Hyperion Development

Task: Object Oriented Design and Design Patterns

www.hyperiondev.com

Introduction

Welcome to the Object Oriented Design and Design Patterns Task!

This task will attempt to provide a brief overview of what design patterns are in the realm of software development. We will also take a closer look a few specific design patterns.

You don't have to take our courses alone! This course has been designed to be taken with an online mentor that marks your submitted code by hand and supports you to achieve your career goals on a daily basis.

To access this mentor support simply navigate to www.hyperiondev.com/support.

What are Design Patterns?

Simply put, a design pattern is a general repeatable solution to commonly occurring problems faced by software developers during software design. They represent the best practices used by experienced object-oriented software developers and are obtained by trial and error by numerous software developers over a substantial period of time. A design pattern is not a finished design that can be transformed directly into code. It is rather a description or template of how to solve a problem that can be used in many different situations.

The Hillside Group (http://hillside.net), which is dedicated to maintaining information about patterns, describes design patterns as follows:

Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.

Design patterns have made a huge impact on object-oriented software design. Not only are they tested solutions to common problems but they also have become a vocabulary for talking about a design. You can therefore explain your design by describing the patterns that you have used.

Benefits of using Design Patterns

- Using them can speed up development process by providing tested, proven development paradigms.
- Reusing design patterns can help prevent subtle issues that may only become visible later in the implementation which can cause major problems.
- It improves code readability for coders and architects familiar with the patterns.
- Design patterns provide general solutions, documented in a format that does not require specifics tied to a particular problem.
- They allow developers to communicate using well-known, well understood names for software interactions.
- Design patterns are constantly improved over time, making them more robust than ad-hoc designs.

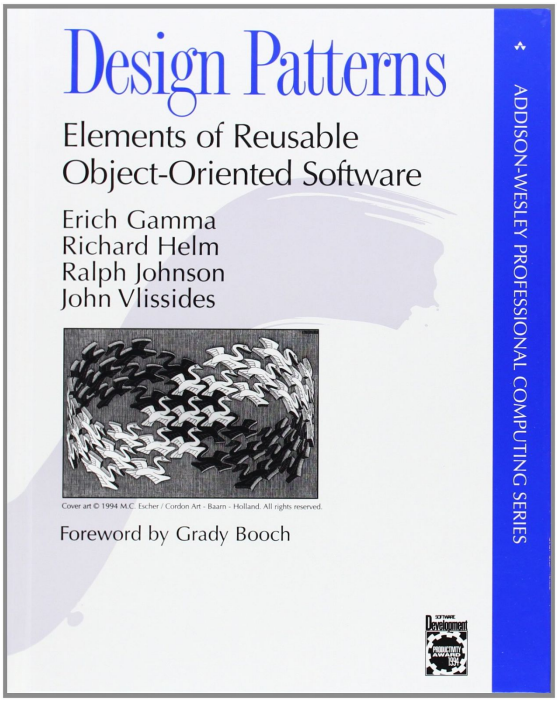
- They are language neutral and so can be applied to any object-orientated programming language.

The Gang of Four

The best known design patterns were described in a book titled Design Patterns - Elements of Reusable Object-Oriented Software. This book initiated the concept of Design Pattern in Software development. It was written by four authors, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, who collectively became known as the “Gang of Four” (GOP).

According to the Gang of Four, design patterns are primarily based on the following principles of object orientated design:

- "Program to an 'interface', not an 'implementation'." (Gang of Four 1995:18)
- Composition over inheritance: "Favor 'object composition' over 'class inheritance'." (Gang of Four 1995:20)



The cover of Design Patterns - Elements of Reusable Object-Oriented Software

Types of Design Patterns

According to the book Design Patterns - Elements of Reusable Object-Oriented Software, design patterns can be classified into three categories:

1. Creational
2. Structural
3. Behavioural

Creational Design Patterns

Creational design patterns provide a way to create objects while hiding the creation logic, instead of instantiating objects directly using the new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

Structural Design Patterns

Structural design patterns are all about class and object composition. Interfaces are composed using the concept of inheritance. The concept of inheritance is also used to define ways to compose objects to obtain new functionalities.

