Design Patterns

Definition

- Wikipedia definition

  “A design pattern is a general repeatable solution to a commonly occurring problem in software design”

- Quote from Christopher Alexander

  “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice” (GoF,1995)

Overview

- Gang of Four (GOF) is
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides
- Four authors published a book titled Design Patterns - Elements of Reusable Object-Oriented Software
- which initiated the concept of Design Pattern in Software development.

Usage of Design Pattern

- Common platform for developers
  - Design patterns provide a standard terminology and are specific to particular scenario.
- Best Practices
  - Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development.
  - Learning these patterns helps unexperienced developers to learn software design in an easy and faster way.

Design Patterns Classification

A Pattern can be classified as

- Creational
  - concern the process of object creation.
- Structural
  - concern with integration and composition of classes and objects
- Behavioral
  - concern with class or object communication.

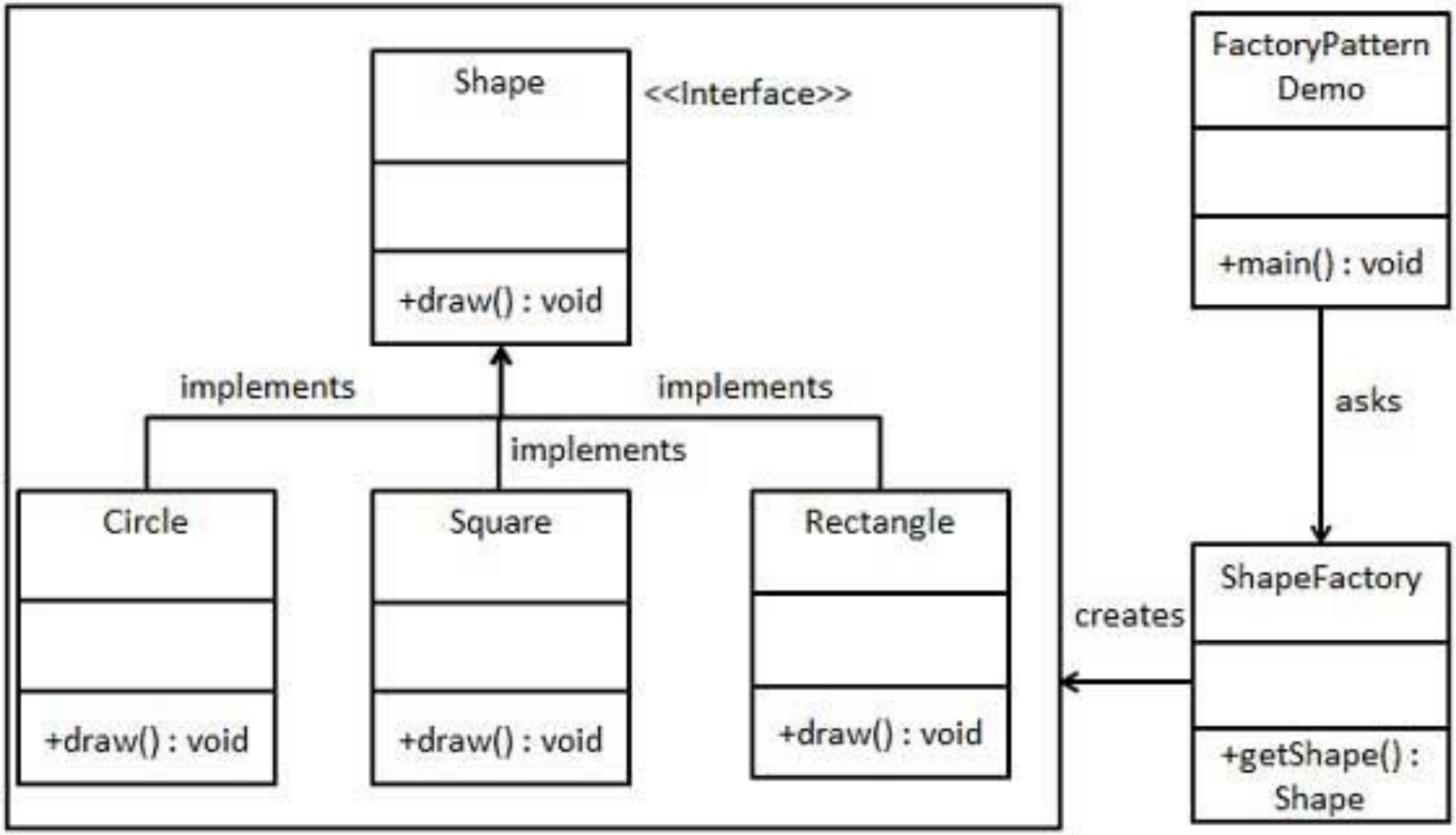
Creational Patterns

- **Abstract Factory**
- **Builder**
- **Factory Method**
- **Prototype**
- **Singleton**

