UECS2344 Software Design: Practical 10

Design Patterns in Java
Adapted from:
http://www.journaldev.com/1827/java-design-patterns-example-tutorial
http://avajava.com/tutorials/categories/design-patterns
http://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
Java Essential Design Patterns (Bevis)

Diagrams from:
http://www.codeproject.com/Articles/430590/Design-Patterns-of-Creational-Design-Patterns

FOR EACH OF THE EXAMPLES GIVEN BELOW, DRAW A CLASS DIAGRAM.

1. Singleton Pattern

Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the Java virtual machine. The singleton class must provide a global access point to get the instance of the class. Singleton pattern is used for logging, driver objects, caching and thread pool.

Singleton design pattern is also used in other design patterns like Facade, etc.. Singleton design pattern is used in core Java classes also, for example java.lang.Runtime, java.awt.Desktop.

To implement Singleton pattern, we have different approaches but all of them have following common concepts.

- Private constructor to restrict instantiation of the class from other classes.
- Private static variable of the same class that is the only instance of the class.
- Public static method that returns the instance of the class - this is the global access point for other classes to get the instance of the singleton class.

Example

DatabaseConnection.java

```
public class DatabaseConnection {

    private static DatabaseConnection instance;

    private DatabaseConnection ()
    { // ignore details here }

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

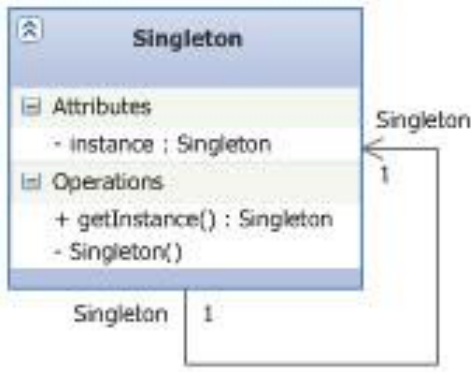
Here is the class to test the DatabaseConnection class.

DatabaseConnectionTest.java

```
public class DatabaseConnectionTest {
    public static void main(String[] args) {
        DatabaseConnection singleton;
        singleton = DatabaseConnection.getInstance();
        System.out.println(singleton.hashCode());

        singleton = DatabaseConnection.getInstance();
        System.out.println(singleton.hashCode());
     }
}
```

The output shows the same hash code, indicating that the same object is always returned by the getInstance() method.

General Class Diagram for Singleton Pattern



2. Adapter Pattern

Adapter design pattern is used so that two unrelated classes or interfaces can work together. The class that joins these unrelated classes or interfaces is called an **Adapter**.

Example 1

As a real life example, we can think of a mobile charger as an adapter because mobile battery needs 3 volts to charge but the normal socket produces either 120V. So the mobile charger works as an adapter between mobile charging socket and the wall socket.

So first of all we will have Socket class that produces volts of 120V.

Socket.java

```
public class Socket {
    public int getVolts(){
        return 120;
    }
}
```

Now we want to build an adapter for a mobile socket that can produce 3 volts. So first of all we will create an adapter interface.

ISocketAdapter.java

```
public interface ISocketAdapter {
    public int getVolts();
}
```

Here is the adapter implementation.

SocketAdapter.java

```
public class SocketAdapter implements ISocketAdapter{

    private Socket socket;

    public SocketAdapter() {
        this.socket = new Socket();
    }

    public int getVolts() {
        return socket.getVolts() / 40;
    }
}
```

Here is a test program to use our adapter implementation.

SocketAdapterTest.java

```
public class SocketAdapterTest {
    public static void main(String[] args) {
        Socket socket1 = new Socket();
        System.out.println("Volts using Socket = " + socket1.getVolts());

        ISocketAdapter socket2 = new SocketAdapter();
        System.out.println("Volts using Adapter = " + socket2.getVolts());
    }
}
```

Output:

```
Volts using Socket = 120
Volts using Adapter = 3
```

The SocketAdapter object uses the Socket object but adapts it to give 3 volts instead of 120 volts.

Example 2

Suppose we have a class named CelsiusReporter. It stores a temperature value in Celsius.

CelsiusReporter.java

```
public class CelsiusReporter {

	private double temperatureInC;

	public CelsiusReporter() {
	}


	public double getTemperature() {
		return temperatureInC;
	}

	public void setTemperature(double temperatureInC) {
		this.temperatureInC = temperatureInC;
```

```
	}
}
```

We want a class that handles both Celsius and Fahrenheit values. Here is the interface that has the required methods.

TemperatureInfo.java

```
public interface TemperatureInfo {

	public double getTemperatureInF();

	public void setTemperatureInF(double temperatureInF);

	public double getTemperatureInC();

	public void setTemperatureInC(double temperatureInC);
}
```

TemperatureAdapter is an adapter. It contains a reference to CelsiusReporter (the adaptee) and implements TemperatureInfo (the interface). If a temperature is in Celsius, TemperatureAdapter utilizes the temperatureInC value from CelsiusReporter. Fahrenheit requests are internally handled in Celsius.

TemperatureAdapter.java

```
public class TemperatureAdapter implements TemperatureInfo {

	private CelsiusReporter celsiusReporter;

	public TemperatureAdapter() {
		celsiusReporter = new CelsiusReporter();
	}

	@Override
	public double getTemperatureInC() {
		return celsiusReporter.getTemperature();
	}