Behavioural Design Patterns

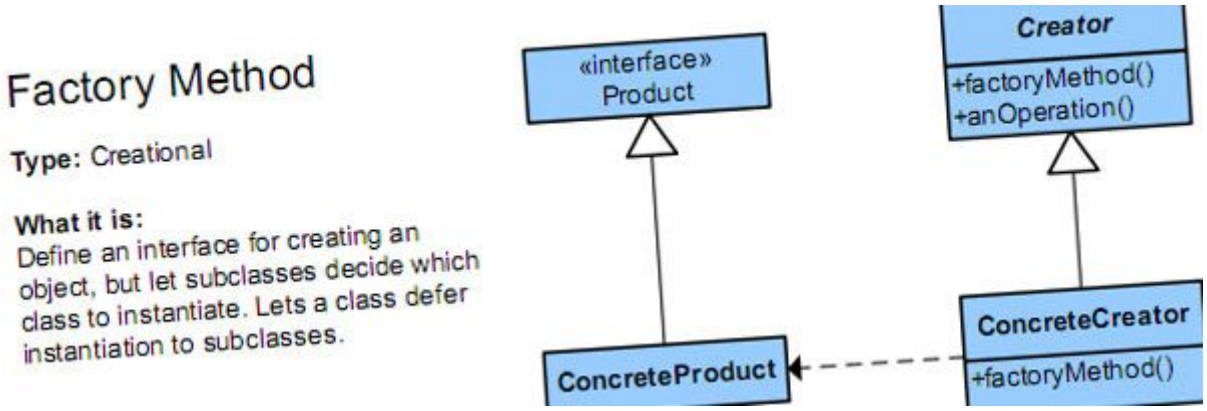
Behavioural design patterns are concerned with communication between objects.

We will now take a closer look at a few popular design patterns.

Factory Pattern

On of the most popular and widely used design patterns in Java is the Factory pattern. The Factory pattern is a creational pattern as it provides one of the best ways to create an object. As the name implies it is a class that acts as a factory of object instances.

The main goal of the Factory Pattern is to encapsulate the creational procedure that may span different classes into one single function. The factory method will be able to return the correct object if it provided with the correct context.



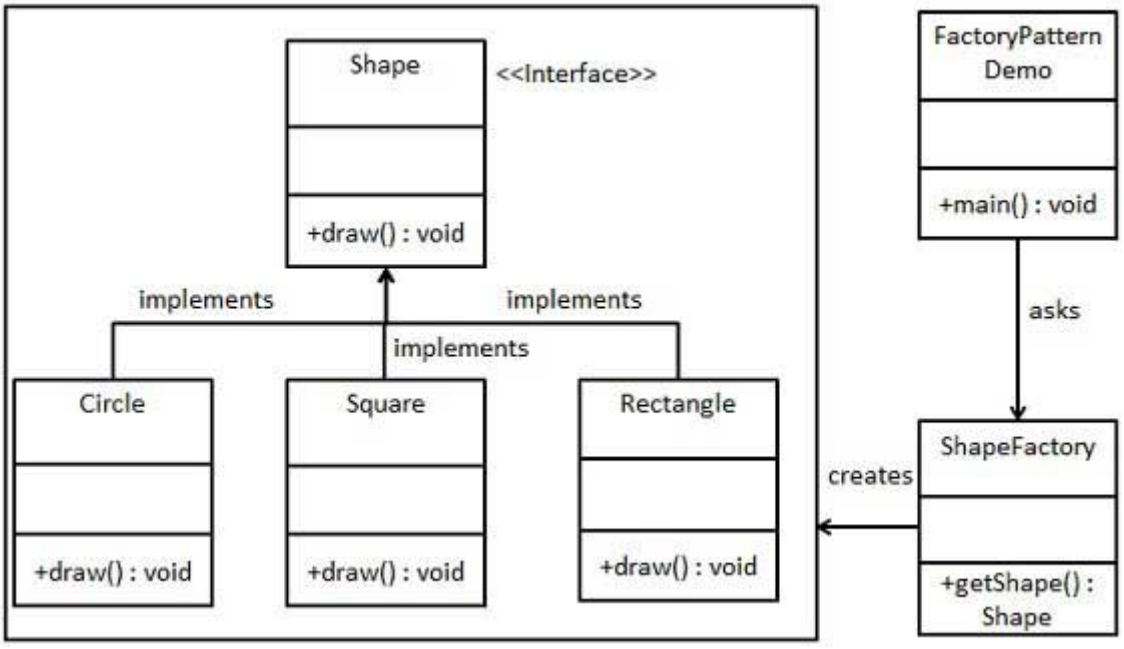
The Factory Method (www.code.tutsplus.com)