Factory Method

- Factory pattern is one of the most used design patterns in Java.
- This pattern provides one of the best ways to create an object.
- We create object without exposing the creation logic to the client and refer to newly created object using a common interface.
- Example
  - Create a *Shape* interface and concrete classes implementing the *Shape* interface.
  - A factory class *ShapeFactory* is defined as a next step.
  - *FactoryPatternDemo*, our demo class will use *ShapeFactory* to get a *Shape* object.
  - It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs.

Factory Method



- **Step 1**
  - **Create an interface.**
  - **Shape.java**

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

- **Step 2**
  - **Create concrete classes implementing the same interface.**
  - **Rectangle.java**

```
public class Rectangle implements Shape {
 @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

- **Square.java**

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

- **Circle.java**

```
public class Circle implements Shape {
  @Override
  public void draw() {
    System.out.println("Inside Circle::draw() method.");
  }
}
```

- **Step 3**
  - **Create a Factory to generate object of concrete class based on given information.**
  - **ShapeFactory.java**

```
public class ShapeFactory {
 //use getShape method to get object of type shape
public Shape getShape(String shapeType){
    if(shapeType == null){
      return null;
    }
```

```
if(shapeType.equalsIgnoreCase("CIRCLE"))
   {
       return new Circle();
    }
else
if(shapeType.equalsIgnoreCase("RECTANGLE"))
    {
       return new Rectangle();
     } else
```

```
if(shapeType.equalsIgnoreCase("SQUARE"))
    {
       return new Square();
     }
     return null;
  }
}
```

- **Step 4**
- **Use the Factory to get object of concrete class by passing an information such as type.**
- **FactoryPatternDemo.java**

```
public class FactoryPatternDemo {
   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");
```

```
//call draw method of Circle
    shape1.draw();

    //get an object of Rectangle and call its draw method.
    Shape shape2 =
            shapeFactory.getShape("RECTANGLE");
   //call draw method of Rectangle
    shape2.draw();
```

```
 //get an object of Square and call its draw method.
    Shape shape3 = shapeFactory.getShape("SQUARE");

    //call draw method of square
    shape3.draw();
  }
}
```

- **Step 5**
- **Verify the output.**

  Inside Circle::draw() method.

  Inside Rectangle::draw() method.

  Inside Square::draw() method.

Factory Method

- **Step 1**
  - **Create an interface.**
  - **Shape.java**

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

- **Step 2**
  - **Create concrete classes implementing the same interface.**
  - **Rectangle.java**

```
public class Rectangle implements
Shape {
 @Override
   public void draw() {
     System.out.println("Inside
Rectangle::draw() method.");
   }
}
```

- **Square.java**

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

- **Circle.java**

```
public class Circle implements Shape {
  @Override
  public void draw() {
    System.out.println("Inside
Circle::draw() method.");
  }
}
```

Factory Method

- **Step 3**
  - **Create a Factory to generate object of concrete class based on given information.**
  - **ShapeFactory.java**

```
public class ShapeFactory {
 //use getShape method to get
object of type shape
public Shape getShape(String
shapeType){
     if(shapeType == null){
       return null;
     }

if(shapeType.equalsIgnoreCase("
CIRCLE"))
   {
       return new Circle();
    }
else
if(shapeType.equalsIgnoreCase("
RECTANGLE"))
    {
       return new Rectangle();
     } else

if(shapeType.equalsIgnoreCase("
SQUARE"))
    {
       return new Square();
     }
     return null;
  }
}
```

Factory Method

- **Step 4**
- **Use the Factory to get object of concrete class by passing an information such as type.**
- **FactoryPatternDemo.java**

```
public class FactoryPatternDemo {
   public static void main(String[] args) {
      ShapeFactory shapeFactory = new
ShapeFactory();

      //get an object of Circle and call its draw
method.
      Shape shape1 =
shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its
draw method.
      Shape shape2 =
shapeFactory.getShape("RECTANGLE");
      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw
method.
      Shape shape3 =
shapeFactory.getShape("SQUARE");