	@Override
	public double getTemperatureInF() {
		return cToF(celsiusReporter.getTemperature());
	}

	@Override
	public void setTemperatureInC(double temperatureInC) {
		celsiusReporter.setTemperature(temperatureInC);
	}

	@Override
	public void setTemperatureInF(double temperatureInF) {
		celsiusReporter.setTemperature(fToC(temperatureInF));
	}

	private double fToC(double f) {
		return ((f - 32) * 5 / 9);
	}

	private double cToF(double c) {
		return ((c * 9 / 5) + 32);
```

```
        }
}
```

The TemperatureAdapterTest class is a client class (that is, it uses the TemperatureAdapter class). First, it creates a TemperatureAdapter object and then demonstrates calls to the adapter.

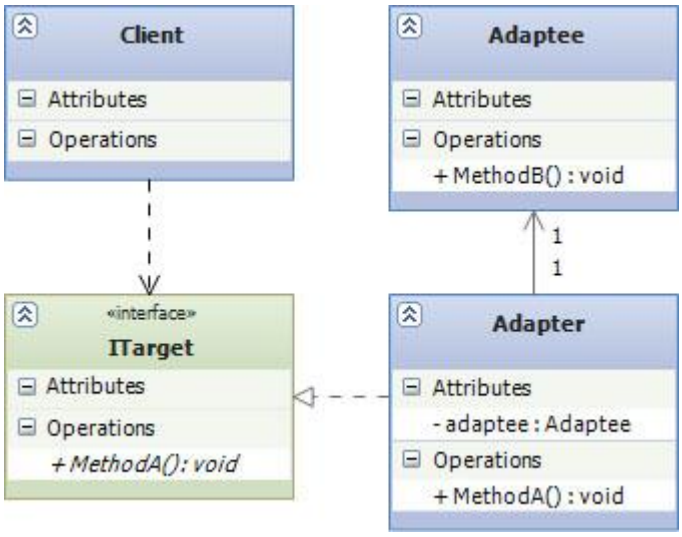
TemperatureAdapterTest.java

```
public class TemperatureAdapterTest {

   public static void main(String[] args) {

      TemperatureInfo tempInfo = new TemperatureAdapter();

      tempInfo.setTemperatureInC(0);
      System.out.println("temp in C:" + tempInfo.getTemperatureInC());
      System.out.println("temp in F:" + tempInfo.getTemperatureInF());


      tempInfo.setTemperatureInF(85);
      System.out.println("temp in C:" + tempInfo.getTemperatureInC());
      System.out.println("temp in F:" + tempInfo.getTemperatureInF());
   }
}
```

Output:

```
temp in C:0.0
temp in F:32.0
temp in C:29.444444444444443
temp in F:85.0
```

General Class Diagram for Adapter Pattern



3. Composite Pattern

Composite pattern is used when we have to represent a structure that consists of a mixture of individual elements and composite elements, and we want to perform similar operations on the individual as well as the composite elements. Composite elements are made up of a combination of the individual elements.

Composite Pattern consists of following objects.

1. ***Leaf*** – It defines the behaviour for the individual elements in the structure. It is the building block for the structure. It does not have references to other elements.
2. ***Composite*** – It consists of a combination of leaf elements.
3. ***Base Component*** – It is the interface or abstract class for all objects in the structure. The client class uses base component to work with the objects in the structure. It contains some methods common to both the leaf and composite objects. In other words, the leaf and composite objects implement the operations in base component.

Example 1

A Drawing is a structure that consists of Objects such as Circle, Rectangle, Triangle, etc. When we fill the Drawing with Colour (say Red), the same Colour also gets applied to all the Objects in the Drawing. So a Drawing is made up of different parts and all the parts have the same operations.

Base Component

We have an interface Shape with a method draw(String fillColour) to draw the shape with the given fill colour.

Shape.java

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

Leaf

The *Leaf* class implements the *Base Component*. *Leaf* objects are the building block for the *Composite*. There can be multiple types of leaf objects such as Triangle, Circle etc.

Triangle.java

```
public class Triangle implements Shape {
    @Override
    public void draw(String fillColor) {
        System.out.println("Drawing Triangle with color "+fillColor);
    }
}
```

Circle.java

```
public class Circle implements Shape {
    @Override
    public void draw(String fillColor) {
        System.out.println("Drawing Circle with color "+fillColor);
    }
}
```

Composite

A *Composite* object contains a group of *Leaf* objects. It contains some methods to add or delete *Leaf* objects from the group. It may also have a method to remove all the elements from the group. The *Composite* in this example is the Drawing.

Drawing.java

```
import java.util.ArrayList;
```

```
import java.util.List;

public class Drawing implements Shape{

    //collection of Shapes
    private List<Shape> shapes = new ArrayList<Shape>();

    @Override
    public void draw(String fillColour) {
        for(Shape sh : shapes) {
            sh.draw(fillColour);
        }
    }

    //adding shape to drawing
    public void add(Shape s){
        this.shapes.add(s);
    }

    //removing shape from drawing
    public void remove(Shape s){
        shapes.remove(s);
    }

    //removing all the shapes
    public void clear(){
        System.out.println("Clearing all the shapes from drawing");
        this.shapes.clear();
    }
}
```

Notice that *Composite* also implements *Base Component* and behaves similar to *Leaf* except that it is a group of *Leaf* elements.

Our Composite Pattern implementation is ready and we can test it.

DrawingTest.java