When to Use the Factory Pattern

If you have multiple different variations of a single entity, the factory pattern can be useful. For example, let's say that you have a Shape class. This class has different variations, such as Circle, Square and Rectangle. You need to be able to create Different Shapes depending on you needs. This is where you can a factory to create the Shapes for you!

Implementation of the Factory Pattern

We are firstly going to create a Shape interface and concrete classes that implement the Shape interface. We then are going to define the factory class ShapeFactory. Our demo class FactoryPatternDemo will use this factory class to get a Shape object.



1. We firstly create an interface.

   Shape.java

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

2. We then create concrete classes to implement the 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.");
   }
}
```

3. We now create a Factory class 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;
   }
}
```

4. The Factory class is then used to get object of concrete class by passing some information. In this case we pass the type of shape.

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 circle
```

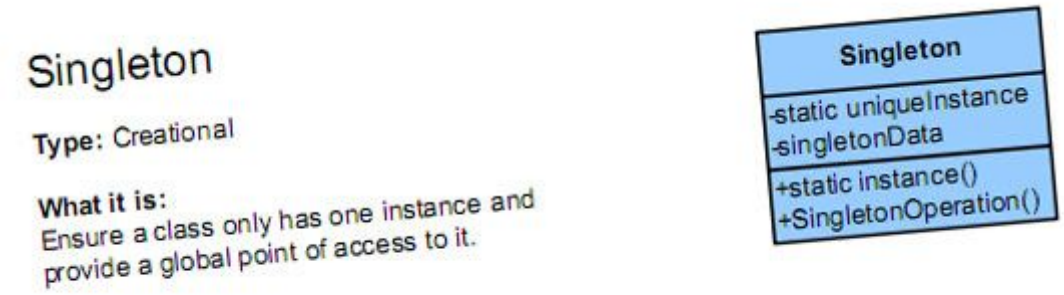
```
        shape3.draw();
    }
}
```

5. The output should be the following:

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

Singleton Pattern

One of the simplest patterns in Java is the Singleton pattern. This creational design pattern involves a single class which is responsible for creating a single object and making sure that only that single object gets created. This class provides a way to access this single object which can be accessed directly without need to instantiate the object of the class.



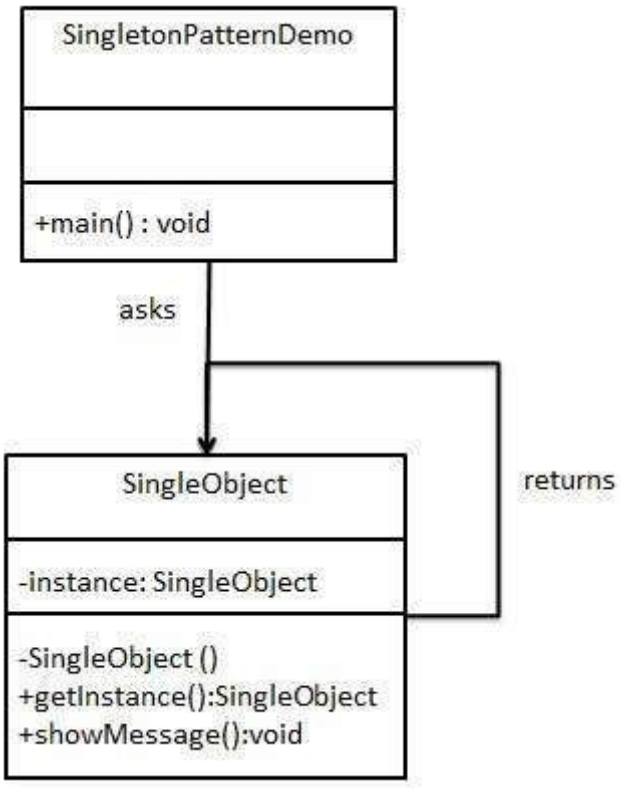
The Singleton Pattern (www.code.tutsplus.com)

When to Use the Singleton Pattern

You can use the singleton pattern to avoid having to pass an instance via constructor or argument, if you need to pass a specific instance from one class to another.

Implementation of the Singleton Pattern

We will now create a SingleObject class which has its constructor as private and a static instance of itself. This class has a static method to get its static instance. The demo class SingletonPatternDemo, will use SingleObject class to get a SingleObject object.



1. We firstly create a Singleton class.

   SingleObject.java