      //call draw method of square
      shape3.draw();
   }
}
```

- **Step 5**
- **Verify the output.**

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

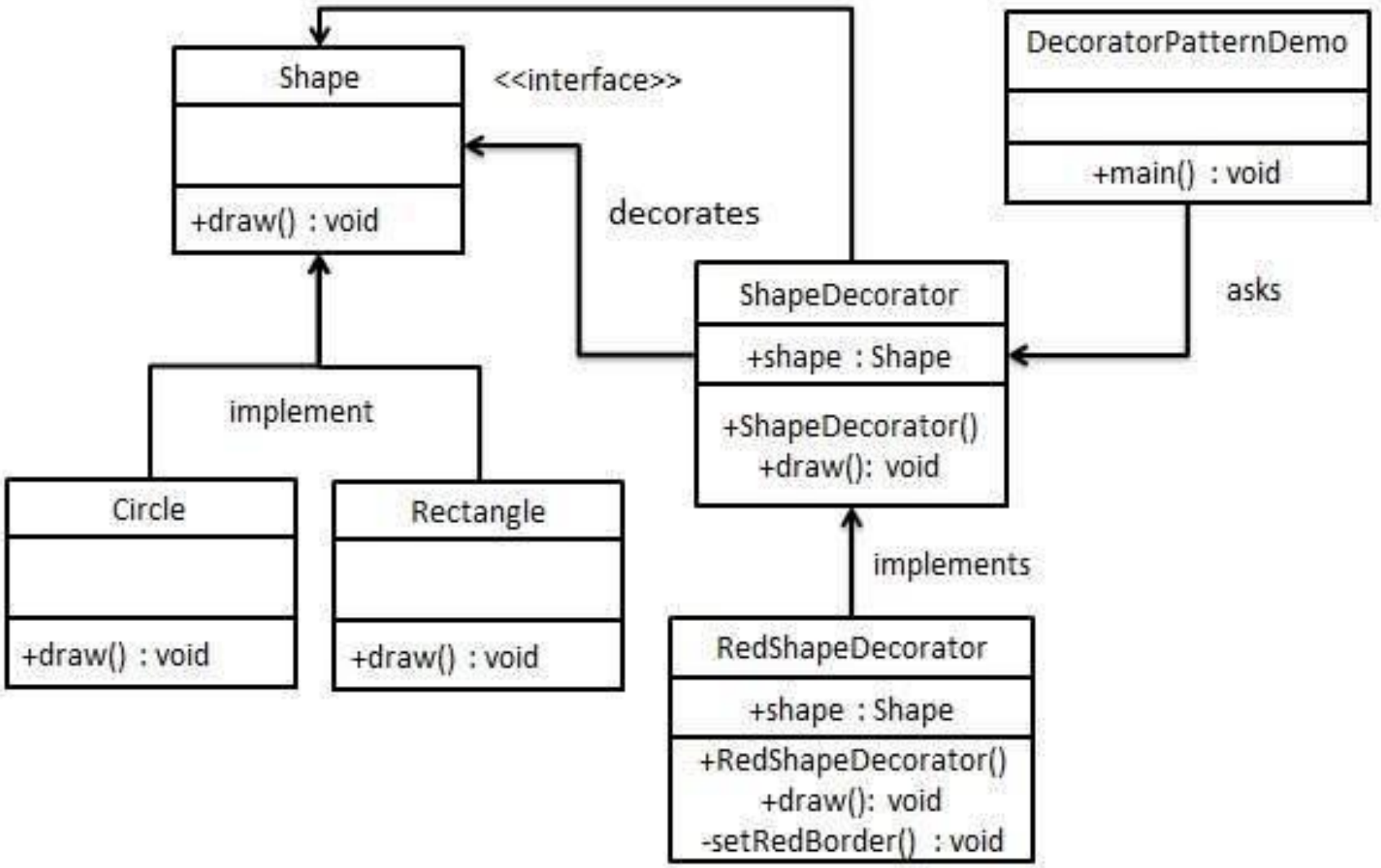
Structural Patterns

- **Adapter**
- **Bridge**
- **Composite**
- **Decorator**
- **Facade**
- **Flyweight**
- **Proxy**

Decorator

- Decorator pattern allows a user to add new functionality to an existing object without altering its structure.
- Acts as a wrapper to existing class.
- This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.
- **Example**
  - we will decorate a shape with some color without altering the shape class.
  - Create a *Shape* interface and concrete classes implementing the *Shape* interface.
  - We will then create an abstract decorator class *ShapeDecorator* implementing the *Shape* interface and having *Shape* object as its instance variable.
  - *RedShapeDecorator* is concrete class implementing *ShapeDecorator*.
  - *DecoratorPatternDemo*, our demo class will use *RedShapeDecorator* to decorate *Shape* objects.

Decorator


Shape
<<interface>>
+draw() : void
DecoratorPatternDemo
+main() : void
decorates
asks
ShapeDecorator
+shape : Shape
+ShapeDecorator()
+draw(): void
implement
Circle
+draw() : void
Rectangle
+draw() : void
implements
RedShapeDecorator
+shape : Shape
+RedShapeDecorator()
+draw(): void
-setRedBorder() : void

Decorator

- **Step 1**
  - **Create an interface.**
  - **Shape.java**

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

- **Step 2**
  - **Create concrete classes implementing the same interface.**
  - **Rectangle.java**