```
public class DrawingTest {

    public static void main(String[] args) {

        Shape tri1 = new Triangle();
        tri1.draw("Yellow");

        Shape cir1 = new Circle();
        cir1.draw("Blue");

        Drawing drawing = new Drawing();
        drawing.add(tri1);
        drawing.add(cir1);

        drawing.draw("Red");

        drawing.clear();

        Shape tri2 = new Triangle();

        drawing.add(tri1);
```

```
        drawing.add(tri2);
        drawing.add(cir);

        drawing.draw("Green");
    }
}
```

Output:

```
Drawing Triangle with color Yellow
Drawing Circle with color Blue
Drawing Triangle with color Red
Drawing Circle with color Red
Clearing all the shapes from drawing
Drawing Triangle with color Green
Drawing Triangle with color Green
Drawing Circle with color Green
```

Important Points about Composite Pattern

- Composite pattern should be applied only when the group of objects has an operation that is also applicable to the individual objects. For the example above, draw() method can be called on the individual (Triangle or Circle) objects as well as the composite (Drawing) objects.
- Composite pattern can be used to create a tree like structure.

Example 2

In the Foobar Motor Company workshop, they build various items from component parts such as nuts, bolts, panels, etc. Each individual component item has an associated description and unit cost, and when items are assembled into larger items, the cost is therefore the sum of its component parts.

The Composite Pattern enables us to treat both individual parts and assemblies (groups) of parts as if they are the same, thus enabling them to be processed in a consistent manner, simplifying the code.

An abstract Item class defines common methods for both parts and assemblies of parts:

Item.java

```
public abstract class Item {

    private String description;

    protected int cost;

    public Item(String description, int cost) {
       this.description = description;
       this.cost = cost;
    }

    public String getDescription() {
       return description;
    }

    public String toString() {
        return description + " (cost " + getCost() + ")";
    }
```

```
    public abstract int getCost();

    public abstract void addItem(Item item);

    public abstract void removeItem(Item item);

}
```

Individual parts are modelled using a Part subclass:

Part.java

```
public class Part extends Item {

    public Part(String description, int cost) {
        super(description, cost);
    }

    public int getCost() {
       return cost;
    }

    public void addItem(Item item) {
       // empty body
    }

    public void removeItem(Item item) {
       // empty body
    }
}
```

Assemblies of parts are modelled using an Assembly subclass:

Assembly.java

```
import java.util.List;
import java.util.ArrayList;

public class Assembly extends Item {

    private List<Item> items;
    public Assembly(String description) {
       super(description, 0);
       items = new ArrayList<Item>();
    }

    // Add cost of all items in list
    public int getCost() {
       int total = 0;

       for (Item item : items) {
           total += item.getCost();
       }
       return total;
    }

    public void addItem(Item item) {
       items.add(item);
```

```
    }

    public void removeItem(Item item) {
       items.remove(item);
    }

}
```

All types of Item objects can now be used in a uniform manner:

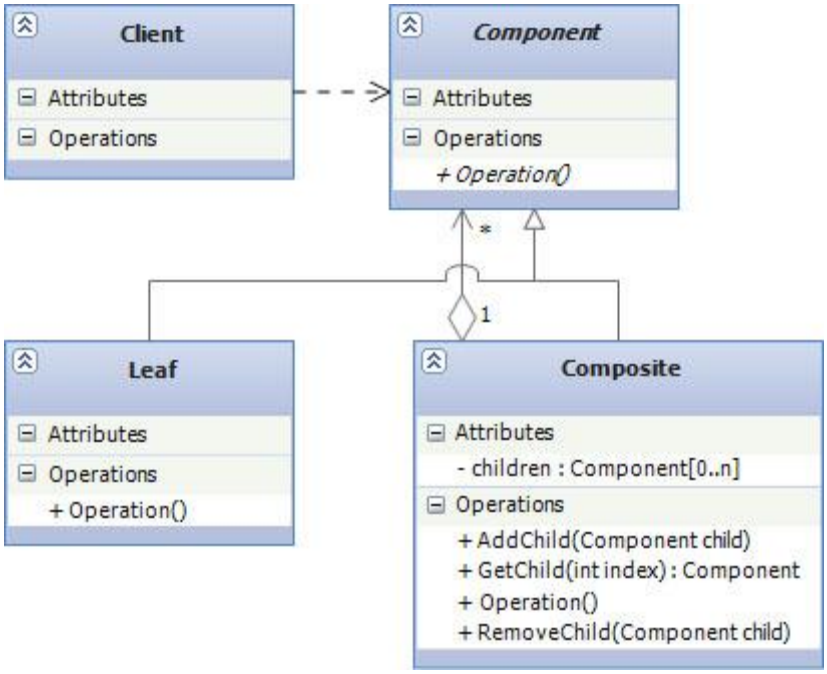
ItemTest.java

```
import java.util.List;
import java.util.ArrayList;

public class ItemTest {

    public static void main(String[] args) {

        Item nut1 = new Part("Nut", 5);
        Item bolt1 = new Part("Bolt", 9);

        Item nut2 = new Part("Nut", 10);
        Item bolt2 = new Part("Bolt", 16);

        Item composite = new Assembly("Composite");
        composite.addItem(nut2);
        composite.addItem(bolt2);;

        List<Item> items = new ArrayList<Item>();
        items.add(nut1);
        items.add(bolt1);
        items.add(composite);

        for (Item anItem : items) {
             System.out.println(anItem);
        }
    }
}
```

Output:

```
Nut (cost 5)
Bolt (cost 9)
Composite (cost 26)
```

General Class Diagram for Composite Pattern



4. Facade Pattern

Facade Pattern is used to help a client class to easily interact with a set of classes. Instead of interacting directly with all the classes in the set, Facade pattern hides the complexities of the set of classes and provides a single class with simplified methods required by the client and delegates calls to the methods of the set of classes.

Example

Suppose we have an application with a subsystem containing a set of classes to use MySql or Oracle database and to generate different types of reports, such as HTML report, PDF report etc. So we will have different set of classes to work with different types of database.

Now a client application can use these classes to get the required database connection and generate reports. But when the complexity increases, the client class will become more complex. So we can apply Facade pattern here and provide an easier way for the client class to use the set of classes.

Set of Classes

We can have two classes, namely MySqlHelper and OracleHelper.

MySqlHelper.java

```
import java.sql.Connection;

public class MySqlHelper {

    public static Connection getMySqlDBConnection(){
        //get MySql DB connection using connection parameters
        return null; // for test purposes
    }

    public void generateMySqlPDFReport(String tableName, Connection con){
```

```
        //get data from table and generate pdf report
    }

    public void generateMySqlHTMLReport(String tableName, Connection con){
        //get data from table and generate html report
    }
}
```

OracleHelper.java

```
import java.sql.Connection;

public class OracleHelper {

    public static Connection getOracleDBConnection(){
        //get Oracle DB connection using connection parameters
        return null; // for test purposes
    }

    public void generateOraclePDFReport(String tableName, Connection con){
        //get data from table and generate pdf report
    }

    public void generateOracleHTMLReport(String tableName, Connection con){
        //get data from table and generate html report
    }
}
```

Facade Class

We can create a Facade class like below.

HelperFacade.java

```
import java.sql.Connection;

public class HelperFacade {

    public static void generateReport(DBTypes dbType,
                                      ReportTypes reportType,
                                      String tableName){
        Connection con = null;

        switch (dbType){

        case MYSQL:
            con = MySqlHelper.getMySqlDBConnection();
            MySqlHelper mySqlHelper = new MySqlHelper();

            switch(reportType){
            case HTML:
                mySqlHelper.generateMySqlHTMLReport(tableName, con);
                break;
            case PDF:
                mySqlHelper.generateMySqlPDFReport(tableName, con);
                break;
            }
            break;
        case ORACLE:
            con = OracleHelper.getOracleDBConnection();
```

```
            OracleHelper oracleHelper = new OracleHelper();

            switch(reportType){
            case HTML:
                oracleHelper.generateOracleHTMLReport(tableName, con);
                break;
            case PDF:
                oracleHelper.generateOraclePDFReport(tableName, con);
                break;
            }
            break;
        }
    }

    public static enum DBTypes{
        MYSQL,ORACLE;
    }

    public static enum ReportTypes{
        HTML,PDF;
    }
}
```

Here is a test program without using Facade and using Facade class.

HelperFacadeTest.java

```
import java.sql.Connection;

public class HelperFacadeTest {

    public static void main(String[] args) {
        String tableName="Employee";

        //generating MySql HTML report and Oracle PDF report
        //  without using Facade
        Connection con = MySqlHelper.getMySqlDBConnection();
        MySqlHelper mySqlHelper = new MySqlHelper();
        mySqlHelper.generateMySqlHTMLReport(tableName, con);

        Connection con1 = OracleHelper.getOracleDBConnection();
        OracleHelper oracleHelper = new OracleHelper();
        oracleHelper.generateOraclePDFReport(tableName, con1);