```
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!");
   }
}
```

2. Now, get the single object from the Singleton class

   SingletonPatternDemo.java

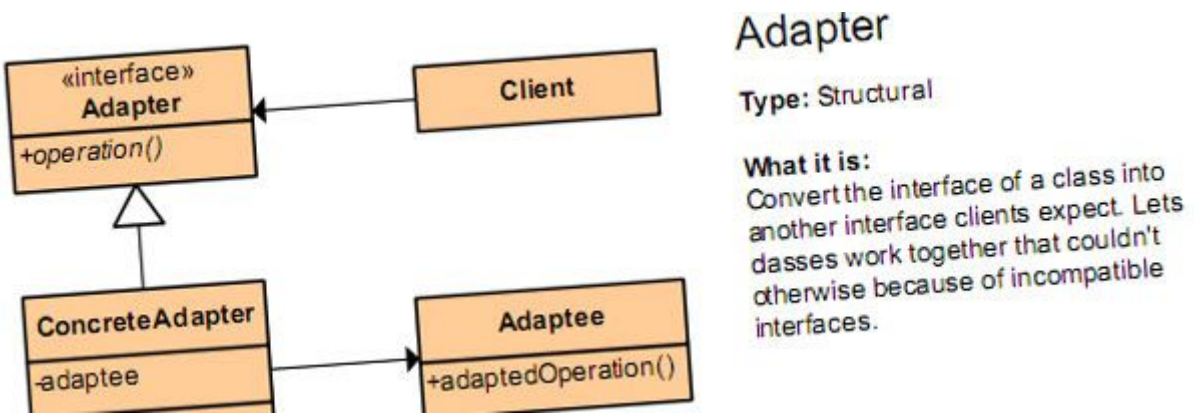
```
public class SingletonPatternDemo {
   public static void main(String[] args) {

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

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

      //show the message
      object.showMessage();
   }
}
```

3. The output should be as follows:

```
Hello World!
```

Adapter Pattern

The Adapter pattern is a structural pattern that works as a bridge between two incompatible interfaces. It involves a single class which is responsible to join functionalities of independent or incompatible interfaces.

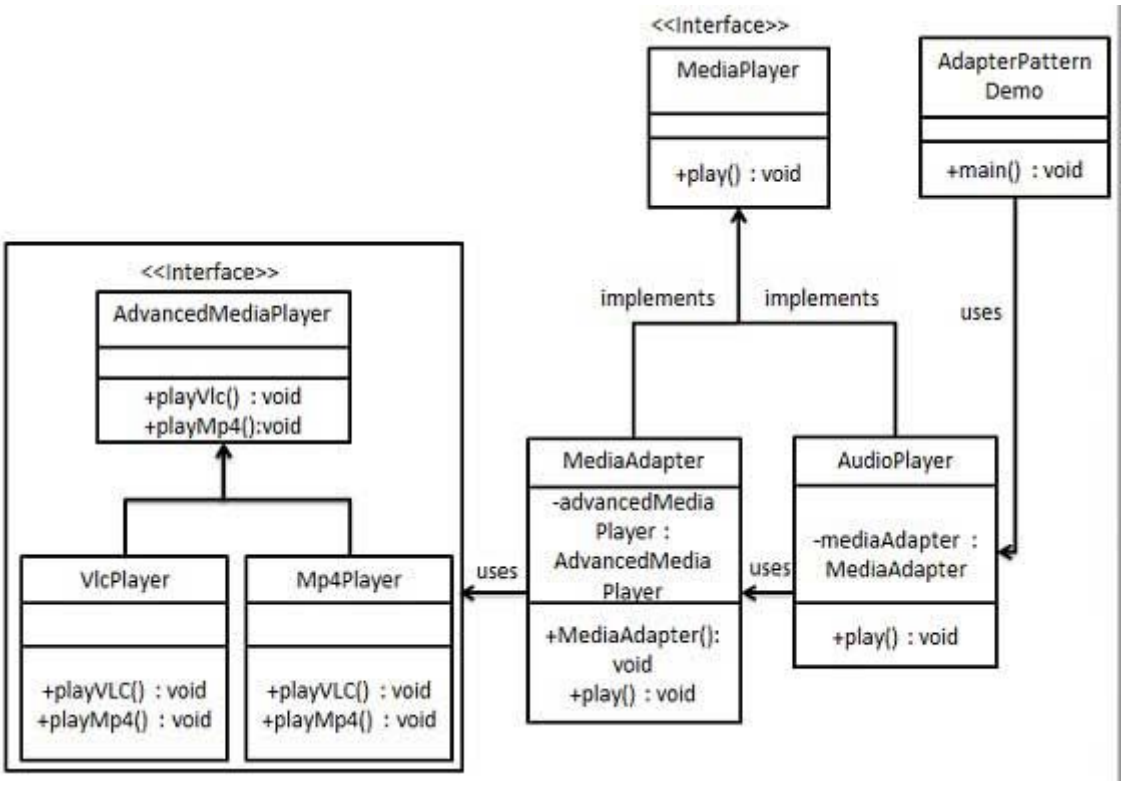


The Adapter Pattern (www.code.tutsplus.com)

The adapter class is also known as a wrapper because it lets you "wrap" actions into a class and reuse these actions in the correct situations.

Implementation of the Adapter Pattern

We will now demonstrate how the Adapter pattern is used in the following example where an audio player device can play mp3 files only and wants to use an advanced audio player capable of playing vlc and mp4 files.



We have a concrete class named AudioPlayer that implements the MediaPlayer interface. AudioPlayer can play mp3 format audio files by default. We also have concrete classes that implement the AdvancedMediaPlayer interface. These classes can play vlc and mp4 format files.

We would like AudioPlayer to play these other formats as well. In order to do 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 and passes it the desired audio type without knowing the actual class which can play the desired format. The demo class, AdapterPatternDemo, will use AudioPlayer class to play various formats.

1. Firstly, create the MediaPlayer and AdvancedMediaPlayer interfaces.

   MediaPlayer.java

```java
public interface MediaPlayer {
   public void play(String audioType, String fileName);
}
```

   AdvancedMediaPlayer.java

```java
public interface AdvancedMediaPlayer {
   public void playVlc(String fileName);
   public void playMp4(String fileName);
}
```

2. Next, create the concrete classes that implement that AdvancedMediaPlayer interface.

   VlcPlayer.java

```java
public class VlcPlayer implements AdvancedMediaPlayer{
   @Override
   public void playVlc(String fileName) {
      System.out.println("Playing vlc file. Name: "+ fileName);
   }

   @Override
   public void playMp4(String fileName) {

   }
}
```

Mp4Player.java

```
public class Mp4Player implements AdvancedMediaPlayer{

   @Override
   public void playVlc(String fileName) {

   }

   @Override
   public void playMp4(String fileName) {
      System.out.println("Playing mp4 file. Name: "+ fileName);
   }
}
```

3. The class MediaAdpter that implements the MediaPlayer interface is then created.

MediaAdapter.java