```
public class Rectangle
implements Shape {
  @Override
  public void draw() {
    System.out.println("Shape:
Rectangle");
  }
}
```

- **Circle.java**

```
public class Circle implements
Shape {

  @Override
  public void draw() {
    System.out.println("Shape:
Circle");
  }
}
```

Decorator

- **Step 3**
  - **Create abstract decorator class implementing the Shape interface.**
  - **ShapeDecorator.java**

```
public abstract class ShapeDecorator implements Shape {
  protected Shape decoratedShape;

  public ShapeDecorator(Shape decoratedShape){
    this.decoratedShape = decoratedShape;
  }

  public void draw(){
    decoratedShape.draw();
  }
}
```

Decorator

- **Step 4**
  - **Create concrete decorator class extending the ShapeDecorator class.**
  - **RedShapeDecorator.java**

```
public class RedShapeDecorator extends ShapeDecorator {
  public RedShapeDecorator(Shape decoratedShape) {
    super(decoratedShape);
  }
  @Override
  public void draw() {
    decoratedShape.draw();
    setRedBorder(decoratedShape);
  }
  private void setRedBorder(Shape decoratedShape){
    System.out.println("Border Color: Red");
  }
}
```

Decorator

- **Step 5**
  - **Use the RedShapeDecorator to decorate Shape objects.**
  - **DecoratorPatternDemo.java**

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

    Shape circle = new Circle();
    Shape redCircle = new RedShapeDecorator(new Circle());
    Shape redRectangle = new RedShapeDecorator(new Rectangle());

    System.out.println("Circle with normal border");
    circle.draw();

    System.out.println("\nCircle of red border");
    redCircle.draw();

    System.out.println("\nRectangle of red border");
    redRectangle.draw();
  }
}
```

Decorator

- **Step 6**
  - **Verify the output.**
  - **Circle with normal border**
  - **Shape: Circle**
  - **Circle of red border**
  - **Shape: Circle**
  - **Border Color: Red**
  - **Rectangle of red border**
  - **Shape: Rectangle**
  - **Border Color: Red**

