Design Patterns

Meredith Brown

SINGLETON

- creational
- a single class is used to create a single instance of an object
- the class also provides a means to access the object via a static method

```
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;
   }
```

tutorialspoint.com

BUILDER

- creational
- used to build complex objects
- the builder class contains all the fields of the object class
- the object class contains a private constructor so so that objects can only be created by the builder

```
BankAccount account = new BankAccount.Builder(1234L)
            .withOwner("Marge")
            .atBranch("Springfield")
            .openingBalance(100)
            .atRate(2.5)
            .build();

BankAccount anotherAccount = new BankAccount.Builder(4567L)
            .withOwner("Homer")
            .atBranch("Springfield")
            .openingBalance(100)
```

source: dzone.com

FACTORY

- creational
- a factory class is used to create objects of one or several different classes

```
public class AnimalFactory {
    public static Dog createDog(String name, Date birthDate) {
        Integer newId = DogHouse.getNumberOfDogs();
        return new Dog(name, birthDate, newId);
    }

    public static Cat createCat(String name, Date birthDate) {
        Integer newId = CatHouse.getNumberOfCats();
        return new Cat(name, birthDate, newId);
    }
}
```

source: Leon

DECORATOR

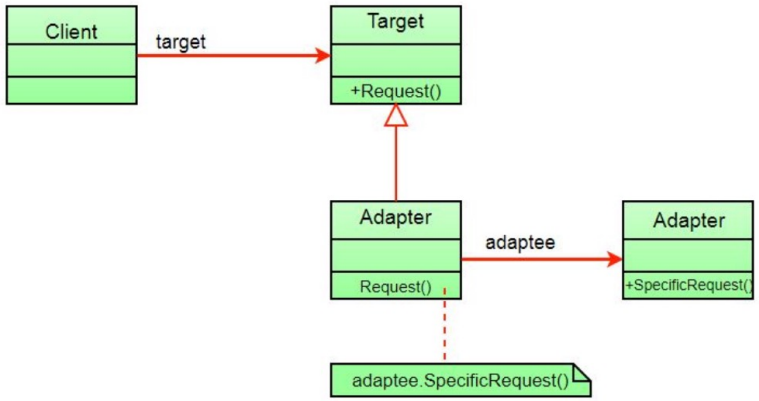
- structural
- gives an existing object new methods/functionality
- a decorator class acts as a wrapper to an existing class

```
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();
   }
}
```

source: tutorialspoint.com

ADAPTER

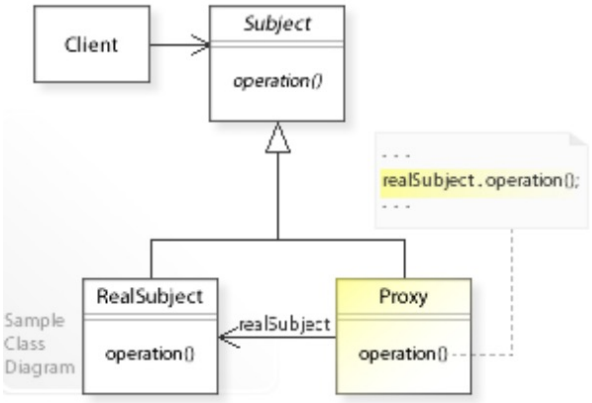
- structural
- connects two independent interfaces by wrapping an existing class with a new interface
- used when a class (client) requires a certain type of object and the object to be used (adaptee) implements an incompatible interface



source: geeksforgeeks.org

PROXY

- structural
- a class represents another class
- the proxy object controls access to the original object
- allows the user to do work before and after the original object is affected



source: howtodoinjava.com

FACADE

- structural
- a facade encapsulates complex code behind a simple interface

```
public void startEngine() {
    fuelInjector.on();
    airFlowController.takeAir();
    fuelInjector.on();
    fuelInjector.inject();
    starter.start();
    coolingController.setTemperatureUpperLimit(DEFAULT_COOLING_TEMP);
    coolingController.run();
    catalyticConverter.on();
}

public void stopEngine() {
    fuelInjector.off();
    catalyticConverter.off();
    coolingController.cool(MAX_ALLOWED_TEMP);
    coolingController.stop();
    airFlowController.off();
}
```

Now, **to start and stop a car, we need only 2 lines of code, instead of 13:**

```
facade.startEngine();
// ...
facade.stopEngine();
```

source: www.baeldung.com

- behavioral
- one object (the observable) notifies one or more other objects (the observers) about a change in the observable's state

```
public class NewsAgency {
    private String news;
    private List<Channel> channels = new ArrayList<>();

    public void addObserver(Channel channel) {
        this.channels.add(channel);
    }

    public void removeObserver(Channel channel) {
        this.channels.remove(channel);
    }

    public void setNews(String news) {
        this.news = news;
        for (Channel channel : this.channels) {
            channel.update(this.news);
        }
    }
}
```

source: www.baeldung.com

- behavioral
- allows the change of behavior at run time
- an interface creates a non-specific method inherited by strategy objects
- the context object's behavior changes based on its strategy object

```
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10,

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

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

source: tutorialspoint.com

- behavioral
- an abstract class specifies a list of operations and the child classes define and implement them

```
abstract class OrderProcessTemplate
{
    public boolean isGift;

    public abstract void doSelect();

    public abstract void doPayment();

    public final void giftWrap()
    {
        try
        {
            System.out.println("Gift wrap successful");
        }
        catch (Exception e)
        {
            System.out.println("Gift wrap unsuccessful");
        }
    }

    public abstract void doDelivery();

    public final void processOrder(boolean isGift)
    {
        doSelect();
        doPayment();
        if (isGift) {
```

source: geeksforgeeks.org

- behavioral
- encapsulates an action (or multiple actions) as an object
- object.execute

```
class StereoOnWithCDCommand implements Command
{
      Stereo stereo;
      public StereoOnWithCDCommand(Stereo stereo)
      {
          this.stereo = stereo;
      }
      public void execute()
      {
          stereo.on();
          stereo.setCD();
          stereo.setVolume(11);
      }
```

source: geeksforgeeks.org