```
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();
      }
   }

   @Override
   public void play(String audioType, String fileName) {

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

4. Then, the concrete class AudioPlayer that implements the MediaPlayer interface is created.

AudioPlayer.java

```
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");
      }
   }
}
```

5. finally , the AudioPlayer is used to play different types of audio formats.

```
public class AdapterPatternDemo {
   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");
   }
}
```

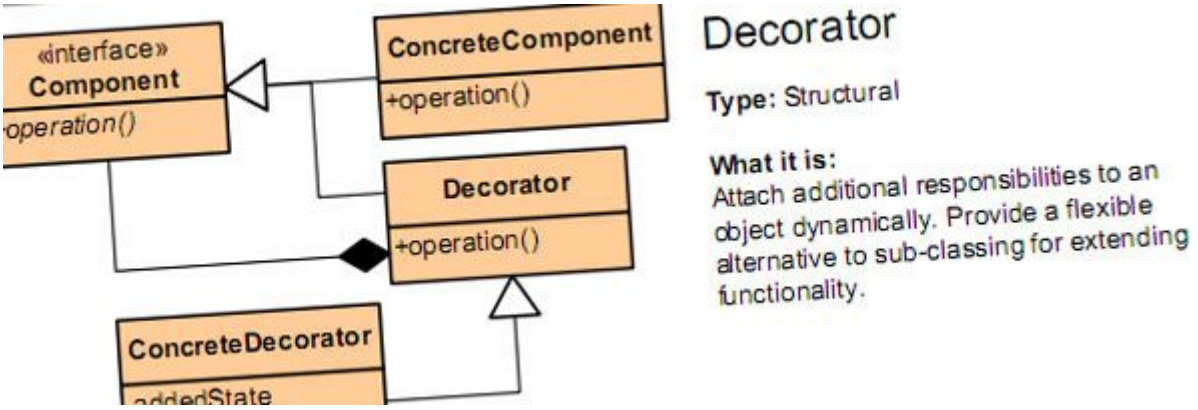
6. The output should be as follows:

```
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
```

```
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
```

Decorator Pattern

The Decorator pattern is a structural design pattern that allows you to add new functionality to an existing object without altering its structure. It creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.



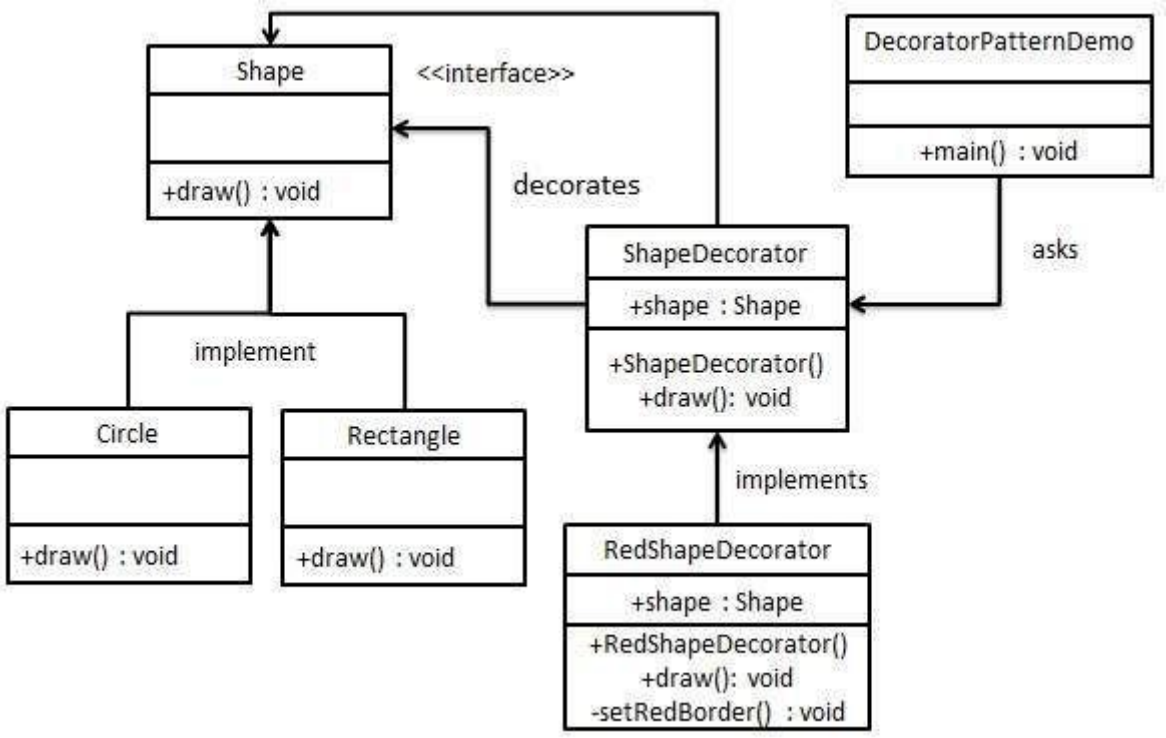
The Decorator Pattern (www.code.tutsplus.com)