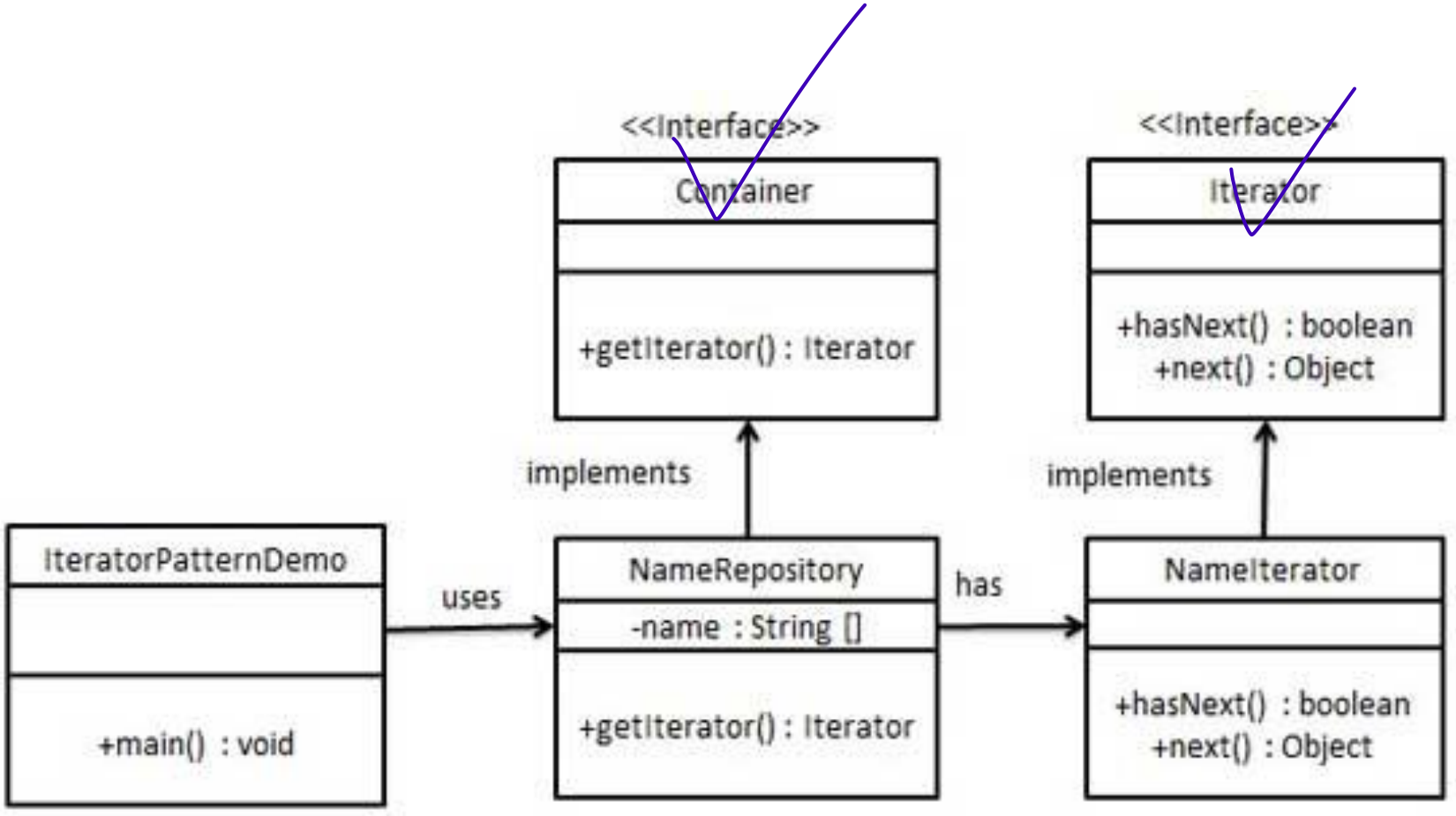
Behavioral Pattern

- **Chain of Responsibility**
- **Command**
- **Iterator**
- **Interpreter**
- **Mediator**
- **Memento**
- **Observer**
- **State**
- **visitor**

Iterator

- Iterator pattern is very commonly used design pattern in Java and .Net programming environment.
- This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.
- Example
  - create a *Iterator* interface which narrates navigation method and a *Container* interface which returns the iterator.
  - Concrete classes implementing the *Container* interface will be responsible to implement *Iterator* interface and use it
  - *IteratorPatternDemo*, our demo class will use *NamesRepository*, a concrete class implementation to print a *Names* stored as a collection in *NamesRepository*.

Iterator


<<Interface>>
Container
+getIterator() : Iterator
<<Interface>>
Iterator
+hasNext() : boolean
+next() : Object
implements
implements
IteratorPatternDemo
+main() : void
uses
NameRepository
-name : String []
+getIterator() : Iterator
has
NameIterator
+hasNext() : boolean
+next() : Object

Iterator

- **Step 1**
  - **Create interfaces.**
  - **Iterator.java**

```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}
```

  - **Container.java**

```
public interface Container {
   public Iterator getIterator();
}
```

Iterator

- **Step 2**
  - **Create concrete class implementing the Container interface. This class has inner class NameIterator implementing the Iterator interface.**
  - **NameRepository.java**

```
public class NameRepository
implements Container {
  public String names[] = {"Robert" ,
"John" ,"Julie" , "Lora"};

  @Override
  public Iterator getIterator() {
    return new NameIterator();
  }
private class NameIterator
implements Iterator {
  int index;
  @Override
  public boolean hasNext() {

    if(index < names.length){
      return true;
    }
    return false;
  }
  @Override
  public Object next() {

    if(this.hasNext()){
      return names[index++];
    }
    return null;
  }
 }
}
```

Iterator

- **Step 3**
  - **Use the NameRepository to get iterator and print names.**
  - **IteratorPatternDemo.java**