        //generating MySql HTML report and Oracle PDF report
        //  using Facade
        HelperFacade.generateReport(HelperFacade.DBTypes.MYSQL,
                                    HelperFacade.ReportTypes.HTML, tableName);
        HelperFacade.generateReport(HelperFacade.DBTypes.ORACLE,
                                    HelperFacade.ReportTypes.PDF, tableName);
    }
}
```

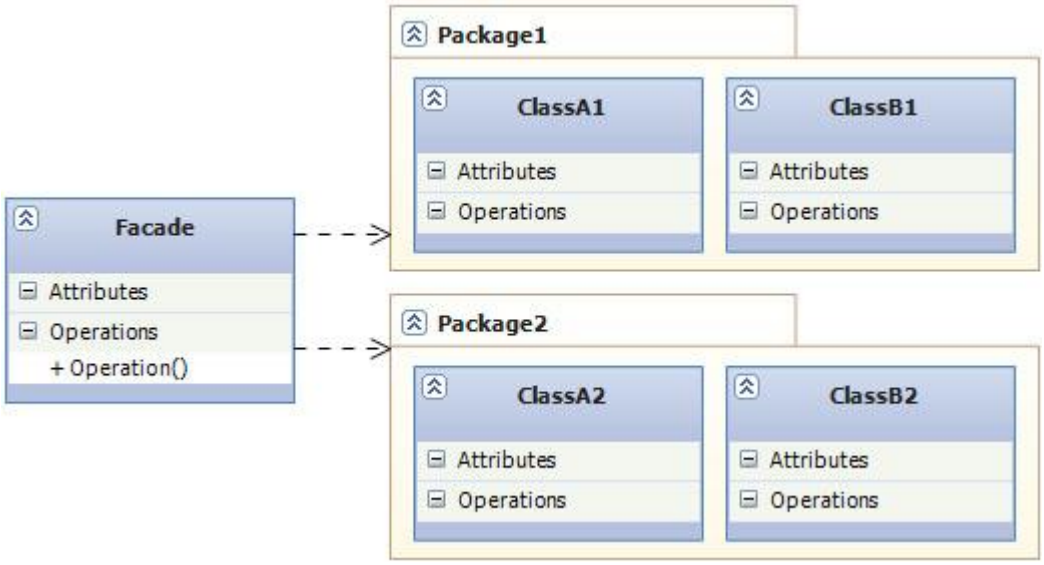
Using Facade class is an easier and cleaner way and avoids having a lot of logic in client class.

Important Points

- Facade pattern should be applied for similar kind of classes - its purpose is to provide a single subsystem interface to these classes.

- Subsystem classes are not aware of Facade and they should not have any reference of the Facade class.

General Class Diagram for Facade Pattern



5. Template Method Pattern

Template Method is used to create a method which defers some of the steps of its implementation to the subclasses. This method defines the steps to execute an algorithm and leaves it to the subclasses to provide the detailed implementation of the steps of the algorithm.

Example

Suppose we want to provide an algorithm to build a house. The steps needed to be performed to build a house are – build foundation, build pillars, build walls, and install windows. The important point is that we cannot change the order of execution because we cannot install windows before building the foundation. So in this case we can create a template method that will provide the overall algorithm with different methods for the steps to build the house.

Now building the foundation for a house is same for all types of house, whether it is a wooden house or a glass house. Similarly, installing windows is the same for all types of house. So we can provide an implementation for these two steps. For the other steps, the details are different for different types of houses.

Template Method Abstract Class

We provide a HouseTemplate class that has a template method for the overall steps for building a house. The detail for some steps is provided in the subclasses of the HouseTemplate class. We make the HouseTemplate class an abstract class since we want the subclasses to provide the detail for the steps. To make sure that subclasses don't override the template method, we make it final.

HouseTemplate.java

```
public abstract class HouseTemplate {
```

```
    //template method, final so subclasses can't override
    public final void buildHouse(){
        buildFoundation();
        buildPillars();
        buildWalls();
        installWindows();
        System.out.println("House is built.");
    }

    //same implementation for all houses
    private void buildFoundation() {
        System.out.println("Building Foundation");
    }

    //methods to be implemented by subclasses
    public abstract void buildWalls();

    public abstract void buildPillars();

    //same implementation for all houses
    private void installWindows() {
        System.out.println("Installing Windows");
    }
}
```

buildHouse() is the template method and defines the order of execution for performing several steps.

Template Method Concrete Classes

We can have different type of houses, such as Wooden House and Glass House.

WoodenHouse.java

```
public class WoodenHouse extends HouseTemplate {

    @Override
    public void buildWalls() {
        System.out.println("Building Wooden Walls");
    }

    @Override
    public void buildPillars() {
        System.out.println("Building Pillars with Wood coating");
    }
}
```

GlassHouse.java

```
public class GlassHouse extends HouseTemplate {

    @Override
    public void buildWalls() {
        System.out.println("Building Glass Walls");
    }

    @Override
    public void buildPillars() {
        System.out.println("Building Pillars with glass coating");
```

```
    }
}
```

Here is a test program.

TemplatePatternTest.java

```
public class TemplateMethodPatternTest {

     public static void main(String[] args) {

        HouseTemplate houseType;

        houseType = new WoodenHouse();
        houseType.buildHouse();
        System.out.println("************");

        houseType = new GlassHouse();
        houseType.buildHouse();
    }
}
```

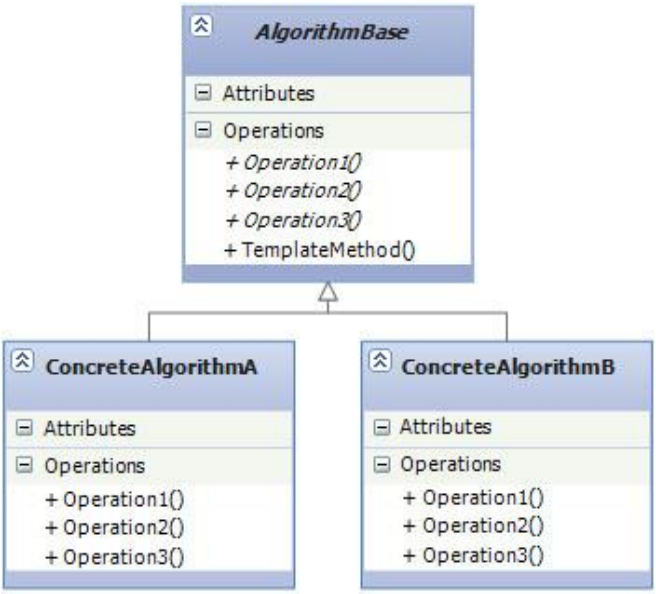
Output:

```
Building Foundation
Building Pillars with Wood coating
Building Wooden Walls
Installing Windows
House is built.
************
Building Foundation
Building Pillars with glass coating
Building Glass Walls
Installing Windows
House is built.
```

Important Points

- Template method should consist of certain steps whose order is fixed and for some of the methods, the implementation is different for the different subclasses. Template method should be final.
- Most of the times, subclasses calls methods from super class but in template pattern, superclass template method calls methods from subclasses, this is known as Hollywood Principle – "don't call us, we'll call you."
- Methods in base class without implementation are referred as **Hooks** and they are intended to be overridden by subclasses. If you don't want some of the methods to be overridden, you can make them final, for example, you can make buildFoundation() method final so that the subclasses do not override it.

General Class Diagram for Template Method Pattern



6. Strategy Pattern

Strategy pattern is used when we have multiple algorithms for a specific task and the client class decides the actual implementation to be used at runtime. The selected implementation can also be changed at run time.

We can define multiple algorithms and let client class pass the algorithm to be used as a parameter.

Example

We have a Payment class where we can use two payment algorithms (or strategies) – by Credit Card or by PayPal. First of all we will create the interface for our strategy with a method to pay the amount.

PaymentStrategy.java

```
public interface PaymentStrategy {
     public void pay(double amount);
}
```

Now we will have to create concrete implementations of algorithms for payment, one using credit/debit card and one through paypal.

CreditCardStrategy.java

```
public class CreditCardStrategy implements PaymentStrategy {

    private String name;
    private String cardNumber;
    private String cvv;
    private String dateOfExpiry;

    public CreditCardStrategy(String nm, String ccNum, String cvv,
                              String expiryDate){
        this.name=nm;
        this.cardNumber=ccNum;
        this.cvv=cvv;
        this.dateOfExpiry=expiryDate;
    }
```

```
    @Override
    public void pay(double amount) {
        System.out.println(amount +" paid with credit/debit card.");
    }
}
```

PaypalStrategy.java

```
public class PaypalStrategy implements PaymentStrategy {

    private String emailId;
    private String password;

    public PaypalStrategy(String email, String pwd){
        this.emailId=email;
        this.password=pwd;
    }

    @Override
    public void pay(double amount) {
        System.out.println(amount + " paid using Paypal.");
    }
}
```

Now our algorithms are ready. The Payment class requires a payment method based on the selected payment strategy.

Payment.java

```
public class Payment {

    private double amount;

    public Payment(double amount) {
        this.amount = amount;
    }

    public void pay(PaymentStrategy paymentMethod){
        paymentMethod.pay(amount);
    }
}
```

Notice that payment method requires the PaymentStrategy object as an argument. It does not store it anywhere as instance variable.

Next is the PaymentTest program.

PaymentTest.java

```
public class PaymentTest {

    public static void main(String[] args) {

       Payment payment = new Payment(100.0);

        //pay by paypal
        PaypalStrategy paymentStrategy1 =
                    new PaypalStrategy("myemail@example.com", "mypwd");
```

```
        payment.pay(paymentStrategy1);

        //pay by credit card
        CreditCardStrategy paymentStrategy2 =
                    new CreditCardStrategy("James Smith", "1234567890123456",
                                           "786", "12/20");
        payment.pay(paymentStrategy2);
    }
}
```

Output:

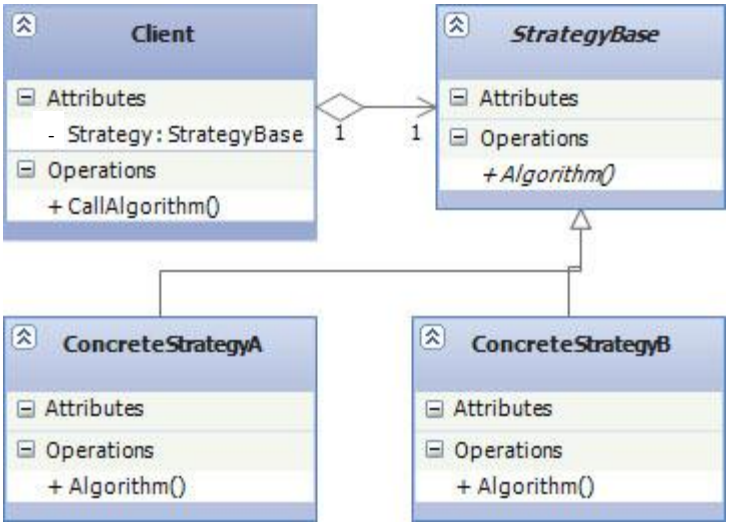
```
100.0 paid using Paypal.
100.0  paid with credit/debit card.
```

Note we did not use an instance variable for the PaymentStrategy as we want to choose the specific strategy to be applied at runtime . So it is passed as an argument.

Important Points

- Strategy pattern is useful when we have multiple algorithms for specific task and we want our application to be flexible to choose any one of the algorithms at runtime.

General Class Diagram for Strategy Pattern



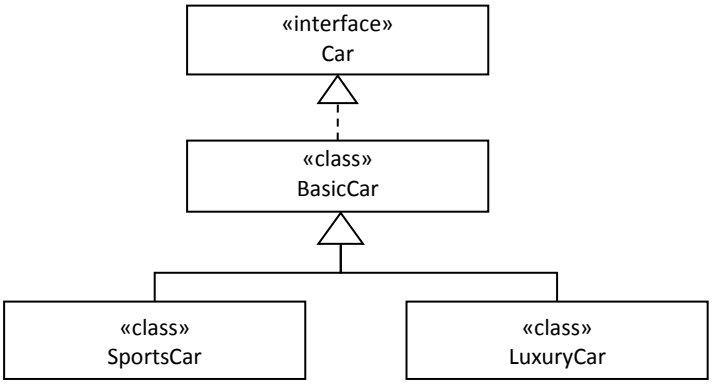
7. Decorator Pattern

Decorator design pattern is used to modify the functionality of a particular object at runtime. The other objects of the same class will not be affected by this. Only that particular object gets the modified behaviour.

We cannot use inheritance for this situation because when inheritance is used to extend the behaviour of a class in a subclass, it is applicable to all instances of the sub class.

Example 1

Suppose we want to implement different kinds of cars – we can create interface Car to define the assemble method for a Basic car. Then we can extend it to SportsCar and LuxuryCar. The interface and classes are shown in the diagram below.



We cannot add any new functionality or remove any existing behaviour at runtime. For example, we cannot get a car at runtime that has both the features of sports car and luxury car.

To solve this kind of programming situation, we apply decorator pattern.

We need to have following types to implement decorator design pattern.

Component Base – it is an interface or abstract class defining the methods that will be implemented. For the above situation, Car will be the *Component Base*.

Car.java

