Design Pattern Examples

Factory Pattern (Creational)

**Goal:** Define an interface for creating an object, but let the classes that implement the interface decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses.

**Plan:** Define a `Shape` interface, then implement concrete classes implementing the `Shape` interface. Then, define a `ShapeFactory` to create `Shape` instances of specific types.

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

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

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

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

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();
        }
    }
}
```
Abstract Factory Pattern (Creational)

**Goal:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

**Plan:** We will use the Shape classes previously defined. Define a Color interface and concrete classes that implement that interface. Then, create an abstract class AbstractFactory which will define generic factory methods and modify our ShapeFactory class, and define a ColorFactory class, that will both implement those methods from the AbstractFactory. Finally, define a FactoryProducer that will return an AbstractFactory object which will be used to create the Shape and Color instances.
public class Green implements Color {
    @Override
    public void fill() {
        System.out.println("Inside Green::fill() method.");
    }
}

public class Blue implements Color {
    @Override
    public void fill() {
        System.out.println("Inside Blue::fill() method.");
    }
}

public abstract class AbstractFactory {
    abstract Color getColor(String color);
    abstract Shape getShape(String shape);
}

public class ShapeFactory extends AbstractFactory {
    @Override
    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;
    }
    @Override
    Color getColor(String color) {
        return null;
    }
}

public class ColorFactory extends AbstractFactory {
    @Override
    public Shape getShape(String shapeType){
        return null;
    }
    @Override
    Color getColor(String color) {
        if(color == null){
            return null;
        }
        if(color.equalsIgnoreCase("RED")){
            return new Red();
        } else if(color.equalsIgnoreCase("GREEN")){
            return new Green();
        }
    }
}
```java
} else if (color.equalsIgnoreCase("BLUE")) {
    return new Blue();
} return null;
}

public class FactoryProducer {

    public static AbstractFactory getFactory(String choice) {
        if (choice.equalsIgnoreCase("SHAPE")) {
            return new ShapeFactory();
        } else if (choice.equalsIgnoreCase("COLOR")) {
            return new ColorFactory();
        }
        return null;
    }

    public static void main(String[] args) {
        // get shape factory
        AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");

        // get an object of Shape Circle
        Shape shape1 = shapeFactory.getShape("CIRCLE");

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

        // get an object of Shape Rectangle
        Shape shape2 = shapeFactory.getShape("RECTANGLE");

        // call draw method of Shape Rectangle
        shape2.draw();

        // get an object of Shape Square
        Shape shape3 = shapeFactory.getShape("SQUARE");

        // call draw method of Shape Square
        shape3.draw();

        // get color factory
        AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");

        // get an object of Color Red
        Color color1 = colorFactory.getColor("RED");

        // call fill method of Red
        color1.fill();

        // get an object of Color Green
        Color color2 = colorFactory.getColor("Green");
    }
```
Null-Object Design Pattern (Behavioral)

**Goal:** Instead of using a null reference to convey absence of an object (for instance, a non-existent customer), one uses an object which implements the expected interface, but whose method body is empty.

**Plan:** Define an AbstractAutoMaker class which will be used for automobile manufacturers. Implement two concrete classes that extend this abstract class – one named RealAutoMaker to be used when the auto maker exists, and one named NullAutoMaker for when the auto maker does not. Finally, create an AutoMakerFactory class that will make AbstractAutoMaker instances.

```java
public abstract class AbstractAutoMaker {
    protected String name;
    public abstract boolean isNil();
    public abstract String getName();
}

public class RealAutoMaker extends AbstractAutoMaker {
    public RealAutoMaker(String name) {
        this.name = name;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public boolean isNil() {
        return false;
    }
}

public class NullAutoMaker extends AbstractAutoMaker {
    @Override
    public String getName() {
        return "Not Available in AutoMaker Database";
    }

    @Override
    public boolean isNil() {
        return true;
    }
}
```
public class AutoMakerFactory {
    public static final String[] names = {"Honda", "Toyota", "Ford"};
    public static AbstractAutoMaker getAutoMaker(String name){
        for (int i = 0; i < names.length; i++) {
            if (names[i].equalsIgnoreCase(name)){
                return new RealAutoMaker(name);
            }
        }
        return new NullAutoMaker();
    }
}

public class NullPatternDemo {
    public static void main(String[] args) {
        AbstractAutoMaker autoMaker1 = AutoMakerFactory.getAutoMaker("Honda");
        AbstractAutoMaker autoMaker2 = AutoMakerFactory.getAutoMaker("Tesla");
        AbstractAutoMaker autoMaker3 = AutoMakerFactory.getAutoMaker("Ford");
        AbstractAutoMaker autoMaker4 = AutoMakerFactory.getAutoMaker("Ferrari");
        System.out.println("AutoMakers");
        System.out.println(autoMaker1.getName());
        System.out.println(autoMaker2.getName());
        System.out.println(autoMaker3.getName());
        System.out.println(autoMaker4.getName());
    }
}

**Adapter Pattern (structural)**

**Goal:** Allow classes to work together that normally could not because of incompatible interfaces, by providing its interface to clients while using the original interface.

**Plan:** Define two interfaces for playing media – a MediaPlayer interface intended for audio and an AdvancedMediaPlayer which can also play VLC and MP4 files. Implement concrete classes for the AdvancedMediaPlayer. Implement a concrete AudioPlayer which implements the MediaPlayer interface and also allow it to play MP4 and VLC files by having a MediaAdapter object that implements the MediaPlayer interface.

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

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

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) {
        // do nothing
    }

public class Mp4Player implements AdvancedMediaPlayer{
    @Override
    public void playVlc(String fileName) {
        // do nothing
    }
    @Override
    public void playMp4(String fileName) {
        System.out.println("Playing mp4 file. Name: "+ fileName);
    }
}

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);
        }
    }
}

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");
}
}

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");
    }
}

References: Design Patterns in Java Tutorial: Simply Easy Learning by tutorialspoint.com,
http://www.tutorialspoint.com/design_pattern/design_pattern_tutorial.pdf