When to Use the Decorator Pattern

The best place to use the decorator pattern is when you have an entity which needs to have new behavior only if the situation requires it.

Implementation of the Decorator Pattern

We will now demonstrate how the Decorator pattern is used in the following example where a shape will be decorated with some colour without altering shape class.



1. Firstly create a Shape interface.

   Shape.java

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

2. Then , create concrete classes which implement the Shape interface.

   Rectangle.java

```java
public class Rectangle implements Shape {

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

   Circle.java

```java
public class Circle implements Shape {

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

3. Then, create an abstract decorator class called ShapeDecorator which implements the Shape interface and has Shape object as its instance variable.

   ShapeDecorator.java

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

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

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

4. Now create RedShapeDecorator which is concrete class implementing ShapeDecorator.

   RedShapeDecorator.java

```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");
   }
}
```

5. Lastly, use the RedShapeDecorator class to decorate Shape objects.

DecoratorPatternDemo.java

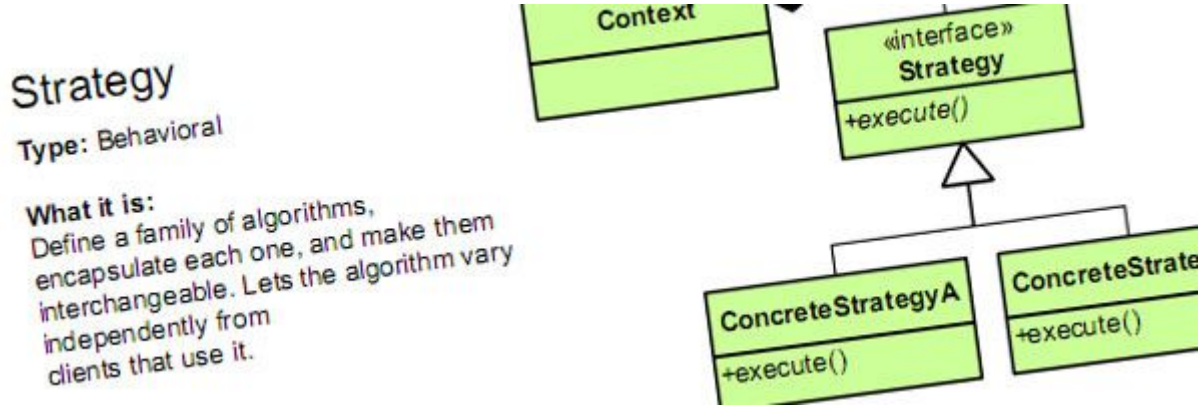
```
public class DecoratorPatternDemo {
   public static void main(String[] args) {

      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();
   }
}
```