```
public interface Car {
    public void assemble();
}
```

Concrete Component – it is the basic implementation of the *Component Base*. We can have BasicCar class as the *Concrete Component*.

BasicCar.java

```
public class BasicCar implements Car {

    @Override
    public void assemble() {
        System.out.print("Basic Car.");
    }
}
```

Decorator – it is the class which implements the *Component Base* and also has a HAS-A relationship with the *Component Base*. The component variable should be accessible to the child decorator classes, so we will make this variable protected.

CarDecorator.java

```
public class CarDecorator implements Car {

    protected Car car;

    public CarDecorator(Car c){
        this.car=c;
    }

    @Override
    public void assemble() {
        this.car.assemble();
    }
}
```

Concrete Decorators – they extend the base decorator functionality and modify the component behavior accordingly. We can have concrete decorator classes as SportsCar and LuxuryCar.

SportsCar.java

```
public class SportsCar extends CarDecorator {

    public SportsCar(Car c) {
        super(c);
    }

    @Override
    public void assemble(){
        car.assemble();
        System.out.print(" Adding features of Sports Car.");
    }
}
```

LuxuryCar.java

```
public class LuxuryCar extends CarDecorator {

    public LuxuryCar(Car c) {
        super(c);
    }

    @Override
    public void assemble(){
        car.assemble();
        System.out.print(" Adding features of Luxury Car.");
    }
}
```

Here is the class to test the Car Decorator.

CarDecoratorTest.java

```
public class CarDecoratorTest {

    public static void main(String[] args) {

        Car sportsCar = new SportsCar(new BasicCar());
        sportsCar.assemble();
        System.out.println("\n*****");
```

```
        Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar()));
        sportsLuxuryCar.assemble();
    }
}
```

Output:

```
Basic Car. Adding features of Sports Car.
*****
Basic Car. Adding features of Luxury Car. Adding features of Sports Car.
```

Notice that test program can create different kinds of Object at runtime and they can specify the order of execution too.

Important Points

- Decorator pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature the same.
- Decorator pattern allows a user to add new functionality to an existing object without altering its structure.
- Decorator pattern is helpful in providing runtime modification abilities, making the program more flexible. It is also easy to extend when the number of choices (decorators) are more.
- Decorator pattern is used a lot in Java IO classes, such as FileReader, BufferedReader etc.

Example 2

We will decorate a Shape object with some colour without altering the Shape class.

Shape.java

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

These are the concrete classes implementing Shape 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");
   }
}
```

We create an abstract decorator class ShapeDecorator implementing the Shape interface and having Shape object as its instance variable.

ShapeDecorator.java

```
public abstract class ShapeDecorator implements Shape {

   protected Shape decoratedShape;

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

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

RedShapeDecorator is concrete class implementing ShapeDecorator.

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");
   }
}
```

This is the program to test RedShapeDecorator.

RedShapeDecoratorTest.java

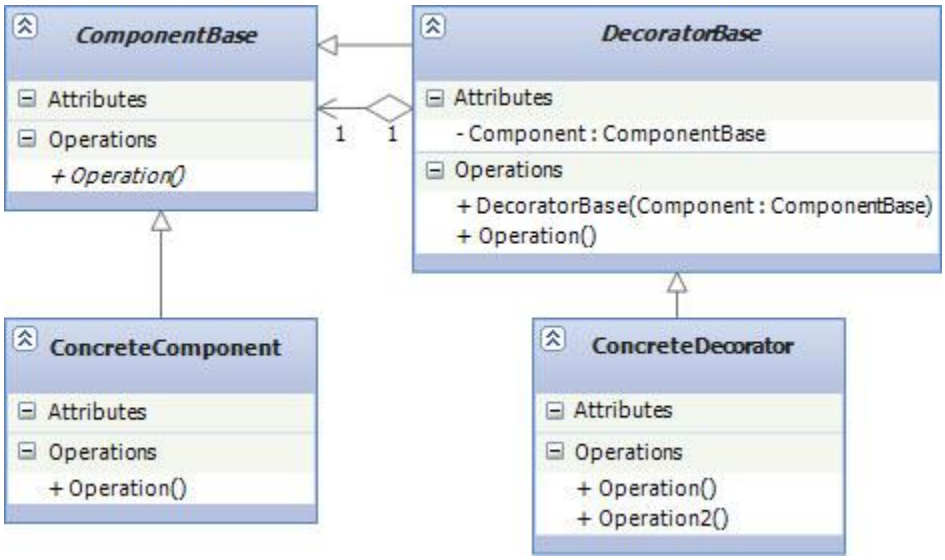
```
public class RedShapeDecoratorTest {
   public static void main(String[] args) {

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

      Shape redCircle = new RedShapeDecorator(new Circle());
      System.out.println("\nCircle of red border");
      redCircle.draw();

      Shape redRectangle = new RedShapeDecorator(new Rectangle());
      System.out.println("\nRectangle of red border");
      redRectangle.draw();
   }
}
```

