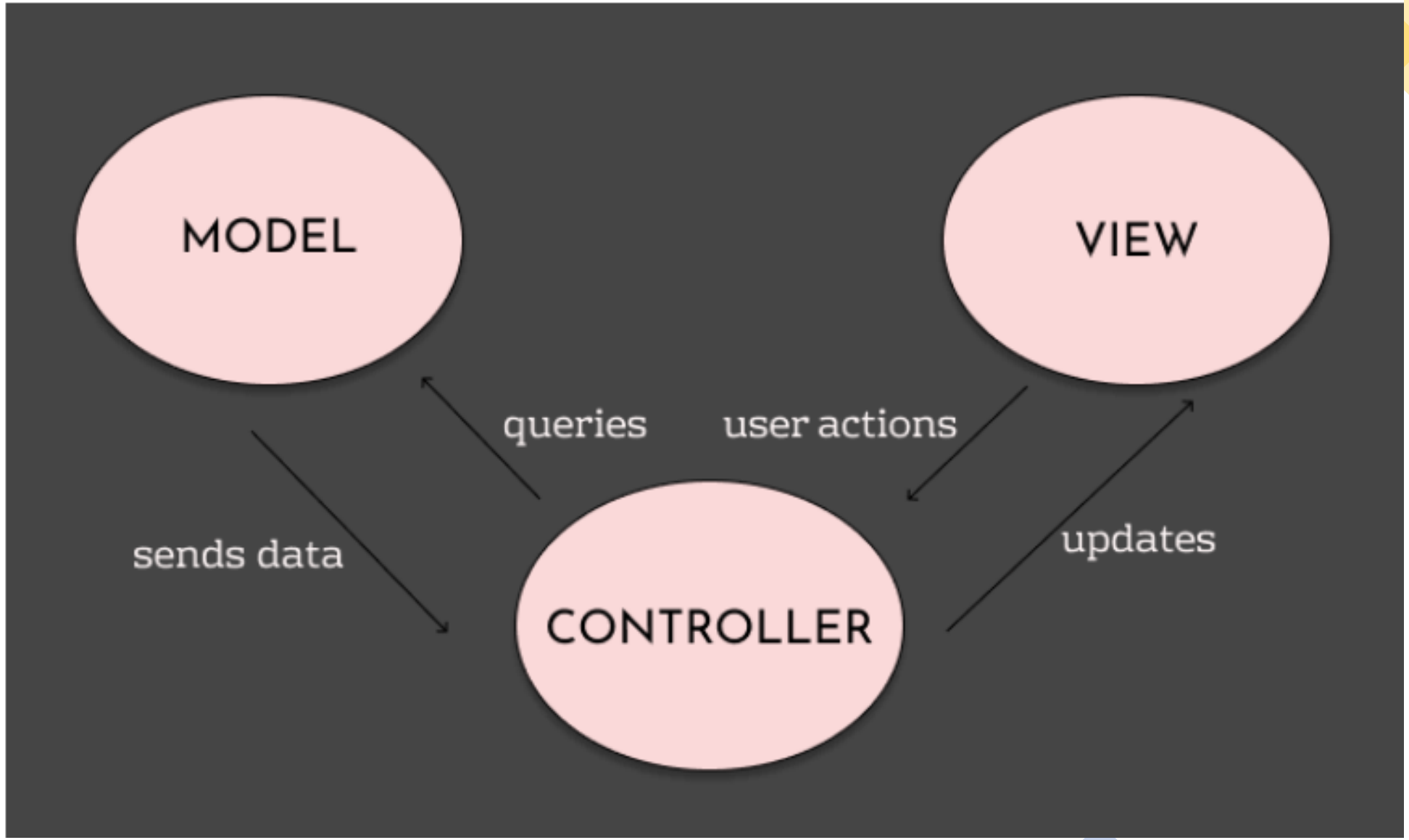
Model-View-Controller Pattern (MVC Pattern)

Model - It is a representation of a real-life instance or object in our code base.

View - View represents the visualization of the data that model contains.

represents the interface through which the user interacts with our application.

Controller - When a user takes an action, the Controller handles the action and updates the Model if necessary. It keeps view and model separate.