6. The output should be as follows:

```
Circle with normal border
Shape: Circle

Circle of red border
Shape: Circle
Border Color: Red

Rectangle of red border
Shape: Rectangle
Border Color: Red
```

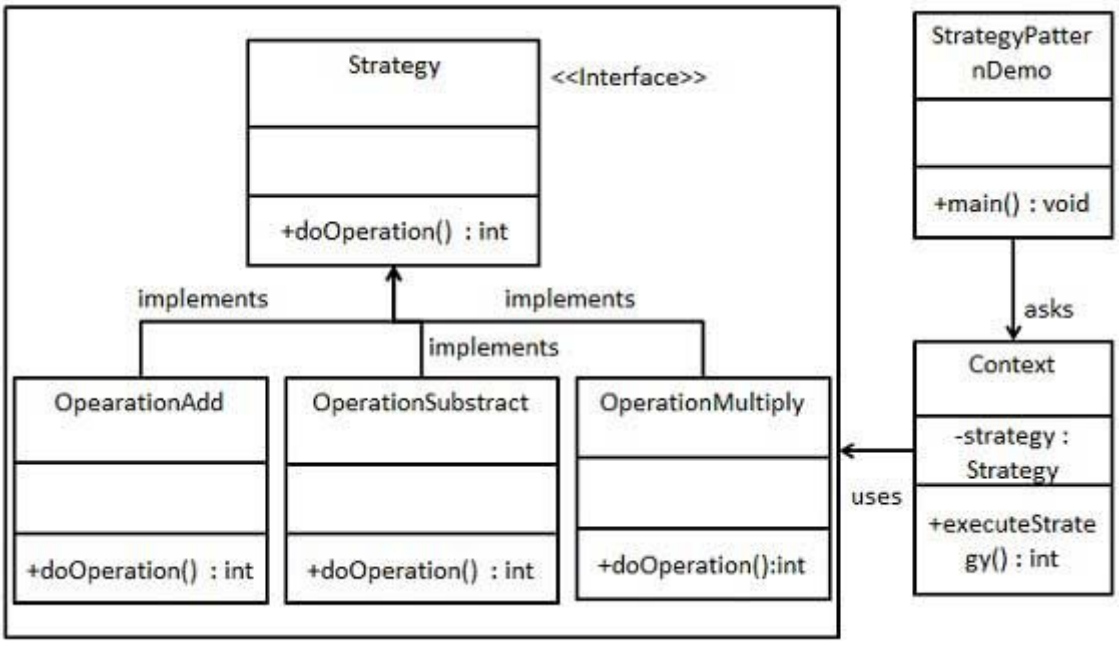
Strategy Pattern

The Strategy pattern is a behavioural design pattern in which a class behavior or its algorithm can be changed at runtime. With this 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.



The Strategy Pattern (www.code.tutsplus.com)

Implementation of the Strategy Pattern



To demonstrate how the Stragey pattern is implemetnted, we are going to create a Strategy interface which defines an action and concrete strategy classes which implement this interface. We then create a Context class which uses a Strategy and finally a demo class called, StrategyPatternDemo, which will use Context and strategy objects to demonstrate change in Context behaviour based on strategy it deploys or uses.

1. Firstly create an interface.

Strategy.java

```
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

2. Then, create concrete classes which implement the interface.

OperationAdd.java

```
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
```

OperationSubstract.java

```
public class OperationSubstract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}
```

OperationMultiply.java

```
public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

3. Now create the Context class.

Context.java

```
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);
```

```
    }
}
```

4. Finally create a demo class that uses the Context class to see change in behaviour when it changes its Strategy.

StrategyPatternDemo.java

```
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));
   }
}
```

5. The output should be as follows:

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
```

Compulsory Task

Follow these steps:

- Download the SingletonExercise.java file.
- Change the Deck class into a Singleton class.
- Change the main method to get the Deck instance rather than creating a new object.
- The program should print all 52 cards in random order.

SHARE YOUR THOUGHTS WITH US

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes. Think the content of this task, or this course as a whole, can be improved or think we've done a good job? Click here to share your thoughts anonymously.