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
```

General Class Diagram for Decorator Pattern



8. Observer Pattern

Observer pattern is useful when you have classes that are interested in the state of an object of another class and they want to get notified whenever there is any change to that object's state.

In observer pattern, the object that watches on the state of another object is called **Observer** and the object that is being watched is called **Subject**.

Subject contains a list of observers to notify of any change in its state, so it should provide methods which observers can use to register and unregister themselves. Subject also contains a method to notify all the observers of any change and it can provide another method to return its state.

Observer should have a method to set the object to watch and another method that will be used by Subject to notify them of any updates.

Example 1

Subject.java

```
import java.util.ArrayList;
import java.util.List;
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();
      }
   }
}
```

Observer.java

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

We create concrete Observer classes.

BinaryObserver.java

```
public class BinaryObserver extends Observer{

   public BinaryObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

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

OctalObserver.java

```
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() ) );
   }
}
```

HexaObserver.java

```
public class HexaObserver extends Observer{

   public HexaObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

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

Here is the test program.

ObserverPatternTest.java

```
public class ObserverPatternTest {

   public static void main(String[] args) {

      Subject subject = new Subject();

      HexaObserver hexa = new HexaObserver(subject);
      OctalObserver octal = new OctalObserver(subject);
      BinaryObserver binary = new BinaryObserver(subject);

      System.out.println("First state change: 15");
      subject.setState(15);

      System.out.println("Second state change: 10");
      subject.setState(10);
   }
}
```

Output:

```
First state change: 15
Hexa String: F
Octal String: 17
Binary String: 1111
Second state change: 10
Hexa String: A
Octal String: 12
Binary String: 1010
```

Example 2

We will implement a simple forum with a topic (the subject) and observers can register to this topic. Whenever any new message is posted to the topic, all the registered observers will be notified.

The Subject interface defines the methods to be implemented by any concrete subject.

Subject.java

```
public interface Subject {

    //methods to register and unregister observers
    public void register(Observer obj);
    public void unregister(Observer obj);

    //method to notify observers of change
    public void notifyObservers();

    //method to get state of subject
    public Object getState(Observer obj);

}
```

Next here is Observer interface.

Observer.java

```
public interface Observer {

    //attach with subject to observe
    public void setSubject(Subject sub);

    //method to update the observer, used by subject
    public void update();
}
```

Here is a concrete implementations of the Subject.

MyTopic.java

```
import java.util.ArrayList;
import java.util.List;

public class MyTopic implements Subject {

    private List<Observer> observers;
    private String message;

    public MyTopic(){
        this.observers=new ArrayList<>();
    }

    @Override
    public void register(Observer obj) {
        if(obj == null) throw new NullPointerException("Null Observer");
        if(!observers.contains(obj))
              observers.add(obj);
    }
```

```
    @Override
    public void unregister(Observer obj) {
        observers.remove(obj);
    }

    @Override
    public void notifyObservers() {
        for (Observer obj : observers) {
            obj.update();
        }
    }

    @Override
    public Object getState(Observer obj) {
        return this.message;
    }

    //method to post message to the topic
    public void postMessage(String msg){
        System.out.println("Message Posted: "+msg);
        this.message=msg;
        notifyObservers();
    }
}
```

Here is the implementation of Observers that will watch over the subject.

MyTopicObserver.java

```
public class MyTopicObserver implements Observer {

    private String name;
    private Subject topic;

    public MyTopicObserver(String nm){
        this.name=nm;
    }
    @Override
    public void update() {
        String msg = (String) topic.getState(this);
        System.out.println(name+":: Opening message::"+msg);
    }

    @Override
    public void setSubject(Subject sub) {
        this.topic=sub;
    }
}
```

Here is a test program for the topic implementation.

ObserverPatternTest.java

```
public class MyTopicObserverTest {

    public static void main(String[] args) {

        //create subject
        MyTopic topic = new MyTopic();
```

```
        //create observers
        Observer obj1 = new MyTopicObserver("Obj1");
        Observer obj2 = new MyTopicObserver("Obj2");
        Observer obj3 = new MyTopicObserver("Obj3");

        //register observers to the subject
        topic.register(obj1);
        topic.register(obj2);
        topic.register(obj3);

        //attach observer to subject
        obj1.setSubject(topic);
        obj2.setSubject(topic);
        obj3.setSubject(topic);

        //now send message to subject
        topic.postMessage("First Message");
        System.out.println();
        topic.postMessage("Second Message");
    }
}
```

Output:

```
Message Posted: First Message
Obj1:: Opening message::First Message
Obj2:: Opening message::First Message
Obj3:: Opening message::First Message

Message Posted: Second Message
Obj1:: Opening message::Second Message
Obj2:: Opening message::Second Message
Obj3:: Opening message::Second Message
```

Observer pattern is also called as publish-subscribe pattern.

General Class Diagram for Observer Pattern