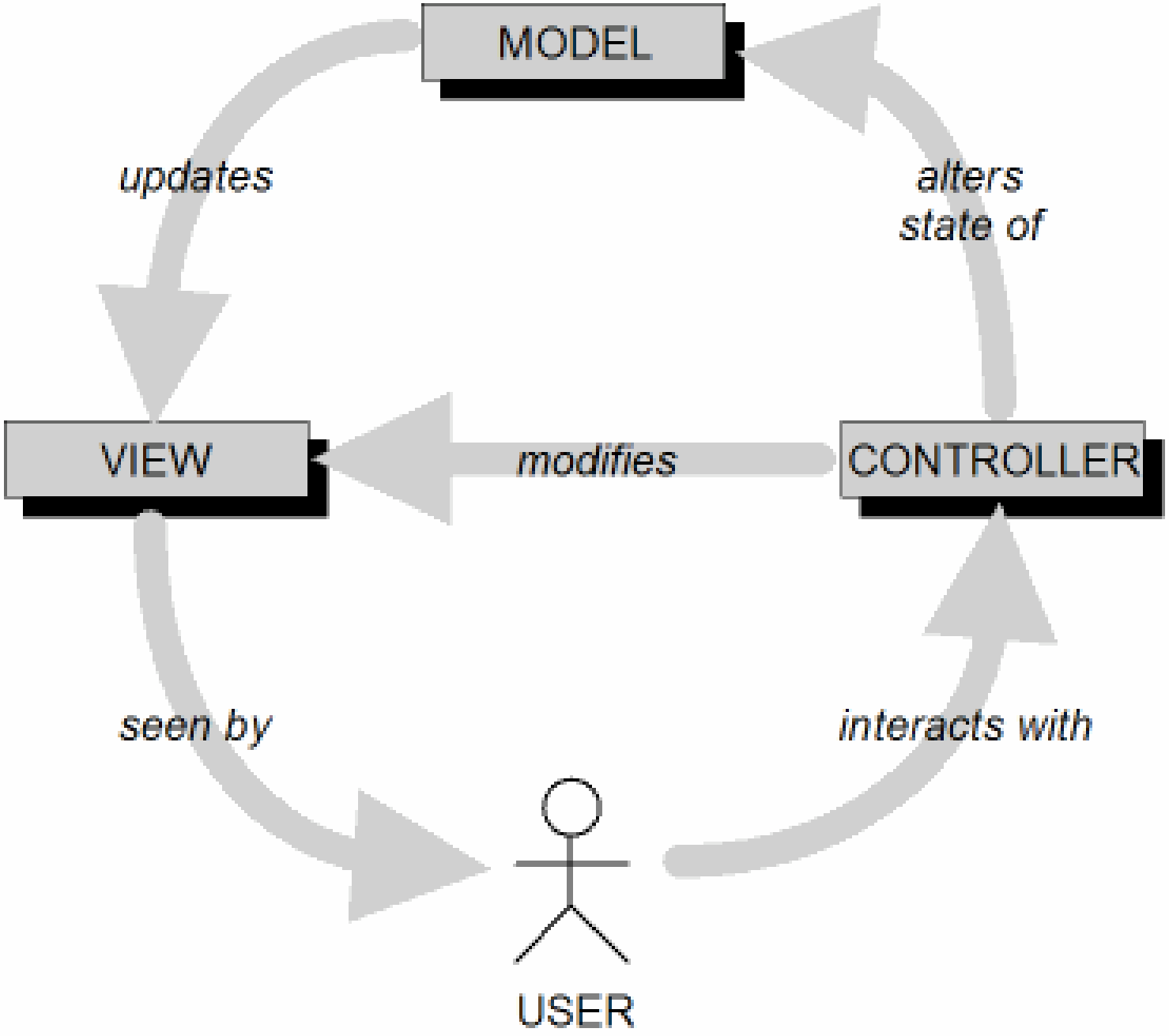
```
public class
IteratorPatternDemo {


   public static void main(String[]
args) {
      NameRepository
namesRepository = new
NameRepository();


      for(Iterator iter =
namesRepository.getIterator();
iter.hasNext();){
         String name =
(String)iter.next();
         System.out.println("Name :
" + name);
      }
   }
}
```