MODEL
VIEW
queries
user actions
sends data
updates
CONTROLLER

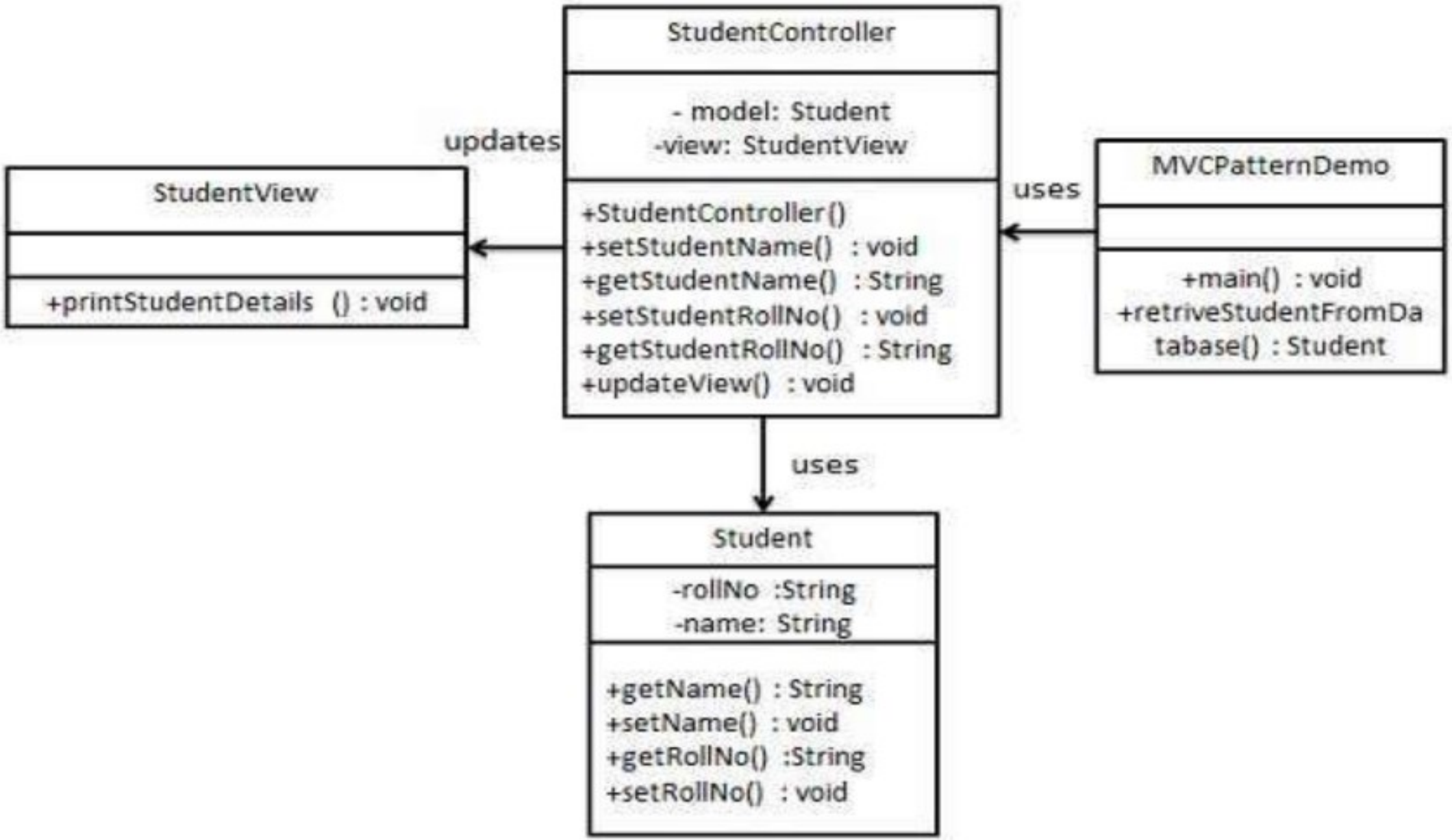
Let’s look at a simple scenario.

If you go to an e-commerce website, the different pages you see are provided by the View layer.

When you click on a particular product to view more, the Controller layer processes the user’s action.

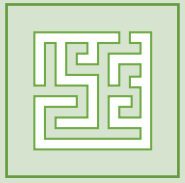
This may involve getting data from a data source using the Model layer.

The data is then bundled up together and arranged in a View layer and displayed to the user. Rinse and repeat.


StudentController
- model: Student
-view: StudentView
+StudentController()
+setStudentName() : void
+getStudentName() : String
+setStudentRollNo() : void
+getStudentRollNo() : String
+updateView() : void
updates
StudentView
+printStudentDetails () : void
uses
MVCPatternDemo
+main() : void
+retriveStudentFromDatabase() : Student
uses
Student
-rollNo :String
-name: String
+getName() : String
+setName() : void
+getRollNo() :String
+setRollNo() : void

Implementation

We are going to create a ***Student*** object acting as a model.***StudentView*** will be a view class which can print student details on console and



StudentController is the controller class responsible to store data in *Student* object and update view ***StudentView*** accordingly.

MVCPatternDemo, our demo class, will use ***StudentController*** to demonstrate use of MVC pattern.

Have
a good
day 😍