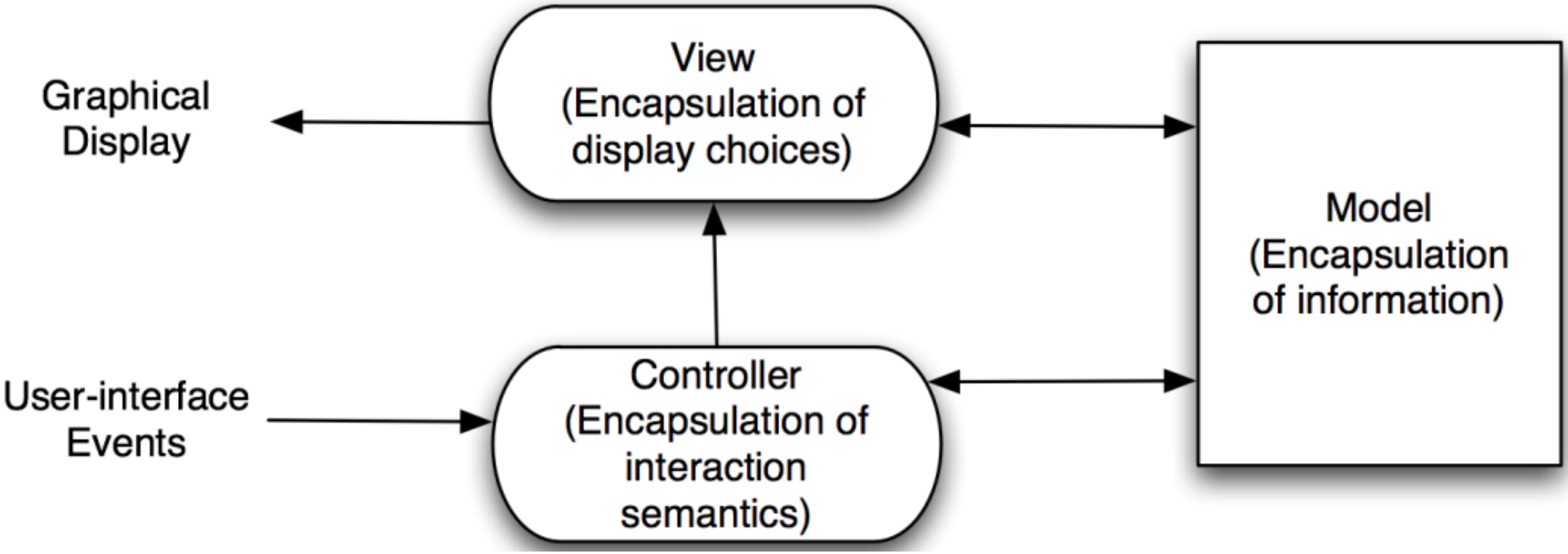
- **Step 4**
- **Verify the output.**

  - **Name : Robert**
  - **Name : John**
  - **Name : Julie**
  - **Name : Lora**

Design Patterns - MVC

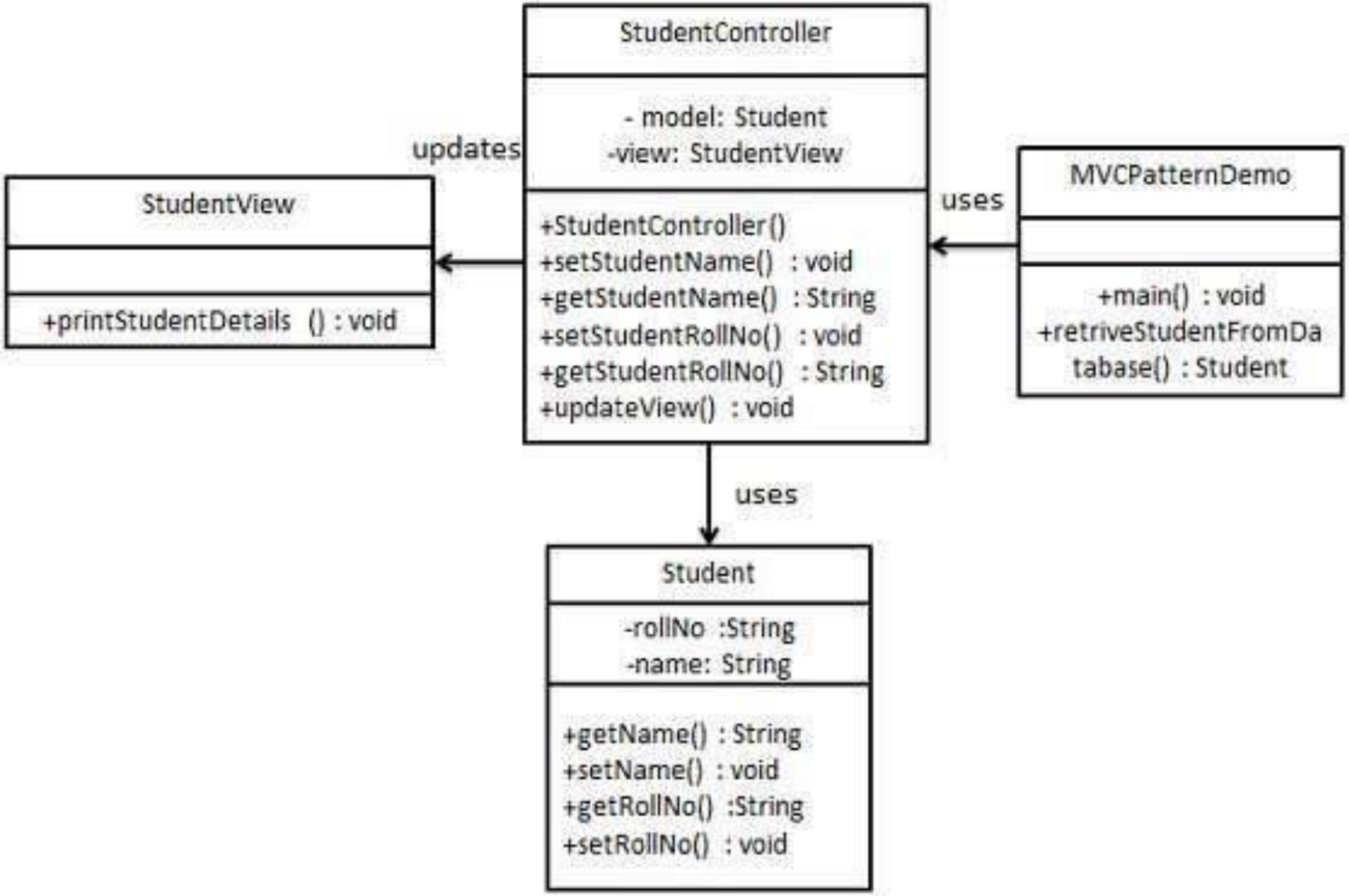
- MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.
  - Model object with data and logic
    - Model represents an object carrying data. It can also have logic to update view if its data changes.
  - View visualization of object
    - View represents the visualization of the data that model contains.
  - Controller controls data from model and updates view if necessary
    - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.


MODEL
updates
alters
state of
VIEW
modifies
CONTROLLER
seen by
interacts with
USER


Graphical Display
View (Encapsulation of display choices)
Model (Encapsulation of information)
User-interface Events
Controller (Encapsulation of interaction semantics)

Implementation - MVC

- Create a Student object acting as a model.
- StudentView will be a view class which can print student details on console
- 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.


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

Step 1: Create Model

Student.java

```
public class Student {
   private String rollNo;
   private String name;
   public String getRollNo() {
      return rollNo;
   }
   public void setRollNo(String rollNo) {
      this.rollNo = rollNo;
   }
```

Step 1: Create Model

Student.java

```
  public String getName() {
      return name;
    }
    public void setName(String name) {
      this.name = name;
    }
}
```

Step 2: Create View

StudentView.java

```
public class StudentView {
   public void printStudentDetails(String studentName,
String studentRollNo){
      System.out.println("Student: ");
      System.out.println("Name: " + studentName);
      System.out.println("Roll No: " + studentRollNo);
   }
}
```

Step 3:Create Controller

StudentController.java

```
public class StudentController {
   private Student model;
   private StudentView view;

   public StudentController(Student model, StudentView view){
      this.model = model;
      this.view = view;
   }

   public void setStudentName(String name){
      model.setName(name);
   }
```

Step 3:Create Controller

StudentController.java

```
   public String getStudentName(){
      return model.getName();
   }
   public void setStudentRollNo(String rollNo){
      model.setRollNo(rollNo);
   }
   public String getStudentRollNo(){
      return model.getRollNo();
   }

   public void updateView(){
      view.printStudentDetails(model.getName(), model.getRollNo());
   }
}
```

Step 4 : MVC design pattern usage

MVCPatternDemo.java

```
public class MVCPatternDemo {
   public static void main(String[] args) {
      //fetch student record based on his roll no from the database
      Student model  = retriveStudentFromDatabase();


      //Create a view : to write student details on console
      StudentView view = new StudentView();
      StudentController controller = new StudentController(model,
view);
      controller.updateView();
```

Step 4 : MVC design pattern usage

MVCPatternDemo.java

```
 //update model data
      controller.setStudentName("John");
      controller.updateView();
   }
 private static Student retriveStudentFromDatabase(){
      Student student = new Student();
      student.setName("Robert");
      student.setRollNo("10");
      return student;
   }
 }
```

Step 5: Verify the output.

- Student:
- Name: Robert
- Roll No: 10
- Student:
- Name: John
- Roll No: 10